Concurrent programming

- synchronous and asynchronous models of communication
- description of concurrent, independent activities
- a *task* is an independent thread of control, with own stack, program counter and local environment.
- Ada tasks communicate through
  - rendezvous (think “meeting someone for a date”)
  - shared variables
  - protected objects
- Java threads communicate through shared objects (preferably synchronized)
- C++ has *had* no core language support for concurrency. Now supported in the new standard.
A task type is a limited type

```
task type Worker;        -- declaration;
                         -- public interface

type Worker_Id is access Worker;

task body Worker is      -- actions performed in lifetime
begin
  loop
    compute;
  end loop;
end Worker;
```

---

-- Runs forever;
-- will be shutdown
-- from the outside.
a task type can be a component of a composite
number of tasks in a program is not fixed at compile-time.

\[
\begin{align*}
W_1, W_2 : \text{Worker}; & \quad \text{-- two individual tasks} \\
\text{type Crew is array (Integer range \textlangle\textrangle) of Worker;}; \\
\text{First_Shift : Crew (1 .. 10); \quad \text{-- group of tasks}} \\
\text{type Monitored is record} \\
& \quad \text{Counter : Integer;} \\
& \quad \text{Agent: Worker;} \\
\text{end record;}
\end{align*}
\]
When does a task start running?

- if statically allocated $\implies$ at the next `begin`
- if dynamically allocated $\implies$ at the point of allocation

```
declare
  W1, W2: Worker;
  Joe: Worker_Id := new Worker; -- Starts working now
  Third_Shift: Crew(1..N); -- N tasks
begin  -- activate W1, W2, and the Third_Shift
  ...
end;  -- wait for them to complete
    -- Joe will keep running
```
a task can perform some actions on request from another task
the interface (declaration) of the task specifies the available actions (entries)
a task can also execute some actions on its own behalf, without external requests or communication

```vhdl
task type Device is
  entry Read (X: out Integer);
  entry Write (X: Integer);
end Device;
```
caller makes explicit request: *entry call*

callee (server) states its availability: *accept statement*

if server is not available, caller blocks and queues up on the entry for later service

if both present and ready, parameters are transmitted to server

server performs action

*out* parameters are transmitted to caller

caller and server continue execution independently
Example: semaphore

Simple mechanism to prevent simultaneous access to a *critical section*: code that cannot be executed by more than one task at a time

```plaintext
task type semaphore is
  entry P;  -- Dijkstra’s terminology
  entry V;  -- from the Dutch
    -- Proberen te verlangen (wait) [P];
    -- verhogen [V] (post when done)
end semaphore;

task body semaphore is
begin
  loop
    accept P;
      -- won’t accept another P
      -- until a caller asks for V
    accept V;
  end loop;
end semaphore;
```
A task that needs exclusive access to the critical section executes:

```
Sema : semaphore;
...
Sema.P;
-- critical section code
Sema.V;
```

If in the meantime another task calls `Sema.P`, it blocks, because the semaphore does not accept a call to `P` until after the next call to `V`: the other task is blocked until the current one releases by making an entry call to `V`.

- programming hazards:
  - someone else may call `V` $\Rightarrow$ race condition
  - no one calls `V` $\Rightarrow$ other callers are `livelocked`
A delay statement can be executed anywhere at any time, to make current task quiescent for a stated interval:

```
delay 0.2;  -- type is Duration, unit is seconds
```

We can also specify that the task stop until a certain specified time:

```
delay until Noon;  -- Noon defined elsewhere
```
need to protect against excessive delays, deadlock, starvation, caused by missing or malfunctioning tasks

- timed entry call: caller waits for rendezvous a stated amount of time:

  ```
  select
    Disk.Write(Value => 12,
                Track => 123); -- Disk is a task
  or
    delay 0.2;
  end select;
  
  if Disk does not accept within 0.2 seconds, go do something else
conditional entry call: caller ready for rendezvous only if no one else is queued, and rendezvous can begin at once:

```pascal
select
  Disk.Write(Value => 12, Track => 123);
else
  Put_Line("device busy");
end select;
```

print message if call cannot be accepted immediately
the server may accept a call only if the internal state of the task is appropriate:

```
select
  when not Full =>
    accept Write (Val: Integer) do ... end;
or
  when not Empty =>
    accept Read (Var: out Integer) do ... end;
or
  delay 0.2; -- maybe something will happen
end select;
```

if several guards are open and callers are present, any one of the calls may be accepted – non-determinism
Two notions

- class Thread
- interface Runnable

An object of class Thread is mapped into an operating system primitive

```java
interface Runnable {
    public void run();
}
```

Any class can become a thread of control by supplying a run method

```java
class R implements Runnable {
    ...}
Thread t = new Thread(new R(...));
t.start();
```
class PingPong extends Thread {
    private String word;
    private int delay;
    PingPong (String whatToSay, int delayTime) {
        word = whatToSay; delay = delayTime;
    }

    public void run () {
        try {
            for (;;) { // infinite loop
                System.out.print(word + " ");
                sleep(delay); // yield processor
            }
        } catch (InterruptedException e) {
            return; // terminate thread
        }
    }
}
public static void main (String[] args) {
    new PingPong("ping", 33).start();  // activate
    new PingPong("pong", 100).start(); // activate
}

- call to **start** activates thread, which executes **run** method
- threads can communicate through shared objects
- classes can have synchronized methods to enforce critical sections
C++ didn’t have native thread support until C++11.
Previously had to use external libraries like pthreads, Boost OpenThreads, etc.
Full state-of-the-art thread support now included in C++.
One-to-one mapping to operating system threads.
Based on the Boost thread library.
class Runnable
{
    std::thread mthread;
    Runnable(Runnable const&) = delete;
    Runnable& operator =(Runnable const&) = delete;

public:
    virtual ~Runnable() { try { stop(); } catch(...) { /* clean up */ } }

    virtual void run() = 0;
    void stop() { mthread.join(); }
    void start()
    { mthread = std::thread(&Runnable::run, *this); }
};
class myThread : public Runnable
{
    protected:
      void run() { /* do something */ }
};

Mutual exclusion can be acheived as follows:

static std::mutex pmm;

void mySynchronizedFunction() {
    std::lock_guard< std::mutex > myLock(pmm);
    // critical area
    // unlocked automatically on return
}
int main()
{
    std::future<int> sol = std::launch::async(subset_sum);
    do_other_stuff(); // while subset sum is computing
    std::cout << "The solution to subset sum is: " << sol.get() << std::endl;
}

Variable sol is called a future (a promise to deliver a result in the future). Method get blocks until the future returns.

Invocation of the asynchronous thread, synchronization and communication between main and asynchronous threads all happen automatically.

Replacing async with sync will cause subset_sum to become a deferred function, which runs entirely during the call to get.
**Future**: an object held by the *receiver* of a communication.

To get the value from a future, call `future::get`.

Function `future::get` will block until the value is available.

Can also call `future::has_value` which checks for a waiting result without blocking.

**Promise**: a channel through which a value is communicated to a future.

The promise object (if any) is handled by the communication *sender*.

Promises can be implicit or explicit.

Values are sent through promises implicitly when the thread returns.

Values are sent explicitly ordinarily using `promise::set_value`.

Explicit normally used for manual thread management (e.g., multiple values must be communicated during the lifetime of a thread.)
Generic programming

Allows for type-independent data structures and functions.

Examples:
- A sorting algorithm has the same structure, regardless of the types being sorted
- Stack primitives have the same semantics, regardless of the objects stored on the stack.

One common use:
- algorithms on containers: updating, iteration, search

Language models:
- **C**: macros (textual substitution) or unsafe casts
- **Ada**: generic units and instantiations
- **C++**, **Java**, **C#**: templates
- **ML**: parametric polymorphism, functors
Parameterizing software components

<table>
<thead>
<tr>
<th>Construct</th>
<th>generic parameter(s) are:</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>bounds, element type</td>
</tr>
<tr>
<td>Ada generic package</td>
<td>values, types, packages</td>
</tr>
<tr>
<td>Ada generic subprogram</td>
<td>values, types</td>
</tr>
<tr>
<td>C++ class template</td>
<td>values, types</td>
</tr>
<tr>
<td>C++ function template</td>
<td>values, types</td>
</tr>
<tr>
<td>Java generics (all)</td>
<td>classes, interfaces</td>
</tr>
<tr>
<td>ML function</td>
<td>implicit</td>
</tr>
<tr>
<td>ML type constructor</td>
<td>types</td>
</tr>
<tr>
<td>ML functor</td>
<td>structures (containing types, values)</td>
</tr>
</tbody>
</table>
template <typename T>
class Vector {
public:
    explicit Vector (size_t); // constructor
    T& operator[] (size_t); // subscript operator
    ... // other operations
private:
    ... // a size and a pointer to an array
};

Vector<int> V1(100); // instantiation
Vector<int> V2; // use default constructor

typedef Vector<employee> Dept; // named instance
template <typename T, unsigned int i>
class Buffer {
    T v[i]; // storage for buffer
    unsigned int sz; // total capacity
    unsigned int count; // current contents
public:
    Buffer () : sz(i), count(0) { }
    T read ();
    void write (const T& elem);
};

Buffer<Shape *, 100> picture;
template <typename T> class List {
    struct Link { // for a list node
        Link *pre, *succ; // doubly linked
        T val;
        Link (Link *p, Link *s, const T& v)
            : pre(p), succ(s), val(v) {} }
    }
    Link *head;
public:
    void print (std::ostream& os) {
        for (Link *p = head; p; p = p->succ)
            // operator<< must exist for T
            // if print will be used.
            os << p->val << "\n";
    }
};
Instantiated implicitly at point of call:

```cpp
template <typename T>
void sort (vector<T>&) { ... }

void testit (vector<int>& vi) {
    sort(vi);  // implicit instantiation
    // can also write sort<int>(vi);
}
```
Template types are not initially known.
Uninstantiated templates are not & cannot be compiled.
Generic definitions must be written completely in header files.
Once fully instantiated, all types become known.
Compiler generates classes, functions from the template.
Compilation proceeds in the usual manner after this.
Compiler may optimize by reusing multiple occurrences of a fully instantiated template.
Templates and regular functions overload each other:

```cpp
template <typename T> class Complex {...};
template <typename T> T sqrt (T); // template
template <typename T> Complex<T> sqrt (Complex<T>); // partial specialization
double sqrt (double); // explicit specialization

testit (Complex<double> cd) {
    sqrt(2); // sqrt<int>
    sqrt(2.0); // sqrt (double): regular function
    sqrt(cd); // sqrt<complex<double> >
}
```

Partial specialization narrows the set of acceptable template parameters. Compiler will select the most specialized (specific) type.
Containers are data structures to manage collections of items

Typical operations: insert, delete, search, count

Typical algorithms over collections use:

- imperative languages: iterators
- functional languages: map, fold, recursion

```java
interface Iterator<E> {
    boolean hasNext ();  // returns true if there are more elements
    E next ();          // returns the next element
    void remove ();     // removes the current element from the collection
}
```
**STL**: A set of useful data structures and algorithms in C++, mostly to handle collections.

- Sequential containers: `list`, `vector`, `deque`
- Associative containers: `set`, `map`

We can *iterate* over these using (what else?) *iterators*.

Iterators provided (for `vector<T>`):

- `vector<T>::iterator`
- `vector<T>::const_iterator`
- `vector<T>::reverse_iterator`
- `vector<T>::const_reverse_iterator`

Iterator concepts: trivial, input, output, forward, bidirectional, and random access.
For standard collection classes, we have member functions `begin` and `end` that return iterators.

We can do the following with an iterator `p`:

- `*p` — “Dereference” it to get the element it points to (trivial)
- `++p, p++` — Advance it to point to the next element (forward)
- `--p, p--` — Retreat it to point to the previous element (bidirectional)
- `p+i, p-i` — Advance/retreat it `i` times (random access)
- `p[i]` — Access index `i` (random access)

A sequence is defined by a pair of iterators:

- the first points to the first element in the sequence.
- the second points to `one past` the last element in the sequence. Cannot deference, but must still be valid.

There are a wide variety of operations that work on sequences.
#include <vector>
#include <string>
#include <iostream>

int main () {
    using namespace std;
    vector<string> ss(20);  // initialize to 20 empty strings
    for (int i = 0; i < 20; i++)
        ss[i] = string(1, 'a'+i);  // assign "a", "b", etc.
    vector<string>::iterator loc =
        find(ss.begin(), ss.end(), "d");  // find first "d"
    cout << "found: " << *loc
        << " at position " << loc - ss.begin() << endl;
}
STL provides a wide variety of standard “algorithms” on sequences.

Example: finding an element that matches a given condition

```cpp
// Find first 7 in the sequence
list<int>::iterator p = find(c.begin(), c.end(), 7);

// Find first number less than 7 in the sequence
bool less_than_7(int v) {
    return v < 7;
}

list<int>::iterator p = find_if(c.begin(), c.end(), less_than_7);
```

// C++11:
```cpp
auto p = find_if(c.begin(), c.end(), less_than_7);
```
Example: doing something for each element of a sequence

It is often useful to pass a function or something that acts like a function:

```cpp
template <typename T>
class Sum {
    T res;
public:
    Sum (T i = 0) : res(i) { } // initialize
    void operator() (T x) { res += x; } // accumulate
    T result () const { return res; } // return sum
};

void f (list<double>& ds) {
    Sum<double> sum;
    sum = for_each(ds.begin(), ds.end(), sum);
    cout << "the sum is " << sum.result() << "\n";
}
```
template<typename Arg, typename Res> struct unary_function {
    typedef Arg argument_type;
    typedef Res result_type;
};

struct R {
    string name; ... }

class R_name_eq : public unary_function<R, bool> {
    string s;
public:
    explicit R_name_eq(const string& ss) : s(ss) { }
    bool operator()(const R& r) const { return r.name == s; }
};

void f(list<R>& lr) {
    list<R>::iterator p = find_if(lr.begin(), lr.end(),
                                    R_name_eq("Joe"));
    ...
}
template <typename Arg, typename Arg2, typename Res>
struct binary_function {
    typedef Arg first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Res result_type;
};

template <typename T>
struct less : public binary_function<T, T, bool> {
    bool operator() (const T& x, const T& y) const {
        return x < y;
    }
};
template <typename BinOp>
class binder2nd
  : public unary_function<typename BinOp::first_argument_type,
   typename BinOp::result_type> {
protected:
  BinOp op;
  typename BinOp::second_argument_type arg2;
public:
  binder2nd (const BinOp& x, 
             const typename BinOp::second_argument_type& v)
    : op(x), arg2(v) { }
  return_type operator() (const argument_type& x) const {
    return op(x, arg2);
  }
};

template <typename BinOp, typename T>
binder2nd<BinOp> bind2nd (const BinOp& op, const T& v) {
  return binder2nd<BinOp> (op, v);
}
Partial application with function objects

void f (const list<int>& xs, int limit) {
  list<int>::const_iterator it =
    find_if(xs.begin(), xs.end(),
            bind2nd(less<int>(), limit));
  int num = it != xs.end() ? *it : limit;
  ...
}

“Is this readable? ... The notation is logical, but it takes some getting used to.” – Stroustrup, p. 520

Equivalent to the following in ML:

fun f xs limit =
  let val optNum = List.find (fn x => x < limit) xs
    val num = Option.getOpt (optNum, limit)
  in ... end
Templates in C++ allow for arbitrary computation to be done at compile time!

```cpp
template <int N> struct Factorial {
    enum { V = N * Factorial<N-1>::V }
};

template <> struct Factorial<1> {
    enum { V = 1 }
};

void f () {
    const int fact12 = Factorial<12>::V;
    cout << fact12 << endl;  // 479001600
}
```
Only class parameters (no value)

Implementation by *type erasure*: all instances share the same code

Unlike C++, generics are fully compilable (uninstantiated).

```java
interface Collection <E> {
    public void add (E x);
    public Iterator <E> iterator ();
}
```

Collection `<Thing>` is a parametrized type

Collection *(by itself)* is a raw type!
class Collection<A extends Comparable<A>> {
    public A max() {
        Iterator<A> xi = this.iterator();
        A biggest = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (biggest.compareTo(x) < 0)
                biggest = x;
        }
        return biggest;
    }
    ...
}
Functors yield *structures*, similar to the way C++ templates yield concrete classes.

Why functors, when we have parametric polymorphic functions and type constructors (e.g., containers)?

- Functors can take structures as arguments. This is not possible with functions or type constructors.
- Sometimes a type needs to be parameterized on a *value*. This is not possible with type constructors.
Example functor: the signature

Similar to an interface (Java) or forward declaration (C++).

```plaintext
signature SET =
sig
  type elem
  type set

  val empty : set
  val singleton : elem -> set
  val member : elem * set -> bool
  val union : set * set -> set
  ...
end
```
functor SetFn (type elem
   val compare : elem * elem -> order) : SET =
structure
   type elem = elem
   datatype set = EMPTY
      | SINGLE of elem
      | PAIR of set * set

   val empty = EMPTY
   val singleton = SINGLE

   fun member (e, EMPTY) = false
   | member (e, SINGLE e') = compare (e, e') = EQUAL
   | member (e, PAIR (s1, s2)) = member (e, s1) orelse
     member (e, s2)

...
structure IntSet = 
    SetFn (type elem = int
           compare = Int.compare)
structure StringSet = 
    SetFn (type elem = string
           compare = String.compare)

fun cmp (is1, is2) = ...

structure IntSetSet = SetFn (type elem = IntSet.set
                              compare = cmp)

Compare functor implementation with a polymorphic type: how are element comparisons done?
I/O for integer types.

Identical implementations, but need separate procedures for strong-typing reasons.

```ada
generic
  type Elem is range <>;  -- any integer type
package Integer_IO is
  procedure Put (Item: Elem);
  ...
end Integer_IO;
```
generic
type Elem is private; -- parameter
package Stacks is
  type Stack is private;
  procedure Push (X: Elem; On: in out Stack);
...
private
  type Cell; -- linked list
  type Stack is access Cell; -- representation
  type Cell is record
    Val: Elem;
    Next: Ptr;
  end record;
end Stacks;
with Stacks;
procedure Test_Stacks is
package Int_Stack
  is new Stacks (Integer);  -- list of integers
package Float_Stack
  is new Stacks (Float);  -- list of floats
S1: Int_Stack.Stack;  -- stack objects
S2: Float_Stack.Stack;
use Int_Stack, Float_Stack;  -- OK, regular packages
begin
  Push(15, S1);
  Push(3.5 * Pi, S2);
  ...
end Test_Stacks;
The syntax is: type T is ...;

<table>
<thead>
<tr>
<th>Restriction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>any type with basic operations (e.g., assignment, equality)</td>
</tr>
<tr>
<td>limited private</td>
<td>any type (no required operations)</td>
</tr>
<tr>
<td>range &lt;&gt;</td>
<td>any integer type (arithmetic operations)</td>
</tr>
<tr>
<td>(&lt;&gt;))</td>
<td>any discrete type (enumeration or integer)</td>
</tr>
<tr>
<td>digits &lt;&gt;</td>
<td>any floating-point type</td>
</tr>
<tr>
<td>delta &lt;&gt;</td>
<td>any fixed-point type</td>
</tr>
</tbody>
</table>

Within the generic, the operations that apply to any type of the class can be used.

The instantiation must use a specific type of the class.
A generic function

generic
  type T is range <>; -- parameter of some integer type
  type Arr is array (Integer range <>) of T; -- parameter is array of those
function Sum_Array (A: Arr) return T;

-- Body identical to non-generic version
function Sum_Array (A: Arr) return T is
  Result: T := 0; -- some integer type
begin
  for J in A’range loop -- array: ’range available
    Result := Result + A(J); -- integer: "+" available
  end loop;
  return Result;
end;
type Apple is range 1..2**15 - 1;
type Production is array (1..12) of Apple;

type Sick_Days is range 1..5;
type Absences is array (1..52) of Sick_Days;

function Get_Crop is new Sum_Array (Apple, Production);
function Lost_Work is new Sum_Array (Sick_Days, Absences);
The only available operations are basic operations, which include assignment and equality.

generic
  type T is private;
procedure Swap (X, Y: in out T);

procedure Swap (X, Y: in out T) is
  Temp: constant T := X;
begin
  X := Y;
  Y := Temp;
end Swap;
A generic sorting routine should apply to any array whose components are comparable, i.e., for which an ordering predicate exists. This class includes more than the numeric types:

```haskell
generic
type T is -- parameter
  private;
with function "<" (X, Y: T) -- parameter
  return Boolean;
type Arr is -- parameter
  array (Integer range <>) of T;
procedure Sort (A: in out Arr);
```
The actual must have a matching signature, not necessarily the same name:

```plaintext
procedure Sort_Up is
    new Sort (Integer, "<", ...);

procedure Sort_Down is
    new Sort (Integer, ">", ...);

type Employee is record ... end record;
function Senior (E1, E2: Employee) return Boolean;
function Rank is new Sort (Employee, Senior, ...);
```
Useful to parameterize containers by size:

generic
  type Elem is private; -- type parameter
  Size: Positive; -- value parameter
package Queues is
  type Queue is private;
  procedure Enqueue (X: Elem; On: in out Queue);
  procedure Dequeue (X: out Elem; From: in out Queue);
  function Full (Q: Queue) return Boolean;
  function Empty (Q: Queue) return Boolean;
private
  type Contents is array (Natural range <>) of Elem;
  type Queue is record
    Front, Back: Natural;
    C: Contents (0 .. Size);
  end record;
end Queues;
generic
  type Real is digits <>; -- any floating type
package Generic_Complex_Types is
  -- complex is a record with two real components
  -- package declares all complex operations:
  --    +, -, Re, Im...
  ...
end Generic_Complex_Types;

We also want to define a package for elementary functions (\texttt{sin}, \texttt{cos}, etc.) on complex numbers. This needs the complex operations, which are parameterized by the corresponding real value.
with Generic_Complex_Types;
generic
  with package Compl is
    new Generic_Complex_Types (<>);
package Generic_Complex_Functions is
  -- trigonometric, exponential,
  -- hyperbolic functions.
...
end Generic_Complex_Functions;

¬ Instantiate complex types with long_float components:

package Long_Complex is
  new Generic_Complex_Types (long_float);

¬ Instantiate complex functions for long_complex types:

package Long_Complex_Functions is
  new Generic_Complex_Functions (long_complex);