Représentation de données

Donnée et représentation

Distinguer entre objet et représentation :

- Objet = donnée du point de vue de l'application
- Représentation = organisation des valeurs composant l'objet (i.e. encodage de l'objet)

Exemple : date du 21 janvier
A : l'entier 21 [jour 21 de l'année]
B : les entiers 21 et 1
C : chaîne "21/01"

Types de données

Définition :
Type = ensemble d'objets possiblement infini

Exemples :
1. Type booléen = $\{\text{vrai}, \text{faux}\}$
2. Type entier = $\{0, 1, -1, 2, -2, \ldots\}$

Types de données II

Classifiés en 2 groupes :

I. **Types simples** (scalaires) :
- normalement prédéfinis (sauf type énuméré)
- indivisibles : entier, caractère, booléen, pointeur, point flottant
- taille de mémoire fixe

II. **Types composés** :
- normalement définis explicitement
- formés à l'aide d'autres types
- type composé homogène : tableau
- type composé hétérogène : produit (eg, enregistrement) ou union d'autres types
- ensemble (Pascal)

Entiers

Unités de mesure = bit, octet, mot (32/64 bits = 4/8 octets)

Représentation des entiers en binaire :
(seul problème : entiers négatifs)

R0 signe+valeur absolue : $R_0(-1) = 1001, R_0(-2) = 1010$
n'est pas très efficace

R1 complément à 1 : $R_1(-1) = 1110, R_1(-2) = 1101$
Cray et CDC : on a $+0$ et -0

R2 complément à 2 : $R_2(-1) = 1111, R_2(-2) = 1110$
plupart des processeurs

complément à deux :
a. $-2^{N-1} \ldots 2^{N-1} - 1$ sur N bits
b. $-2^{N-1} \leq k < 0$ est représenté par $2^N + k$
c. $R_2(a+b) = R_2(a) + R_2(b) \bmod 2^N$ si $a+b \in -2^{N-1} \ldots 2^{N-1} - 1$

Entiers II

C permet de choisir la taille des entiers et signé ou pas :
`signed/unsigned` et `short/long`

Note 1 : `char` est un type entier qui occupe une "unité" d'espace (1 octet)
Exemple : `unsigned char x;` $0 \leq x \leq 255$

Note 2 : `char` $\leq$ `short` $\leq$ `int` $\leq$ `long`

Note 3 : `#include <limits.h>` pour avoir les limites de chaque type : `INT_MIN/INT_MAX`

Entiers III

ordre des octets dans un mot représentant un entier :

I. “big-endian” (MIPS, SPARC, M68000)
II. “little-endian” (Intel, DEC Alpha)

Exemple : l'entier 16 bits 258 = 1*256 + 2

`0000000100000010` — big-endian
`0000001000000001` — little-endian

⇒ problèmes lors de la communication entre ordinateurs

⇒ Il faut convertir dans une représentation “standard” pour envoyer la donnée (Perl : **`pack`** et **`unpack`**)

Type sous-intervalle

(sous-ensemble du type entier)

Pascal : $\langle \text{const}_1 \rangle .. \langle \text{const}_2 \rangle$

Nombre bits pour représentation :

$$\left\lceil \log_2(\text{const}_2 - \text{const}_1 + 1) \right\rceil$$

Exemples : `1..2` $\rightarrow$ 1 bit, `5..7` $\rightarrow$ 2 bits

Souvent les compilateurs arrondissent vers l'octet le plus proche pour un accès rapide

Utilisé en Pascal pour définir les bornes des tableaux :

```
var T1 : array [10..20] of integer;
    T2 : array [char] of integer;
```

Type énuméré

Chaque objet élément du type a un nom symbolique

Exemple en Pascal :

```
var j : (lundi,mardi,mercredi,jeudi,
         vendredi,samedi,dimanche);

begin
  j := mercredi;
  if j = lundi then ...;
  j := succ(j); (* j=jeudi *)
  j := pred(j); (* j=mercredi *)
end
```

Représenté comme type sous-intervalle Le type booléen de Pascal est un type énuméré.

Type énuméré II

Exemple en C :

```
enum Jours {lundi=1,mardi,mercredi,jeudi,vendredi,
      samedi,dimanche=99} j;
```

traités comme entiers :
`j=99` est OK.

Type réel

point flottant ou fixe

point flottant : $1.45 \cdot 10^{-11} = 14.5 \cdot 10^{-12} = 0.0145 \cdot 10^{-9}$
binaire : $-1.01 \cdot 2^4 = -101 \cdot 2^2 = \cdots$

signe+exposant+mantisse
nombre normalisé : $1 \leq$ mantisse < 2

représentation (IEEE 754 précision simple) :
nombre $x = (-1)^s 2^{e-b}(1+f)$ est stocké par
- s signe en 1 bit
- e exposant biaisé en 8 bits, $0 < e < 255$
- $b = 127$ biais
- $0 < f < 1$ fraction en 23 bits

Type réel II

IEEE-754 :

- précision double : exposant de 11 bits, fraction de 52 bits, biais $b = 1023$
- précision double extendue : ≥ 64 bits (128 : quadruple)
- valeurs spéciales :

valeur	exposant e	fraction f
$\pm\infty$	$11\cdots1$ binaire	0
±0	0	0
NaN	$11\cdots1$ binaire	non-zéro

  NaN : n'est pas un nombre (*Not A Number*)
- arithmétique étendue :
  $(+\infty)+(+\infty) = +\infty$, $(+\infty)+(-\infty) = \text{NaN}$
  $1/(+\infty) = +0$, $1/(-\infty) = -0$
  $\log(-0) = \text{NaN}$, $\log(+0) = -\infty$
- exceptions (pas celles de Java !)

Tableaux

Pascal :

`var T : array [`⟨type$_{\text{index}}$⟩`] of` ⟨type$_{\text{élem}}$⟩

type$_{\text{index}}$ doit être sous-intervalle ou énuméré

```
var B : array [boolean] of real;
var Col : array [(rouge,vert,bleu)] of boolean;
var I : array [10..12] of integer;
var T2 : array [0..4] of array [0..9] of real;

B[false] := 1.5; Col[vert] := true;
I[11] := 123;
T2[2][3] := 3.2; T2[3,4] := 4e-11
```

En Pascal, les indices font partie du type tableau ! ! — très mauvaise idée.

Tableaux II

C : ⟨type$_{\text{élem}}$⟩ `T[`⟨constante⟩`] ;`

```
int T[2];          T[0] = 123;
float T[5][10];    T[2][3] = 1.5;
int D[] = {2,3,4};  /* tableau à 3 élems */
```

- borne inférieure fixée à 0
- dépassement de bornes est permis :

`int X[2]; X[2]=4;` n'est pas toujours une erreur

Algol 68 : tableaux dynamiques

```
int n;
...
{
  [-n:n] real petit tableau;
}
```

tranches : `a[3 :8] := b[1 :6] ;`
bornes accéssibles : `lwb` et `upb`, dépassement interdit

Tableaux III

Exemple : tableau en Pascal

```
var T array[bas..haut] of integer;
```

stockage typique :
élément $T[i]$ est à l'adresse $a_i = D + (i - \text{bas})E$
où
- D est l'adresse de base de T
- E est la taille d'un élément (en octets)

donc $a_i = K + iE$ avec $K = D - E \cdot \text{bas}$
- K peut être calculé en avance si tableau statique
- multiplication rapide (décalage) si E est multiple de 2

Tableaux IV

tableaux multi-dimensionels

– row-major : (tableau 2D=tableau de rangées)

$$a_{ij} = D + (i - b_1)(h_2 - b_2 + 1)E + (j - b_2)E$$
$$= K + iR + jE$$

– column-major : (tableau 2D=tableau de colonnes)

$a_{ij} = D + (j - b_2)(h_1 - b_1 + 1)E + (i - b_1)E$

C, Java : row-major

FORTRAN : column-major

tableaux dynamiques : il faut stocker les bornes

— accès aux éléments prend plus de temps

Enregistrements

(=produit cartésien de types)

groupe ordonné d'objets accessibles par leur nom
- Nom connu à la compilation (sélecteur de champ)
- Offset connu à la compilation (contrairement aux tableaux)
- En général hétérogène

Représentation : disposition contiguë des champs en mémoire

Enregistrements II

Pascal :

```
record
  a : char;                       (* adr rel. = 0 *)
  b : array [1..2] of integer; (* adr rel. = 1 *)
  c : real;                       (* adr rel. = 9 *)
end
```

C : contrôle fin (au niveau des bits) sur la disposition des champs

```
struct ByteS
  { unsigned int a : 3; /* 0..7 */
    unsigned int b : 1; /* 0..1 */
    unsigned int c : 4; /* 0..15 */
  } x;
```

Les contraintes de la machine peuvent affecter la position : e.g. entier (réel) à des adresses qui sont des multiples de 4 (8)

langages OO : objets («évolution de l'enregistrement»)

Unions

type union = (union de types) En C la syntaxe est similaire aux structures

```
union UnionABC
  { char a;    /* adr relative = 0 */
    int b;     /* adr relative = 0 */
    double c;  /* adr relative = 0 */
  } abc;
```

Taille = taille du plus grand champs

Seulement un des champs est actif à un instant donné et c'est impossible de savoir lequel uniquement en examinant l'objet

Unions II

Algol :

```
union (int, real) z;
z := 9.998;
...
case (z) in
  (int i): i := i+2,
  (real r): r := r-0.02
esac
```

(représentation de z n'est pas accéssible)

Unions III

usage de `union` en C

stocker l'état

```
struct
 { enum {CHAR, INT} sorte;
   union
    { char a;
      int b;
    } val;
 } T[10];
 ...
for (i=0; i<10; i++)
  switch (T[i].sorte) {
    case CHAR: printf("%c ", T[i].val.a);
               break;
    case INT:  printf("%d", T[i].val.b);
               break;
  }
```

Unions IV

usage de `union` en C (cont.)

accéder la même valeur de manières altérnatives

```
union {
  unsigned char bytes[4];
  unsigned int i;
} mon_entier;
mon_entier.i=1;
if (mon_entier.bytes[0]==1)
  puts("little endian");
else
  puts("big endian");
```

Ensembles

[type très rare]

l'exception : Pascal

```
var s1, s2, s3 : set of 1..5;

s1 := [1,4,5];   (* 0000000000011001 = 25 *)
s2 := [2,4];     (* 0000000000001010 = 10 *)
s3 := s1 + s2;   (* 0000000000011011 = 27 *)
s3 := s1 * s2;   (* 0000000000001000 = 8  *)
s3 := s1 - s2;   (* 0000000000010001 = 17 *)
if 2 in s1 then ...;
```

représentation : un mot machine avec 1 bit par élément possible (1=présent, 0=absent)

Pointeurs

pointeur = objet qui permet un accès indirect à un autre objet

pourquoi ?

exemple : liste chaînée
chaque élément est un enregistrement

```
typedef struct Liste_s {
  int valeur;
  struct Liste_s prochain;
} elem;
```

erreur ! (taille de `elem`= ?)

solution :

```
typedef struct Liste_s {
  int valeur;
  struct Liste_s* prochain;
} elem;
```

Pointeurs II

- La taille du pointeur est indépendante de la taille de l'objet pointé (typiquement 1 mot machine)
- En général, un pointeur est représenté par l'adresse en mémoire de l'objet pointé
- Cas spécial : pointeur nul
  `nil` en Pascal, `NULL` en C
  =adresse invalide
- Type de pointeur inclut le type de l'objet pointé (en général)

Pointeurs III

I. Applications
- Structures de données dont la taille varie à l'exécution (chaînes, listes, arbres, graphes)
- Manipulation rapide d'objets (la copie d'un pointeur est plus rapide que la copie de l'objet pointé)

  Exemple : trier des enregistrements

II. Opérations principales sur les pointeurs
- Allocation d'un objet dans le tas (trouve espace inutilisé + marque comme utilisé)
- Récupération de l'espace alloué à un objet (marque inutilisé)
- Indirection (accès à l'objet pointé)

Pointeurs IV

```
Pascal                  C                              SIMULA

DECLARATION DE TYPE ET VARIABLE
  type R =              struct R                        class R;
    record               {                               begin
     x, y : integer;      int x, y;                       integer x, y;
    end;                 };                              end;

  var p, q : ^char;     char *p, *q;
      a, b : ^R;        struct R *a, *b;                ref(R) a, b;

INDIRECTION
  p^ := 'X';            *p = 'X';
  a^.x := 20;           (*a).x = 20; a->x = 20;         a.x := 20;

AFFECTATION DE POINTEUR
  q := p;               q = p;
  b := a;               b = a;                          b :- a;
  b := nil;             b = NULL;                       b :- none;

AFFECTATION DE L'OBJET POINTE
  b^ := a^;             *b = *a;                        b := a;

EGALITE
  if a=b then ...       if (a==b) then ...              if a==b then ...

ALLOCATION
  new(a);               a = (struct R*) malloc(         a :- new R;
                              sizeof(struct R));

RECUPERATION
  dispose(a);           free(a);                        **AUTOMATIQUE**
```

Pointeurs V

Il y a des dangers importants lorsque la récupération de la mémoire est sous la responsabilité du programmeur(récupération manuelle)

I. récupérer l'espace **trop tôt** : *pointeurs fous* (pointeur vers un espace réservé pour un autre usage)

- pointeur à une variable déjà relâchée,
- pointeur à une variable locale ou un paramètre (alloués sur la pile),
- pointeur à une adresse invalide,
- accès à l'extérieur des bornes d'un tableau.

une solution : «tombes»

Pointeurs VI

II. récupérer l'espace **trop tard** (ou même jamais) : *fuites de mémoire*

```
new(p);  (* allocation de l'objet 1 *)
p^ := 10;
new(q);  (* allocation de l'objet 2 *)
p := q;  (* fuite de memoire: l'objet 1 n'est plus
            accessible et ne peut plus etre recupere *)
```

- objet mort = objet qui n'est plus accessible par le programme
- Une fois qu'un objet est mort il le restera pour toujours
- fuite de mémoire = création d'objets morts
- Bogue insidieuse car le programme peut marcher longtemps sans problèmes mais tranquillement la performance se dégrade et cause éventuellement un arrêt total (e.g. Microsoft Word)

Pointeurs VII

Lisp, SIMULA et Java possèdent des «ramasses-mièttes» (*garbage collectors*) qui récupèrent automatiquement l'espace associé aux objets morts.

Deux techniques répandues :

1. Associer un *compteur de référence* à chaque objet = nombre de pointeurs vers l'objet. Récupérer espace lorsque compteur = 0. Ne fonctionne pas dans le cas des structures cycliques.

2. À partir des variables globales et la pile, *traverser tous les objets atteignables* en passant par des pointeurs et marquer ces objets. À la fin, les objets non-marqués sont récupérés.

Pointeurs VIII

C possède plusieurs routines de gestion mémoire (disponibles dans `stdlib.h`)

```
void *malloc (int taille_bloc);
void *calloc (int nb_elem, int taille_elem);
void *realloc (void *ptr, int nouvelle_taille);
void free (void *ptr);
```

- `malloc` alloue un bloc, `calloc` alloue un tableau et `realloc` change la taille d'un objet alloué (ce qui demande parfois d'allouer un nouveau bloc et de copier l'objet)
- Le pointeur retourné est de type `void*` (pointeur universel) et il faut le convertir vers le bon type de pointeur avec un «cast»
- `NULL` est retourné si la mémoire est épuisée
- C++ possède 2 opérateurs de gestion mémoire de plus haut niveau que C

```
int *p;
char *s;
p = new int;      // p = (int*)malloc(sizeof(int));
s = new char[5]; // s = (char*)calloc(5,sizeof(char));
delete p;         // free(p);
delete [] s;      // free(s);
```

Pointeurs IX

Le langage C permet une forme d'arithmétique sur les pointeurs (opérateurs + et -)

- `&`⟨var⟩ : pointeur vers ⟨var⟩
- `p+n` (où `n` est de type entier et `p` est de type `T *`) : pointeur vers n-ième objet de type `T` après objet pointé par `p`
- `p2-p1` (où `p1` et `p2` sont de type `T *`) : nombre d'objets de type `T` de `*p1` à `*p2`
- `t` (où `t` est un tableau) : pointeur vers premier élément du tableau
- C définit : `x[y]` = `*(x+y)` = `*(y+x)`
  Donc (!) : `1[t]` = `t[1]` et `1[t1][t2]` = `t2[t1[1]]`
- Les opérateurs ++ et -- sont utilisables sur les pointeurs pour avancer ou reculer un élément

Pointeurs X

Le traitement de chaînes de caractères en C est basé sur les pointeurs

```
char t[4];
char *p, *q;

p = "ABC";

putchar (p[2]);       /* imprime C */
putchar ("XYZ"[2]);   /* imprime Z */

q = t;                /* copier dans t */
while (*p != '\0')
  *q++ = *p++;
*q = '\0';

if (t == "ABC") ...; /* toujours faux! */

/* utiliser strcmp(t,"ABC") de <string.h> */

*p='Z'; /* erreur: pointeur vers une constante */
```

Typage et sûreté

La notion de type est utile pour définir la notion d'erreur

Exemples

1. `int *p; ... abs(p) ...` est incorrect car (en C) l'argument de `abs` doit être de type entier
2. `int t[10]; ... t[1.5] ...` est incorrect car l'index doit être de type entier
3. enregistrement variant en Pascal (comme `union` en C)

Déf : Une erreur de type survient lorsqu'une fonction (ou opérateur) est appelée avec un argument qui n'est pas du type attendu

Typage et sûreté II

Les vérifications de type permettent de garantir qu'aucune opération incorrecte ne sera exécutée

I. Vérifications de type à la compilation (**typage statique**)
⇒ exécution plus rapide
⇒ programme plus *sûr* : moindre erreurs de type

II. Vérifications de type à l'exécution (**typage dynamique**)
⇒ requis dans certains cas (Lisp, unions de C)
⇒ programme moins sûr

Équivalence de types

Est-ce que les variables ont le même type ?

```
struct Sif  {int   i; float f;} reg1;
struct Sif2 {int   i; float f;} reg2;
struct Sfi  {float f; int   i;} reg3;
```

équivalence par nom ou par structure

C : par nom pour `struct` et `union`, par structure pour autres

Ada : par nom (les types suivants ne sont pas compatibles)

```
type celsius is new FLOAT;
type fahrenheit is new FLOAT;
```

Algol 68 : par structure (les types suivants sont compatibles)

```
mode a = struct (int val, ref a next);
mode b = struct (int val,
    ref struct (int val, ref b next) next);
```

Conversion de types

I. explicite

Ada : `FLOAT(I)` (fonction)
C, C++, Java : `(float) i` (opérateur)

II. implicite (coercition)

– élargissement (*widening*) :
  `short int`→`int`→`float`→`double`
– rétrécissement (*narrowing*) `int i=2.99;`
– (etc.)

Conversion de types II

conversion automatique : utile mais aussi une cause d'erreurs

I. perte de précision `int`→`float`

```
int z=1234567890;
float f=z; /* z=1234567936.0 */
```

II. conversion `signed`→`unsigned`

```
    unsigned int u=0;
    if (u<-1) puts("Bizarre.");
```

III. ou une conversion souhaitée. . .

```
float C, F;
...
C=(F-32)*(5/9);
```

hélas, pas de conversion automatique pour `5/9`
Pascal : division entière `div` et flottante `/`