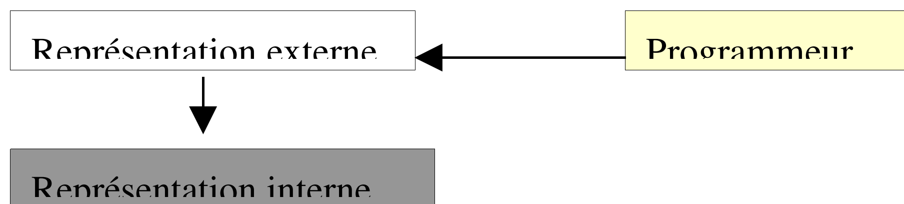
Type Abstrait de Données

Définition et implémentation

Dans une programmation en large il est nécessaire d'identifier et de spécifier les données assez tôt dans le processus de développement.
Une fois la donnée complexe identifiée, nous nous intéressons à sa spécification. Le but de cette spécification est de définir l'interface d'utilisation (**représentation externe**) de cette donnée et de lui donner une sémantique abstraite indépendante de l'implantation (**représentation interne**).

Les langages de programmation impératifs typés offrent des types dits prédéfinis. Ceci conduit à une utilisation naturelle et simple des variables définies à partir de ces types. Ces types sont souvent définis d'une façon abstraite puis implémentés dans le langage de programmation.



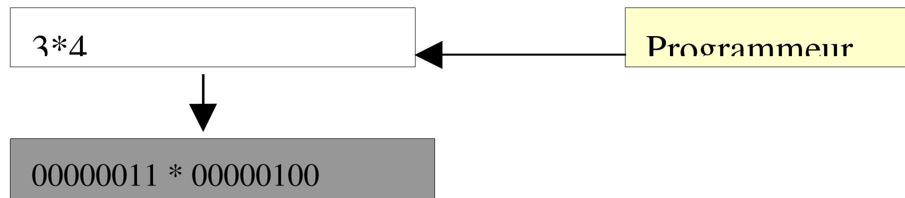
Exemple les entiers :

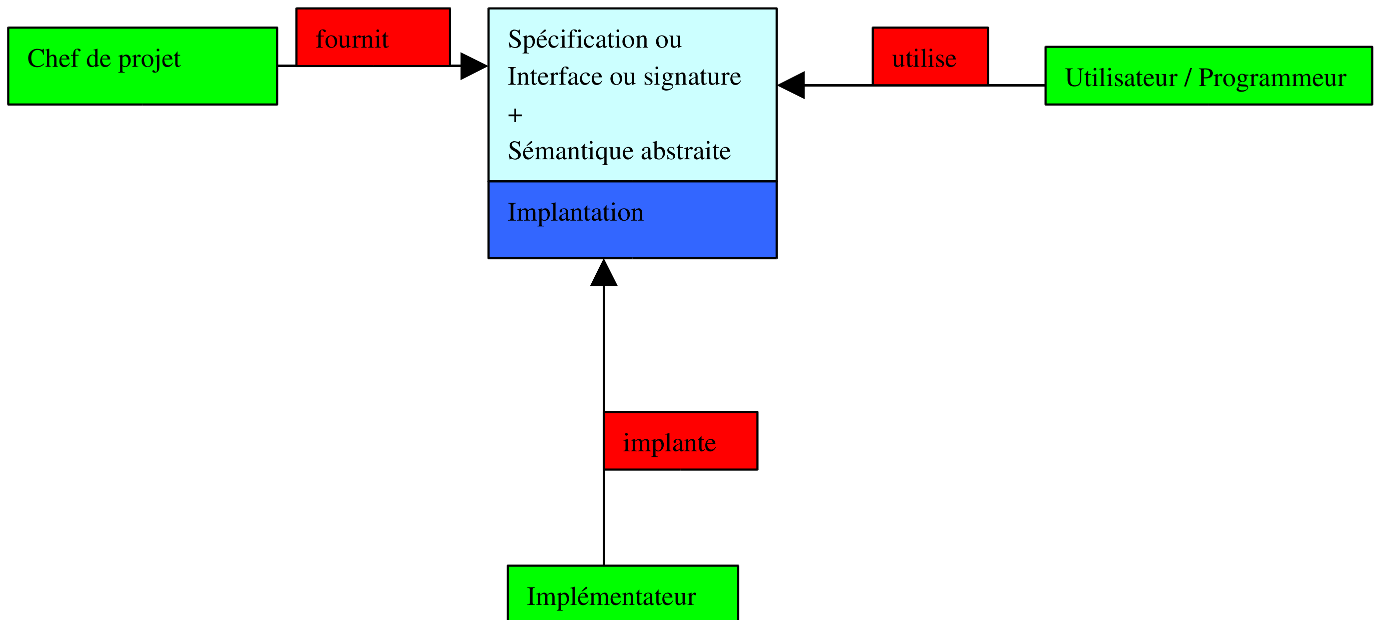
Représentation externe des entiers en C :

```
int          /* le nom du type
5, -6, 21    /* des nombres
+, -, *, /   /* les opérations permises
```

Ceci représente l'interface des entiers. Cette représentation externe donne la possibilité d'une utilisation naturelle des entiers et elle est incomparablement plus facile à utiliser qu'une représentation interne qui est dans ce cas une représentation binaire.

Exemple : Multiplication





Nous nous intéressons aux types abstraits de données les plus souvent utilisés. Nous proposons la classification suivante :

Les types à structure linéaire : Liste, Pile, File
Les types à structure arborescente : Arbre Binaire, Forêt
Les Graphes

Spécification

Fonctionnelle(Signature)

Donner un nom significatif au type
Donner les profils des opérations du type
Une opération est considérée comme une fonction

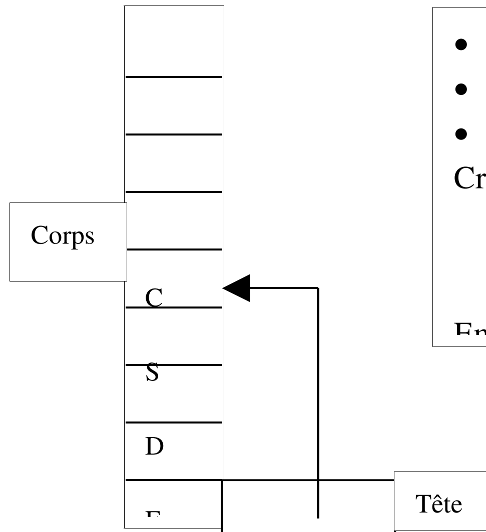
Sémantique

Pour donner une sémantique abstraite nous utilisons le langage de la logique équationnelle. Nous caractérisons chaque opération par des axiomes. Les opérations sont considérées comme fonctions totales ou fonctions partielles dont la restriction s'exprime par des pré conditions.

Une façon pour construire les axiomes nous conduit à partager les opérations du type en constructeurs et opérateurs. Un constructeur est indispensable à la représentation des valeurs du type. Chaque axiome est construit par l'application d'un opérateur sur un constructeur si la précondition est satisfaite.

Exemple : Les Piles

Une pile est une structure possédant une seule tête de lecture/écriture ce qui conduit à la gérer selon la politique '' premier entré dernier sorti ou dernier entré premier sorti (LIFO)''.



- La pile doit être vide à la création
- Une pile vide est construite par Créer
- Une pile non vide est construite par

Créer suivie par une suite d'Empiler

Empiler(Empiler(Empiler(Empiler(Empiler(créer, A), E), D), S), C)

Spécification d'une pile d'entier

Fonctionnelle(Signature)

Type PILE
Utilise Entier, Booléen

Constructeurs

Créer : → Pile
Empiler : Pile X Entier → Pile

Opérateurs

Dépiler : Pile → Pile
Vide : Pile → Booléen
Top : Pile → Entier

Sémantique

P : pile
x : Entier

Préconditions

Dépiler(p), Top(p) déf ssi Vide(p) = False

Axiomes

Vide(Créer) = True
Vide(Empiler(p,x)) = False
Dépiler(Empiler(p,x)) = p
Top(Empiler(p,x))= x

Implémentation en langage C

L'implémentation d'un type abstrait consiste à :

- Implémenter une représentation interne du type
- Implémenter les fonctions
- Assurer la séparation entre l'implémentation de l'interface (signature) et le corps. Deux grands avantages :
  1. Permet de modifier le corps sans toucher à l'interface donc sans modifier les programmes utilisateurs
  2. manipuler par abstraction
- Assurer la protection de la représentation interne

1. **Implanter une représentation interne du type et implémenter les fonctions**

Nous sommes souvent devant un choix entre une implémentation **statique** ou une implémentation **dynamique :**

- Statique : signifie qu'au chargement du programme, à l'exécution, la réservation des variables est effectuée et ces variables ne changent pas de place par rapport à l'espace mémoire du programme. (tableau)

Avantages : accès rapide
Inconvénients : occupation inutile de l'espace mémoire pendant l'exécution

- Dynamique : les réservations et libérations des variables s'effectuent en cours d'exécution. (pointeurs)

Avantages : optimisation de l'espace mémoire occupée
Inconvénients : l'efficacité est diminuée par la gestion de la mémoire qui se fait en cours d'exécution.

Implémentation du type pile de 10 entiers:
Le choix entre fonction et procédure est décidé à partir du point de vue utilisation.
En sachant qu'en langage C les sous-programmes sont représentés sous forme de fonctions C. La différence est aperçue à l'utilisation : une fonction est utilisée comme une expression, par contre une procédure est utilisée comme une instruction.
Nous considérons dans ce qui suit l'utilisation de error() qui permettra l'échappement et la sortie immédiate du sous-programme.

Statique

```
#include <stdio.h>
#define N 10;
typedef struct pile
{int indice; int t[N] ;} piles ;

void creer_pile (piles *p) {  (*p).indice = -1 ; } /* la pile est créée vide */

void empiler (piles *p , int e)
{ if ((*p).indice ) == N-1) error() ;
                                        /* précondition  en liaison avec l'implémentation statique */
          (*p).indice ++ ; (*p).t[(*p).indice] = e; }

void depiler (piles *p)
 { if (vide(*p)) error() ;  /* precondition*/
          (*p).indice-- ;}

int vide (piles p) { return (p.indice == -1);}

int top (piles p)  { if (vide(p)) error () ; return (p.t[p.indice]);}
```

Dynamique

```
#include <stdio.h>
typedef struct cel {int info; struct cel *suiv;} cels;
typedef  cels * piles;

piles creer () {return NULL;}

void empiler (piles *p, int e)
{ piles q;
 q = (piles) malloc (sizeof(cels));
q->info = e; q->suiv = *p; *p = q;}

void depiler ( piles *p)
{ if vide (*p) error() ;
 *p = (*p)->suiv ;}

int vide(piles p) { return p==NULL;}

int top (piles p) { if vide(p)  error() ; return p->info ;}
```