Data representation

Chapter 4 in the book

Imperative languages

- Emphasis in imperative languages on data structures with assignable components.
- Size and layout data structures tended to be fixed at compile time.
- Dynamical data structures are implemented using **pointers**.
- Storage in imperative languages is typically allocated and desallocated explicitly.

Data structures

- Primitive types

  – built-in primitive types for representing integers, reals, characters, booleans...

  – Examples of C types:

  int, float, char

  – Examples of Pascal types:

  integer, boolean, real

- **Layout of basic types:**

  Basic types are laid out by using an address to save the value and the machine representation of the value.

  In general, characters fit in a byte, integers in a word and real numbers in two contiguous words.

- Enumerated types

  – In Pascal:

```
type PrimaryColors = {Red,Green,Blue};
var PrimaryColors first_coat;
```

  – In ANSI C:

```
typedef enum {Red, Green, Blue} PrimaryColors;
PrimaryColors first_coat;
```

Arrays

- Most languages provide a mechanism to define vector or array types.
- An array is a collection of values, all belonging to the same type. Individual values in an array, called its **elements** are accessed by their **index**.
- Declaration of an array:
  – In C:

```
int a[3];
```

  is an array of 3 elements. The 3 elements can be accessed by $a[0]$, $a[1]$ and $a[2]$.

  – In Pascal:

  In Pascal the array indices can be drawn from any arbitrary range of integers.

```
var a: array [0..2] of integer;
var b: array [7..9] of integer;
```

  a is an array containing 3 elements: $a[0]$, $a[1]$ and $a[2]$.

  b is an array containing 3 elements: $b[7]$, $b[8]$ and $b[9]$.

- Almost all imperative languages allow arrays of more than one dimension.

  – In Pascal:

```
var c: array [0..2,0..2] of integer;
```

  is a two dimensions array.

  We can access an element as $c[1, 2]$.

  – The declaration is C is:

```
int c[3][3];
```

  We can access an element as $c[1][2]$.

- The array bounds determine the amount of storage needed for an array.
- The bounds are not needed until storage is allocated.

Bounds checking

- Consider the declarations:

```
int a[3]; (C)
var a: array [0..2] of integer; (Pascal)
```

- What happens if we write $a[5]$?
- It is an out-of-bounds access to the array.
- Pascal and Java insist that every access be within bounds. C makes no such claim. C assumes you know what you are doing and you get what you deserve.
- Many occurrences of out-of-bounds accesses cannot be detected by a compiler (e.g. consider $a[i]$ where i is an integer whose value is read from input). This means that i is known at running time and we need to insert checks to see if the index is indeed within bounds.

  This is the reason why C does not do the checking.

Array layout

- The **layout** of an array determines the machine address of an element $A[i]$ relative to the address of the first element.
- The layout is composed of 2 parts:
  - a part that is precomputed when the array is declared and
  - a part that is computed at run time (depending on an array subscript).
- Let consider the following Pascal declaration of an array:

```
var A: array[low..high] of T
```

- One dimension array:

  $i * w + (base - low * w)$

  – $A[low]$ begins at location $base$ and

  – w is the width of each array element.

- Two dimensions array - layout for $a[i_1][i_2]$:

  $i_1 * w_1 + i_2 * w_2 + (base - low_1 * w_1 - low_2 * w_2)$

- Efficient computation of the layout:

  The number of instructions needed to compute compute the layout is independent of the value of i. Each element can therefore be accessed in constant time, providing random access to array elements.

Random access

- The term **Random access** refers to examining or changing an arbitrary element that is specified by its position in the array.

  These operation are constant time operations for an array.

Storage allocation

- Let consider the declaration:

```
var A: array[1..c] of integer;
```

- If the value of c is known at compile time, then the bounds can be computed at compile time and the array layout allocation is done.

- If the value of c is unknown at compile time, they will not be known until runtime, then the array layout and allocation are deferred to run time and done dynamically.

Records

- Often it makes sense to group data of various types into a single structure.

- Example: The data concerning an employee in a company.

  An employee is defined by multiple attributes: name, department, salary...

  – Pascal and C allows all values associated with employee to be collected together into one structure using the concept of **records**:

  – In Pascal:

```
type Department = {Sales, Marketing, Personnel,
Engineering};
var employee : record
  name : array [0..29] of character;
  dept: Department;
  salary: real
end;
```

– In C:

```
typedef enum {Sales, Marketing, Personnel,
Engineering} Department;
struct {
char name[30];
Department dept;
float salary;
} employee;
```

The **fields** of the records can be accessed by employee.name, employee.dept and employee.salary.

Variant and union

- Consider the case where there are 2 classes of employees: permanent and hourly.

  Only permanent employees have deduction (retirement...) whereas only hourly employees have overtime wages.

- A **variant record** is a record having a part common to all records of that type, and a variant part, specific to some subset of the records.

- In Pascal:

```
type Department = (Sales, Marketing, Personnel,
Engineering, Accounting);
type EmployeeCategory = (Permanent, Hourly);
var employee : record
    name  : array [0..29] of character;
    dept  : Department;
    salary: real;
    case emptype: EmployeeCategory of
         Permanent:
           deductions: real;
         Hourly:
           overtime: real
    end;
```

employee.deductions makes sense only when employee.emptype = Permanent.

Similarly, employee.overtime makes sense only when employee.emptype = Hourly.

What happens if we try to access employee.overtime when employee.emptype = Permanent?

As in the case with array bounds checking, it is not always possible at compile time to determine whether or not a field access in a variant record is legal. Languages sometimes insist on runtime checks, but more often than not, do not perform any checking at all. Thus they leave this burden to the programmer to not do illegal accesses, giving garbage values (silently!) when such illegal accesses are made.

- In C:

```
typedef enum {Sales, Marketing, Personnel,
Engineering, Accounting} Department;
typedef enum {Permanent, Hourly} EmployeeCategory;

struct {
   char       name[30];
   Department dept;
   float      salary;
   EmployeeCategory: emptype;
   union {
        float deductions;
        float overtime;
    }
} employee;
```

As usual, C leaves the burden on the programmer to use the fields deductions and overtime wisely.

Pointers

- Each variable is associated with a location (address) in memory (store) and a value.
- The fundamental constructs of the imperative languages are the assignment operation (write values to location) and read values from locations.
- Examples:

  In C:

```
int i, j;
float f;

i = 4;
j = 8;

i = i + j;
```

  In Pascal

```
var i, j: integer;
var f: real;

i := 4;
j := 8;
i := i + j;
```

- Why $4 = i$; and $i + j = i$; are invalid in C?
- Why $4 := i$; and $i + j := i$; are invalid in Pascal?

Pointers

- Pointers are variables whose values range over the set of locations in the memory.
-
```
int i, j;
float f;
int *p;
float *q;
i = 4;
j = 8;
p = &i;
i = *p + j;
```
  - p is a pointer to integer. The value of p is a location. Let l_p be the this value.
  - q is a pointer to float. The value of q is a location. Let l_q be the this value.
  - & is an operation that gets the location associated with an expression. The value of &i is l_i.

    $p = \&i$ places l_i in the location l_p.
  - $*$ is the dereference operation that works over a pointer. $*p$ determine the value at location l_p.

Arrays and pointers in C

- Arrays and pointers are intimately related in C.
- In the declaration:

  `int a[3];`

  a is a pointer to the zeroth element $a[0]$

  $a + i$ (addition to a pointer) points to $a[i]$..

Dynamic storage allocation

- Pointers are very powerful since they allow us to access values of variables without naming the variable itself.
- Pointers are efficient. Rather than move or copy a large data structure in memory, it is more efficient to move or copy a pointer to the data structure.
- Pointers are useful to implement dynamic data structure that grow and shrink during execution.

  Example: Linked lists.

Abusing pointers

- C and its descendants give users complete freedom in manipulating pointers.
- C was originally seen as a language to program components of an operating system and it makes available to access particular locations in memory.
- Programming in C with pointers is close from assembly-level.
- Pointers can point anywhere in memory. This freedom is abused mostly unknowingly by programmers who write application code. Abusing pointers needs an intimate knowledge of how memory is allocated for variables in a program by the compiler.
- Pointers must be used wisely: to implement recursive datatypes mainly.
- Compilers optimize programs so in a vast majority of situation high-level code is no worse in performance than equivalent assembly-level code.

Types

- Variable bindings: the type of a variable

  A **variable binding** associates a property with a variable.

  Pascal has static bindings of types and dynamic binding of values to variables. Lisp has dynamic bindings of both values and types.
- Type systems:

  A type system is a set of rules for associating a type with expressions in the language. A type system rejects an expression if it does not associate a type with an expression.
- Basic rules of type checking:
  - When a function from a set A to a set B is applied to an element of set A, the result is an element of set B.
  - Arithmetic operators
  - Coercion: Implicit type conversion
  - Polymorphism

    A polymorphic function can take parameters of different types.

    In Pascal and C the only polymorphic functions are operations in built-in types. But there are polymorphic data types.
  - Type equivalence
  - Static and dynamic checking