Lectures 5-6

Imperative languages

Iztok Savnik, FAMNIT

March, 2024.

Literature

- Textbooks:
  - John Mitchell, Concepts in Programming Languages, Cambridge Univ Press, 2003 (Chapters 5 and 8)
  - Michael L. Scott, Programming Language Pragmatics (3rd ed.), Elsevier, 2009 (Chapters 6 and 8)
- Many examples are from:
  - Emmanuel Chailloux, Pascal Manoury, Bruno Pagano, Developing Applications With Objective Caml, English translation, O'REILLY, 2000

Outline

- Introduction
- Memory and variables
- Sequences, conditional statements and blocks
- Loops
- Procedures and functions
- Records
- Pointers
- Arrays
- Sets, unions, dictionaries

Functional vs. imperative approach

- Functional programming
  - Abstract computation model is λ-calculus
  - Program is a function represented by a λ-term,
    - The outcome is obtained by its reduction
  - Functions do not have side effects
    - Variables outside the function can not be altered
- Imperative programming
  - Abstract computation model is a Turing machine
  - Program is a sequence of instructions
    - Program has states; results are computed by executing instructions
  - Instructions change the state of main memory
    - The outcome is obtained when the final state is reached

Imperative programming

- Early imperative programming languages
  - Fortran, 1954; mathematical formulas
    - Still popular programming language
  - BASIC, 1960; Beginners' All-purpose Symbolic Instruction Code
  - Pascal, 1970; Algorithms + Data Structures = Programs
  - C, 1972; Constructs map efficiently to typical machine instructions
    - January 2021, C was ranked first in the TIOBE index
    - On top of lists: on demand, job offers, Web search results
  - Fortran is still among the most popular languages for numeric processing

Example

Let us compute the greatest common divisor of two integers

C

```c
int gcd(int  x, int y) {
   while (y != 0) {
        int t = x % y;
        x = y; y = t;
    }
    return x;
}
```

OCaml

```ocaml
let rec gcd x y =
    if y = 0 then x
    else gcd y (x mod y);;
```

<u>Functional</u>: values, recursion

<u>Imperative</u>: variables, loops, sequences

Structured control

- Structured programming, 1970
  - Emerged in late 1950s with ALGOL 58, ALGOL 60
  - Coined by Edsger W. Dijkstra
    - Paper “Go To Statement Considered Harmful", open letter, 1968
  - Structured and unstructured control flow
    - Unstructured: GOTO statements
    - Structured: syntactical constructs direct computation
  - »Revolution« in software engineering
    - Top-down design (i.e., progressive refinement)
    - Modularization of code
    - Structured types (records, sets, pointers multidim. arrays)
    - Descriptive variable and constant names
    - Extensive commenting conventions

Structured control

- Strong influence on imperative programming languages
  - Most structured programming ideas were implemented in imperative languages
  - Pascal, C, Modula, Ada, Oberon, Java, C#, …
  - But also in ML, Scala, F#, ...
- Most of modern algorithms can be elegantly expressed in imperative languages
  - All classical algorithms implemented in imperative languages (Dijkstra, Floyd, Knuth, ...)

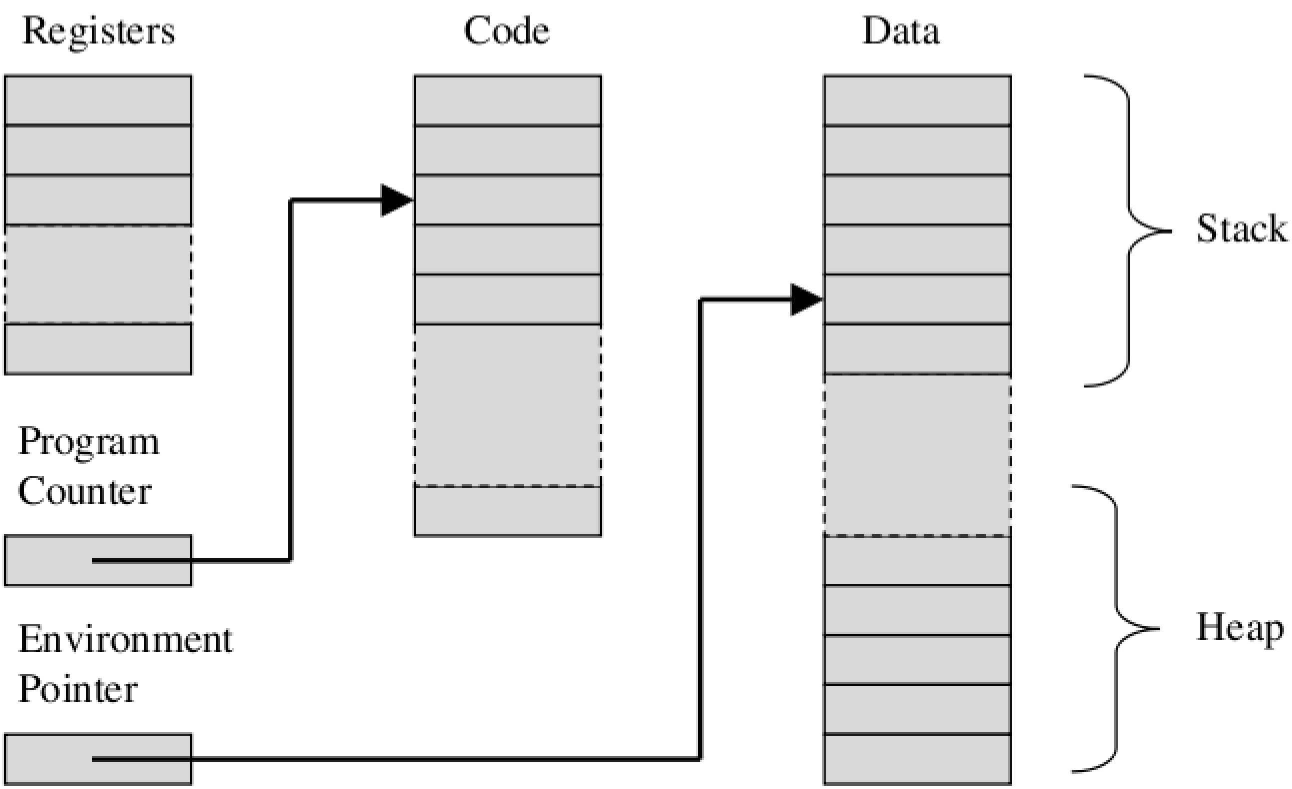
Concepts of imperative languages

- Read-write memory, variables
- Instructions and sequences of instructions
- Blocks
- Conditional statements
- Loops - conditional loops, iterations through ranges or through containers
- Procedures and functions
- Records and arrays
- Sets, unions and dictionaries

Outline

- Introduction
- <u>Memory and variables</u>
- Sequences, conditional statements and blocks
- Loops
- Procedures and functions
- Records
- Pointers
- Arrays
- Sets, unions, dictionaries

Program memory



Variables

- Memory of a program is organised into <u>memory cells</u>
- Any cell can be accessed via its <u>address</u>; an integer, usually in range 0 – ($2^{62} - 1$)
- <u>Variable</u> is a symbolic name for a memory space of given size
  - Variable can be accessed by using the name instead of the address of the memory cell
- <u>Namespaces</u> in a program
  - A program context (block, function, module)
  - A namespace includes information about variables
    - Name (identifier), the address of the beginning of memory space, the size of allocated memory
  - Namespaces are often organized hierarchically

Operations with variables

- Program must <u>allocate</u> the memory space before the variable is used
  - The allocation can be either static or dynamic
- Program <u>reads</u> the contents of the memory in the moment we refer to the identifier (name)
- The contents of the memory referred by a variable is <u>changed</u> by assignment
- The variable is <u>freed</u> either on the end of execution or on demand
- Possible problems:
  - Read/write unallocated memory, concurrent write, memory leak
  - We will study these problems in lecture on memory management

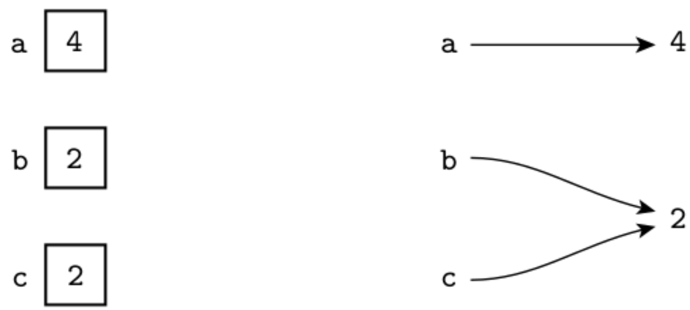
Models of variables

```
d = a;
a = b + c;
```

- Two models of variabes
  - Value model and reference model
- <u>Value model</u> of variables
  - Variable is a named container for a value
  - Location and value (see variable a)
    - l-value = refers to the location of a variable (left-hand side of assignment statements)
    - r-value = refers to the value of a variable (expressions that denote values)
    - both l-values and r-values can be complicated expressions
  - An expression can be either an l-value or an r-value, depending on the context
  - C, C++, Pascal, Ada, Java (simple values), etc.

Models of variables

- Reference model of variables




```
b := 2;
c := b;
a := b + c;
```

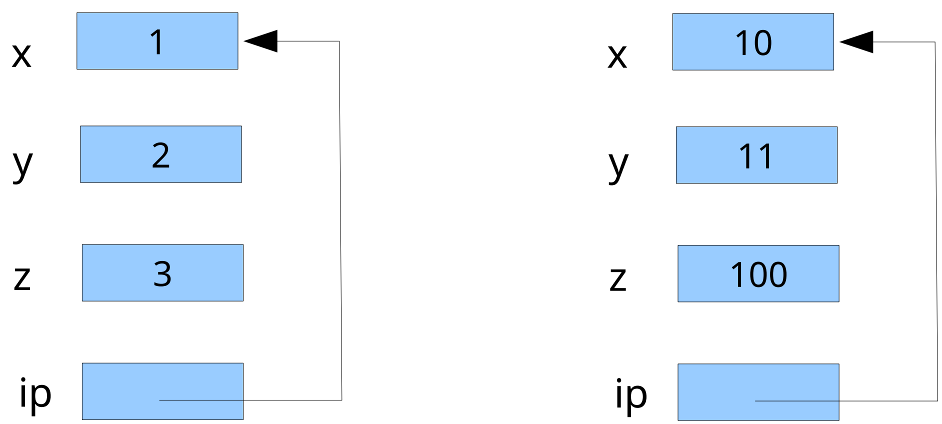
  - A variable is a named reference to a value
    - every variable is an l-value
    - variable in a context of an r-value must be dereferenced
    - dereference is automatic in most PL but not in ML
  - Reference model is (not) more expensive
    - Use multiple copies of immutable objects
  - Algol68, Clu, Lisp/Scheme, Python, ML, Haskell, and Smalltalk

Variables in C

- Two important operators
  - Operator »&«: returns address of variable
  - Operator »*«: returns value of variable (from an address)

```
int x = 1, y = 2, z = 3;
int *ip;
ip = &x;
y = *ip;
*ip = 0;
*ip = *ip + 10;
y = *ip + 1;
z = *ip * 10;
```

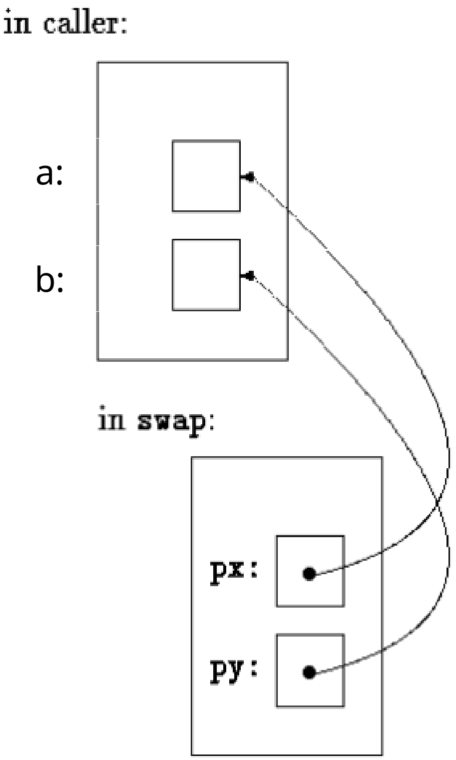
Value model



Two important operators

– Operator »&«: returns address of variable
– Operator »*«: returns value of a variable

```
swap(&a, &b);
...
void swap(int *px, int *py) {
   int temp;
   /* interchange *px and *py */
   temp = *px;
   *px = *py;
   *py = temp;
}
```

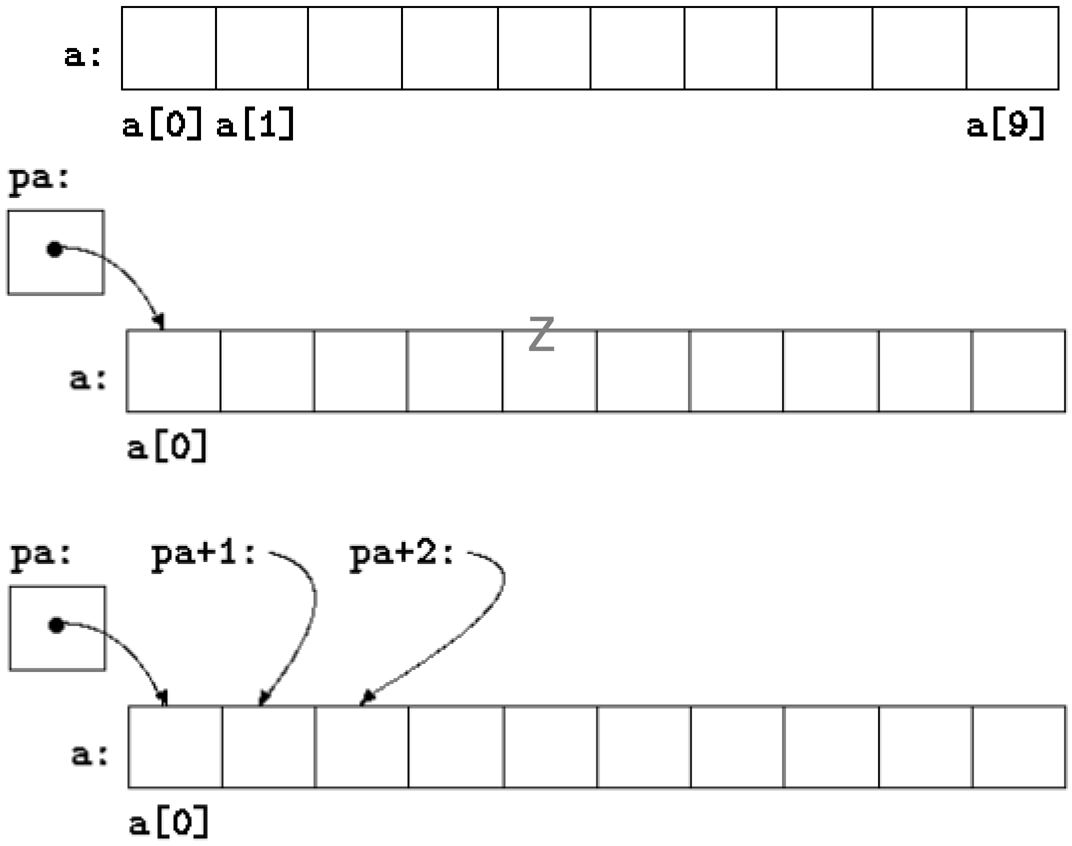


Pointers and arrays in C

- C has <u>pointer arithmetic</u>

```
int a[10];
// a[0], a[1],..., a[9]
pa = &a[0];
```

```
/* strlen: return length of string s */
int strlen(char *s) {
  int n;
  for (n = 0; *s != '\0', s++)
     n++;
  return n;
}
```



Variables in OCaml

```
type 'a ref = {mutable contents: 'a}
```

- Variables are implemented by using a <u>reference type</u>
- OCaml has weaker, but safer model of a variable
  - Reference is <u>initialised</u> on creation by the referenced value
  - Memory space is <u>automatically allocated</u> using the type of referenced value
  - <u>Assignment</u> is a special function with resulting type unit
  - The result of <u>reading</u> has the type of referenced variable
- Drawbacks of the model
  - Functions cannot be referenced
  - We do not have full access to the program's memory – no pointers, no pointer arithmetic

Examples of variables in Ocaml

```
# let x = ref 2 ;;                                    (* declaration and allocation
val x : int ref = {contents=2}
# !x;;                                                (* reading, notice operator '!'
- : int = 2
# x ;;                                                (* reading of the reference
- : int ref = {contents=2}
# x := 5; !x;;                                        (* assignment
- : int = 5
# x := !x * !x; !x;;                                  (* reading, operation, and assignment
- : int = 25
# let l = ref [1;2;3];;
val l : int list ref = {contents = [1; 2; 3]}
# !l;;
- : int list = [1; 2; 3]
# l := 0::!l; !l;;
- : int list = [0; 1; 2; 3]
```

Outline

- Introduction
- Memory and variables
- <u>Sequences, conditional statements and blocks</u>
- Loops
- Procedures and functions
- Records
- Pointers
- Arrays
- Sets, unions, dictionaries

Sequence

- Sequence is foundamental abstraction used to describe algorithms
  - Von Neumann's instruction cycle

```
LOOP: execute *PC;      // execute instruction referenced by PC
      PC++;             // increment programm counter (PC) by 1
      goto LOOP;        // loop
```

- Sequence of instructions change the state of variables (memory)

```
int t = x % y;
x = y;
y = t;
```

Sequences in OCaml

- Syntax of OCaml sequences

```
let t = ref 0
and x = ref 42
and y = ref 28 in begin
    t:= !x mod !y;
    x:=!y;
    y:=!x ;
end;;
```

```
<expr1>; <expr2>; <expr3>  (* list of exprs *)
```

- Every expression in a sequence must be of type `unit`.

```
# print_int 1; 2; 3;;
Warning 10: this expression should have type unit.
1- : int = 3
```

```
# print_int 2; ignore 4; 6;;
2- : int = 6
```

Blocks

- Imperative languages are typicaly block-structured languages
- Block is a sequence of statements enclosed by some structural language form
  - Begin-end blocks, loop blocks, function body, etc.

```
begin
   let t = ref 0
   and x = ref 42
   and y = ref 28 in
   begin
         t:= !x mod !y;
         x:=!y;
         y:=!x;
    end;
end;;
```

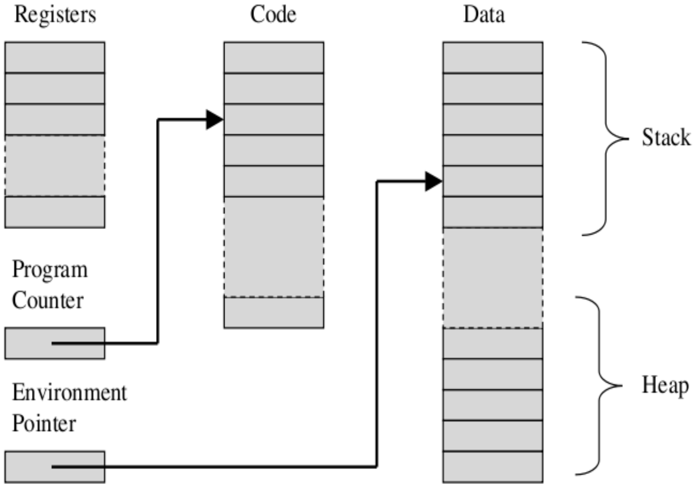
outer block

```
{int x = 2;
     { int y = 3;
        x = y+2;
     }
}
```

inner block

Blocks

- Each block is represented using <u>activation record</u>
  - Includes parameters and local variables
  - Includes memory location for return value
  - Includes control pointers to be detailed in next lectures
  - Control pointers are used to control computation
- Activation records are allocated on <u>program stack</u>
  - Presented in lecture on Memory management



Conditional statements

JE/JZ	Jump if equal/Jump if zero
JNE/JNZ	Jump if not equal/Jump if not zero
JA/JNBE	Jump if above/Jump if not below or equal
JAE/JNB	Jump if above or equal/Jump if not below
JB/JNAE	Jump if below/Jump if not above or equal
JBE/JNA	Jump if below or equal/Jump if not above
JG/JNLE	Jump if greater/Jump if not less or equal
JGE/JNL	Jump if greater or equal/Jump if not less
JL/JNGE	Jump if less/Jump if not greater or equal
JLE/JNG	Jump if less or equal/Jump if not greater
JC	Jump if carry
JNC	Jump if not carry

- Machine languages use instructions for <u>conditional jumps</u>
  - Initial imperative approach
- OCaml syntax

```
if <cond_expr> then <expr_true> else <expr_false> ;;
```

- Conditional statements is a concept shared between imperative and functional languages
  - Both branches must agree on type in Ocaml
  - Conditional statement in Ocaml has value

```
# (if 1 = 0 then 1 else 2) + 10;;
- : int = 12
```

Outline

- Introduction
- Memory and variables
- Sequences, conditional statements and blocks
- <u>Loops</u>
- Procedures and functions
- Records
- Pointers
- Arrays
- Sets, unions, dictionaries

Loops – while-do

- Repeat a block of commands while (or until) a condition is satisfied
  - Loop body changes the state of program
- Statement `while` in OCaml

```
while <cond_expr> do
   <sequence>
done
```

```
# let gcd (x,y) =
      let t = ref 0 in
      while !y != 0 do
         t := !x mod !y; x := !y; y := !t
      done; !x;;
val gcd : int ref * int ref -> int = <fun>
# let a = ref 42 and b = ref 28;;
val a : int ref = {contents = 42}
val b : int ref = {contents = 28}
# gcd (a,b);;
- : int = 14
# (!a,!b);;     (* passing references! *)
- : int * int = (14, 0)
```

Loops – `for` statement

- Statement `for` is classical construct of imperative programming languages
- Statement `for` in OCaml

```
for <sym> = <exp1> to <exp2> do
    <exp3>
done
for <sym> = <exp1> downto <exp2> do
    <exp3>
done
```

```
/* atoi: convert s to integer */
int atoi(char s[])
{
    int i, n, sign;
    for (i = 0; isspace(s[i]); i++) ;       /* skip white space */
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')  i++;    /* skip sign */
    for (n = 0; isdigit(s[i]); i++)          /* convert s to integer */
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

- Used in
  - Impertive
  - Script
  - OO
  - Moduar

  programming languages

Ocaml: for->while statement

```
let is_digit = function '0' .. '9' -> true | _ -> false;;
let is_white = function ' ' | '\n' | '\t' -> true | _ -> false;;
```

```
let int_of_string s =
   begin
      let i = ref 0 and n = ref 0 in
      while is_white(s.[!i]) do i := !i+1; done;
      let sign = (if s.[!i]='-' then -1 else 1) in
      if s.[!i]='+' || s.[!i]='-' then i := !i+1;
      while is_digit(s.[!i]) do
         n := 10 * !n + (int_of_char(s.[!i]) - int_of_char('0'));
         i := !i+1;
      done;
      sign * !n;
   end;;
```

```
# let s = " -12\n";;
val s : string = " -12\n"
# int_of_string s;;
- : int = -12
```

Loops – do-while statement

- Loop condition is at the end of loop block
  - do-while syntax in C prog. lang. (also in Java)
  - Not included in Ocaml!

```
do <sequence>
while <cond_expr>
```

  - <u>repeat-until</u> in Pascal
- Example:
  - Kernighan & Ritchie: The C programming language

```c
/* itoa: convert n to characters in s */
void itoa(int n, char s[]) {
  int i, sign;
  if ((sign = n) < 0) /* record sign */
    n = -n;            /* make n positive */
  i = 0;
  do {     /* generate digits in reverse order */
     s[i++] = n % 10 + '0';     /* get next digit */
  } while ((n /= 10) > 0);       /* delete it */
  if (sign < 0)
     s[i++] = '-';
  s[i] = '\0';
  reverse(s);
}
```

Ocaml: do-while->while statement

```
let string_of_int n =
  begin
     let s = Bytes.make 10 ' ' and sign = n and nr = ref n and i = ref 0 in
     if sign<0 then nr := -n;
     Bytes.set s !i (char_of_int(!nr mod 10 + int_of_char('0')));
     nr := !nr / 10;
     while (!nr > 0) do
        i := !i+1;
        Bytes.set s !i (char_of_int(!nr mod 10 + int_of_char('0')));
        nr := !nr / 10;
     done;
     if (sign < 0) then begin i := !i+1; Bytes.set s !i '-'; end;
     reverse(Bytes.to_string (Bytes.trim s));
  end;;
```

Loop control

- Loop control in C programming language
  - Jumping out of a loop – break
  - Jumping to a loop condition – continue
  - Not included in Ocaml!

```
for (i = 0; i < n; i++)
    if (a[i] < 0)
        /* skip negative elements */
       continue;
    ...
        /* do positive elements */
```

```
/* trim: remove trailing blanks, tabs, newlines */
int trim(char s[]) {
    int n;
    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

  - Kernighan & Ritchie: The C programming language

Outline

- Introduction
- Memory and variables
- Sequences, conditional statements and blocks
- Loops
- <u>Procedures and functions</u>
- Records
- Pointers
- Arrays
- Sets, unions, dictionaries

Procedures and functions

- Abstraction is a process by which the programmer can associate a symbol or a pattern with a programming language construct.
  - Control and data abstractions
- Subroutines are the principal mechanism for control abstraction.
  - Part of program with well defined input and output is abstracted as subrutine, procedure, or function.
  - Subrutine performs operation on behalf of caller.
  - Caller passes arguments to subrutine by using parameters
  - Subrutine that returns values is a function.
  - Subrutine that does not is called a procedure.

Procedures and functions

- Most subroutines have parameters
- Procedure was first abstraction in Algol-family of programming languages
  - Formal and actual parameters of procedure

```
procedure Proc(First : Integer; Second: Character);
Proc(24,'h');
```

  - Actual parameters are mapped to formal parameters
- The most common parameter-passing modes
  - Some languages define a single set of rules that apply to all parameters (C, Java, Fortran, ML, and Lisp)
  - Others have more modes of parameter passing (Pascal, C++, Ada, ...)

Parameter passing

- Input and output of procedure is realized by means of parameter passing
  - Passing values
    - C (only cbv), Java, Ocaml, C++, Pascal, ...
  - Passing references
    - Pascal, C++, Fortran, ...

```
Procedure Square(Index : Integer;
                     Var Result : Integer);
  Begin
    Result := Index * Index;
  End
```

- Other parameter passing issues
  - Passing structured things
    - arrays, structures, objects
  - Missing and default parameters
  - Named parameters
  - Variable-length argument lists

Passing values

- The most commonly used method
  - Values of actual parameters are copied to formal parameters
  - Java uses only this method (arrays and structures are identified by references)
- Parameter is seen as <u>local variable</u> of procedure
  - It is initialized by the value of actual parameter

Java

```java
int plus(int a, int b) {
  a += b;
  return a;
}
int f() {
  int x = 3; int y = 4;
  return plus(x, y);
}
```

Ocaml

```ocaml
# let plus ((a:int), (b:int)) : int = a + b;;
val plus : int * int -> int = <fun>
# let f () =
    let x = 3 and y = 4
    in plus (x,y);;
val f : unit -> int = <fun>
# f ();;
- : int = 7
```

Passing references

- Reference to variable is passed to procedure
  - Code of procedure is changing passed variable
  - All changes are retained after the call
  - Passed variable and formal parameter are aliases
- Best method for larger structures!

C

```c
void plus(int a, int *b) {
   *b += a;
}
...
int x = 3;
plus(4, &x);
// x == 7
...
```

Ocaml

```ocaml
# let plus ((a:int), (b:int ref)) : unit =
    b := a + !b;;
val plus : int * int ref -> unit = <fun>
# let a = 4 and b = ref 3;;
val a : int = 4
val b : int ref = {contents = 3}
# plus (a,b);;
- : unit = ()
# !b;;
- : int = 7
```

Variations on value and reference parameters

- C++ references
  - In C++, C references are made explicit
  - C++ implements call-by-reference

```
void swap(int &a, int &b) { int t = a; a = b; b = t; }
```

- Call-by-sharing
  - Barbara Liskov, CLU (also Smalltalk)
  - Objects (identifiers) are references
  - No need to pass reference (to references)
  - Just pass reference

Variations on value and reference parameters

- Call-by-value/Result
  - Actual params are copied to formal params initially
  - Result is copied back to actual parameter before exit
- Read-only parameters
  - Modula-3 provided read-only params
  - Parameter values can not be changed
  - Read-only params are available also in C (const)
- Parameters in Ada,
  - Modes: in, out, in out (also in PL/SQL)
  - Named parameters / position parameters

Variations on value and reference parameters

- Default values of parameters
  - Ada, Oracle PL/SQL
- Variable length argument lists
  - Programming language C, Perl, ...
  - No type-checking, no control, may be dangerous

Outline

- Introduction
- Memory and variables
- Sequences, conditional statements and blocks
- Loops
- Procedures and functions
- <u>Records</u>
- Pointers
- Arrays
- Sets, unions, dictionaries

Type declaration

- Type is defined from simpler types
  - By using type constructors
  - *, |, record, list, array, ...
- Type definition in Ocaml
- No parametrized (polymorphic) types in imperative languages!
  - Just concrete
- Records, Pointers and Arrays !
  - Types of imperative programming languages
  - Presented in this section

type name = typedef ;;

type $name_1$ = $typedef_1$

and $name_2$ = $typedef_2$

...

and $name_n$ = $typedef_n$;;

Records

- Record types allow related data of heterogeneous types to be stored and manipulated together.
- Records in programming languages
  - Originally introduced by Cobol
  - In Algol 68 called them structures (also in C)
    - They use the keword `struct`
    - Later in Fortran 90 they named them *record type*
  - In C++ structures are special form of a class
  - Java has no notion of a structure
  - C# and Swift use reference model for classes and value model for the type `struct` (no inheritance)

Records

- In C, a simple record might be defined as follows:
- Each of the record components is known as a field.
- To refer to a given field of a record, most languages use “dot” notation:

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
};
```

```
element copper;
const double AN = 6.022e23;     /* Avogadro's number */
...
copper.name[0] = 'C'; copper.name[1] = 'u';
double atoms = mass / copper.atomic_weight * AN;
```

Records in Ocaml

- Record is a product with named components
- Record type definition and record constr. in Ocaml

```
type name = { name₁ : t₁ ; . . . ; nameₙ : tₙ }
{ name₁ = expr₁ ; . . . ; nameₙ = exprₙ }
```

- Record components can be defined mutable
  - Component assignment operation

```
type name = { ...; mutable nameᵢ: tᵢ ; ... }
```

```
expr1.name <- expr2
```

```
# type complex = { mutable re:float;
                   mutable im:float } ;;
type complex = { mutable re : float;
                 mutable im : float; }
# let c = {re=2.;im=0.} ;;
val c : complex = {re=2; im=0}
```

```
# c.im <- 3.;;
- : unit = ()
# c;;
- : complex = {re = 2.; im = 3.}
# c = {im=3.;re=3.} ;;
- : bool = true
```

Records in Ocaml

- Operations:
  - Accessing components
  - Pattern matching

```
expr.name
```

$\{ \text{name}_i = p_i \ ; \ . \ . \ . \ ; \ \text{name}_j = p_j \ \}$

```
# let add_complex c1 c2 = {re=c1.re+.c2.re; im=c1.im+.c2.im};;
val add_complex : complex -> complex -> complex = <fun>
# add_complex c c ;;
- : complex = {re=4; im=6}
# let mult_complex c1 c2 = match (c1,c2) with
({re=x1;im=y1},{re=x2;im=y2}) -> {re=x1*.x2-.y1*.y2; im=x1*.y2+.x2*.y1} ;;
val mult_complex : complex -> complex -> complex = <fun>
# mult_complex c c ;;
- : complex = {re=-5; im=12}
```

Example in Ocaml

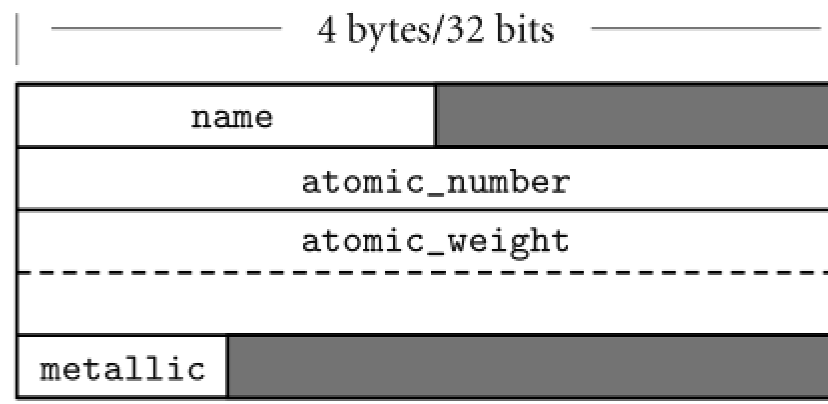
```
# type point = { mutable xc : float; mutable yc : float } ;;
type point = { mutable xc: float; mutable yc: float }
# let p = { xc = 1.0; yc = 0.0 } ;;
val p : point = {xc=1; yc=0}
# p.xc <- 3.0 ;;
- : unit = ()
```

```
# let moveto p dx dy =
    begin
      p.xc <- p.xc +. dx;
      p.yc <- p.yc +. dy;
    end;;
val moveto : point -> float -> float -> unit = <fun>
# moveto p 1.1 2.2 ;;
- : unit = ()
# p ;;
- : point = {xc=4.1; yc=2.2}
```

Memory layout for records

- The fields of a record are usually stored in adjacent locations in memory.
- Compiler keeps track of the offset of each field within each rec. type.
- Value model (of var.)
  - Nested records are embedded in parent record
- Reference model
  - Fields are references to in another location.

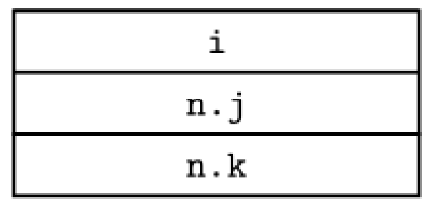
```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
};
```



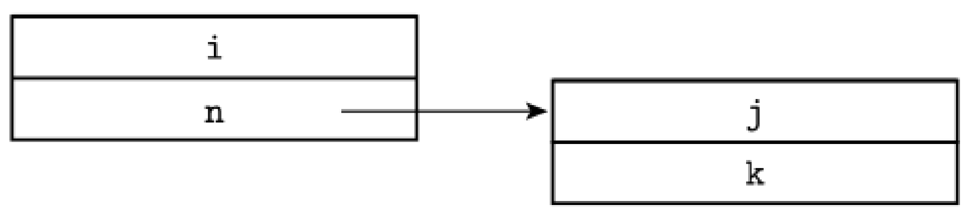
Memory layout for records

- Layout of memory for a nested struct (class) in C (top) and Java (bottom).

```
struct T {
    int j;
    int k;
};
struct S {
    int i;
    struct T n;
};
```




```
class T {
    public int j;
    public int k;
}
class S {
    public int i;
    public T n;
}
```



Outline

- Introduction
- Memory and variables
- Sequences, conditional statements and blocks
- Loops
- Procedures and functions
- Records
- Pointers
- Arrays
- Sets, unions, dictionaries

Pointers

- A pointer is a reference to an object in memory
  - There were attempts to call it reference
  - Pointer is usually represented by a memory address
  - A pointer can be typed (ML,C++,Java, …)
    - PL then knows the structure and size of referenced object
    - The access to the object can now be checked by a compiler
  - A pointer can be untyped (Lisp,C)
    - Programmer must know the object pointed to by a pointer
    - Compiler does not know the structure of object
    - Therefore, it can not check the access to the object

Pointers and recursive structures

- A recursive data structure includes at least one reference to an object of the same type
  - A recursive structure can be implemented by using a structure that includes components
    - Records, products, lists, arrays and unions.
- Languages using reference model
  - Components include references to other objects
  - No need to define pointers; they are defined implicitly
- Languages using value model
  - Components must include pointers to objects and not object values

Pointers

- Pointed location
  - In some languages pointers are restricted to refer to objects on heap (Java, Pascal, Ada, Modula)
    - Object is created with operation new() that returns pointer
    - This is the only way you can create a pointer
  - Other languages use pointers that can point to any location (C, C++)
    - This languages use operator address-of ‘&’
- Disposing allocated objects
  - Some languages use explicit operation for releasing the allocate memory space (C,C++,Pascal)
    - Possible errors: memory leak, accessing disposed object
  - Others use automatic memory management (Java,C#)

Pointers

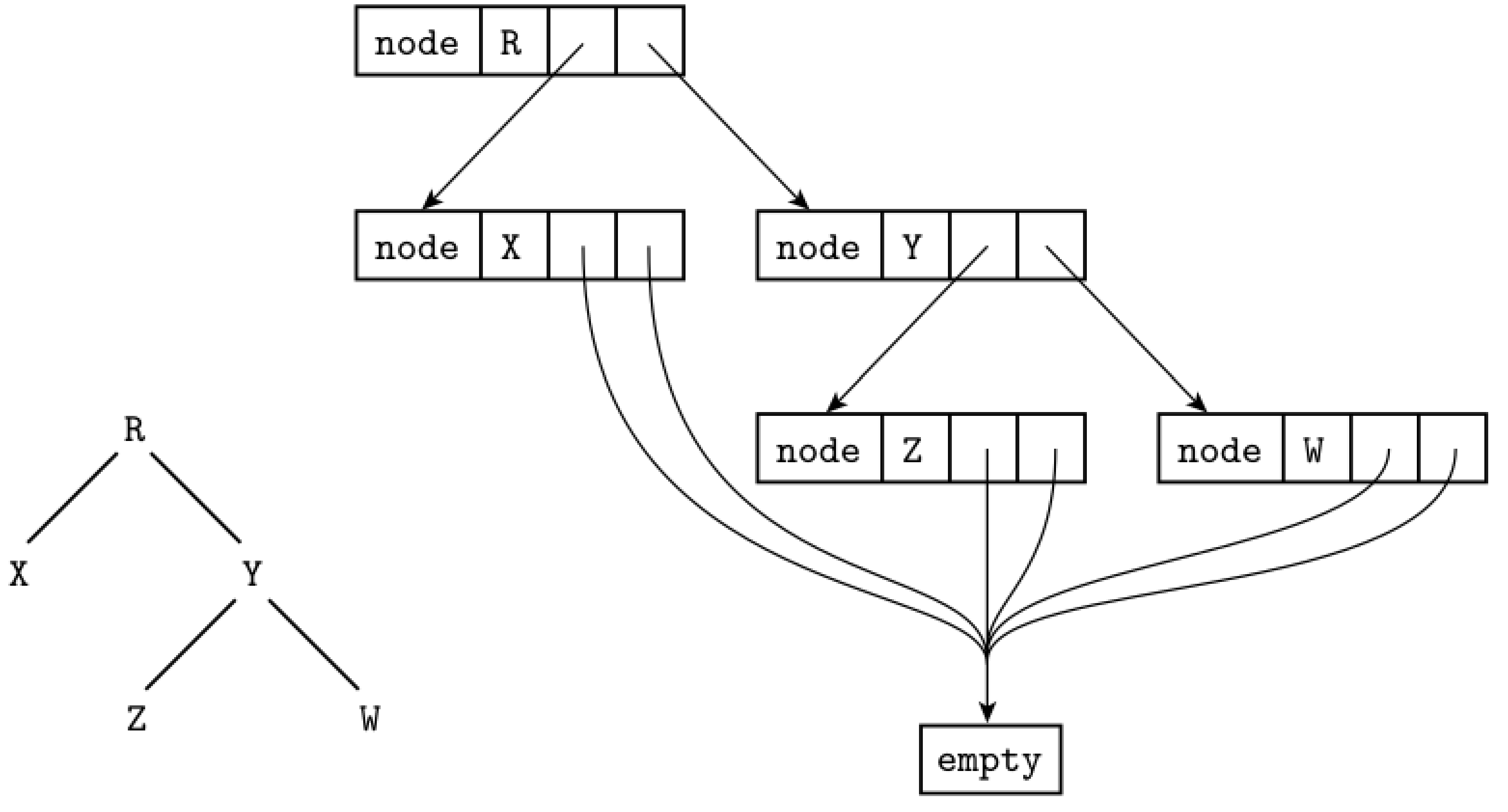
- Operations on pointers depend on the model of variables
  - Allocation and deallocation of objects on the heap
    - Reference model usually implies automatic memory management
    - Value model often implies manual storage allocation/deallocation
  - Dereferencing a pointer to access an objects to which it points
    - Need to dereference in the case of reference model
    - No need to dereference in the case of value model
  - Assignment of one pointer to another
    - In the case of a reference model, pointers are copied as references
    - In case of value model, an assignment copies the value, so the pointers have to be used

Reference model

- Recursive data structures include the pointers to structures of the same kind
- An example of a recursive data structure in Ocaml
  - ML uses references to idenfify tuples, lists, records, arrays, ...

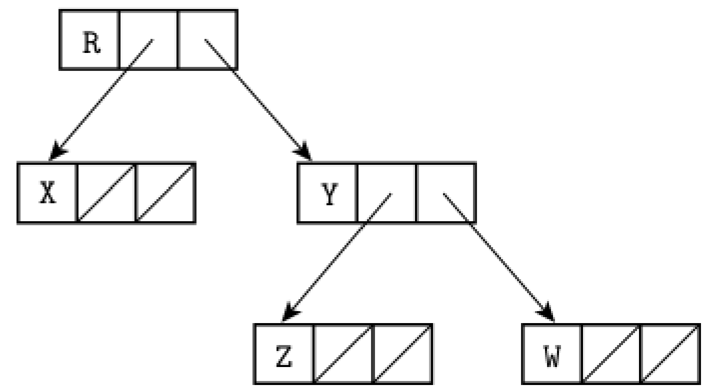
```
# type ctree = Empty | Node of char * ctree * ctree;;
type ctree = Empty | Node of char * ctree * ctree
# Node('R',Node('X',Empty,Empty),
           Node('Y',Node('Z',Empty,Empty),
                    Node('W',Empty,Empty)))
```

Recursive types in OCaml



Value model

- Recursive data structures in languages with explicit pointers
  - Imperative languages with the value model of variables
  - C, C++, ML, ...
- Example in Pascal and C




```
type chr_tree_ptr = ^chr_tree;
     chr_tree = record
        left, right : chr_tree_ptr;
        val : char
     end;

new(my_ptr);
```

```
struct chr_tree {
    struct chr_tree *left, *right;
    char val;
};

my_ptr = malloc(sizeof(struct chr_tree));
```

Example: Implementation of lists

```
# type 'a rnode = { mutable cont: 'a; mutable next: 'a rlist }
    and  'a rlist = Nil | Elm of 'a rnode;;
type 'a rnode = { mutable cont : 'a; mutable next : 'a rlist; }
and 'a rlist = Nil | Elm of 'a rnode
```

```
# let l1 = Elm {cont = 1; next = Elm {cont = 2; next = Nil}};;
val l1 : int rlist =  Elm {cont = 1; next = Elm {cont = 2; next = Nil}}
# let cons v l = Elm {cont=v; next=l};;
val cons : 'a -> 'a rlist -> 'a rlist = <fun>
```

```
# let ( ** ) v l = cons v l;;
val ( ** ) : 'a -> 'a rlist -> 'a rlist = <fun>
# let l2 = cons 3 (cons 4 Nil));;
val l2 : int rlist =  Elm {cont = 3; next = Elm {cont = 4; next = Nil}}
# let l3 = 5**6**Nil;;
val l3 : int rlist = Elm {cont = 5; next = Elm {cont = 6; next = Nil}}
```

Example: Implementation of lists

```
# exception EmptyList;;
exception EmptyList
# let head l = match l with Nil -> raise EmptyList | Elm r -> r.cont;;
val head : 'a rlist -> 'a = <fun>
# let tail l = match l with Nil -> raise EmptyList | Elm r -> r.next;;
val tail : 'a rlist -> 'a rlist = <fun>
# head l1;;
- : int = 1
# tail l1;;
- : int rlist = Elm {cont = 2; next = Nil}
```

```
# let rec length l = match l with Nil -> 0 | Elm {next=t} -> 1+length t;;
val length : 'a rlist -> int = <fun>
# length l1;;
- : int = 2
```

Example: Implementation of lists

```
# let rec append l1 l2 = match l1,l2 with
  Elm r1,_ -> Elm {cont=r1.cont; next=append r1.next l2}
| Nil,Elm r2 -> Elm {cont=r2.cont; next=append Nil r2.next}
| Nil,Nil -> Nil;;
val append : 'a rlist -> 'a rlist -> 'a rlist = <fun>
```

```
# append l1 l2;;
- : int rlist =
Elm {cont=1; next=Elm {cont=2; next=Elm {cont=3; next=Elm {cont=4; next=Nil}}}}
# l1;;
- : int rlist = Elm {cont=1;  next=Elm {cont=2; next=Nil }}
# l2;;
- : int rlist = Elm {cont=3;next=Elm {cont=4;next=Nil }}
```

Example: Implementation of lists

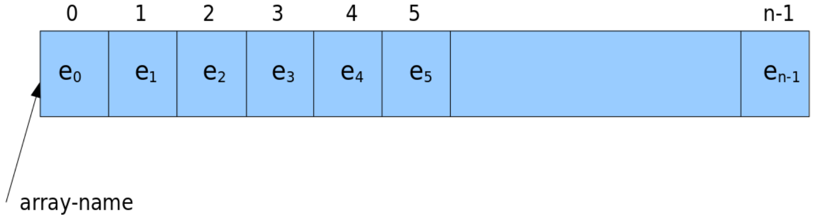
```
# let rec append1 l1 l2 = match l1 with
  Nil -> l2
| Elm r when r.next=Nil -> r.next <- l2; l1
| Elm r -> ignore (append1 r.next l2); l1;;
val append1 : 'a rlist -> 'a rlist -> 'a rlist = <fun>
```

```
# append1 l1 l2;;
- : int rlist =
Elm {cont=1;  next=Elm {cont=2; next=Elm {cont=3; next=Elm {cont=4; next=Nil}}}}
# l1;;
- : int rlist =
Elm {cont=1;  next=Elm {cont=2; next=Elm {cont=3; next=Elm {cont=4; next=Nil}}}}
# l2;;
- : int rlist = Elm {cont=3;next=Elm {cont=4;next=Nil}}
```

Outline

- Introduction
- Memory and variables
- Sequences, conditional statements and blocks
- Loops
- Procedures and functions
- Records
- Pointers
- Arrays
- Sets, unions, dictionaries

Arrays



- Arrays are data structures holding the finite number of elements of certain data type
- Semantically, array is a mapping from an index type to a component or element type.
  - Index is usually an integer but many PLs can use discrete type
- In imperative languages an array is an important data structure
  - C, C++, Java, Fortran, Pascal, ...
  - Similar role in imperative PL as lists have in functional PL.
- An array is by definition mutable, but its size is fixed

Syntax and operations

- Accessing elements of array
  - Most languages append index delimited by a variant of parentheses to the array name (a(), a[], a{}, ...)
  - Indexes of arrays are usually of integer type but can be also of discrete type
- Declaration of an array
  - Indexes in most languages are defined by range
  - Index in C starts with 0

```
char[] upper;          /* Java */
char upper[];          /* alternative declaration */
upper = new char[26];
char upper[26];     /* C */
character, dimension (1:26) :: upper  /* Fortran */
character (26) upper    /* shorthand notation */
var upper : array ['a'..'z'] of char;   /* Pascal */
```

Dimensions, Bounds, and Allocation

- In prev. examples, the shape of the array (including bounds) was specified in the declaration.
- For such static shape arrays, storage can be managed in the usual way
  - Static allocation for arrays whose lifetime is the entire program;
  - Stack allocation for arrays whose lifetime is an invocation of a subroutine;
  - Heap allocation for dynamically allocated arrays with more general lifetime.
- Storage management is more complex for arrays
  - Whose shape is not known until elaboration time, or
  - Whose shape may change during execution.

Dimensions, Bounds, and Allocation

- For dynamic arrays, compiler must
  - Allocate space and make shape info available at run time
  - Some PLs allow the number and bounds of dimensions to be dynamic, others allow just bounds to be dynamic
- Allocation of dynamic arrays
  - Local array may still be allocated in the stack.
    - Shape, is known at elaboration time
  - An array whose size may change is allocated in the heap.
- Descriptors, or dope vectors, hold shape information at run time
  - Offsets for record components, lower bound, the size and upper bound of each dimension
  - Dope vector may be stored in activation record on stack, or together with an array on heap

Memory Layout

- Arrays in most language implementations are stored in contiguous locations in memory.
  - One-dimensional array: one elem. after another
  - Multi-dimensional array: row-major, column-major
    - Important for nested loops to access all the elements of a large, multidi-mensional array.
    - Speed of such loops depends heavily effectiveness of caching
  - True multidimensional arrays use contiguous layout

```
for (i = 0; i < N; i++) {          /* rows */
    for (j = 0; j < N; j++) {      /* columns */
        ... A[i][j] ...
    }
}
```

- Row-Pointer Layout
  - Not stored contiguously, but in blocks including 1d arrays
  - Advantages: variable sized of rows, initialized from pieces
  - C, Ocaml, Java, C# (many provide both layouts)

Arrays in OCaml

```
# let v = [| 3.14; 6.28; 9.42 |] ;;
val v : float array = [|3.14; 6.28; 9.42|]
```

- Elements can be enumerated between [|...|]
- Arrays are integrated into Ocaml
  - (but not so profoundly as lists)
- Similarly to lists, there is a module Array that includes all necessary operations
- Create an array

```
# let v = Array.create 3 3.14;;
val v : float array = [|3.14; 3.14; 3.14|]
```

- Access/update an array element
  - Accessing an element
  - Setting new value

$expr_1$.($expr_2$)

$expr_1$.($expr_2$) <- $expr_3$

Arrays in OCaml

- Example:
- Array index must not go accross the borders

```
# v.(1) ;;
- : float = 3.14
# v.(0) <- 100.0 ;;
- : unit = ()
# v ;;
- : float array = [|100; 3.14; 3.14|]
```

```
# v.(-1) +. 4.0;;
Uncaught exception: Invalid_argument("Array.get")
```

- Checking that the index is not used outside borders is expensive
  - Some languages do not check this by default (C)

Functions on arrays

```
# let n = 10;
val n : int = 10
# let v = Array.create n 0;;
val v:int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

```
# for i=0 to (n-1) do v.(i)<-i done;;
- : unit = ()
# v;;
- : int array = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]
```

```
# let reverse v =
      let tmp=ref 0
      and n = Array.length(v)
      in for i=0 to (n/2-1) do
             tmp := v.(i);
             v.(i) <- v.(n-i-1);
             v.(n-i-1) <- (!tmp);
         done;;
- : unit = ()
# reverse(v);;
- : int array = [|9; 8; 7; 6; 5; 4; 3; 2; 1; 0|]
```

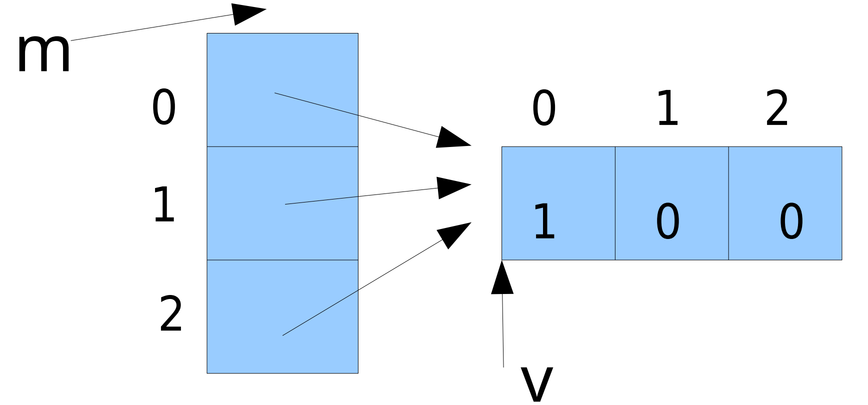
```
# let u = [|2;3|];;
val u : int array = [|2; 3|]
# let m = 2;;
val m : int = 2
# let subarray u v =
      let found = ref false
      and i = ref 0
      in while ((!i<=(n-m)) && not !found) do
             found := true;
             for j=0 to (m-1) do
                 if v.(!i+j) != u.(j) then
                    found := false
             done;
             i := !i+1
         done;
      !found;;
val subarray : 'a array -> 'a array -> bool = <fun>
# subarray u v;;
- : bool = true
```

Example: subarray()

```
# let prefix u v i =
    let found = ref true
    and m = Array.length(u)
    in for j=0 to (m-1) do
           if v.(i+j) != u.(j) then
              found := false
           done;
        !found;;
val prefix : ’a array -> ’a array -> int ->
                   bool = <fun>
# prefix u v 0;;
- : bool = false
# prefix u v 2;;
- : bool = true
```

```
# let subarray u v =
    let found = ref false
    and i = ref 0
    and m = Array.length(u)
    and n = Array.length(v)
    in while ((!i<=(n-m)) && not !found) do
            found := prefix u v !i;
            i  := !i+1
        done;
    !found;;
val subarray : 'a array -> 'a array ->
                   bool = <fun>
```

Matrix in Ocaml is array of arrays




```
# let v = Array.create 3 0;;
val v : int array = [|0; 0; 0|]
# let m = Array.create 3 v;;
val m : int array array =
      [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
```

```
# v.(0) <- 1;;
- : unit = ()
# m;;
- : int array array =
  [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
```

```
# let v2 = Array.copy v ;;
val v2 : int array = [|1; 0; 0|]
# let m2 = Array.copy m ;;
val m2 : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
# v.(1)<- 352;;
- : unit = ()
# v2;;
- : int array = [|1; 0; 0|]
# m2 ;;
```

Matrices in Ocaml

```
# let m = Array.create_matrix 4 4 0;;
val m : int array array = [|[|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]|]
# for i=0 to 3 do m.(i).(i) <- 1; done;;
- : unit = ()
# m;;
- : int array array = [|[|1; 0; 0; 0|]; [|0; 1; 0; 0|]; [|0; 0; 1; 0|]; [|0; 0; 0; 1|]|]
# m.(1);;
- : int array = [|0; 1; 0; 0|]
```

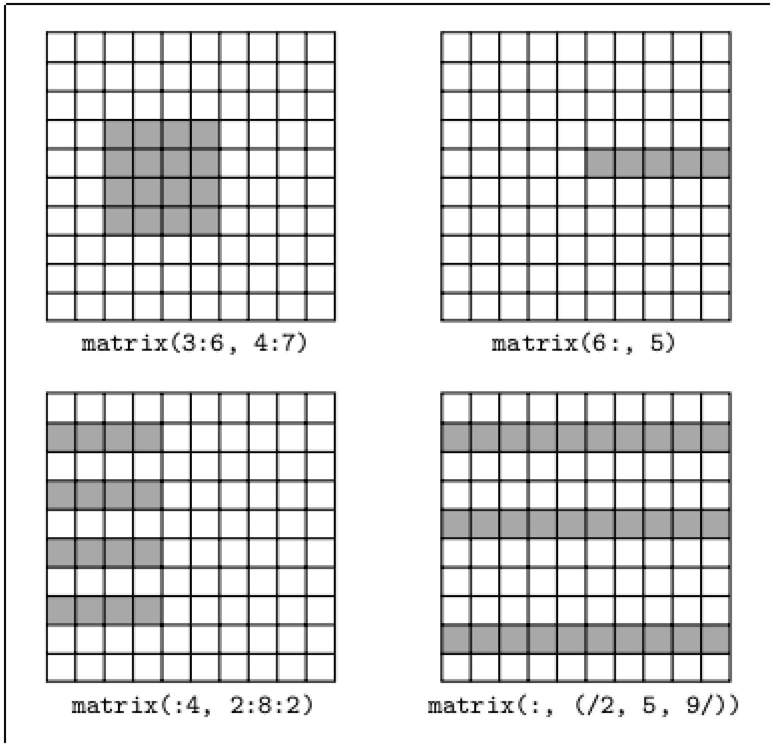
Operations on matrices

```
# let add_mat a b =
    let r = Array.create_matrix n m 0.0 in
      for i = 0 to (n-1) do
        for j = 0 to (m-1) do
          r.(i).(j) <- a.(i).(j) +. b.(i).(j)
        done
      done ; r;;
val add_mat : float array array -> float array array -> float array array = <fun>
# a.(0).(0) <- 1.0; a.(1).(1) <- 2.0; a.(2).(2) <- 3.0;;
- : unit = ()
# b.(0).(2) <- 1.0; b.(1).(1) <- 2.0; b.(2).(0) <- 3.0;;
- : unit = ()
# add_mat a b;;
- : float array array = [|[|1.; 0.; 1.|]; [|0.; 4.; 0.|]; [|3.; 0.; 3.|]|]
```

Matrices

- Multidimensional arrays
  - Declaration
  - Arrays of arrays
    - C, C++, ML, Java
  - Two-dimensional array
    - One block of memory
    - Ada, Fortran
- Slices
  - A slice is a rectangular portion of an array.
  - R, Fortran, Python

```
/* C */
double mat[10][10];
/* Ocaml */
type 'a matrix = array array 'a;;
/* Modula-3 */
VAR mat : ARRAY [1..10] OF ARRAY [1..10] OF REAL;
/* Ada */
mat1 : array (1..10, 1..10) of real;
```



Outline

- Introduction
- Memory and variables
- Sequences, conditional statements and blocks
- Loops
- Procedures and functions
- Records
- Pointers
- Arrays
- Sets, unions, dictionaries

Sets

- A set stores unique values, without any particular order
- Basic operations
  - Set ops: create, delete, add_element, delete_element
  - Boolean ops: membership, subset, equality, disjoint
  - Set algebra: union, difference, intersection
- Implementation
  - There are many different ways of implementing sets
    - Each with serious weaknesses for some purposes
    - For any specific purpose, it is not hard to implement set functionality using commonly available data structures

Sets

- Implementation
  - Lists, arrays (unefficient)
  - Bitstrings (storage efficient, converted to instructions)
  - Binary search trees (library: Ocaml, Haskell)
  - Hash tables
  - Dictionary representation of sets
- Sets in programing languages
  - Libraries: C++, Java, .NET, Ruby, Ocaml, Swift, Erlang
  - Build-in: Javascript, Python, Pascal

Sets in Phyton

- Examples:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)

thisset.add("orange")
thisset.remove("banana")
thisset.discard("banana")  # not ∃ -> no error
del thisset                # delete complete set

set2 = {1, 2, 3}
set3 = set1.union(set2)    # {3, 'b', 'a', 2, 1, 'c'}
```

Sets in Phyton

•add()	Adds an element to the set
•remove()	Removes the specified element
•discard()	Remove the specified item
•pop()	Removes an element from the set
•clear()	Removes all the elements from the set
•copy()	Returns a copy of the set
•union()	Return a set containing the union of sets
•intersection()	Returns a set, that is the intersection of two other sets
•difference()	Returns a set contains the difference betw two or more sets
•isdisjoint()	Returns whether two sets have a intersection or not
•Issubset()	Returns whether another set contains this set or not
•issuperset()	Returns whether this set contains another set or not

•

•... and more

Unions

- Type constructed by union
  - Make a new type by taking the union of existing types
- Unions in Ocaml
  - Type definition
  - Construction of instance
  - Pattern matching
- Union in other languages
  - Tagged union:
    - ML-family, Haskell
    - Pascal, Ada, Modula2
      - Also called: Variant records in Pascal
  - Untagged union: C, C++

```
type name = ...
      | Name_i ...
      | Name_j of t_j ...
      | Name_k of t_k * ...* t_l ... ;;
```

Unions in C

- Type that allows multiple different values to be stored in the same memory space
  - Size = the largest component

```
union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

```
union Data {
   int i;
   float f;
   char str[20];
} data;
```

```
int main( ) {
   union Data data;
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);
   return 0;
}
```

Variant records in Pascal

- Parts of records are variant
  - Type tag is used to differentiate among the variants
  - Records must include the largest variant

```
type paytype=(salaried, hourly);
var employee: record
            id: integer;
            dept: array [1...3] of char;
            age: integer;
            case payclass: paytype of
                salaried: (monthlyrate: real; stardate: integer);
                hourly: (hourrate: real; reghours: integer; overtime: integer);
      end;
```

Dictionaries

- Alternative names
  - Associative array, map, symbol table
- A store of key/value pairs
  - Keys and values are of arbitrary type
  - Operations provided
    - Create, access, update, delete, to-list, keys, ...
- Programming languages
  - Initial implementations
    - TMG (1965, compiler-compiler), SETL (late 1960s), Snobol (1969)
  - Script languages
    - AWK, Rexx, Perl, PHP, Tcl, JavaScript, Python, Ruby, Go, Lua

Dictionaries

- Other languages
  - C++, Java, Scala, Erlang, OCaml, Haskell

- Implementation of a dictionary
  - Hash tables
    - `O(1)`
  - Search trees
    - Binary search trees, B+-trees, ...
- Very popular and useful data structure
  - Any data structure can be represented

Python dictionary operation

- Creation
  - `D = {}, D = {’key1’:value1,’key2’:value2, … }`
  - `dict(name1=value1,name2=value2, …)`
  - From a list: of pairs, names, ...
- Access by a key
  - `D[’name’], D[’name1’][’name2’], ’name’ in D`
  - `D.get(key)`
- Update and delete a key
  - `D.update(D1)`
  - `del D[key], D.pop(key)`
- Reading keys, values and key/value pairs
  - `D.keys(), D.values(), D.items()`