Cours de Génie Logiciel
Contrôle Qualité en Programmation

Laurent Henocque
http://laurent.henocque.free.fr/
Enseignant Chercheur ESIL/INFO France
http://laurent.henocque.perso.esil.univmed.fr/

version 1.4 en date du 17 Novembre 2008

Introduction

Le génie logiciel est une discipline qui vise à structurer et organiser l'ensemble des activités liées à la réalisation de logiciels, et à promouvoir des niveaux de qualité croissants.

L'expérience des succès et échecs de l'industrie informatique a permis de dégager des concepts assez fédérateurs, et de mesurer leur efficacité. L'activité des chercheurs dans des domaines aussi variés que la psychologie, la linguistique, l'ergonomie et l'informatique évidemment a permis de faire apparaître des notions dont certaines font preuve aujourd'hui d'une grande acceptation du marché. La programmation orientée objet en est un exemple, avec son langage phare qui est C++. De très grands projets ont été réalisés selon des méthodes de spécification et de garantie de qualité souvent dérivées de méthodes utilisées pour l'industrie spatiale. On sait que l'efficacité maximale sur un projet informatique est obtenue pour deux personnes travaillant pendant six mois. Mais les grands logiciels sont réalisés par des équipes immenses (une centaine d'ingénieurs) en plusieurs années.

Ce cours a pour projet de donner les bases de techniques de travail reconnues comme nécessaires pendant tout le cycle de vie du logiciel, et d'apprendre les méthodes qui permettent d'affronter systématiquement et de résoudre les problèmes.

Fréquemment, les cours de génie logiciel s'appuient sur la chronologie naturelle des projets, en débutant par la spécification, pour continuer par la conception et finir par la programmation et la gestion de projet. Ce cours aborde les aspects individuels de la programmation, sur la base du langage C++, mais gagne a être abordé avant même que de parler de conception et de spécification Les raisons sont notamment que :

- nous pensons que la qualité des étapes préalables à la programmation effective d'un logiciel, à savoir cahier des charges et conception, dépendent de la connaissance de principes de qualité fondamentaux dans la programmation.

- le langage C++ couvre par sa puissance expressive une partie très importante des besoins liés à la rédaction de documents de spécification et à une conception orientée objet.

L'utilisation de C++ ne doit pas faire oublier que le concepteur (au sens large) d'un logiciel doit faire preuve d'ouverture d'esprit, et de malléabilité. En effet, une fois maîtrisées les étapes de spécification et de conception, l'étude préalable à la réalisation d'un logiciel conduit souvent à l'utilisation de langages ou d'outils dont on n'est pas forcément familier.

La démarche qualité présentée dans ce cours sur la base du langage C++ peut être vue comme un exemple de ce qui doit être effectivement mis en oeuvre quels que soient le langage, la méthode et les outils utilisés au final.

Plan

Nous verrons donc dans une première partie les concepts fondamentaux de qualité logicielle, et notamment les notions de contrats, d'invariants et une discussion sur la notion de test.

Ensuite, nous évaluerons de quelle manière l'activité de programmation peut être organisée, dans un objectif de qualité. Une troisième partie liste des considérations techniques sur la qualité. Enfin, certains aspects humains (psychologiques notamment) seront abordés.

Partie 1 Qualité en programmation: Notions fondamentales

1.1. Le contrat

Principe fondamental dans toutes les activités industrielles ou plus généralement économiques, le contrat est aussi un fondement de toutes les étapes de l'élaboration d'un logiciel, de sa spécification jusqu'à son abandon. Bien sûr, en fonction du point auquel on se situe, la nature et les effets des différents contrats seront variables. Toutefois, on peut reconnaître parmi les fonctions du contrat les éléments fondamentaux suivants:

• le contrat garantit une communication sans défaut, par un examen exhaustif de tous les cas nécessaires. Un bon contrat lève toute ambiguité entre ses parties.

• le contrat est un support fondamental de la simplicité des solutions mises en oeuvre pour le respecter. En effet, il permet de travailler en monde clos, et de ne pas anticiper des évolutions improbables.

En matière de génie logiciel, les documents contractuels sont nombreux. L'ensemble de ces pièces décrit les réponses aux questions suivantes :

• "quoi" : quelles sont les fonctionnalités à implanter,

• "comment ": comment accède t'on à ces fonctionnalités,

• "quand" quelles sont les limites hors desquelles ces fonctionnalités ne sont pas accessibles (pas implantées).

Ces contrats lient dans tous les cas des "clients" et des "fournisseurs". Trois niveaux fondamentaux de contrat sont usuels en informatique, et lient des parties distinctes. Le plus haut niveau permet la communication entre acheteur du logiciel et prestataire, c'est le cahier des charges (spécification). Le niveau suivant lie les membres d'une équipe

d'informaticiens, c'est la conception. Enfin, avec la granularité nécessaire, le dernier niveau lie les programmes entre eux.

1.1.1. La spécification : contrat entre client et fournisseur

Un contrat fondamental lie le prestataire d'une solution informatique et son client: la spécification. Son document de référence est le cahier des charges. Une fois accepté, il constitue un engagement des deux parties. Le client admet que ses besoins y sont exprimés de façon correcte et complète, et donc accepte d'avance toute solution conforme. Le prestataire sait de son côté que les solutions qu'il envisage sont réalisables (dans les délais, ceux ci pouvant apparaître comme un besoin dans la spécification). Il a la garantie qu'aucun besoin nouveau ne viendra briser l'édifice qu'il va construire.

L'acceptation du cahier des charges a donc pour effet de figer la partie initiale de l'étude, permettant à la conception, puis au développement, de se faire sur des bases solides. Il ne faudrait pas croire pour autant que la spécification est réalisée indépendamment de ces phases ultérieures. L'engagement de réalisation rend implicite la validation technique des solutions envisagées. Cette validation peut aller jusqu'à la programmation de certains modules pour tester une faisabilité. Par ailleurs, l'expression des besoins par le client est parfois floue, et nécessite des interactions entre les parties sur la base d'un prototype. Dans ce cas, des itérations successives conduiront par exemple à l'acceptation d'un prototype qui sera jugé adéquat.

Certaines activités logicielles ne comportent pas de client tangible, et l'on pourrait être tenté de passer outre une phase de spécification contractuelle. C'est le cas notamment dans l'édition de produit logiciel, ou l'on tente de répondre à un besoin qui n'est pas toujours exprimé (sauf peut être par une étude de marché sérieuse). Il est pourtant nécessaire d'exprimer de manière écrite les engagements que prendront les différents acteurs. La renégotiation en cours de réalisation d'un point de la spécification est une opération qui peut conduire à des dépassements considérables de budget. Un document écrit entre le service de recherche et développement et sa direction est le meilleur témoin des responsabilités de chacun.

Dans la mesure ou la spécification est une vue exacte du logiciel à réaliser, ce document est la première pierre de sa documentation, et sa qualité initiale sera un profit à toutes les étapes du travail et en particulier à la fin.

Dans de nombreux cas, la pression exercée par l'extérieur sur le prestataire pour faire démarrer les choses le plus vite possible aura pour effet de réduire le temps passé sur la rédaction d'un document, sinon sur la réflexion associée. Le prestataire doit avoir la force de lutter pour écarter les points d'ombre, et éviter de laisser des bombes à retardement dans la spécification.

Au niveau de la spécification, le "quoi" reflète l'ensemble des fonctionnalités du produit, le "comment" sera souvent la spécification de l'interface homme machine, et le "quand" correspondra à la mention explicite rigoureuse des différentes limites de fonctionnement du système (pas plus de 3 utilisateurs, pas de facture à zéro, etc…).

1.1.2. La conception : contrat liant les membres de l'équipe

Le contrat qui lie l'ensemble des développeurs entre eux et avec le chef de projet est la conception. La conception décrit de manière détaillée l'ensemble des solutions techniques devant être mises en oeuvre pour implanter ce qui a été spécifié. Il est formellement impossible de faire plus, ou moins, que ce que la conception a décrit. Aucune initiative individuelle n'est possible dans ce cadre. Toutes les influences et toutes les contraintes ont été prises en compte pour concevoir, et l'initiative d'un membre de l'équipe n'ayant pas de vision d'ensemble est à proscrire (rappelons que personne n'a jamais une vision d'ensemble suffisamment détaillée d'un grand projet pour prendre une décision technique très pointue).

La conception détaillée reflète évidemment tous les éléments annoncés dans la spécification. Elle y ajoute de façon aussi détaillée que souhaitable les conditions "quoi; comment, quand" des différents modules qu'elle décrit.

1.1.3. contrat entre programmes

Un logiciel est découpé en éléments logiques indépendants (en général des classes et opérations). Il est bon de décrire comment chaque élément s'engage sur les points suivants :

- quoi : quelle est la fonctionnalité implantée par ce programme. Bien sûr, dire ce que fait une fonction est une condition incontournable[1].
- comment : quelles sont les règles de nommage (noms de fonction, types des arguments, types en général) pour obtenir cette fonctionnalité.
- quand : quelles sont les données de base pour lesquelles son fonctionnement est garanti (contraintes de domaine).

Le corollaire de ces engagements est une facilité de développement accrue. En effet, on ne peut exiger d'une fonction de prévoir, voire de traiter un cas exceptionnel. Hors de son domaine de définition officiel, on doit considérer qu'une fonction produit des résultats indéterminés (il est bon de penser que le programme s'interrompra).

En d'autres termes, on ne doit pas considérer qu'une fonction traite une exception. Traiter une exception, c'est à dire s'en sortir honorablement, revient à ajouter l'exception au domaine de définition, même si le résultat du calcul sur cette valeur est sujet à caution, ou même illogique.

Exemple : supposons qu'on implémente la division réelle y = f(x,z) = x/z. Le cas z égale zéro suggère deux attitudes.

- 1: Cette situation exceptionnelle ne doit sous aucun prétexte se produire lors de l'utilisation du logiciel, et ne peut se produire si les conditions d'exécution sont correctes par ailleurs. On décide alors de ne pas tester la valeur 0 pour z dans la version définitive (commerciale) du programme f. Bien sûr, il peut s'avérer utile de tester ce cas pendant le développement et les tests. Mais ce test ne participe pas à la logique du programme. L'appel f(x,0) interromp le programme, en fournissant plus ou moins d'informations utiles pour débugger.

```
float divise(float x, float z)
{
#ifndef NO_DEBUG
      if (z == 0) then {
            printf ("Division par zéro");
            exit (1);
      }
#endif
      return x/z;
}
```

- 2: Le cas peut légitimement se présenter car les programmes n'ont pas le contrôle de toutes les données en entrée, et une décision de type moyen terme doit être prise, sans interrompre le programme[2].Dans ce cas, zéro est une valeur participant au domaine de définition de la fonction.

```
float divise(float x, float z)
{
      if (z == 0) then {
            return MAX_FLOAT; //mauvais mais peut être acceptable
      } else {
            return x/z;
      }
}
```

En dépit de son apparente plus grande simplicité, l'approche 2 conduit à une programmation beaucoup plus difficile dans la réalité. Une raison en est que, lorsque cela n'est pas explicitement mentionné dans la conception, laisser la porte ouverte à la prise en compte de situations exceptionnelles conduit à compliquer considérablement les algorithmes. Noter aussi que le cas 1 correspond en fait à un programme plus simple au final, et que le test de domaine possède sa logique propre indépendamment de la fonctionnalité.

Il faut noter que les langages de programmation modernes (C++, Java) permettent de gérer les exceptions. Il s'agit d'un mécanisme de contrôle particulier, mais la prise en charge de ces exceptions rentre dans le contrat au sens propre (donc ne contredit pas ce qui est dit globalement dans cette section)

Deuxième pôle du contrat "interne" entre les différents modules d'un logiciel, l'interface de programmation (API : Application Programming Interface en anglais) doit être définie une fois pour toutes de façon très rigoureuse, et ne plus varier. Ainsi, les différents développeurs d'une équipe peuvent utiliser dans leurs programmes les formes définitives des appels de fonctions, noms de types et noms de variables.

Dans l'exemple de la division, l'ordre des deux arguments est défini à priori. On peut également définir des règles génériques. Par exemple décider que dans toutes les fonctions portant sur un tableau dont la taille est fournie en paramètre, la taille précède le tableau lui même dans la liste des arguments.

Il est d'usage que sur l'ensemble d'un projet, ou sur même sur une gamme de produits, soient définis des guides de style pour la définition des types et des signatures de fonctions et méthodes[3] (y compris leur nommage). On pourra décider que toutes les fonctions du module graphique commencent par "Gr", que tous les objets externes aient des noms commençant par une lettre capitale, que tous les objets statiques soient en majuscules, qu'aucun nom de variable ne fasse moins de six caractères etc…

---

[1]Il arrive toutefois que lorsque le développement n'est pas bien encadré on finisse par ne plus savoir ce que fait exactement une fonction. Ce n'est pas rare.

[2] écriture de la chaine "#@@@?!!ERREUR" ou encore "infini" dans un fichier texte, ou de la valeur NULL dans un fichier base de données.

[3]On appelle signature d'une fonction l'énoncé exact de son en tête mentionnant la totalité des types des arguments avec leurs atributs annexes : passage par référence, caractère constant, etc…

1.2. La notion d'utilisateur parfait

La conception et la programmation gagnent considérablement au respect du "principe de l'utilisateur parfait"

> Principe de l'utilisateur parfait: tout programme s'engage à se comporter comme un utilisateur parfait de ses ressources (les fonctions qu'il appelle), c'est à dire à ne jamais appeler une fonction dans des conditions qui la mettent hors de son domaine de définition.
>
> Autrement dit : l'utilisateur d'un programme garantit qu'il n'activera jamais ce programme dans des conditions inacceptables.

Ce principe est le réciproque du contrat signé par chaque programme :

> Tout programme qui fournit un service au système s'engage à fonctionner correctement lorsque ses éléments de calculs sont valides.

Autrement dit : la responsabilité d'un élément de programme est limitée à son domaine de fonctionnement, mais elle est entière dans ce domaine.

- Aucune fonction ne doit s'attendre à gérer un cas ou un autre élément de programme fournirait des données incorrectes.
- Aucune fonction ne doit tester les arguments de son calcul pour leur correction.

En termes plus mathématiques, on considère qu'une fonction possède un domaine de définition, et un domaine d'application. Le programme qui implémente la fonction ne doit pas tester l'appartenance de ses opérandes au domaine de définition de la fonction. Par compte, il doit garantir sa correction (résultats justes) et sa complétude (couverture de tout le domaine de définition).

Bien entendu, il est nécessaire de fournir de manière séparée des fonctions qui testent l'appartenance d'arguments au domaine.

1.3. L'utilisation d'invariants

Le principe d'utilisateur parfait stipule qu'un programme ne teste pas l'appartenance de ses arguments au domaine de définition. Pourtant, du point de vue du développeur, ce test doit être effectué, pour des raisons évidentes de détection d'erreurs pendant le développement. Les tests de domaine, mais ce ne sont pas les seuls figurent parmi les invariants des programmes. Un invariant est une propriété logique qui est toujours vraie pendant l'exécution d'une partie d'un programme, soit qu'elle soit un prérequis (c'est alors bien un invariant de domaine de définition), soit qu'elle soit implicite du fait de la nature du programme (la définition et la vérification des invariants implicites est un outil de base dans la preuve formelle de programmes).

Les invariants de domaine sont appelés des **préconditions**, et les invariants implicites portant sur la valeur que doit retourner une fonction s'appelent les **postconditions**. Ce ne sont pas les seuls. Par exemple les paramètres intervenant dans une boucle peuvent être soumis à un certain nombre d'invariants (le test de continuation de la boucle en est un).

Dans tous les cas, on gagne à ajouter au programme des opérations supplémentaires permettant la vérification des invariants. Comme cette vérification n'est utile que pendant le développement, et plus lorsque le produit aura été testé et

ne violera plus aucun invariant, on doit utiliser un macro processeur pour que le test puisse être retiré dans tous les programmes par une simple option de compilation. L'exemple le plus fameux de ce qu'il est possible de faire est la macro "ASSERT" du C ANSI. En voici la définition :

```
#define assert(test)                                                \
      if (!(test))      {                                           \
            printf ("assert violé %s ligne %d du fichier %s\n",     \
                  #test,                                            \
                  __LINE__,                                         \
                  __FILE__);                                        \
            abort ();                                               \
      }                                                             \
```

Donnons un exemple. Supposons que le assert du programme suivant se trouve à la ligne 215 d'un fichier appelé test.cc :

```
#include <assert.h>
…
float un_sur_x (float x) {
      assert (x!=0);
      return 1/x;
}
…
```

Un programme permettant le test de cette fonction sera par exemple:

```
main () {
      …
      un_sur_x (0);
      …
}
…
```

Alors l'application du préprocesseur C (cpp) sur ce fichier, par la commande

- `cpp test.cc`

produira un fichier test.i dans lequel le texte de la fonction `un_sur_x` aura été modifié de la façon suivante :

```
…
float un_sur_x (float x) {
      if (!(x!=0))      {
            printf ("assert violé %s ligne %d du fichier %s\n",
                  "x!=0",
                  215,
                  "test.cc");
            abort ();
      }
      return 1/x;
}
```

La compilation du fichier test.cc débute de façon invisible par un appel du préprocesseur C. La commande de compilation de test.cc est par exemple

- `CC -o test test.cc`

L'appel malencontreux de `un_sur_x (0)` dans le main provoquera la sortie :

```
>test
assert violé x!=0 ligne 215 du fichier test.cc
abort (core dumped)
>
```

Telle qu'elle a été définie, la macro assert est traduite par un test qui sera toujours effectué. Or on souhaite pouvoir produire des versions finales des programmes qui ne fassent plus ces tests. On utilise alors les primitives de compilation

conditionnelle #ifdef (vraie si un symbole est défini) ou #ifndef (vraie si un symbole n'est pas défini). Voici maintenant assert sous une forme assez proche de celle avec laquelle il est généralement défini dans le fichier "assert.h".

```
#ifndef NDEBUG
#define assert(test)                                              \
      if (!(test))      {                                         \
            printf ("assert violé %s ligne %d du fichier %s\n",   \
                  #test,                                          \
                  __LINE__,                                       \
                  __FILE__);                                      \
            abort ();                                             \
      }                                                           \
#else
#define assert(test)
#endif
```

La compilation de test par "`CC -o test test.cc`" produira le même résultat que précédemment. Le symbole NDEBUG n'étant pas défini, l'implantation de assert choisie par le préprocesseur sera celle qui fait le test. Par contre, avec

- `CC -o test -DNDEBUG test.cc`

toutes les utilisations de assert dans le fichier test.cc seront substituées par rien. Tout se passe alors comme si l'on avait écrit :

```
float un_sur_x (float x) {
      return 1/x;
}
```

L'appel malencontreux de `un_sur_x (0)` dans le main ne provoquera peut être plus d'erreur, mais le résultat de la division sera indéterminé, ou bien produira une sortie du type:

```
>test
test :: run time error 3715 : zero divide
>
```

On voit combien il est préférable de disposer en commentaire des noms de fichiers et des numéros de ligne.

1.4 Différentes sortes d'invariants

Les pré conditions et post conditions sont les formes d'invariants les plus connues. Les programmes comportent d'ailleurs généralement essentiellement des pré conditions. Toutefois, d'autres catégories peuvent être rencontrées.

les invariants de compatibilité système et/ou architecture

De tels invariants décrivent la dépendance d'un programme à une condition liée à la machine qui pourrait ne pas être maintenue sur une autre plate forme, engendrant ainsi un problème de portage. L'exemple le plus connu est celui liè à la taille des types fondamentaux :

```
assert(sizeof(t_int)>=4); // ce programme requiert des int d'au moins 4 octets
```

les invariants de compatibilité de bibliothèques

Ici, on doit garantir poar exemple que des types de base utilisés par deux bibliothèques différentes ont les mêmes caractéristiques. Par exemple :

```
assert(sizeof(lib1_int) == sizeof(lib2_int)); // même taille d'int
```

les invariants d'étape dans les algorithmes

Un tel invariant décrit une condition connue comme devant être vraie en un point d'un algorithme (pointeur non nul par exemple). L'exemple typique se rencontre à la sortie d'une boucle de recherche. Supposons que la boucle parcourt une structure de données à la recherche d'un plus grand élément, accédé par un pointeur. Si la structure de données n'est pas vide, ce qui a été testé auparavant, la boucle doit produire un pointeur non nul vers le plus grand élément. Cette propriété doit être documentée et enforcée par un assert.

les variants de boucle

Un tel invariant s'appelle un « variant » parce qu'il décrit le fait que deux itérations successives d'une boucle modifient la valeur d'une variable de contrôle d'une manière telle que l'on sait que le programme va s'arrêter. Ce concept est utilisé en preuve de programmes, et peut être utilisé dans les boucles à critère d'arrêt non explicite (autre qu'une boucle « for (int i = 0; i < 10 ; i++) » dont on ne sait pas à priori qu'elle s'arrêtera un jour.

En C et C++, il faut utiliser une variable auxiliaire pour comparer l'état d'une variable à chaque itération avec sa valeur antérieure.

```
int lastloopvar = 10000;
int loopvar = 7;
while (1){                              // pourrait ne jamais s'arrêter
      if (loopvar == 0) break;          // sortira t'on un jour ?
      assert (lastloopvar > loopvar);   // certainement !!
      lastloopvar = loopvar;
      loopvar --;
}
```

les invariants de classe

Ces invariants décrivent des conditions vraies de toutes les instances d'une classe à tout moment de leur existence compris entre la fin de leur construction et le début de leur destruction.

```
class ThisClassIsChecked {
      int integrity(){}
      ThisClassIsChecked(){
            ... initialize ...
            assert(integrity());
      }
      ~ThisClassIsChecked(){
            assert(integrity());
            ... delete everything ...
      }
      method_A(){
            assert(integrity()); // avant de commencer
            ... do everything ...
            assert(integrity()); // avant de sortir
      }
      ...
}
```

1.4. Qu'est ce qu'un test

Les tests réalisés lors de l'exécution d'un programme relèvent de deux problématiques complémentaires. Certains décident de l'appartenance d'un jeu de données au domaine de définition d'une fonction, d'autres identifient l'appartenance à une partie d'un tel ensemble.

Par exemple, imaginons une fonction définie par intervalles sur le domaine des nombres réels strictement positifs:

- $x \leq 0$ fonction non définie *test de domaine*
- $x > 0$ et $x < 3$ $f(x)=1/x$ *test de partie*
- $x \geq 3$ $f(x)=1/3$ *test de partie*

Principe des tests de domaine: une fonction qui implante un algorithme ne teste jamais l'appartenance de ses arguments au domaine de définition.

Ce test s'il s'avère cependant utile est implanté par une fonction spécialisée à valeur booléenne, ou dans des cas triviaux par un simple test du langage.

Ce principe sert trois objectifs fondamentaux de concision des programmes, de facilité de programmation, et de performance.

- concision certains tests étant supprimés en tête de fonctions qui implantent des algorithmes, le texte du programme se trouve donc réduit à l'expression simple des algorithmes.

- facilité le programmeur d'un algorithme ne s'attend pas à gérer des situations d'exceptions, puisqu'il n'en écrit pas les tests. La réflexion qui est aboutit au programme est donc débarassée de ces considérations auxilliaires.

- performance l'économie provenant de la non réalisation de certains tests est réelle.

- maintenabilité si l'on utilise des invariants à la place

Par exemple, soit f une fonction à valeur entière prenant deux arguments entiers non nuls. Voici le programme que l'on est tenté d'écrire :

```
int f(int x, int y)     {
      if (x == 0) {
            printf ("f :: erreur : argument x nul");
            return 0;//par exemple, mais quelle valeur conviendrait?
      } else if (y == 0) {
            printf ("f :: erreur : argument y nul");
            return 0;//par exemple, mais quelle valeur conviendrait?
      } else {
            … algorithmes de f
      }
}
```

La même version, avec le lancement d'exceptions.:

```
int f(int x, int y)throws NullArgumentException{
      if (x == 0) {
            throw new NullArgumentException();
      } else if (y == 0) {
            throw new NullArgumentException();
            //il faudrait deux classes, pour distinguer x, et y, ou un
commentaire, mais comment sera-t-il utilisé?
      } else {
            … algorithmes de f
      }
}
```

Très souvent, le programmeur est pris par le temps, et ne peut consacrer toute l'énergie utile à la prise en compte des exceptions. On observe alors des programmes structurés comme suit:

```
int f(int x, int y)     {
      if (x && y) {
            … algorithmes de f
```

```
      } // pas de else, pas le temps
}
```

L'absence de "else" dans ce programme donne une bonne idée de ce que peut être une valeur *indéterminée* (certains compilateurs émettront de sévères avertissements puisque f ne retourne pas toujours de valeur). Nous recommandons définitivement la version suivante :

```
int f(int x, int y)     {
      … algorithmes de f
}
```

Et plus encore définitivement la version suivante :

```
int f(int x, int y)     {
      assert(x!=0);
      assert(y!=0);
      … algorithmes de f
}
```

Cette approche garantit aussi de meilleures performances. En effet, lors de l'exécution d'un programme, il est fréquent qu'une condition d'appartenance à un domaine soit toujours vérifiée de manière implicite lors d'un fonctionnement correct à tous égards. Par exemple, un programme qui construit une liste en y ajoutant des éléments ne génère jamais de situation où la liste serait vide. Toutes les fonctions susceptibles d'être appelées sur cette liste qui testeraient sa non nullité le feraient donc inutilement. En reprenant l'exemple f ci dessus, on peut concevoir une situation ou le test systématique d'appartenance au domaine dans la fonction f est une perte sèche pour la performance du programme. Cette situation est très fréquente :

```
{
      …
      int j=1;
      for (i=1;1≤100000;i++)  {
            j = j * f(i,2*i); //appel de f toujours valide
      }
      …     //utilisation de j
}
```

Bien sûr, le gain potentiel en performance est plus net que le gain en concision. On objectera à l'approche présentée ci dessus que si le test de domaine n'est pas réalisé par la fonction elle même, et qu'il doit être fait de manière systématique, on devra le répéter dans les programmes qui l'appellent. La solution est alors de définir pour les besoins

ad hoc des programmes qui risquent systématiquement de rencontrer les conditions d'erreur une fonction supplémentaire, ou une macro, qui masquera le test proprement dit. Par exemple, en C++, on peut définir une fonction qui surcharge le symbole f et qui effectue les tests :

```
inline int f(int x, int y, int& erreur) {
      // le paramètre erreur passé par référence permet de contrôler
      // dans le programme principal les cas d'erreur
      if (x!=0 && y!=0) then {
            erreur = 0;
            return f(x,y)
      } else
            return erreur = 1;
}

main () {
      …
      int erreur = 0;
      int j;
      while ( true )    {
            … lecture des paramères x et y
            j=f(x,y,erreur);
            if (erreur) break;
            … utilisation de j
      }
}
```

Cette approche est confortée par un autre argument. S'il est vrai dans un cas particulier que le test de domaine d'une fonction doive être effectué de façon quasiment systématique, il sera rarement le cas que les erreurs soient prises en compte de manière uniforme par tous les programmes, ni qu'une valeur de retour discriminante de la condition d'erreur puisse être choisie qui convienne dans tous les cas.

Le principe des tests qui a été énoncé au début de ce paragraphe peut être reformulé de la façon suivante :

Principe de séparation des algorithmes et des conditions : Une fonction qui implémente un algorithme ne teste jamais que les conditions de mise en oeuvre de l'algorithme sont vérifiées.

Enfin, les programmes qui prennent abusivement en compte le contrôle des situations hors domaine sont les moins susceptibles d'être réutilisés pour d'autres projets. En effet, la manière effective de traiter une exception est très spécifique d'un projet (tel programmeur sait que dans le projet en cours une fonction ne renverra jamais la valeur 1000, et utilise donc cette valeur comme témoin d'erreur). Alors que l'algorithme proprement dit possède une portée beaucoup plus générale. Les programmes qui ne testent pas leurs conditions d'execution ont donc plus de chances d'être réutilisés que les autres.

1.5. Qu'est ce qu'une exception

Les langages orientés objet modernes (C++, Java, C#,...) permettent la prise en compte d'exceptions. Le lancement d'une exception par un programme permet d'interrompre de façon brutale la pile d'appels en cours, et de transférer le contrôle à une section alternative du programme.

La mise en œuvre des exceptions traduit une évolution au sein des langages de programmation de moyens d'interruption brutale mais contrôlée de l'exécution obtenue pour le langage C avec les fonctions "setjump()" et "longjump()".

Les exceptions permettent de détecter la présence d'une situation ne permettant pas l'exécution normale d'un programme, tout en laissant au programme client la possibilité de survivre, de la manière qu'il souhaite, à la situation, ou de laisser passer l'exception, en délégant ainsi son (absence de) traitement au monde extérieur (un autre programme, ou le système).

Le choix de définir et de mettre en œuvre certaines exceptions est un choix de conception important, normalement complémentaire de la mise en œuvre d'invariants. En particulier, il est déraisonnable de définir une hiérarchie d'exceptions pour la totalité des préconditions et invariants d'un programme.

On met en œuvre des exceptions par exemple dans une programme devant manipuler le contenu d'un fichier. Si l'ouverture du fichier échoue (pour des raisons qui sont extérieures au programme), l'exécution ne peut continuer. Toutefois, cette erreur doit pouvoir être récupérée à plus haut niveau. On lui associe donc une exception. Dans le même cadre, on pourra associer des exceptions à la présence dans le fichier d'éléments non reconnus, ou de données incompatibles avec le traitement.

On ne met pas en œuvre d'exceptions spécifiques pour tester qu'une fonction est appelée avec des arguments situés dans son domaine de défintion. Il faut noter que le langage java s'appuye sur le mécanisme des exceptions pour l'implantation de "assert".

1.6. Capitaliser

L'activité de programmation est couteuse, et autant que possible les organisations doivent mettre en place une forme de recyclage du travail. Les efforts passés doivent être capitalisés et pouvoir resservir. Traduit en termes de programmes, on dira que les programmes écrits doivent être réutilisables.

Il est vrai que de nombreux outils logiciels de base se rencontrent dans tous les projets. Citons par exemple les listes, les tableaux réallouables, les tables hashées, les B-arbres. On observe souvent qu'une équipe de programmeurs ne s'entende pas sur la spécification de ces éléments de base et qu'un projet intègre des versions légèrement différentes des mêmes programmes. A fortiori, on conçoit que de projet en projet les outils conçus antérieurement puissent être perdus. Permettre la réutilisabilité suppose d'avoir une approche "produit" dans la réalisation des outils de base. C'est à dire qu'il faut de manière adaptée spécifier, concevoir, implanter, documenter, livrer et maintenir la bibliothèque qui implante une fonctionnalité d'usage général.

- spécifier : les besoins auxquels répond une gestion de liste par exemple doivent être clairement définis. Pour garantir leur réutilisabilité ils doivent être universels, donc indépendants de toute spécificité applicative. Si de telles spécificités existent cependant, elles sont décrites à part en s'appuyant sur le noyau fondamental. Lorsqu'un projet fait apparaître des usages spécifiques pour un module, il sera souvent possible de définir les variations au schéma fondamental par héritage.

- concevoir : pour garantir la réutilisabilité, la conception doit prévoir une interface de portabilité (indépendance des architectures matérielle et système), et doit obéir aux différents principes généraux, dont celui de l'utilisateur parfait. La conception de l'interface homme machine, et de ses relations avec l'applicatif présente à cet égard un caractère saillant dans l'ensemble de la conception. C'est souvent elle qui conditionnera le niveau de modularité et de réutilisabilité de l'ensemble.

- programmer : les outils de base font l'objet de modules indépendants, groupés dans une ou plusieurs bibliothèques. Ces bibliothèques sont accessibles à l'ensemble des membres de l'équipe, et font l'objet d'une rigoureuse gestion de versions. Les programmes eux mêmes sont protégés des malversations par des contrôles d'invariants dans leur version de développement. Un tel programme ne doit jamais "planter" s'il est mal utilisé, mais doit se comporter comme une boite noire. On ne doit jamais conduire un utilisateur à utiliser un débugger pour naviguer dans des sources inconnus à la recherche de l'origine d'une erreur.

- documenter : de bons documents de spécification et de conception constituent déjà une documentation adéquate. Il faut la compléter par un "manuel de l'utilisateur" qui décrit notamment des cas d'utilisation détaillés et des exemples.

- livrer : un utilitaire doit être disponible aux développeurs sous deux versions. La première, de développement, effectue tous les tests de domaines nécessaires, et garantit de ne jamais "planter" sans dire pourquoi elle plante. Par exemple le déréférencement d'un pointeur suppose que sa valeur ne soit pas égale à zéro[4]. La rencontre par un programme d'un pointeur nul en situation inopportune doit être détectée et provoquer un arrêt de l'exécution avant même que l'erreur physique ne se produise (SEGV - segmentation violation-, ou BUS - bus error - par exemple sous Unix).

- La seconde version de la bibliothèque est utilisée lorsque le programme a été intégralement testé. Elle ne fait plus aucune vérification de domaine. En cas de "plantage", il est alors conseillé de refaire une édition de liens avec la bibliothèque de tests plutôt que de courir après l'erreur via un débugger. Cette dernière possibilité risque fort d'être impossible si l'exécutable livré a été dépouillé ("strippé") de ses symboles, ou a été compilé en optimisé.

Les algorithmes utiles dans les cas les plus courants de gestion de structures de données comme les arbres balancés ou les listes sont tellement connus que de nombreuses versions en existent dans le domaine public. Il est possible de partir d'un des sources disponibles sur le marché plutôt que de zéro. Il est conseillé aux organisations de concevoir et d'implanter leurs propres outils de base. Ainsi, elles possèdent la maîtrise d'un élément qui s'avère central dans leurs projets. Toutefois, il faut encourager l'ensemble des membres des équipes de développement à connaître et à faire vivre les outils qu'elles utilisent. La compétence sur ces programmes ne doit pas rester le fait d'un individu isolé. De même, en amont, il faut que les étapes de conception et de spécification de ces outils à caractère général voient la participation de tous leurs futurs utilisateurs.

Lorsqu'en un point éventuellement avancé du projet un individu identifie une fonctionnalité pouvant raisonnablement être considérée comme d'usage public, et réutilisable, les décisions relatives à sa spécification et à sa conception doivent être prises par l'équipe au complet.

---

[4]On peut aller plus loin et tester si le pointeur est correctement aligné (multiple de 4 en général), et si sa valeur est comprise dans le domaine des adresses valides (utilisation des symboles _etext et _edata en C par exemple).

1.6. Documenter

La documentation des programmes est un art difficile. Il y a trois niveaux fondamentaux de documentation :

- 1 la documentation interne du programme,
- 2 le manuel de référence,
- 3 le manuel d'utilisateur et d'exemples.

1.6.1. La documentation interne

La plus grande difficulté en matière de documentation interne est de trouver la juste limite entre trop en faire ou pas assez. En fait, aucune information nécessaire à la compréhension rapide d'un programme ne devrait manquer a son lecteur.

> On doit considérer que la structure logique d'un programme (les tests et les boucles) parle d'elle même au lecteur lui même programmeur. Une documentation intégrée strictement redondante avec cette logique est inutile.

Ainsi, l'algorithme ne doit pas être expliqué dans le détail. Peut être que quelques lignes d'explication initiales sur les principes généraux suivis seront suffisantes. Par contre :

> Quand une expression évaluée dans un test n'est pas suffisament explicite pour le comprendre, une ligne de commentaire est nécessaire.

Dans le cas ou un test complexe est réalisé, il est parfois utile de reporter ce test dans une fonction à valeur booléenne dont le nom sera explicite de ce qui est testé. Ainsi, le corps du programme aura une expression proche de l'algorithme.

De même :

> Lorsqu'un programme possède un effet de bord non évident, cela doit être documenté.

Observons maintenant que les invariants mentionnés explicitement dans les programmes, et testés, constituent une documentation fondamentale. En effet, leur conformité avec les algorithmes est garantie (justement parce qu'ils sont testés). La pratique du développement avec contrôle d'invariants montre que très souvent, leur présence éclaire considérablement le sens et la fonction des programmes ou ils apparaissent.

> Les invariants constituent le fondement de la documentation interne.

Tous les invariants ne peuvent raisonnablement pas être testés. C'est notamment le cas de certaines postconditions qui soit sont vérifiées de manière évidente à la lecture du programme, soit demanderaient pour être vérifiées de pratiquement dupliquer le programme.

```
Par exemple, imaginons une fonction ajoutant un élément en fin d'une liste
chainée simple. La première action de cette fonction est de parcourir la liste,
pour en atteindre la fin. L'ajout est ensuite immédiat.
Tester que l'élement ajouté est bien situé en queue de liste doit impérativement
procéder à un nouveau parcours. C'est inutile.
```

Dans ce cas, nous dirons que :

> Un invariant qui ne peut, ou ne doit, raisonnablement pas être testé doit être mentionné en commentaire.

Les différents principes que nous avons définis, s'ils sont suivis, permettent de fournir au lecteur futur d'un programme une documentation de qualité, en général très succinte et précise (notamment du fait des invariants), si possible complète et exacte, qui ne donne jamais au programmeur l'impression de fournir des informations redondantes avec le programme lui même, et donc encourage à la maintenir en même temps que les programmes.

1.6.2. Le manuel de référence

Ce document décrit dans le détail exactement ce que fait chaque fonction, sous quelles conditions, et avec quels paramètres. Aucune spécificité non implicite ne doit y manquer, afin de permettre à tout nouveau programmeur ou utilisateur de trouver toutes les informations nécessaires à son activité. Le manuel de référence comporte une partie destinée à l'utilisateur final. Cette partie doit être strictement conforme à ce qui est mentionnée dans le cahier des charges. L'autre partie du manuel de référence peut être obtenue par extraction de parties de programmes lorsque la documentation interne est faite rigoureusement.

> Automatiser l'extraction du manuel de référence est un moyen de conduire à une documentation interne de qualité.

De plus, cette automatisation peut aussi permettre de contrôler plus facilement (sinon automatiquement) la conformité de ce qui est implanté effectivement avec le cahier des charges. Le lanage Java intègre cette fonctionnalité au langage grâce à l'outil Javadoc, qui exploite des commentaires spéciaux dals le source et génère une documentation navigable par un browser html.

1.6.3. Le manuel d'exemples

Ce manuel donne des exemples d'utilisation de l'outil, et en général guide de façon progressive vers la maîtrise de ses différents concepts. C'est donc un cours sur papier, avec fréquemment une implication assez forte de l'utilisateur par le biais de l'ordinateur (écriture de programmes). Ce document n'est pas forcément rédigé par les programmeurs eux mêmes.

Partie 2 : Qualité en programmation: organisation de l'activité

2.1. Compiler et exécuter dès le début de son activité

Tous les aspects du développement peuvent être source de difficultés. Il est important de détecter les défauts du système que l'on réalise aussi rapidement que possible. Les deux outils de diagnostic fondamentaux sont le compilateur et la machine elle même lors de l'exécution du programme. Il faut les utiliser:

- compiler aussi tôt que possible, avec tous les fichiers include des bibliothèques que l'on va utiliser.
- linker avec les bibliothèques que l'on doit utiliser par la suite.
- exécuter le programme de façon à pouvoir le tester.

On ne doit pas programmer en passant par de longues phases au cours desquelles toute compilation serait impossible.

Nous venons de voir ce problème selon le point de vue du programmeur. Les mêmes contraintes existent au niveau du projet. Tous les modules d'un applicatif doivent pouvoir être réunis et testés ensemble très tôt dans le projet et aussi souvent que possible.

2.2. Sauver le temps

Dans un projet informatique l'économie est essentiellement une économie de temps. En général, pour gagner du temps sur l'ensemble d'un projet il convient déjà d'en perdre un peu (en apparence seulement) dans les phases initiales de spécification et de conception en prenant soin de faire que ces phases aboutissent à des documents contractuels *utilisables*.

Ensuite, on constate que la conception choisie peut avoir un effet important sur le temps de développement. Si la contrainte temps est un besoin exprimé de façon claire par le client, cela a pu orienter la conception dans une direction très spécifique. Il est parfois vrai que certains choix techniques permettent de gagner du temps sur l'ensemble d'un projet (implantation préalable d'un langage ad hoc, bootstrapping, développement en pipe line notamment) mais leur effet est rarement mesurable, dans la mesure où l'on n'essaye jamais les voies alternatives. Donc nous allons juste nous efforcer de décrire ici les attitudes individuelles et collectives qui permettent des économies de temps manifestes.

- 1 Ne pas programmer de fonctions inutiles. Il faut implanter tout et seulement ce que la conception a défini.

- 2 Communiquer l'information. Cette communication est essentiellement réalisée par l'utilisation de messageries et la mise en place d'un service d'aide en ligne sur tous les programmes. L'aide en ligne est garantie correcte et mise à jour à toute modification de son objet.

- 3 Réduire le temps d'accès à l'information. Des outils adaptés doivent être développés pour éviter au programmeur de chercher les signatures de fonctions notamment. Cela peut être obtenu soit par des outils dynamiques (recherche / extraction de texte par exemple) soit par des outils statiques : extraction automatique de définitions et de commentaires des fichiers include par exemple.

- 4 Documenter ses programmes. Seules les astuces non triviales doivent être documentées. L'essentiel de la logique d'un programme même inconnu est évident au programmeur averti, pourvu que ce programme mentionne clairement les invariants dont il dépend.

- 5 Protéger ses programmes. Implanter le test des invariants via une compilation conditionnelle. De cette manière la version de test du programme ne plantera jamais et garantit ses sorties. Le gain de temps fourni par cette approche est considérable car elle permet de presque totalement supprimer les recours au débugger.

- 6 Réduire la charge des machines. L'expérience montre que dans des projets importants la progression vers le but s'accompagne d'une montée en charge du système. Les fichiers à compiler sont plus longs, les fichiers inclus sont plus nombreux, les données sur lesquelles travaillent les programmes sont plus volumineuses. A intervalles réguliers il convient de prendre les décisions qui s'imposent pour que la compilation et les tests se fassent en un temps acceptable.

Il y a certainement d'autres attitudes positives en regard du gain (ou de la "non perte") de temps, mais celles ci nous paraissent les plus saillantes.

2.3. Savoir automatiser

L'ensemble de l'activité de développement relève de l'automatisation et un bon programmeur automatise toujours une partie de son activité. Par exemple, la recherche d'information (dans quel fichier se trouve la déclaration de cette fonction?) est facilitée par des programmes systèmes (bat sur PC ou shell sous Unix par exemple) utilisant le programme Grep. De même, des actions répétitives comme la substitution dans n fichiers d'un nom de symbole par un autre peuvent être réalisées par l'utilisation de l'outil .

Bien utiliser les outils disponibles pour gagner du temps et des efforts rébarbatifs est fondamental dans l'activité de développement.

Pourtant, on a souvent vu des programmeurs passer une journée entière à développer un outil à usage unique, et dont l'effet désastreux sur la qualité globale du logiciel (inutile complexité, absence de documentation) fait oublier les maigres bénéfices que l'on en retire par ailleurs sur le moment.

La mise en oeuvre d'un automatisme annexe doit prendre un temps négligeable au regard du problème posé (disons que cela se compte en minutes en général).

A titre d'exemple, le programme shell suivant permet par exemple d'appliquer récursivement une commande sur tous les sous répertoires du répertoire courant. Un tel utilitaire permet de régler ce problème une fois pour toutes et d'éviter de prendre en compte la descente récursive dans les shells que l'on écrit par ailleurs.

```
#!/bin/sh -e
if [ $# -ne 2 ]
then
      echo usage $0 dir command
      echo this program performs command on all files in the directory
      echo and recurses down all subdirs
      exit 1
fi
dir=$1
command=$2
echo -------------- recursive $command in directory $dir
for i in `ls -A $dir`
do
      $command $dir/$i
      if [ -d $dir/$i ]
      then
            $0 $dir/$i $command
      fi
done
```

Un autre exemple est un shell qui permet la création automatisée d'une révision des programmes sur lesquels on travaille. Ce shell réalise la copie des sources dans un répertoire créé automatiquement et procède à leur compilation automatiquement. Il comporte des exemples de calculs et de tests en shell.

```
#!/usr/bin/sh

topvers=`ls .. | grep -v src | sort | tail -1|cut -c2-10`
echo current most recent version of project is v$topvers
majorvers=`echo $topvers|cut  -f1 -d\.`
echo major revision number is $majorvers
minorvers=`echo $topvers|cut  -f2 -d\.`
echo minor revision number is $minorvers
nextminor=`expr $minorvers + 1`
nextmajor=`expr $majorvers + 1`
newdmaj=v$nextmajor.0
newdmin=v$majorvers.$nextminor
echo new minor revision name would be $newdmin
echo new major revision name would be $newdmaj
echo "choose one (default is cancel) [major/minor] : \c"
read rep
case $rep in
      major ) newdir=$newdmaj;;
      minor ) newdir=$newdmin;;
      *) echo exiting ... ; exit 1;;
esac
echo new revision directory to create is ../$newdir
echo "perform ? (default is cancel) [yes] : \c"
read rep
case $rep in
      yes ) echo create ../$newdir; mkdir ../$newdir ;;
      *) echo exiting ... ; exit 1;;
esac
for file in `cat projectfiles`
do
      cp $file ../$newdir
done
cd ../$newdir
make
if [ $? = 0 ]
then
      echo CREATION of new revision $newdir COMPLETED;
else
      echo CREATION of new revision $newdir FAILED;
      cd ..; mv $newdir v$topvers.b.$newdir;
```

```
fi
```

2.4. Organiser l'accès à l'information

Un programmeur aujourd'hui doit gérer des ensembles de programmes qui ajoutés les uns aux autres représentent des dizaines de milliers de lignes de code. Avec des langages modernes comme Ada ou C++, la densité d' informations pertinentes dans les programmes est grande. La volume des données à mémoriser par le programmeur est tel que c'est impossible. A cela, deux réponses sont données: structurer les fichiers de manière logique, et organiser l'accès à l'information.

- structurer : par exemple on décide que chaque classe d'objets (C++) est décrite dans un jeu de fichiers spécifiques, quelle qu'en soit la complexité. Ou encore, on décide de grouper dans un fichier unique les programmes relevant de l'impression, ou de la trace, pour toutes les classes. Dans la pratique, la décision prise n'est pas indifférente, les deux approches ayant leurs avantages. En effet, les programmes de trace auront tous des traits communs faisant que leur présence groupée dans un fichier unique permet d'en programmer rapidement de nouveaux "par l'exemple". D'un autre côté, le groupement par classes reflète la structure logique interne des classes dans les fichiers, notamment l'encapsulation des fonctions.
  Quelle que soit l'approche, on aura dans certains cas besoin d'informations orthogonales.

- accéder : puisque la structure de fichiers ne permet pas simultanément les multiples points de vue qui peuvent être nécessaires pour la disposer de toutes les informations utiles, il est nécessaire de se donner des outils (il s'agit bien de programmes) permettant de retrouver et de lister une information aussi rapidement que possible. Par exemple, on voudra consulter très rapidement le texte d'une méthode de trace implantée pour une classe connue. Ou encore on voudra retrouver rapidement le texte de la déclaration d'une fonction pour en connaître la signature exacte (besoins fréquents).
  Sous Unix, les outils de base "grep", "sed" et "awk" sont précieux pour concevoir des programmes de recherche d'information dans des sources. Ils doivent être utilisés en l'absence de meilleur outil. Les environnements de développement modernes (Jbuilder, Visual, Eclipse) offrent de multiples possibilités.

Principe d'accès à l'information : un environnement de développement doit toujours être complété par des programmes éventuellement spécifiques permettant la recherche optimale des informations utiles.

Lorsque de tels outils sont absents, l'activité du programmeur est progressivement saturée par des taches annexes de recherche d'information qui effondrent la qualité du travail et le rendement. Les possibilités de la mémoire humaine ne permettent pas de basculer trop longtemps sur une tâche auxiliaire. De surcroît, un risque naturel est associé à la recherche manuelle d'information : l'effet encyclopédie buissonnière.

Effet encyclopédie buissonnière : lors d'une recherche trop longue, l'esprit est attiré vers une partie des programmes éloignée de la préoccupation courante qui semble justifier une intervention immédiate. Ainsi le programmeur peut abandonner sa tâche actuelle pour une cascade d'opérations effectuées en parfait désordre et sans planification.

Exemple : un shell qui permet d'extraire le texte d'un fichier autour d'une ligne donnée:

```
#!/bin/sh
# ce fichier decoupe une fenetre dans un fichier a partir d'une position donnee
nlines=${3:-10}
headcount=`expr $2 + $nlines - 1`

#echo $nlines
#echo $headcount
echo EXTRACT ----------------------------------------
echo
head -$headcount  $1 | tail -$nlines
echo
echo ------------------------------------------------
```

Les exemples suivants sont des programmes qui permettent de chercher dans un répertoire donné (ici une application à XWindow) toutes les déclarations d'une classe ou d'un type, etc..., même si l'on ne dispose pas d'interface graphique (ces programmes marchent en mode ligne).

le script searchinc

Ce shell recherche une combinaison de patterns ($1 et $2) séparés par des caractères sans importance dans un fichier.

```
#!/bin/sh
cd /usr/include
echo PARTIAL MATCH
grep -n "$1.*$2" X11/*
grep -n "$1.*$2" Xm/*
echo TOTAL MATCH
grep -n "$1.*$2[  ();{]" X11/*
grep -n "$1.*$2[  ();{]" Xm/*
```

le fichier awk awkfile

Ce programme awk formate la sortie fournie par seachinc, afin que le résultat puisse être exécuté immédiatement (par extract) afin d'en voir plus.

```
BEGIN { FS=":" }
/PARTIAL/ { print $0 "-----------------------"; next }
/TOTAL/ { print $0 "-----------------------" ; next}
{
      print $3;
      print "        extract /usr/include/" $1 " " $2
}
```

le shell de recherche search

```
searchinc $1 $2 | awk -f awkfile
```

l'exemple stype (cherche un nom de type C en fin de déclaration)

```
searchinc  } $1 | awk -f awkfile
```

l'exemple sdefine (cherche une définition de macro)

```
searchinc "#define" $1 | awk -f awkfile
```

l'exemple stype

```
searchinc typedef $1 | awk -f awkfile
```

l'exemple sclass

```
searchinc  class $1 | awk -f awkfile
```

2.5. Ne pas dupliquer

Tout projet logiciel consiste en plusieurs fichiers. Chacun des fichiers groupe des fonctionnalités voisines. Par exemple, un fichier décrit les fonctions d'entrée sortie, un autre les impressions, un troisième des algorithmes de calcul… Il arrive souvent qu'un même fragment de programme doive être répété en diverses parties du logiciel, voire même dans des fichiers distincts. Il va de soi que si ce traitement possède une description naturelle sous la forme d'une fonction, cette dernière figurera dans un fichier indépendant regroupant divers programmes jouant le rôle d'outils[5]. Il existe toutefois des situations où le morceau de code en question ne peut pas être traduit sous la forme d'une fonction ou de quelque autre unité du langage syntaxiquement valide.

Lorsqu'on ne peut factoriser un traitement par le biais d'une construction du langage cible, on doit utiliser une fonction de texte, par le biais d'un processeur de macros. De la sorte, le traitement partagé est défini une seule fois pour tous les fichiers qui y ont recours. Toute modification du programme se fera au niveau de la macro, et sera donc automatiquement répercutée à tous les endroits nécessaires.

L'alternative honorable serait de faire figurer à côté du fragment de programme considéré, et ce dans chaque fichier où il apparaît, un commentaire décrivant les répercussions nécessaires d'une mise à jour. L'expérience montre que ces commentaires ne sont jamais présents, et que des programmes contenant des doublons sont difficiles à gérer même par leur auteur, qui oublie rapidement l'ensemble des occurrences du schéma. Bien souvent, si le schéma doit être modifié

[5]En C++, on l'appelera util.cc ("tools.cc"), ou divers.cc ("misc.cc") par exemple.

ultérieurement, le programme n'atteindra de nouveau un point stable qu'après une série de cycles mise au point / erreur pouvant dériver dans des proportions inacceptables.

L'exemple suivant montre comment utiliser le pré processeur C pour factoriser un traitement C++ qui ne pourrait être implanté par une fonction de façon aussi immédiate. La macro INIT_LOG déclare en effet une variable VAR_LOG qui est initialisée d'une façon standard, puis utilisée localement à des besoins divers. Si le mécanisme d'initialisation change, la macro seule doit être modifiée.

```
Exemple de trois fichiers
// fichier header.h
#define VAR_LOG var_i_1
#define INIT_LOG  \
      obj VAR_LOG ;           \
      VAR_LOG.init("select * from users where name = ",name,"and …");
…

// fichier fic_imprime.cc
#include "header.h"
void imprime (char * name)
{
      INIT_LOG;
      VAR_LOG.print ();
}
…

// fichier fic_reset.cc
#include "header.h"
void imprime (char * name)
{
      INIT_LOG;
      VAR_LOG.reset ();
}
```

En aucun cas une partie de programme ne doit être dupliquée dans plusieurs parties d'un logiciel. Le faire est s'exposer à des difficultés de maintenance considérables.

2.5.1. Une seule exception

S'il est fondamentalement dangereux de dupliquer des programmes se trouvant dans des parties éloignées d'un applicatif (notamment dans des fichiers distincts), il existe toutefois une exception à cette règle simple : le cas où les expressions à répéter se trouvent en séquence dans un programme. Par exemple, il est inutile d'implanter une gestion de listes sous prétexte qu'un programme traite successivement trois cas différents d'un même problème.

Dans ce cas, si l'on veut réduire le risque d'erreurs, et améliorer la lisibilité du programme, on fera souvent appel au pré processeur. Voici un exemple typique d'initialisation d'un tableau dont la taille est connue d'avance:

```
int tab[20];
int i = 0;
tab[i] = 12; i = i + 1;
tab[i++] = 13;
tab[i] = 14;
tab[++i]=15; i++;
…
```

Voici une version utilisant les services du pré processeur :

```
int tab[20];
int i = 0;
#define Store(n) tab[i]=n; i++;
      Store(12);
      Store(13);
      Store(14);
      Store(15);
      Store(16);
#undef Store
```

Autre exemple, avec des comparaisons répétées de chaînes de caractères (cas fréquent) :

```
// version naturelle :
if (!strcmp(str,"-help") {
      // code pour help
} else if (!strcmp(str,"-com1") {
      // code pour com1
} else if (!strcmp(str,"-com2") {
      // code pour com2
} else if (!strcmp(str,"-com3") {
      // code pour com3
} else { // code de l'erreur
}
```

dont on peut sensiblement améliorer la lisibilité par :

```
#define IFSTR(chaine,bloc) if (!strcmp(str,chaine)) { bloc } else
      IFSTR("-help", /*code pour help*/)
      IFSTR("-com1", /*code pour com1*/)
      IFSTR("-com2", /*code pour com2*/)
      IFSTR("-com3", /*code pour com3*/)
```

```
      {/*code de l'erreur*/}
#undef IFSTR
```

2.6. Ne maintenir qu'un seul exemplaire de chaque programme.

La nécessité de porter un logiciel sur différents environnements conduit souvent à modifier la version d'origine d'un programme dans de telles proportions que l'on se trouve réduit à en maintenir séparément deux versions.

La première façon d'éviter ce problème consiste à grouper dans un fichier unique les fonctions qui dépendent de l'environnement matériel et logiciel, et de proposer par ce fichier une interface abstraite dite "de portabilité" d'accès aux fonctionnalités souhaitées. Ainsi, lors d'un portage, seul ce fichier sera adapté, et on pourra tolérer d'en avoir une version par environnement. La raison auxiliaire qui pousse à cette solution est le fait que les algorithmes ne changent pas, eux, et donc ne doivent pas être sensibles au portage.

Il arrive parfois que des algorithmes doivent être adaptés en fonction de spécificités environnementales qui ne peuvent pas être extraites du code, car elles ne se traduisent pas par des fonctions, mais par des fragments non syntaxiquement isolables du langage. Dans ce cas, et quel qu'en soit le coût apparent en termes de lisibilité du programme, il faut utiliser un pré processeur.

La mauvaise lisibilité est compensée par la possibilité de maintenir correctement les mêmes algorithmes sur tous les environnements sans rien oublier. Un cas fréquent est celui des spécificités (et parfois bugs) de compilateurs :

```
void fonct (int *tab)
{
      //calcul de la longueur du tableau
      int i = 0;
      for (; tab[i] != 0 ; i++)
#ifdef SOLAR_V2
      {;}   // ne compile pas correctement sinon
#else
      ;     // compile correctement sur toutes les autres machines
            // et avec tous les autres compilateurs
#endif
      …
}
```

Cet exemple montre que l'on ne peut pas procéder autrement que par compilation conditionnelle pour à la fois:

- 1 permettre la compilation sur toutes les plate formes,
- 2 permettre la mise au point de tous les algorithmes sans jamais oublier de version,
- 3 réduire le nombre des fichiers source utilisés,
- 4 permettre l'utilisation optimale des ressources offertes par chaque environnement,
- 5 se protéger contre l'oubli potentiel des conditions dans lesquelles la variabilité apparaît.

Le dernier point 5 est fondamental, et à ce titre, le programme ci dessus est meilleur que le suivant, car un commentaire présente une moins grande résistance à l'usure du temps (et au despotisme de certains programmeurs) :

```
void fonct (int *tab)
{
      //calcul de la longueur du tableau
      int i = 0;
      for (; tab[i] != 0 ; i++) {;} // {;} nécessaire sur SOLAR_V2
      …
}
```

Par ailleurs, ce dernier programme contredit le point 4, puisque du code inutile est compilé presque partout, pour les beaux yeux de SOLAR_V2 seulement.

2.7. Avoir une démarche produit

Citons rapidement quelques concepts clef à garder en mémoire lors du développement. Avoir une démarche produit est grandement synonyme de "Penser aux autres".

- écrire la documentation interne et externe de ses programmes, même lorsque ce n'est pas demandé
- tester ses programmes dans une gamme raisonnable de conditions d'utilisation. Cette activité peut être automatisée par des programmes réalisant des tests automatiquement. Dans ce cas, le test sera réalisé implicitement par le contrôle des invariants, ce qui implique que les programmes en contiennent.
- assurer la compatibilité ascendante de ses programmes : on ne doit pas prendre la décision de changer la signature ou les caractéristiques d'une fonction. Tout programme ancien qui utilisait une fonction (d'une bibliothèque par exemple) doit pouvoir compiler et s'exécuter sans différence avec la nouvelle version.
- assurer la portabilité : le programmeur doit utiliser, si une telle interface n'est pas définie pour le projet ou l'organisation tout entière, une interface de programmation pour les fonctions système, pour les fonctions graphiques, et une interfac pour les types de données de base (entiers signés de deux ou quatre octets par exemple).
- permettre la réentrance : un programme doit être conçu pour pouvoir fonctionner sur deux (ou plus) jeux indépendants de données dans une même session. Cela interdit formellement l'utilisation anarchique de variables globales. Très souvent aujourd'hui, une programmation à caractère évènementiel permet d'enregistrer dans une structure de données des fonctions appelées automatiquement à certains moments. Ainsi, un programme peut appeler un autre programme qui en cascade appelle le premier, etc. Une telle situation requiert également une parfaite réentrance.
- isoler les algorithmes de leurs interfaces, et les fournir de façon séparée sous forme d'une bibliothèque à usage public.

Partie 3 : Qualité en programmation: quelques aspects techniques

3.1. Distinguer les différents types de paramètres

Toute application informatique requiert un certain nombre de données pour son fonctionnement. On appelle paramètres les données qui ne sont pas constantes pour toutes les versions ou toutes les sessions d'exécution d'un programme. Ces paramètres sont de plusieurs types.

3.1.1. Paramètres machine

Les paramètres machine sont ceux qui sont liés à l'architecture **physique** de la machine. On sait par exemple que telle machine utilise des mots de 32 bits. Si l'on doit faire dépendre des programmes de cette contingence, "Taille de mot" est alors un paramètre machine. Bien que les compilateurs permettent de connaître dynamiquement les tailles des structures (instruction "sizeof" du langage C), il importe souvent pour des raisons de portabilité de se prémunir contre les variations de taille des types fondamentaux (le type entier signé par exemple). Un autre exemple est le nombre de registres internes au processeur. L'écriture d'un ramasse miettes ("garbage collector " en anglais) en dépendra certainement.

Le pré processeur C permet de prendre en compte des symboles définis automatiquement par le compilateur, ou explicitement par la commande de compilation, pour s'adapter aux variations d'architectures physique. Le symbole correspondant s'appelle ARCH sous Unix. Sa valeur lors d'une compilation est connue et peut être prise en compte.

```
#if ARCH == CRAY_XMP    //définition d'un entier de 4 octets
      typedef short int       int32
#else
      typedef int             int32
#endif
```

3.1.2. Paramètres système

Le système d'exploitation pose également des conditions sur l'exécution des programmes. Par exemple, le nombre de fichiers qu'il est possible d'ouvrir simultanément est variable, comme le nombre de processus, ou de threads qu'il est possible d'exécuter à un moment donné.

Un logiciel performant doit s'adapter automatiquement aux ressources disponibles, de façon à en faire profiter l'utilisateur du programme. C'est une mauvaise pratique que de choisir des valeurs minorantes pour les niveaux de ressources disponibles (limiter arbitrairement à 16 le nombre de fichiers ouvrables simultanément par exemple) et de ne pas rendre dynamiques leur prise en compte.

Les paramètres systèmes sont accessibles par un jeu de symboles définis pour le pré processeur afin de permettre des compilations conditionnelles ou la prise en compte de bornes. Ces symboles sont définis dans les fichiers d'inclusion système.

```
const int num_threads = MAX_THREADS; // valeur dépendant du système
```

3.1.3. Paramètres de configuration

Pour un même système d'exploitation, sur une même machine, un logiciel peut s'attendre encore à un certain niveau de variabilité externe. En effet, l'organisation même de la machine, le nombre de disques, les partitions, la structure des répertoires, l'emplacement des fichiers utiles, la disponibilité des entrées sorties et d'autres encore sont variables.

Dans certains cas, celui de programmes simples, ces contingences n'influent que sur la compilation. Il faut parfois adapter le programme chargé de la construction des exécutables pour qu'il puisse exploiter toutes les ressources en fonction de leur emplacement. Par exemple, si l'on doit changer de compilateur ou de bibliothèques d'objets, on ne remplace pas les anciens par les nouveaux. On adapte ou on crée un nouveau makefile pour accéder aux bonnes ressources.

Lorsqu'un programme effectue des opérations d'entrée sortie plus complexes, et notamment ouvre et ferme des fichiers, il doit permettre à l'utilisateur final de configurer lui même le logiciel pour décider de l'extérieur quels répertoires accéder.

Dans ces deux cas, le paramétrage externe se fait par l'utilisation de variables d'environnement". Ces variables sont définies au niveau du système par une instruction spéciale du langage interprété de commande. On peut aussi les déclarer dans des fichiers de commande exécutables. Sous MS DOS, le fichier "autoexec.bat" joue ce rôle, comme tout fichier "*.bat" contenant une séquence valide de commandes DOS. De même sous UNIX, les fichiers de commande shell ".login", ".profile", ".cshrc" (lus par sh, csh, ou ksh principalement) contiennent des définitions de variables d'environnement:

```
#ligne reconnue par csh
setenv PATH .:../bin:/usr/bin
#lignes reconnues par sh
set LD_LIBRARY_PATH=/usr/lib:/usr/lib/X11:/usr/lib/motif
export LD_LIBRARY_PATH
```

Les variables d'environnement qui sont définies lors de l'appel d'un programme lui sont passées automatiquement par la fonction système permettant le lancement ("" en C sous Unix, avec diverses variantes).

Typiquement, le programme une fois actif peut alors consulter les valeurs des variables d'environnement présentes dans un tableau bidimensionnel de chaînes de caractères. Ce tableau est obtenu comme le contenu de la variable appelée "environ".;.

```
char *path = NULL;

for (int i = 0; environ[i][0] != NULL ; i++)    {
      char *varName =   environ[i][0];
      char *varValue =  environ[i][1];
      // code qui dépend de varName et stocke varValue si besoin est
      if (!strcmp (varName, "PATH"))      {
            path = environ[i][1];
      }
}
```

3.1.4. Paramètres d'exécution

Les données qui peuvent varier d'une exécution à une autre et qui ne dépendent ni de la machine, ni du système, ni de la configuration sont les paramètres d'exécution. Il existe plusieurs manières de les fournir à un programme (liste non exhaustive) :

- par l'environnement (comme ci dessus), en définissant des variables

```
>setenv PARAM_DEFAUT_CARRE 3
>carre
9
>...
```

- par ce qu'on appelle la ligne de commande, en ajoutant un ou plusieurs paramètres lors de l'appel du programme, après son nom

```
>carre -operande 4
16
>carre
9
```

Ici, on dispose d'un arsenal équivalent au traitement des variables d'environnement. Un paramètre formel de la fonction principale du programme appelée en C, contiennent le nombre et la liste des paramètres passés lors de l'appel. Voici un exemple de ce que pourrait être un programme C++ pour carre (prenant en compte également l'option du paragraphe ultérieur sur les fichiers) :

```
class ArgvData {
      int operande;
      int interactif;
      char *fichier;
      void parseArgs(const int argc, const char ** argv);
      public:
      ArgvData(const int argc, const char ** argv)   {
            interactif=true;
            operande=0;
            fichier=NULL;
            parseArgs(argc,argv);
      }
};

void Argvdata::parseArgs (const int argc, const char ** argv){
      for (int i = 0; i<argc ; i++) {
            if (!strcmp(argv[argc],"operande") {
                  operande = atoi (argv[i]);
            } else if (!strcmp(argv[argc],"fichier") {
                  fichier = … // lecture fichier de nom argv[i]
            } else if (!strcmp(argv[argc],"interactif")    {
                  interactif = true;
            }
      }
}

main (const int argc, const char ** argv){
      ArgvData *data = new ArgvData(argc, argv);
      if  (data->interactif)  {
            while (!cin.eof()) { // lire et ecrire sur les io standard
                  int val;
                  cin >> val;
                  cout << val*val;
            }
      } else if (!data->fichier){
            cout << data->operande * data->operande << endl;
      } else {/*cas ou on lit dans un fichier*/}
}
```

- par des fichiers textes (y compris l'entrée standard)

```
>carre -interactif
```

```
3
9
5
25
^D
>echo 12 > datacarre
>carre -fichier datacarre
144
>…
```

- et également en environnement multitâches par de la mémoire partagée, par des fichiers d'entrée sortie (sockets tcp/ip, pipes Unix, lignes série et parallèles). Il est difficile ici de donner un exemple simple de programme qui utilise ces notions. Disons simplement que le programme communique avec d'autres programmes à l'aide de dispositifs divers, et qu'une interface de programmation permet de lire et d'écrire sur ces dispositifs, à l'aide de fonctions read et write.

3.1.5. Constantes

Les constantes sont le dernier type de paramètre d'un programme. Ils ne peuvent varier que d'une compilation à une autre. Bien sûr, une constante ne doit jamais être mentionnée en clair dans un programme à l'endroit (à tous les endroits) ou on l'utilise. Sa valeur ne doit être fournie qu'en un seul point : à l'endroit où elle décrit la valeur d'un symbole.
Il existe plusieurs manières de les décrire:

```
//#define         max_fenetres      20          /* cas 1 */
//#define         max_fenetres      ((int)20)   /* cas 2 */
//int             max_fenetres =    20;   // cas 3
//static int      max_fenetres =    20;   // cas 4
const int         max_fenetres =    20;   // cas 5

if (num_fenetres == max_fenetres) { /* code */ }
```

Dans cet exemple, toutes les alternatives commentées conviendraient. Toutefois, le cas 5 est la meilleure solution. En effet, il permet un typage rigoureux, tout en évitant au compilateur de générer une variable.

- 5 permet au compilateur de donner les commentaires les plus précis en cas de mauvaise utilisation de la constante.

- Les cas 1 et 2 évitent de générer de symbole C++, et 2 permet aussi en fait un contrôle de type, mais d'une manière moins limpide que 5.

- 3 génère un symbole externe qui sera référencé dans le code objet associé au fichier ou dans une bibliothèque contenant cet objet. Il est alors accessible à d'autres programmes pour lecture et modification éventuelle, ce qui est formellement contradictoire avec le concept de *constante*.

- 4 réduit un peu ce défaut en limitant la portée du symbole au fichier compilé, mais permet encore la modification de cette valeur pendant l'exécution.

3.2. Gestion des exceptions

Une exception est une situation non prévue par un programme. Par exemple, donnons nous une fonction Ajoute prévue pour ajouter un élément à une liste non vide. Supposons qu'elle reçoive une liste vide comme paramètre d'appel. C'est alors une exception. Ici, plusieurs cas sont possibles : soit la fonction appelante exécute un mauvais algorithme et a engendré un appel invalide à Ajoute, soit le cas d'une liste vide n'a pas été prévu mais doit l'être et donc la fonction Ajoute est mal spécifiée[6].

Dans les deux cas, on ne peut demander à la fonction (ou à son programmeur) de prévoir cette situation exceptionnelle pour éviter l'erreur.

Un algorithme ne doit sous aucun prétexte prévoir le cas de situations exceptionnelles.

On voit que des situations où il est demandé à une fonction de prendre en compte des situations sensiblement hors de son domaine de définition ne peuvent pas être qualifiées d'exceptions. Ne méritent ce titre que les situations dont la prise en compte n'a pas été prévue lors de la spécification du système ou bien lors de la conception des programmes.

En d'autres termes, s'il est convenu que la division de deux nombres flottants doive s'accommoder d'un diviseur nul, l'irruption de ce diviseur nul n'est pas une exception, même si son traitement suit une toute autre route que le traitement habituel, et même si cela lève une condition d'erreur. Donc on peut sans crainte affirmer que :

Un logiciel sans défaut ne rencontre jamais d'exceptions.

Mais il est aussi vrai que :

Un logiciel commence à être utilisé avant d'être totalement dépourvu de défauts.

Puisque des exceptions devront être prises en compte, il faut savoir comment. Les deux grandes époques dans la vie d'un logiciel, à savoir le développement et tests, puis la diffusion et maintenance, ne vont pas demander la même attitude devant le problème des exceptions.

3.2.1. Période de développement et tests

Lors du développement et des tests, une exception demande une correction immédiate. Le meilleur moyen d'obtenir du programmeur[7] une intervention immédiate est de la rendre nécessaire, par une interruption subite du programme lors de la rencontre de l'exception. Lorsque les différents programmes sont implantés conformément à leur spécification et vérifient leurs invariants de domaine de définition par le biais de la macro assert par exemple, cet effet est instantané, puisque assert provoque une interruption d'exécution ("Abort").

[6] un dernier cas non totalement improbable est celui ou un ou plusieurs bits auraient été modifiés dans la mémoire vive par un rayon gamma ou béta.

[7] soi même le plus souvent !!

3.2.2. Période d'exploitation

Lorsqu'un logiciel est utilisé effectivement, le client demande en général une robustesse incompatible avec la brutalité de assert. Même si le programme rencontre une situation inattendue, l'utilisateur souhaite pour le moins pouvoir sauver le fruit de ses efforts récents, et si possible continuer à travailler en continuant comme si de rien n'était. Pas question d'interrompre le programme donc.

Toutefois, pour l'équipe de maintenance, il serait souhaitable de disposer d'informations précises sur les invariants violés pour permettre une correction rapide des défauts. Peut on concilier continuation d'exécution et "logging"[8]?

Permettre la continuation d'un programme lors d'une exception suppose la capacité à le replacer dans des conditions d'exécution saines. Toutefois, puisque les fonctions effectivement implantées *ne gèrent pas* les exceptions (bien qu'elles en *détectent* certaines), cette opération ne peut se faire via une continuation normale d'exécution de la fonction qui rencontre l'erreur.

En C++, comme en Java, les exceptions sont intégrées au niveau du langage. Il existe un mécanisme en C pour y parvenir, au moyen des fonctions "setjump" et "longjump". La première permet de mémoriser un environnement d'exécution, et la seconde d'y revenir brutalement. Un programme doté d'une interface homme machine est souvent bâti sur une boucle sans fin ("main loop"), qui est un bon candidat pour une continuation d'exécution en cas d'erreur.

Il va de soi qu'un tel saut en hauteur s'accompagne de quelques scories (mémoire non libérée, dessins non terminés par exemple) et que le système peut progressivement se trouver dans une situation inextricable du point de vue des données.

Techniquement, l'implantation de ce mécanisme est facile par une définition appropriée de la macro assert, qui écrit le texte de l'assertion violée dans un fichier (assert.log par exemple), et provoque ensuite l'appel à longjump pour retrouver des conditions d'exécution antérieures.

Si le site client est accessible en télémaintenance par un modem, il suffit d'aller lire à intervalles réguliers le fichier assert.log pour effectuer de façon invisible un déboguage très efficace et sans douleur excessive pour le client.

3.3. Gestion de la mémoire

Les compilateurs permettent de réserver deux types d'espace dans la mémoire vive de la machine: la mémoire statique et la mémoire automatique. Un troisième type d'espace de stockage est fourni à la demande par le système tant que c'est possible : la mémoire dynamique.

Ce dernier type d'espace est le plus difficile à gérer. C'est en fait le seul à devoir l'être. Ce paragraphe étudie les problèmes liés à l'allocation de mémoire dynamique, après un rappel des notions utiles concernant les variables globales et les paramètres formels, afin de comprendre les rapports délicats que ces trois concepts possèdent.

- 1 la mémoire statique : cet espace est associé au code du programme, et sous Unix il en est très proche physiquement. Il est réservé par la déclaration d'une variable globale (hors de la portée d'une fonction). Il est parfois utile de limiter la portée de l'identificateur de la variable au fichier qui la déclare. C' est obtenu à l'aide du mot clef "static" en C et C++. Par insertion d'une déclaration de variable statique dans un bloc (le corps d'une fonction le plus souvent), on obtient une variable globale à visibilité réduite au bloc en question.

```
static int globale1=0;  //globale non utilisable d'un autre fichier
```

[8] un fichier de "log" stocke des informations caractéristiques sur les états successifs du système, ou les interventions qu'on y fait (opérations sur la base de données par exemple), ou encore des évènements : une exception est un évènement.

```
char *nom="mon prog";   //globale consultable par un autre fichier

void fonction ()
{
      static int compteurDAppels = 0; //globale visible dans fonction
      compteurDAppels ++;
      cout<< nom << " " << globale1 << " " << compteurDAppels << "\n";
}
//compteurDAppels = 0;  // provoquerait une ERREUR de compilation ici
```

Il faut noter ici que le recours à l'utilisation de variables globales, pratique quand on écrit de petits programmes, est à déconseiller ou même à proscrire dans les projets importants. En effet cela interdit d'appeler les mêmes fonctions pour des usages différents au cours d'une même session d'exécution, et notamment de les appeler récursivement (on dit que l'utilisation de variables globales rend les programmes non réentrants).

On peut toujours se passer de variables globales. La solution consiste alors à déclarer une structure particulière pour stocker toutes les données globales, à en allouer un exemplaire par exemple dans la fonction principale (main) du module, et passer cet objet par référence à toutes les fonctions qui le nécessitent.

```
struct Globales {
      int globale1;
      char *nom;
      int compteurDAppelsFonction;
      Globales():
            globale1(0),
            nom("mon prog"),
            compteurDAppelsFonction(0) {}
};


void fonction (Globales &glob)
{
      glob.compteurDAppelsFonction ++;
}

main ()
{
      Globales glob();  //crée et initialise
      fonction (glob);  //passe la référence
}
```

Une telle pratique est très saine et doit être recommandée sans réserves. Dans l'exemple ci dessus, les variables globales sont en fait allouées sur la pile d'exécution, avec les données automatiques. Comme la fonction "main" est la première

appelée, et la dernière à terminer, on voit que la "durée de vie" de nos globales est égale à celle du programme lui même. C'est bien l'effet cherché.

Dans certains cas, il peut être préférable d'allouer dynamiquement les globales, ce qui permet de travailler au sein du même programme sur plusieurs contextes différents. Voici un exemple :

```
#include "Globales.h"
void fonction (Globales *glob)
{
      glob->compteurDAppelsFonction ++;
}

main ()
{
      Globales glob1 = new Globales;      //crée et initialise
      fonction (glob1);             //passe le pointeur
      Globales glob2 = new Globales;      //crée et initialise
      fonction (glob2);             //passe le pointeur

      delete glob1;
      delete glob2;
}
```

Enfin, il est possible d'utiliser une classe (qualifiée d'"utility", pour grouper les données et fonctions globales:

```
class Globales {
     public:
      int globale1;
      char *nom;
      int compteurDAppelsFonction;
      Globales():globale1(0),nom("mon prog"),compteurDAppelsFonction(0) {}

     void fonction ()   { compteurDAppelsFonction ++; }
};

main ()
{
      Globales glob();  //crée et initialise
      glob.fonction (); //appel
}
```

- 2 la mémoire automatique : l'espace correspondant est alloué sur la pile d'exécution sans intervention particulière, pour stocker les paramètres formels et les variables locales des fonctions. La mémoire automatique est entièrement prise en charge par le programme généré par le compilateur. La mémoire automatique est rendue dès que la fonction qui en a fait la demande termine.

```
void fonction (char *str)
{
      const int longueurMax = 20;
      char tab[longueurMax ]; //espace disponible pendant l'exécution
                  //de la fonction mais plus après
      assert (strlen(str)<longueurMax);
      strcpy (tab,str); //copie de str dans tab
      // … manipulations de tab puis retour
}
```

La durée de vie de la mémoire automatique est limitée à l'exécution de la fonction. On voit toute de suite qu'il est impossible d'utiliser l'adresse d'une zone automatique pour accéder à son contenu après que la fonction qui en avait fait la réservation ait terminé. En particulier, il serait déraisonnable de fournir comme valeur de retour d'une fonction l'adresse d'une de ses variables automatiques (et non la valeur de cette variable, ce qui est bien sûr permis). Exemple :

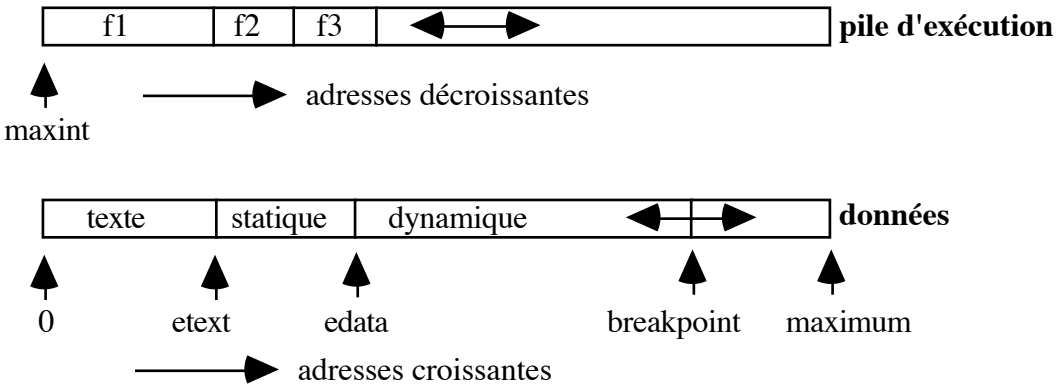
```
int * fonction (int *i)
{
      int k = *i;
      k *= 2;
      return &k;
}

void main ()
{
      int i       = 323;
      int *j      = fonction (&i);  // passage de i autorisé
      cout << *j << endl;           // effets imprévisibles
}
```

Même si le compilateur autorise cette écriture, et même si l'exécution du programme est satisfaisante (on obtient l'affichage de "626"), elle doit être farouchement interdite.

- 3 la mémoire dynamique : c'est l'espace total disponible pour le programme (hors pile d'exécution liée à la mémoire automatique). Cet espace est égal lors du lancement du programme à celui strictement nécessaire au chargement dans la mémoire vive du code programme et des données statiques. Un programme peut ensuite demander explicitement de la mémoire à l'aide de la fonction "sbrk".

Pour récapituler, disons que deux zones de données fondamentales sont accessibles : la pile d'exécution ("stack" en anglais), et la pile des données ("heap"), dont voici une représentation graphique :



Cette représentation n'est que virtuelle, car dans la machine des zones apparemment contiguës[9] pour le programme sont en fait souvent disjointes (du fait de la pagination notamment). Toutefois, cette indirection est transparente pour le programmeur, pour qui les adresses des zones de mémoire qu'il utilise se suivent de façon continue.

- Noter toutefois que sous Unix, les adresses des variables de la pile d'exécution décroissent avec les appels récursifs de fonctions (les entiers correspondant à ces adresses sont en fait négatifs). La comparaison d'une adresse à 0 et aux valeurs de etext, edata et du breakpoint (obtenu par un appel de la fonction brk()) permet ainsi de savoir de quel type d'adresse il s'agit[10].

Pour obtenir l'espace nécessaire au stockage de ses données, un programme (malloc) demande au système le déplacement du "breakpoint", dans les limites permises par la configuration de la machine[11]. Ensuite, l'espace est distribué à la demande aux fonctions par le biais des fonctions malloc, calloc, realloc, et libéré au moyen de free. Cette interface est devenue un standard, et est disponible dans presque tous les environnements. Toutefois, l'implantation effective de ces fonctions diffère d'une version à l'autre pour de nombreuses raisons, notamment :

- la stratégie d'allocation peut par exemple privilégier l'allocation performante des petits blocs, en réservant d'office des espaces importants pour ces zones.

- des contrôles plus ou moins sophistiqués peuvent être effectués pour vérifier que des blocs demandés par "malloc" ne sont pas oubliés, ou bien libérés plusieurs fois.

- un allocateur pourra peut être compacter l'espace automatiquement lors de "realloc", et permettre ainsi (dans de rares cas) de réduire la place demandée par le programme au système (diminution du breakpoint, qui d'habitude ne cesse de croître).

---

[9] On peut le vérifier en comparant les adresses des zones mémoire.

[10] Un programme qui dépendrait de ces conditions risque fort toutefois de nécessiter quelques modifications lors d'un portage sur un système différent.

[11] Cela dépend de la mémoire vive effectivement disponible sur la machine en termes de nombre d'octets, comme de la mémoire virtuelle disponible sur le disque (on parle aussi de zone de "swap").

- un allocateur saura peut être récupérer automatiquement l'espace oublié, agissant ainsi comme un ramasse miettes ("garbage collector").

- un allocateur pourra privilégier la rapidité de délivrance des blocs (temps d'exécution des fonctions "malloc" etc.) au détriment peut être de l'espace perdu.

La mémoire allouée sur la pile d'exécution par le compilateur est dite mémoire automatique. Cette dernière est en effet libérée dès que la fonction qui avait demandé de l'espace pour une variable automatique retourne. La mémoire dynamique quand à elle, une fois allouée, est réservée jusqu'à sa libération explicite par free. Si l'on oublie de procéder à un appel de free, on maintient définitivement le caractère réservé d'un certain espace mémoire, empêchant de ce fait son utilisation pour de nouveaux appels de malloc. Si cet oubli est répété de nombreuses fois, le programme peut progressivement réserver pour n'en rien faire la totalité de l'espace de mémoire vive disponible sur la machine, jusqu'à finalement s'arrêter pour allocation impossible (plus du tout de place). Nous pouvons donc affirmer :

Toute zone de mémoire dynamique qui n'est plus utile doit être libérée.

3.4. Conventions générales pour l'allocation de mémoire

Les programmes requièrent tous d'allouer un certain espace mémoire dynamiquement pour gérer leurs différents états. Par exemple, un éditeur de texte garde une liste complexe d'informations permettant la construction du document final. Comme la structure même du document réalisé par l'utilisateur ne peut être prévue d'avance (et à fortiori sa taille), l'espace nécessaire est alloué par le programme au fil de son utilisation. C'est de la mémoire dynamique.

Certaines fonctions d'un programme vont donc allouer de la mémoire. D'autres vont en libérer. Un besoin fondamental reste cependant à satisfaire : toute zone inutilisée doit être désavouée.

Mais par qui?

Nous devons satisfaire un double objectif de performance espace et temps (éviter de dupliquer les informations) et de qualité (garantir la libération effective de toutes les zones allouées dynamiquement). Observons ici les différentes situations que l'on rencontre habituellement. Considérons le cas d'une fonction A qui appelle une fonction B. A passe à B un paramètre P qui est l'adresse d'une zone de données particulière. B utilise ces informations, les intègre dans une structure de données R et retourne à A une valeur qui est l'adresse de cette structure (un point d'entrée).

```
void *A()
{
      void *P = 0;
      //initialisation de P
      void *R = B(P);
      return R;
}
```

- 1 P est alloué dynamiquement par A, passé à B, utilisé dans R et n'est plus utilisé par A, qui travaille ensuite sur la base de R.

Alors, P n'aurait pas dû être alloué par A, mais directement par B, au besoin avec des données d'initialisation passées en paramètre.

- 2 P est alloué dynamiquement par A, passé à B, utilisé dans R et dans A par la suite.

S'il est entendu que les modifications faites à P après l'appel de B doivent cependant figurer dans R, il est clair que B ne doit pas avoir procédé à une copie de son argument P pour l'intégrer à R. Sinon, il faut se poser la question du bien fondé de l'utilisation de P dans A après l'appel de B, et voir si l'on ne devrait pas se placer dans le cas 1.

- 3 P est alloué automatiquement par A, passé à B, utilisé dans R et n'est plus utilisé par A.

Une copie de P doit être réalisée dans B, pour l'insérer à la structure R. Toutefois, on peut se poser la question de l'utilité de la déclaration de P dans A, et voir si l'on ne devrait pas passer des paramètres appropriés à B pour qu'il crée seul les structures.

- 4 P est alloué automatiquement par A, passé à B, utilisé dans R et dans A par la suite.

Comme B a du procéder à une copie de P pour R (cf. 3 ci dessus), les utilisations postérieures de P n'auront aucun effet sur R.

3.4.1. Principe 1

En aucun cas une fonction ne peut renvoyer l'adresse d'un espace alloué automatiquement.

3.4.2. Principe 2

Lorsqu'une fonction retourne un pointeur, il faut savoir de façon non ambiguë (documentée) si le pointeur désigne une zone allouée par la fonction, ou bien une zone accessible par l'un de ses paramètres, et donc allouée antérieurement.

On ne peut donc avoir de mécanisme mixte : valeur allouée dans certains cas, non allouée dans d'autres.

3.4.3. Principe 3

Lorsqu'une fonction prend un pointeur en paramètre, elle n'effectue jamais de duplication des données.

Plusieurs arguments confortent cette position :

- faisabilité : la copie n'est possible que lorsque on connaît la taille et la structure des données. La fonction appelante possède peut être cette information, qui reste sans utilité pour la fonction appelée.

- économie : l'autre alternative, souvent employée, est celle de la copie systématique des paramètres en entrée. Elle se traduit par des coûts d'exécution temps/espace plus importants, et par le risque de voir proliférer des zones non libérées.

- sécurité : la fonction appelante sait exactement dans quelles conditions la copie est nécessaire, en fonction de ses propres traitements, et peut la réaliser elle même.

Il peut être utile dans certains cas de posséder malgré tout des fonctions faisant une copie de leurs arguments en entrée. Pour distinguer de manière explicite le cas des fonctions réalisant une copie de celles qui n'en font pas, on peut utiliser la convention suivante : surcharger la fonction qui ne procède à aucune copie (et qui prend un pointeur en paramètre) par une fonction de même nom qui prenne une référence constante au paramètre).

```
struct Essai {
      int i;
      int j;
      Essai (int _i, int _j):i(_i), j(_j) {}
};
void fonct (Essai *e)
{
      // pas de copie
      // différents calculs
}

void fonct (const Essai &e)
{
      // copie initiale
      Essai * f = new Essai (e.i,e.j);
      fonct (f);
}

void main ()
{
      Essai *e = new Essai (1,2);
      fonct (e);        //pas de copie
      fonct (*e); // copie
}
```

Noter toutefois que toutes les fonctions prenant des références constantes à des structures ne procèdent pas à une copie.

Lors du développement d'un logiciel, il doit être fait obligation aux développeurs de se conformer aux règles définies. En particulier, il doit être fait mention dans la documentation:

- du fait qu'une fonction duplique certains de ses arguments, le défaut étant raisonnablement qu'elle ne le fasse pas,
- du fait qu'elle alloue elle même sa valeur de retour ou non.

Partie 4 : Qualité en programmation: aspects humains

4.1. Eviter de généraliser et d'abstraire

Un besoin consiste souvent en un cas particulier d'un certain nombre de mécanismes plus généraux. La relative autonomie du concepteur, puis du programmeur peut conduire à implanter non pas un programme qui corresponde strictement au besoin, mais une version arbitrairement plus générale. Les motivations pour le faire sont diverses:

- 1 le goût de résoudre un "problème" difficile,
- 2 le goût d'une certaine esthétique des programmes,
- 3 la croyance que l'on se protège d'avance contre des évolutions futures du besoin client.
- 4 la croyance que le même programme puisse servir dans un autre projet
- 5 l'impression d'améliorer la qualité du logiciel

Tous ces points sont contestables. Pour 1, il est clair que ni le programmeur ni le chef de projet ne doivent chercher à se faire plaisir. La jubilation d'un projet terminé, testé, documenté puis vendu dans les temps dépasse considérablement le plaisir d'un exploit local.

2 est à proscrire dans tous les cas. Seule compte l'implantation des fonctionnalités selon les règles établies par la spécification puis la conception.

3 est toujours faux. En premier lieu disons qu'un projet non terminé dans les délais parce que tel programmeur n'a pu gérer la complexité de son œuvre n'aura de toute façon jamais à évoluer, puisque le client aura disparu. Ensuite, on observe que les demandes d'évolution émises par les clients ne sont jamais celles que l'on attend, et il se peut fort bien qu'un programme inutilement magnifique soit tout simplement abandonné dans la suite.

4 procède du même schéma de pensée que 3. Le programmeur n'est pas le client.

Pour 5, on observe que l'effet inverse se produit. Les programmes soit disant génériques sont peu testés dans leurs parties limites, et des problèmes se rencontrent fréquemment lorsqu'un logiciel qui en contient est livré.

Enfin, il est fréquent de sous estimer la complexité du développement d'une fonctionnalité plus générique que ce qui est nécessaire. Les cas limites sont toujours les plus difficiles à traiter, et au sein d'un projet logiciel conséquent on risque de voir apparaître des problèmes aux limites dans des programmes qui somme toute ne devraient faire que des choses simples. La perte est double, et le risque est grand.

4.2. Savoir déboguer

Les programmeurs font toujours des erreurs. Le niveau d'erreur le plus bas, et le plus impardonnable, est la faute de syntaxe décelée par le compilateur (oubli de point virgule par exemple, commentaire non terminé, chaîne de caractères non terminée, par exemple). Mis a part celà, on observe deux types de programmes erronnés :

- 1 le programmes corrects mais qui implantent un algorithme faux,
- 2 les programmes qui implanteraient un algorithme correct, s'ils ne contenaient des erreurs.

Dans les deux cas, le programme erronné risque fort de violer un invariant majeur. Si cet invariant est testé, le programme sera interrompu au plus tôt, évitant à l'erreur encore inconnue d'aller produire des effts dans des portions du programmes étrangères à l'erreur elle même, situation qui rend plus délicate la découverte du bug. Cet argument devrait encore inciter à l'utilisation systématique de tests d'invariants.

Notons que la situation 1 est souvent plus difficile à corriger que la seconde. En effet, en présence d'un bug, le programmeur remet rarement en cause les principes du programme qu'il a écrit[12].

Plaçons nous dans le cas d'un programmeur qui recherche un bug d'un programe qu'il a lui même écrit, et supposons que l'algorithme est correct.

4.2.1. aspects psychologiques

L'erreur est toujours une faute d'inattention, et en général ce qu'on appelle un "acte manqué" en psychologie. Tout se passe comme si un mécanisme inconscient avait conduit à l'erreur (par exemple une substitution de noms de variables), et bloquait le mécanisme de mémoire qui permettrait de trouver la solution au problème. On en a la preuve au moment ou l'on trouve cette solution en général. Il arrive que la découverte de l'erreur se produise comme par magie, à un moment ou l'on n'y pense presque plus, en feuilletant négligeamment le source. Cela donne des pistes sur les attitudes à avoir en face d'un bug :

- décontraction : la pression que l'on s'impose pour chercher une erreur est plus génératrice de renforcement du blocage qu'autre chose.

- faire un effort calme et patient de mémoire : passer les symboles et les programmes en revue, en répétant mentalement leur rôle précisément, va généralement faire disparaître le blocage.

- demander l'aide d'une personne extérieure : comme elle n'aura pas le blocage, lui expliquer le rôle des éléments du programme fautif et la logique générale du programme suffira souvent à lui permettre de trouver l'erreur. Souvent, le fait d'expliquer permet au programmeur de faire effectivement l'effort calme et patient en question ci dessus, et de trouver l'erreur lui même.

4.2.2. aspects statistiques

Lorsqu'un programme "plante", et que l'on a des indications sur l'endroit dans le programme ou l'exécution s'est arrêtée, cette information est précieuse car dans une très grande majorité des cas, et quelles que soient les apparences, les instructions erronnées sont très proches de l'endroit où le programme s'arrête.

4.2.3. savoir isoler les conditions d'apparition du bug

Dans certains cas, un bug sera mieux compris si l'on série très précisément les conditions de son apparition (sous forme par exemple d'une séquence particulière d'appels de fonctions), en se plaçant non pas das la fonction supposée fautive (au cas ou on la connaisse), mais au niveau de fonctions qui appellent cette dernière. Spécialement dans le cas ou l'on n'est pas soi même le programmeur des éléments incriminés, il est nécessaire de produire une telle séquence "minimale" conduisant à l'erreur, pour permettre à l'auteur un diagnostic rapide et fiable.

---

[12]C'est à ce point vrai que les utilisateurs novices d'ordinateurs incriminent d'abord la machine, puis le compilateur, avant d'admettre que l'erreur peut être de leur fait!

Pour réduire les conditions d'erreur à leur plus simple expression il suffit en général de supprimer des portions du programme de façon itérative tant que l'erreur est maintenue. Il est en général inutile de chercher la cause de l'erreur tant qu'on n'a pas procédé à cette réduction à la séquence minimale.

4.2.4. savoir procéder par différences élémentaires

Il arrive que pour comprendre le mécanisme d'un bug (dans ce cas souvent une erreur dans l'algorithme) il faille procéder par différences élémentaires pour observer le comportement de l'algorithme, après avoir isolé les conditions fautives selon la démarche décrite ci dessus. Il est alors fondamental de rester aussi proche que possible des conditions minimales génératrices de l'erreur : si une modification élémentaire maintient la situation d'erreur, alors elle n'est pas discriminante et doit être annulée avant l'essai d'une nouvelle modification. Cette technique permet de travailler sur la base de conditions stables, et d'acquérir un maximum d'informations lorsqu'une modification élémentaire supprime le bug.

4.3. Etre souple et curieux

Enfin, voici une liste non exhaustive de quelques qualités demandées à l'informaticien.

4.3.1. Autonomie

L'informaticien ne doit pas être une charge pour l'équipe, notamment en termes de demande d'information. Mais il ne doit pas non plus être une charge par la génération de difficultés pouvant peser sur tout le monde. En conséquence, un comportement abusivement renfermé est également néfaste. Tout n'est qu' équilibre! Disons seulement qu'une démarche individuelle d'accès à l'information, associée à des attitudes propres à promouvoir la qualité et le respect des spécifications, permettra souvent de limiter les interactions avec les autres programmeurs à leur partie la plus utile. En d'autres termes, on peut appliquer aux relations entre individus le principe de l'utilisateur parfait! Cest une boutade, mais elle est assez réaliste.

4.3.2. Adaptabilité à tout langage

Il n'y a pas de mauvais langage. Chacun possède des fonctionnalités facilitant certains pans de l'activité du programmeur (en rapprochant la structure du programme de la spécification). Les défauts peuvent souvent être compensés par l'utilisation de techniques qui ne sont pas l'apanage de C et C++. En particulier, les pré processeurs ne sont pas dédiés à un langage particulier, et peuvent être utilisés pour faire du contrôle d'invariant soumis à compilation conditionnelle (cf la macro assert) dans tout langage.

4.3.3. Apprendre à utiliser tous les outils

De très nombreux outils, en complément des pré processeurs, permettent d'éviter des opérations fastidieuses sur les programmes, et parfois de doter les programmes de compléments utiles. Il faut savoir en lire les documentations, et tenter de les utiliser chaque fois que possible, de façon à en atteindre une maîtise croissante, et ne jamais se contenter de ce qu'on sait.

Partie 5 : Eléments de gestion de configuration

Le programmeur doit impérativement gérer, même en étant isolé, les versions successives de ses sources. Pour cela, il dispose sous Unix d'outils éprouvés comme **sccs** et **rcs**, voire **get admin** et **delta** en cas d'absence des deux premiers. Sur un pc, un outil appelé « pvcs » apporte des fonctionnalités comparables. Voir l'annexe Unix de ce cours pour des informations plus complètes. Ces outils permettent de conserver économiquement (par différences successives) les versions successives d'un fichier texte, et surtout de contrôler (donc interdire) les accès simultanés pour modification à un même fichier par des utilisateurs différents. Par ailleurs, il est nécessaire de contrôler les versions successives d'un produit, en étant capable de revenir à la version exacte de tous les sources impliqués dans la construction de chaque version du produit. Le projet « config » décrit précisément une structure complète de configuration logicielle pour un produit, utilisable pour les projets, et assortie d'outils de génération automatique d'environnements.

Partie 6 : La ligne de compilation avec intégration des tests

Les tests unitaires valident l'implantation de programmes élémentaires comme la description d'une classe. Il est utile d'intégrer dans la ligne de compilation (habituellement gérée par l'outil « make » sous Unix comme sur pc) des tests unitaires qui sont capables de détecter toute régression d'un programme due à une modification erronée aussi tôt qu'elle se produit. Pour le garantir, il suffit de compiler et exécuter les programmes de tests qui dépendent de la bibliothèque à chaque invocation de « make », et de contrôler leur bonne exécution. Noter que par défaut, une commande unix qui ne retourne pas 0 interromp « make. Voir le projet « tools » pour plus de détails. Voici une version abrégée du makefile de tools, à méditer :

```
SCCSID=@(#)makefile     1.11  16 Nov 1995

CCC=CC
#FLAGS= -g -Wp,-nosplice
FLAGS= -g
INC=
LIBS=
GET=sget #defined in the project # using make -g (autoget)

SC=SCCS/s.

.SUFFIXES:.cc .do .rs .tc .ts .tx .i
#.cc source c plus plus
#.do objet c++ compile sans l'option NDEBUG
#.rs un fichier reference de session d'execution pour tests
#.tc un fichier source c++ de tests
#.tx un executable de tests
#.ts une session de tests
#.i un fichier passe par cpp et indente (si possible)
```

```
INCLUDES=gendefs.h trace.h integ.h
MOREINCS=named.h typed.h display.h
TESTS=list.ts assoc.ts quark.ts array.ts narray.ts iterator.ts types.ts
memory.ts

all: $(INCLUDES) libtools.d.a $(TESTS) libtools.a $(MOREINCS)

############################################ libraries

libtools.d.a:list.do assoc.do quark.do array.do
      ar rv $@ $?
#     ranlib $@

libtools.a:list.o assoc.o quark.o array.o
      ar rv $@ $?
#     ranlib $@

############################################ suffix rules
# building objects with assertion checking
.cc.do:
      $(CCC) -c $(FLAGS) $(INC) $*.cc
      mv $*.o $*.do

## additional dependencies for .cc.do
list.do: $(INCLUDES) list.h list.cc
assoc.do: $(INCLUDES) list.h assoc.h assoc.cc
quark.do: $(INCLUDES) list.h assoc.h quark.h quark.cc
array.do: $(INCLUDES) array.h array.cc

############################################
# building test exes (.tx) from archived test sources (.tc)
.tc.tx:
      @echo "building test program $*.tx _____"
      @cp $*.tc $*.t.cc
      @chmod +w $*.t.cc
      @purify -exit-status -windows=no -viewfile=$*.pv -logfile=$*.log $(CCC)
$(FLAGS) -DTESTING $*.t.cc -o $*.tx libtools.d.a
      @rm -f $*.t.cc $*.t.o
      @echo done
list.tx:list.do list.tc
assoc.tx:list.do assoc.do assoc.tc
quark.tx:list.do assoc.do quark.do quark.tc
array.tx:array.tc array.h
```

```
narray.tx:narray.tc narray.h
iterator.tx:iterator.tc iterator.h
types.tx:types.tc types.h
memory.tx:memory.tc memory.h

###############################################
# comparing the execution of a test program to a stored sample (.rs)
.tx.ts:
	@echo differences between $*.tx and $*.rs
	@$*.tx | tee $*.tmp
	@diff $*.rs $*.tmp > tmp
	@mv tmp $*.ts
	@rm $*.pv $*.log
	@rm -f $*.tmp
	@echo no differences

## additional dependencies for tx.ts.
list.ts:list.rs
assoc.ts:assoc.rs
quark.ts:quark.rs
array.ts:array.rs array.tx
narray.ts:narray.rs narray.tx
iterator.ts:iterator.rs iterator.tx
memory.ts:memory.rs memory.tx
types.ts:types.rs types.tx
	@echo differences between $*.tx and $*.rs
	@-types.tx | tee types.tmp
	@diff types.rs types.tmp > tmp
	@mv tmp $*.ts
	@rm $*.pv $*.log
	@rm -f $*.tmp
	@echo no differences

###############################################
.cc.o:
	$(CCC) -c $(FLAGS)-DNDEBUG  $(INC) $*.cc -o $*.o

## additional dependencies for .cc.o
list.o:$(INCLUDES) list.h list.cc
assoc.o:$(INCLUDES) list.h assoc.h assoc.cc
quark.o:$(INCLUDES) list.h assoc.h quark.h quark.cc
array.o:$(INCLUDES) array.h array.cc
```

```
############################################ other
.cc:
	$(CCC) $(FLAGS) $(INC) $*.cc
	$(CCC) $(FLAGS) $*.o -o $* $(LIBS)

.cc.ix:
	$(CCC) -E $(FLAGS) -DTEST $(INC) $*.cc > $*.i
	# not available # indent $*.i
	$(CCC) $(FLAGS) $*.i -o $*.ix $(LIBS)

########################################### cleaning dir
clean:
	@ touch ,dummyfile
	rm -f *.o *.a *.do ,* *.ts a.out core *.tx *.old
	- sccs get makefile sget
	chmod +x sget
```

Annexe : quelques commandes Unix très utiles

make

Cette commande permet de déclencher des traitements de façon sélective quand des dates de fichiers sont modifiées. Les règles de déclenchement sont décrites dans un fichier. Make utilise par défaut un fichier programme appelé « makefile » ou « Makefile », s'il existe.

```
make -f monfichier.mak  une_cible // exploite un fichier spécifique
```

Les règles du makefile ont un format particulier, et reposent sur l'utilisation de caractères invisibles (tabulation notamment), ce qui demande un peu de soin, au début. Si make est invoqué sans mentionner un nom de cible particulier, la première règle du fichier est considérée, et bien sûr toutes celles nécessaires en cascade.

Il est traditionnel que cette première règle porte le nom de cible « all » sans que ce nom ait une importance particulière.

```
cible: dépendance_1 dépendance_2 ... dépendance_n
```

```
[tab] commande shell
[tab] commande shell
[tab] commande shell

dépendance_1: ...
      ...
```

La suite de commandes de cible est déclenchée dès lors que :

- aucun fichier de nom « cible » ne se trouve dans le répertoire courant, ou bien
- un des fichiers du répertoire courant appelé dépendance_i est plus récent que cible

Si il existe des règles dont les noms de cible correspondent aux dépendances, ces règles sont envisagées avant l'exécution de la règle de niveau supérieur.

Si une règle ne produit pas par les commandes associées un fichier dont le nom corresponde à la cible, elle sera déclenchée systématiquement à chaque exécution de make pour cette cible.

Cet outil puissant est utilisé par les programmeurs pour compiler leurs exécutables : un programme dépend de bibliothèques, qui dépendent de fichiers .o qui dépendent de fichiers source, qui dépendent de fichiers .h par exemple, et également par le système pour garder cohérents des ensembles de fichiers suite à une modification de l'un d'entre eux par l'ingénieur système.

```
pgm: a.o b.o
      cc a.o b.o -o pgm
a.o: a.h a.c
      cc -c a.c
b.o: a.h b.h b.c
      cc -c b.c
```

Si l'on modifie a.h, les trois règles ci dessus déclenchent en cascade (dans l'ordre 2, 3, 1). Si b.h est touché seules 3 et 1 déclenchent. 1 déclenche dans les deux cas car 2 et 3 ont pour effet de produire des fichiers nouveaux appelés a.o et b.o, dont 1 dépend.

On dispose avec make de variables prédéfinies, dont les plus utiles sont

- $* : le basename de la cible (privée de son suffixe)
- $@ : le nom complet de la cible, dans la partie commande
- $< : la dépendance plus récente que la cible, pour une règle générique, dans la partie commande
- $? : la liste des dépendances plus récente que la cible

et également :

- $@@ : le nom complet de la cible, dans la partie dépendances

- $% : le nom de la cible dans une archive

On a aussi des règles génériques (celle ci dit comment compiler par défaut tout .c en .o)

```
.c.o:
      CC -c $*.c -I../include
```

ou encore

```
.c.o:
      CC -c $< -I../include
```

On peut générer des bibliothèques sans faire de calculs inutiles (la règle ci dessous n'ajoute à lib.a que les seuls fichiers qui viennent d'être recompilés

```
lib.a: fic1.o fic2.o fic3.o ....
      ar rv $@ $?
```

Make possède de nombreuses autres fonctionnalités, dont une partie non portable. L'outil existe également sur PC avec les compilateurs traditionnels C++.

SCCS

Cette commande permet de contrôler les versions successives de documents « texte ». Elle sert également de système d'archivage, puisque les différentes versions successives d'un même document texte sont accessibles par leur numéro de version, stockées dans un format économique (par différences -ou deltas- successifs).

Pour utiliser sccs, il faut créer un répertoire appelé SCCS dans le répertoire courant.

sccs create fic

crée un fichier archive appelé s.fic dans SCCS

sccs edit fic

copie la dernière version de fic en mode rw (dans le but de le modifier) dans le répertoire courant, à condition qu'il ne soit pas déjà édité, et crée un fichier de lock dans SCCS appelé p.fic. L'existence de ce fichier interdit à un autre utilisateur de prendre ce m^me fichier en édition.

sccs delta fic

ajoute l'état actuel de fic dans le répertroire courant à SCCS/s.fic comme dernière version. Supprime fic de . ou bien cahnge son mode en READONLY, et supprime SCCS/p.fic. Au passage,delta demande à l'utilisateur de fournir des commentaires, accessibles plus atrd par sccs prt.

sccs get fic

copie la dernière version de fic en mode r (impossible de le modifier) dans le répertoire courant, à condition qu'il ne soit pas déjà édité par soi même.

sccs info

donne la liste de tous les fichiers actuellement edites

sccs prs fic

affiche des informations sur les versions successives de fic.

sccs diffs fic

calcule les différences entre le fichier présent dans le répertoire courant, et la dernière version archivée en delta.

sccs unedit fic, sccs unget fic

annule une prise en édition, ou supprime le fichier, avec vérification.

L'outil sccs reconnait dans les fichiers des séquences de caractères prédéfinies, dont la plus utilisée est « %W% %G% ». Lorsque le fichier n'est pas édité, i.e. obtenu par « sccs get », ce schéma est substitué par le nom du fichier et la date et la version de ce fichier. Il est ainsi possible de faire figurer dans le produit d'une compilation les noms et version et dates des fichiers utilisés pour produire la version. La séquence @(#) est quand à elle reconnue par la commande Unix **what**, qui permet d'extraire ces informations, même d'un exécutable (pour lequel l'utilisation de grep ne donne pas de bons résultats). Donc on écrit souvent « @(#) %W% %G% » en commentaire d'un fichier utilisé directement (par exemple un source shell), ou comme valeur d'une variable de type chaine qui figurera dans l'exécutable généré par compilation.

Sur certains systèmes où sccs n'existe pas, un système comparable appelé **rcs** est parfois disponible, et sinon, les commandes système « get », « admin » et « delta » permettent d'atteindre le même type de fonctionnalité.

diff

Diff fic1 fic2 calcule les différences entre deux fichiers. Des options permettent de contrôler le calcul des différences (taille de plages de raccord), l'affichage des résultats, et même de générer automatiquement un script permettant de changer fic1 en fic2 si c'est possible, afin de générer un patch automatique (il suffit d'envoyer le patch plutôt que la nouvelle version du fichier).

find

Cette commande permet de rechercher des fichiers à partir d'un noeud du système de fichiers selon des combinaisons variées de critères (sauf le contenu : il faut utiliser « grep »), et éventuellement d'appliquer des commandes automatiquement sur les fichiers trouvés (option -exec)

```
find /usr/bin -name fic1 -print
find / -name « *sh* » ! -user root -print
find . -name « *.* » -exec rm -f {} \;
find / \( -name core -o -name tmp \) -exec rm -f {} \;
```

file

Cette commande fournit des informations (limitées) sur le type d'un fichier (exécutable, texte, source shell par exmemple)

sed

Cette commande permet de filtrer un fichier ligne par ligne (sed = stream editor). On peut remplacer ainsi du texte par un autre texte de façon assez riche.

```
sed -e « s/toto/une belle variable/g » fic1 > fic2
sed -f sedfile fic1 > fic2
```

et voici fichier *sedfile* correspondant:

```
s/bozo\([0-9]*\)king/c'est le numero \1/
s/[ \t][ \t]*/ /g
```

(\1 est une variable texte : la première zone parenthèsée, et la deuxième ligne substitue toute répétition de deux ou plus espace ou tabulation par un seul espace, avec répétition sur toute la ligne)

Dans un fichier sed on peut donc faire suivre plusieurs commandes de substitutions qui sont appliquées en cascade sur la même ligne : la deuxième opère sur le résultat de la première. Si une ligne ne contient pas le schéma de substitution elle est laissée inchangée. Il est possible avec de concaténer des lignes, mais c'est assez difficile à réaliser et d'expérience peu portable (quoique cela doive s'améliorer).

awk

Cette commande permet d'extraire des colonnes de fichiers, **et de faire des calculs sur ces colonnes**. La notion de colonne est souple, et repose sur un séparateur que l'on peut paramétrer (par défaut l'espace). awk possède deux niveaux de séparateurs. On peut passer des variables à awk, et utiliser les valeurs de ces variables dans le programme awk utilisé.

Pour extraire simplement des colonnes, on utilise plutôt la commande cut ci dessous.

cut

Permet d'extraire une information d'une ligne de texte (un peu comme awk, mais sans la complexité du langage awk)

```
> cut -f2 -d" "
1 2 3 4
2
4 5 6 7
5
```

ar

Cette commande est l'archiveur Unix. Il est utilisé principalement pour construire les bibliothèques, mais il peut servir à grouper toute sorte de fichiers, texte ou non. Les options sont :

- v (verbose)
- t (listing)
- r (replace)
- a (add)
- x (extract)
- d (delete).

```
ar rv archive liste de fichiers
```

Dans le cas des archives de fichiers objet, une commanade appelée ranlib permet de créer un dictionnaire des symboles de l'archive, qui détecte les doublons de fichiers et de symboles le cas échéant. Dans le cas de très petits fichiers, ar peut être avantageuses sur tar, qui aligne les blocs et perd de la place.

tar

Cette commande permet de combiner des ensembles de fichiers en un seul. Le fichier cible peut être soit un fichier disque authentique, soit un device (streamer notamment), sur lequel le fichier est écrit séquentiellement. Les options sont v pour verbose et f pour indiquer le nom de l'archive (sinon c'est /dev/tape par défaut,ou la valeur de la variable $TAPE):

- r pour append à la fin de l'archive
- u pour update : mise à la fin de l'archive si le fichier n'y est pas déjà
- x pour extract
- t pour listing
- c pour create
- p pour conserver aux fichiers leur mode d'origine, indépendamment de l'umask courant
- L pour suivre les liens symoboliques, h pour le contraire
- w pour un fonctionnement interactif fichier par fichier
-

Tar permet de transporter une arborescence de fichiers là ou un seul fichier peut être transmis à la fois : cas d'une bande magnétique de type streamer, ou cas d'une commande ftp par exemple.

```
tar xvf /dev/rmt0                    #pour lire sur cette bande et extraire les fichiers
tar tvf archive_lo        #pour consulter le contenu du fichier tar archive_lo
tar cvfh archive_lo ./*  # pour faire une archive du contenu de mon répertoire courant, en suivant les liens
```

Il est en général, comme dans l'exemple ci dessus, de compier les fichiers dans l'archive en utilisant un adressage relatif. Avec un adressage absolu (tar cvf /usr/home par exemple), on peut avoir des difficultés à extraire le contenu de l'archive si un répertoire a disparu ou a été renommé.

Tar est utile à l'ingénieur système pour préparer des arborescences de fichiers qui doivent être reproduites plusieurs fois. Dans ce cas, une archive contient une arborescence relative, qui peut être extraite n'importe où.

Certaines implantations de tar offrent un accès fichier par fichier (insertion, suppression, extraction rapide) pour les archives qui sont sur disque. C'est évidemment impossible sur une bande streamer.

compress / uncompress

La commande compress appliquée à un fichier dont le suffixe n'est pas .Z le comprime en générant un .Z. Le fichier de départ est alors supprimé automatiquement. Appliquée à un .Z, uncompress produit sa décompression.

L'algorithme utilisé est celui de Lempel Ziv, avec effacement locaux de la table des symboles si c'est utile.

gzip / gunzip

Ces outils du domaine public sont installés sur certaines machines, et proposent des fonctionnalités d'archivage avec compression sur des fichiers disque, en conservant les arborescences et la possibilité d'accéder aux fichiers de façon sélective (contrairement à tar). Une fonctionnalité comparable est approchée avec une combinaison de ar et de compress/uncompress, sans les informations de répertoires, sauf à archiver des archives de façon récursive.