Principles of Programming Languages

-- Imperative languages Part 1: basics

What's imperative programming ?

- Method of programming that:
  - Has fully specified control flow
    - Control flow is 'step wise' managed.
  - Has fully specified data management
    - Data is managed by 'named items'
- Programming is very explicit
  - 'do this, then that'
  - Data X added to Data y
- Fits well in common human thought patterns
  - Cooking recipes, building plans, etc

Pro's – con's

- Pro
  - High performance
  - Easy to understand programs
- Con
  - Very machine oriented
    - Where/when to allocate data
    - Exactly which step follows which

Data

- Data is usually 'typed'
  - A type
    - Representation in memory
    - A set of properties
- Most data is 'named'
  - Int x;
- Some data is 'anonymous'
  - 4 * 5
- Data declaration
  - Relationship between names, types and representations

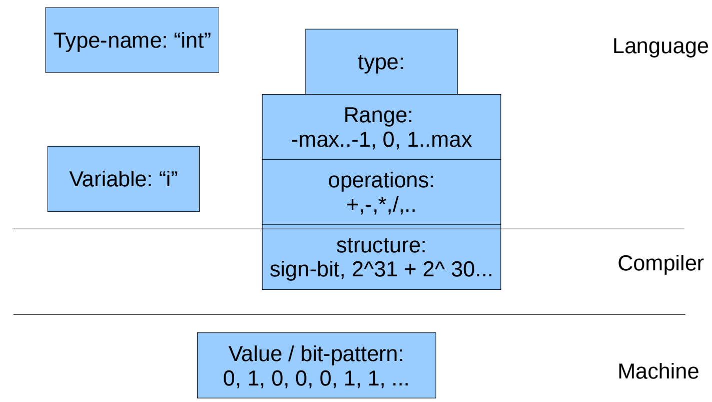
Data declaration

- A range of memory addresses needs to be given structure to be meaningful
  - Data declation gives
    - Name to address range
    - Imposes structure on address range
- A variable is
  - A combination of name, type, value.
- Types can be named too

C	Ada, Pascal, Modula
int a, b;	A, B: Integer

Data declarations

- And graphically:



Data declarations

- Types of declarations / declaration modifiers
  - ...many...

C/C++/Java	Ada
int a = 5, b = 5;	A,B: Integer := 5;

Or;

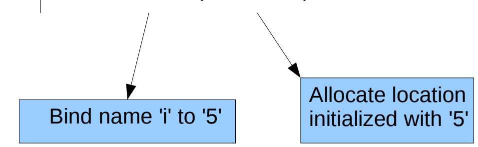
A: Integer := 5;
B: Integer := 5;

Both ways reasonable:
- one initialization expression
- 2 variables --- 2 expressions

Data declarations

- Split allocation and naming of variables ?

C/C++/Java	Algol 68
int a = 5, b = 5;	ref int i = (loc int := 5)



*The same holds for some OO languages: the constant '5' is an object and 'i' is a reference to it.

Data declarations

- Declaration of constants

C/C++/Java	Ada
const int a = 5, b = 5;	A,B : constant Integer := 5;

In almost all cases, compiler/language-rules enforce 'const'.

Uninitialized Variables

- Some languages allow data declarations without initialization expressions (C, Ada, etc)
- Usage of uninitialized data causes bugs, language designers can:
  - Ignore (program problem)
  - Runtime checks
  - Silently initialize uninitialized declarations
  - Language allows only initialized variables
  - Meta value (outside of type) that declares initialization state.
    - Integer = {omega, min_int, ... -1, 0, 1, ... max_int}
    - Omega + 5 = omega

Renaming and aliasing

- Provide an alternative name for an expression

Ada	C++
K : Integer renames L;	int &K = L

Ada	C++
K : Integer renames A((N+M)/2)	int &K = A[(N+M)/2]

Can cause many surprises as changing L will change K..

Overloading

- Binds one name to multiple objects
  - Opposite to renaming (multiple names for 1 object)
- Context in which name is used tells which **actual** object to use
  - Its a 'foot' long
  - My foot's too long
- Math operators usually overloaded (implicitly)
  - 4 + 5 (instead of add_int(4,5))
  - 4.1 + 5.1 (instead of add_real(4.1, 5.1))

Primitive types

- Hardware usually provides:
  - Character (0..255)
  - Integer, 16 / 32 / 64 bit
    - Unsigned (0..65536), signed (-32768 ...32767)
  - Real, 32 / 64 bit.
- The 'none' type is sometimes also explicit (no bits in memory associated)
  - 'void' in Java/C/C++
- Range / precision is language specific:

Ada
type dollar is range 0 .. 99 -- in cents type hour is range 0 .. 59   -- in minutes

Type constructors

- Build new types from existing types
  - Can be used directly/anonymously

C/C++	Ada
int a[10];	A: array (Integer range 1 .. 10) of Integer

- Constructed types can be named:

C/C++	Ada
typedef int10[10]; int10 a;	type int10 is array (Integer range 1 .. 10) of Integer A: int10;

C/C++: it looks like a[10] is valid, however, the array index is from 0 .. 9 !!

Compound types

- Types consisting of multiple different types
  - Combination of 'int' and 'double' as a single type for example
- Lets look at a number of type-constructors first..

Enumeration types

- Defines set of disjunct names as values of the type (names, instead of bits in memory)

C	Ada

enum colors {
    red, yellow, green
}
	type color is {
	    red, yellow, green
	};

enum colors c;	C: color

typedef enum {
    true, false
} boolean;
	Type Boolean is {
	        True, False
	};

Enumeration types

- Most languages allow copying and comparison on enumeration types
  - Some also allow >,<,<=,>= be giving each name an implicit integer based on ordering in declaration

C	Ada

enum days { mon, tue, wed, thu, fri };
 int x = wed > thu;
	type days is {
	 mon, tue, wed, thu, fri
	 }
	 X : Integer := wed > thu;

Note: some languages have the 'ASCII' character set as enum (ex: versions of modula)

Arrays

- Series of known number of items of some type
  - Can be indexed with an ordinal number
  - Often have operators to get lower/upper bound / index of first/last element in series.

Java	C/C++	Ada
A[0]	a[0]	A'First
not available	not available	A'Last
a.length	sizeof(a)	A'Length

Ada/Modula: lower/upper bounds defined by programmer
Java/C/C++: lower bound always '0'
C/C++: sizeof delivers size in bytes, not in #elements..
#elements = 'sizeof(a) / sizeof(a[0])'

Compound values / aggregates

Int a[] = {3,5,6}

A: array (Integer range 1 .. 3) of Integer :=
{3,4,5};

Dynamic vs static array bounds

Java	C/C++	Ada
int []a = new int[k];	<not possible>	a: array (Integer range M .. N) of Integer;

* Flexible arrays: resize arrays after they have already been allocated
  - YES: Python
  - NO: Ada, C/C++, Java

** Can simulate flexible arrays in C using malloc/realloc
** can simulate in general using indirection: abstract away from array implementation using a module with get(array, index), set(array, index, value) functions

Array forms

- Array of arrays
  - Every element can have a different size sub-array
  - Can reassign a sub-array to another array
    - a[i] = new int[1000];
  - Accessed mostly via a[i,j,..] = ...
- Multidimensional arrays
  - Every element has an equal size sub-array
  - Accessed mostly via a[i][j]... = ...

	Multi-dim	arrays-of-arrays
Ada	yes	yes
Java	no	yes
C/C++	yes	no

Slicing an array

- Access a range of array elements

Ada

```
A(1 .. 3) := (10,11,12);
```


Array indexing methods

- Enumerations sometimes allowed:
  - enum material{ .., coal, ...}; price[coal] = 3.45;
- Characters sometimes allowed
  - price['c'] = 3.45
- Some languages allow all types:
  - Python
    - price[“string”] = 3.45;
  - ABC
    - Given arbitrary datatype 'coal'
      - PUT 3.45 IN price[coal]
      - WRITE price[coal]
  - “associative array”

Sequences / lists

- Some languages have list as a basic datatype
  - Array = fixed size
  - Sequence = series of unknown number of elements, all of the same type
    - accessing via iteration: “first”, “last”, “previous” and “next”
      - Are all operators in language
      - No 'length' operator (or it would be an array)
  - Character strings are logically sequences but modeled mostly as arrays.

Sets and Bags

- Set
  - Set of values with a fixed type
  - No duplicate values
- Bag
  - Duplicates allowed
  - Implemented in "ABC"

Interesting operators over sets:
- powerset S: set of all subsets (SETL)
- union (S1 + S2)
- intersection (S1 - S2)

Modula-2:

```
TYPE CharSetType = SET OF CHAR;
VAR CharSet         : CharSetType;

CharSet := CharSetType { 'a', 'b', 'c' }
IF 'b' in CharSet THEN
        writeln ...
END IF;
```


Records and Pointers

- Record: a group of a fixed number of types

C	Ada

struct person {
    const char *name;
    unsigned age;
    int account;
};
	type Person is record
	    Name: String
	    Age: Natural;
	    Account: Integer;
	end record;

Struct person p;	P: Person;
p.age = 3;	P.Age := 3;

Recursive records

```
// can not write:
struct tree {
    struct tree left;
    struct tree right;
    int value;
};
// as something of X bytes can
// not contain 2 * X bytes + sizeof(int)
```

```
-- can not write
type Tree is record
      Left, Right: Tree;
      Value: Integer;
end record;
```

```
// correct version:
struct tree {
    struct tree *left, *right;
    int value;
}

// here left and right are pointers
 (potentially 'NULL')
```

```
-- correct
type Tree; -- forward declaration of type
type TreeAccessType is access Tree;
type Tree is record
      Left, Right: TreeAccessType;
      Value: Integer;
end record;
```


Recursive types

- Pointer access via operators:

C	Ada
Tree *a;	A: TreeAccessType
Tree t = *a	T: Tree := A.all;
(*a).value	A.all.value
a->value	A.value

Note: The -> in C is not actually needed, the language/compiler would be fine with '.' always (like Ada). Language designers: . And -> to make code more understandable.

Record Allocation

C	Ada
Struct tree *t = (struct tree*) malloc(sizeof(tree));	T: Tree := new Tree

- Notes:
  - Malloc() is a library function in C, 'new' is a builtin operator in Ada.
  - Pointers for efficiency: copying a pointer is more efficient than copying the data!!

Data aliasing

- Creating a pointer to a variable

C	Ada
Int i = 5;	I: aliased Integer
int *ip = &i;	type IntAccess is access Integer; IP : IntAccess := I'Access;
*ip = 123;	IP.all = 123;
printf(“i = %d\n”, i);	PUT(I);

References

int i = 5;	I: Integer;
int &ir = i;	IR: Integer renames I;
ir = 10;	IR := 7;
printf(“%d\n”, i);	PUT(I);

- Notes:
  - Dangling references: reference used while referred to object is gone
  - Pointers not valid across machine boundaries...
  - Need to check almost every pointer usage....
  - Some (imperative) languages have no pointers: lists/trees/etc are builtin types
    - Orca, ABC, SETL, etc.

Unions

- A record contains fields 0 (of type x) **AND** 1 (of type y), **AND** 2, **AND** 3..., etc
- A union contains fields 0 (of type x) **OR** 1 (of type y) **OR** 2 **OR** 3, etc.
  - Reuse the same memory for all fields ?
  - How to decide which field is valid ?
    - Undescriminated: C/C++
    - Descriminated: Ada
  - Access to fields of union the same as a record

Unions

C:

```
union container {
      int wheels;
      long cans_of_beans;
}
```

Ada:

```
type Content is (Has_Wheels,
      Has_Cans);

type Container(Status: Content) is record
   case status is
      when Has_Wheels =>
            wheels: Natural;
      when Has_Cans =>
            cans_of_beans: BigNatural;
   end case
end record;
```

```
Union container c;
c.wheels = 3;
printf("%ld\n", c.cans_of_beans);
```

```
c: Container(Has_Wheels);
c.wheels = 3;
Put(c.cans_of_beans); // RUNTIME ERROR
```


Functions as data types

- Useful by, for example:
  - Instead of testing which function to call, call a function over a pointer for each case:
    - F = array[i]
    - F();
  - Functions as data, then functions have types

C	Ada
// declare function pointer variable int (*convert)(float);	-- type declaration type Converter is access function (F:Float) return Integer;
// implicit conversion: int i = convert(1.23);	-- declare variable of function pointer type Convert: Converter;
// explicit conversion: int i = (*convert)(1.23);	-- call over function pointer: I = Convert(1.23);

Type Orthogonality

- Combination orthogonality
    - If one member of X can be combined with Y, then all of X can:
      - If have types and typed-declarations, then all types can be fitted for all declarations: 'T x = <expr>' then allow 'int x = <expr>' and 'float x = <expr>'
  - Sort orthogonality
    - If one member of X can be meaningfully combined with Y, then all of X can be meaningfully
    - Allow any type to be used as a subtype in arrays/records/unions/sets/lists
      - Array of records
      - Records containing arrays
      - Sets of lists of arrays, etc.

Type Orthogonality

- Number orthogonality
  - When one of X is allowed meaningfully, then 0,1,2, etc instances are meaningfully allowed also.
    - If one statement is allowed as a sub-statement, then 0, 1, 2, 3, etc statements are allowed also
- Orthogonality is good: no surprises
  - Forces language designer to make language/implementation 'clean'

Restricted types

- Some languages restrict the value range of some type:
  - Integer only allowed value from X to Y
  - List only allowed to have at max N nodes
  - Etc (however, generally, only 'simple' restrictions)
- In general: a good idea to have the most precise type for a data type:
  - int weekday; // Is 10000 is day-of-the-week ? (could be 'error value' ?)
  - int weekday[1..7];
    - Better describes day-of-the-week: more readable code
    - No checks needed to see if value in range

Type Equivalence

- When are types equivalent ?
  - "Integer [2..10]" == "Integer [0..8]" ?
  - "Integer [2..10]" == "Integer[0..3]" ?
  - "typedef int x" == "int" ?
  - "struct x {int p;}" == "struct y {int z;}" ?
- Structural equivalence
  - Algol 68
    - Mode t1 = struct(int val; ref mode t1);
    - Mode t2 = struct(int val; ref mode t2);
    - Mode t3 = struct(int val1; ref struct (int val2; ref mode t3));
- Name equivalence
  - What about anonymous types ?
    - X: int[10][13];

Type Equivalence

- Name equivalence in Ada

```
IA1: array(Integer range 1..10) of Integer
IA2: array(Integer range 1..10) of Integer

is shorthand for:

type anon_1 is array(Integer range 1..10) of Integer;
IA1: anon_1;
type anon_2 is array(Integer range 1..10) of Integer;
IA2: anon_2;


-- Which means that IA1 and IA2 are not assignment compatible
-- as they have different types !!
```


Type Equivalence

- Structural equivalence in C:

```
typedef int apples;
typedef int pears;

apples a = 3;
pears p = 9;

p = a + 1; // no problem: int + int = int
           // even though pears/apples are different types/concepts !
```

Coercions / Contexts / Casts

- Need to convert one type into another
  - To fill gaps in type system
  - Force reinterpretation of memory
- Coercions: implicit type conversions
  - Float x = 3.14 + 5;
    - // float + integer = float
  - Are coercions good/bad ?
    - Compiler does things behind programmer's back...
    - Float x = 3.14 + float(5);
    - Float x = 3.14 + 5.0;
    - Float x = float(int(3.14) + 5))
- C/C++ has coercions, Ada does not

Coercions / Contexts / Casts

- How to coerce depends on context
- If coercion has too little context: manually cast
  - Cast = name type of result, let coercion protocol work it out
    - (C) int a = (int) 3.14; *// coercion protocol casts*
  - Conversion = a function call that does the work
    - (Ada) A: Integer := Integer(3.14); *-- 'Integer()' is function call*
  - *'voiding coercion':* take value and throw it away:
      - int foo() { return 5; }
      - foo(); // return value thrown away (don't have to write 'int d = foo();')
    - Downside:
      - void foo() { “hello”; } // is ok: “string” converted to “void”

Assignments

- Usually:
  - Lvalue = rvalue
    - Lvalue indicates memory address, rvalue delivers a value and is in the form of an expression
  - differentiate syntactically between assignment and equality
    - C: '=' and '=='   Ada: ':=' and '='
  - Does not create a permanent relation:
    - X=4
    - X=6; // valid
- ABC: PUT 1234 in weight[car]

Assignments

- Completely evaluate the RHS ?
  - Given array slice copy:
    - Ada:
      - S: array (Integer range 1..7) of Character := {'a','b','c','d',e',f','g'};
      - S(4..6) := S(3..5)
        - -- first copies 'c','d','e' to temporary
      - PUT(S); -- results in 'a','b',c','c','d','e','g'
    - Alternatively (not Ada!)
      - Copy characters one-by-one to destination (without temp)
      - S(4) := S(3); S(5) := S(4); S(6) := S(5);
        - abccccg

Assignment operators

- Shorthand: Instead of “Dest = dest <op> expr”
  - Have: “Dest <opx> expr”
  - C: opx can be: +=, -=, *=, etc
  - Modula2/3: +:=, -:=, *:=, etc.
- C optimized this more with
  - ++X *// instead of 'x = x + 1; x;'*
    - *last x voidable, can also have –X (but not **X, //X ^^X, %%X, etc ?)*
    - *pre-increment*
  - *X++ // instead of 'int tmp = x; x = x + 1; tmp;'*
    - *last tmp voidable, can also have ++X*
    - *post-decrement*

Assignment operators

- *Assignment operators help programmer*
      - *A[2*i*j/k+g^d] = A[2*i*j/k+g^d] + 1*
    - *Vs*
      - *A[2*i*j/k+g^d] ++;*
  - *Helps compilers too to generate better code (without complex analysises) as index expression only evaluated once.*

Infix notation

- “4+5”
- Important terms:
  - Operator priority: * > +, * > /, / > +
    - 5 / 4 + 2 == (5 / 4) + 2
  - “(“ and “)” used to override operator priorities
  - left/right associative
    - Evaluation order on terms with the same priority
      - “X := Y := Z := 3” == “X := (Y := (Z := 3))”
        - *Multiple assignment*
  - Monadic (or unary) operators take 1 argument
  - Dyadic (or binary) operators take 2

Infix notation

- Most Monadic operators are prefix notated
  - -3
  - -foo()
- Example exception
  - X++
- Difficult to get right:
    - a/b*c
      - (which was usually meant to do (a/(b*c)) instead of (a/b)*c
  - or
    - B: Boolean
    - B := 0 <= x <= 5 *-- as in std. math notation*

Conditional expression

- Q := IF x /= 0 THEN 1/x ELSE 0;
  - Ternary expression (takes 3 operands)
- In C:
  - Q = x != 0 ? 1/x : 0;

Postfix/prefix notation

- \+ 4 5 *// instead of 4 + 5*
  - *3 + 4 * 5 == + 3 * 4 5*
- *4 5 + // instead of 4 + 5*
  - *3 4 5 * +*

Lazy Evaluation

- Only evaluate what must be evaluated:
    - IF i<10 AND x[i] > 10 THEN ...
  - Only evaluate x[i] if i <10 !
  - “short cut operators” or “short circuit operators”
- Must used in 'functional programming'
  - Later..

C	ADA
if ((x < 10) && (x[i] > 10)) ...	if x<10 and then x[i] > 10 then .... end if

External state: Output

- Interface with operating system / other programs / humans
  - Some languages implicitly convert values to strings
    - printf(“%d”, a); Put(a);
  - “%d” is format string: decimal
    - Many languages have only one predefined format.
  - What if you want the string ?
    - Some languages use a 'fake' file..
  - Many use polymorphic functions (C,ADA)
    - functions that can be applied to many types
    - Mostly excludes strong type checking (C)

External state: Input

- Problem: malformed input, we need two return values (error value + return value)
  - C:
    - int e = scanf(“%d”, &i)
      - Reads one integer and stores it in 'i'
      - Return value of scanf signals error.
  - Ada:
    - Get(I);
      - Raises exception on error (see later)

Flow of Control

- Imperative languages: lots of control
  - Sequencer tells which to run next, normally textually/logically following
    - i++; j++; I := I + 1; J = J + 1
  - 'goto statement'
    - Dijkstra paper: 'goto considered harmful”
    - Currently: use gotos sparingly but there are cases where its ok (see Linux kernel source code)

```
        P = <allocate resource>
        Code1;
        if (error1) goto handle_error;
        Code2;
        if (error2) goto handle_error;
        goto success;
handle_error:
        <free resource P>
        return 0;
success:
        return P;
```

```
Goto Handle_Error1
    ...

<<Handle_Error1>>
```


Flow-of-control: Selection

- Based on value of condition, execute one of the other of two pieces of code:

```
if (x < y) {
        printf(“hi\n”);
} else {
        printf(“lo\n”);
}
```

```
if x < y then
        Put(“hi”);
else
        Put(“lo”);
end if
```

'Else' can be optional in most languages:

```
if (x < y) {
        printf(“hi\n”);
}
```

```
if x < y then
        Put(“hi”);
end if
```


Flow-of-control: Selection

- Case / switch statements:

```
switch (n) {
        case 0: printf(“0”); break;
        case 1: printf(“1”); break;
        default: printf(“?”); break;
}
```

```
case N is
        when 0 => Put(“0”);
        when 1 => Put(“1”);
        when others =>  Put(“?”);
end case;

-- Ada note: “case” != “select”
-- select has to do with tasks
```

case/switch should atleast have a specification that explains what happens if:
- multiple value for the same action ? (Ada + C)
- value ranges for an action ? (Ada yes, C no)
- must the values be compile time constants ? (Ada+C)
  - can values occur multiple times ? (error for Ada+C)
- must the values cover the possible values from the case/switch expression ?
  C: no, Ada: yes
- does control flow from case to the next case or to the end of the switch ?
  C,C++: to next case; ada,C#, pascal, modula-X: to end of switch

Flow-of-control: Selection

- For Ada, case && union types cooperate nicely:
  - Type-definition
  - usage

```
type Container(Status: Content) is record
   case status is
      when Has_Wheels =>
         wheels: Natural;
      when Has_Cans =>
         cans_of_beans: BigNatural;
   end case
end record;

c: Container;
...
case c.Status is
      when Has_Wheels => ...
      when Has_Cans => ..
end case;
```


Procedures

- When set of statements can be thought of in independence:
  - Give set of statements a 'name'
  - Give inputs 'names'
  - Control returns after procedure finishes (more later)
  - Difference between 'function' and 'procedure': functions have return-values.

```
read_line();
```

```
read_line;
-- in ada: no arguments needed then
-- no () needed either
```


Repetition

- Two kinds:
  - Repetition over a precalculated(finite) set
    - 'for each in X', 'for(X in ...)', 'for(..;x;..)'
  - Repetition for as long as a condition is met
    - 'while (X)'

```
Int sum = 0;
int i;

for (i=0; i<n; i++)
      sum += a[i];
```

```
Sum: Integer;

Sum := 0;
for I in 0..N loop
      Sum := Sum + A[I];
end loop;
```

'i<n' is tested each iteration
-> for loop actually a 'while' loop !
Initialization, condition and update expressions can be arbitrary: great flexibility = lots of bugs?

Very inflexible..

Repetition: 'for'

- Who declares the control variable ?
  - C: the programmer, Ada: the language
- Is the control variable alive after the loop ?
  - What is its value after ?
- Can the control variable be changed in the loop ?
  - C: yes, Ada: no
- Can the loop bounds be changed inside the loop ?
  - C: yes, Ada: no (loop bounds evaluated **before** the loop runs)

Repetition: While--do

- While e do loop ... end loop
  - Ada
- do .. while(e);
  - C
- while(e) ... <do>;
  - C (no equivalent in Ada)
- Repeat ... until(e);
- while(e) ... repeat;

Note: best avoid “do .. while(e);' and 'repeat ... until(e) as they do not handle 'empty' sequences / sets !

Loop-exits

- Exit a loop in the middle of an iteration
  - Some critique: breaks all/nothing semantics of loop iterations, programmer should write smaller loops / better loop exits tests / etc

```
while (! end_of_file(f)) {
      line = read_next_line(f);
      if (line_is_empty(line))
            break;
      process_line(line);
}
```

```
while not End_Of_File(f) loop
      line := Read_Next_Line(f);
      if Line_Is_Empty(line) then
            exit;
      end if;
      Process_Line(line);
end loop;
```

Note: in C the 'break' statement is reused for switch-case termination.

Loop-exits

- Multi-loop early exit
  - Exit from multiple nested loops in one 'exit'

```
// hack:

for (i=0;i<10;i++) {
   while (! end_of_file(f)) {
      line = read_next_line(f);
      if (line_is_empty(line))
            goto exit_all_loops;
      process_line(line);
   }
}
exit_all_loops:
```

```
Read_Data : for I in 0..10 loop

      while not End_Of_File(f) loop
            line := Read_Next_Line(f);
            if Line_Is_Empty(line) then
                  exit Read_Data when true;
            end if;
            Process_Line(line);
      end loop

end loop Read_Data;
```


Run-time Error Handling

- 3 types:
  - domain/data errors (sync)
    - Integer overflow
  - Resource exhaustion (sync+async)
    - Out of memory/disk
  - Loss of facilities (async)
    - Network hangup
- Alternatively:
  - Reproducable / intermittent
  - Internal / externally caused
- Important: Allow program to fix error: signals, error-codes & exceptions

Loop-exits

- Sometimes an iteration needs to give up control to the next iteration before finishing

```
while (! end_of_file(f)) {
        line = read_next_line(f);
        if (line_is_empty(line))
                continue;
        process_line(line);
}
```

<not in Ada>

Signals

- A procedure is called (possibly asynchronously) upon error
  - Signal statement associates an error-condition to a error-handling procedure

C:

```
signal(SIGFPE, close_done_app);
```

Ada:

```
Attach_Handler(Close_DownApp, Control_C_Hit);
```

Problem: the signal-handler procedure has no access to the local variables of the currently active procedure/function (which may have caused the error).

Exceptions

- A set of statements associated with a set of error-conditions and handler statements

<C doesn't have exceptions>

<Ada>

```
begin
        Put(1/X);
exception
        when Constraint_Error =>
                Put("Division by zero");
end
```

Note: Ada 'exception' blocks can be placed before each 'end' AND can access all variables in the 'begin-end' block.

Exceptions: Problems

- Don't know which statement caused the exception:
  - Put(1/X);
  - Put(1/Y);
  - Put(1/X);
- Exceptions are coarsely grouped
  - Ada: only 4 groups
    - Constraint_Error, Program_Error, Storage_Error, Tasking_Error

Exceptions: more information ?

- Ada has Exception_Occurrence and 'exception'
  - Exception_Occurrence contains exception data
  - Exception is the type of exception that occurred

```
begin
      put(1/X)
exception
      -- EO is of type Exception_Occurance
      when EO: Constraint_Error =>
                  Put("division by zero");
                  Put(Exception_Message(EO));
end
```


Exceptions ?

- Exception specification should answer:
  - Can programmer define extra exceptions ? Ada:Yes, Java: Yes, C: no, signals are fixed
  - How are exceptions represented ?
    - Ada: declared names, C: signal-number, Java: objects
  - Can additional information be retrieved?
    - Ada: yes, C: no, signal occurrence only, Java: in object
  - What happens if exception is ignored?
    - Ada: exception is propagated to dynamically enclosed exception statement (if none found, program aborts), C: program aborts
  - Is there a 'catch-all' exception type ? Ada:yes, C:no
  - Can exceptions be used as normal variables?
    - Ada: no (only Exception_Occurance variable), C:yes, signal-number is integer
  - Can programmer explicitly cause exceptions?
    - Ada: 'raise Contraint_Error', C: kill(SIGFPE)
- NOTE: error-handling is still VERY MUCH an open research area !

Next week: more imperative programming + example languages