Types

```
data type: size, structure, encoding, semantics

type specification: Ts := (T,Is,F)   T: base set,
                                    Is: invarians,  Is(ts) == true
                                     F: operations
type relaization:   TT := (ro,I,S)  ro: E*xTs  representation function
                                     I: E->Bool  invariant
                                     S: S = {S1,...,Sn} operations

Classifications
===============
statically/dynamically typed
strong/weak type system

E.g.: Python: dynamically typed strong(ly) type(d)system
~ polymorphism: not very statically typed

type conversions : widening or narrowing.   int i{3.14}, // error: narrowing
OOP: upcast OK, downcast: ? (checked downcast: dynamic_cast, or (CC) in Java)

Type taxonomy:
==============
  Primitive
    Pointer
    Scalar
      Real  (fixed and floating point representation)
      Integral (incl. char, bool, signed, ...)
  Composite/Aggregate
    Enumeration
    Iterated
      Array
    Union
      tagged (Algol68, Pascal variant record, C++ variant)
      non tagged (C union)
    Cartesian product types, Tuples
      Struct/Class
    ...
    Set  (Pascal,ADA)
    String (C-like, terminator char: '\0', Pascal-like: size at the beginning)
      class


Value semantics vs. reference semantics


Type equivalence  nominal or structural
  ADA R1: Array (1..10) of Integer
      R2,R3: Array (1..10) of Integer  R2<=>R3
```

Object-oriented programming

History
=======

Simula 67
- class == block + name + instantiation
  - attribute == variables in block scope
  - member func == local functions
  - constructor == block runs until first detach (co-routines)
  - local data/method is accessible via scope
  - garbage collection
- inheritance
- virtual function

Smalltalk (Alan Kay) from 70s.. (major version 1980)
- object
  - instance of class
  - store state (and it is not accessible even for the same class)
  - send/receive message
  - dynamic: object (structure) can be changed in runtime
- classes are first-class objects:
  they are created by sendig "subclass" message to class "Object"

Methodology
===========
- OO Analysis
  - abstraction
    - narrow the word to important features
  - classification
    - inner state messages to receive (and how to respond)
  - generalization/specialization

Class := data structure + methods + identity
    e.g. IndexError <: ColumnIndexError

Object creation
===============
 - Constructor: set invariants
 - Factory
 - Cloning

Copy of objects
 - value semantics: C++, C# struct
 - reference semantics: Java, Scala, ADA
 (C++: copy constr., op=. ADA: limited type)

Static members
Static member function
  Global (func. Prog. + current direction in C++)

Inheritance
===========
 - is multiple inheritance exists?
 - how diamond shape inheritance solved
   - Scala traits
   - C++ virtual/non-virtual base classes

Polymorphism

- inclusion polymorphism
- Liskov substitutional principle
- dynamic binding
- Mixin
- CRTP

Safe redefinition in subclasses

UML

- static model: structure of classes
- dynamic model: behavior of classes

Generics

We want to write the software as generic as we can
(but only with no or minimal performance loss)

Polymorphism
- Universal
  - Subtype/inclusion
  - Parametric
 (- raw (structural subtyping))
- Ad-hoc
  - name based (overloading)
  - type cast (coercion)

No Polimorfism: monomorphic type system
   (mostly: Pascal, C (but void*))

generics (templates)
types and algorithms

Constrained generics (hu: Sablon-szerzodés model)
 - Ada "with"
 - Eiffel: subtyping, Java: subtyping + interfaces
 - C++20: Concepts

Implementation
- type erasure (hu: típustörlés) (Java) no specialization
- instantiation (Ada, Eiffel, C++) C#??
  Ada: explicit instantiation, C++: implicit + explicit also possible
- func. prog: ML, Haskell, Clean

Generics + inheritance
  Mixin
  CRTP -> F-bounded polymorphism

Concurrency

Moore's law 1965
https://en.wikipedia.org/wiki/Moore%27s_law#/media/
File:Moore's_Law_Transistor_Count_1971-2018.png

# of integrated circuit transistors duplicates in 24 month
   (at least for the next 10 years)
for long time the speed also duplicated in the same period

Herb Sutter: The Free lunch is Over 2005 DrDobb's Journal
http://www.gotw.ca/publications/concurrency-ddj.htm
The clock speed cannot be increased
--> applications should be utilize multicore systems

Concurrency =/= Parallelism

Why to use concurrent solutions?
More intuitive to solve the problem
 - Server or graphical applications
 - X-Window over DOS - non-preemptive scheduling
  (thread runs untill some external event occurs, mostly I/O)
Running on different devices
More efficient/scalable

Concurrency levels
==========

o Process level
  + Separate address space: better for security and stability
  + No shared memory: no data race
  + Can run on physically different hardware nodes
  - Costly to start a new process (spawn or fork-exec)
  - Costly to switch between processes
  - Costly to communicate
      (IPC = signals, messages, semaphores, shared mem., pipe)
o Thread level
  o Thread as user space lib
    + Faster start and context switch
    - I/O can block the whole branch of threads
  o Thread as kernel lib (M:N)
    + Real concurrent behavior
    - More expensive start and switch
      (still slightly cheaper than start a new process)
o Corutines
  o Stackless
  o Stackfull

Amdahl's law:
  P portion of parallelly executable code
  N execution units

  Performance gain is:

$$\frac{1}{(1-P) + \frac{P}{N}} \xrightarrow{N \to \infty} \frac{1}{1-P}$$

  e.g. if P = 0.95   N->OO   20x
          P = 0.90   10x

Communication and synchronization models

Issues
======

Data race

 Concurrent non-atomic actions on the same memory location
 at least one of them update

Solution:
 busy waiting / spin lock
 mutex/semaphore (semaphore: Dijkstra 1968)
 lock_guards
 conditional variables
    spurious wake up

data channels (Golang)

Deadlock
 - Job interview :)

```
 bool operator<(A a, B b)
 {
   lock_guard(a);
   lock_guard(b);
   return a < b;
 }
```

t1: x < y   t2 : y < x

All of these conditions should occur:
- mutual exclusion
- hold and wait locking
- no preemption (like database manages)
- circular dependencies

Solution:
 algorithms, like Banker's one (???)
 setting the lock always in some predefined order
 detecting deadlock and interact (databases)

```
std::lock(Args... args);
```

Starvation
==========
(resource) starvation
e.g. naive RW lock approaches

C++11 memory model

std::atomic<>

sequential consistency
release-acquare
release-consume
relaxed

- promise-future
- std::async()
- C++17 parallel STL

Others
======

SIMD == Single Instruction Multiple Data
RCU == Read-copy-update patterns

False claims:
 If it concurrent: it is faster
 - Actually, it can be even slower
 - Start a (system) thread is expensive.
   Parallel STL in C++17 starts new threads only over ~10000 elems
Easy to write a sequential prototype and than rewrite it parallel

Hard to debug
  n statements
  t threads   (nt)!
possible execution paths = -----
                           n!^t

Correctness

```
Correctness of software
=======================
- bug finding
  - testing (dynamic)
  - test driven development
  - levels
    - unit
    - integration
    - system
    - acceptance
  - approaches
    - white-box
    - black-box
    - grey-box
  - dynamic analysis
  - static analysis
  - code review
- verification
  - model checking
  - mathematical methods
  - practical:
    - contract

Static analysis
  - pattern matching (context free)
  - AST-based
  - flow sensitive
  - path sensitive  (symbolic execution)

Static safety
- type system
  - Strong type system != Static type system
  - e.g. Python has non-static strong type system
  - Concept: type system for templates

Dynamic safety
- Contract
- Eiffel (Bertrand Meyer): "Design by Contract" (2000)
  - Eiffel: first order predicate logic
    (including quantifiers: there_exists, for_all)

  across my_list as l
     all l.item.some_property = some_value end

  - precondition  "require"
  - postcondition "ensure"
  - invariants    "invariant"
  - assertions    "check"

e.g.
  invariant
    valid_capacity: capacity >0
    valid_item_count: item_count >=0
                      and item_count <= capacity
    definition_of_empty:(empty implies item_count = 0)
                      and (item_count = 0 implies empty)
    definition_of_full: (full implies item_count = capacity)
                      and (item_count = capacity implies full)
```

```
Redefinition - refining of contract
  e.g. inheritance
  Liskov Substitutional Principle

  Der : Base;
  Base b;
  Derived d;

  P1 is stronger than P2  iff P1 => P2

  <Pre',Post'> : Derived  is a sub-specification of
  <Pre, Post>  : Base
    iff Pre => Pre' and Post' => Post

                    ( x'->y' ) <: ( x -> y)
                      x' <: x  &&  y <: y'

  covariant return type
  contravariant input parameters
                  (by type theory, identical by practice)

see:
https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)
Eiffel makes itt wrong :
https://www1.icsi.berkeley.edu/~sather/Documentation/EclecticTutorial/
node15.html

In C++:

Base           { X  f(Y); }
Derived : Base { X' f(Y); }


- Other:
  - assert
  - C++2b Contracts
  - C++11 Concept - axioms -- removed from standard
    "operator< is transitive, non-reflexive"
  - Java ??

Open issues: WCET (Worst Case Execution Time)
  - HUME
  - Rust
```