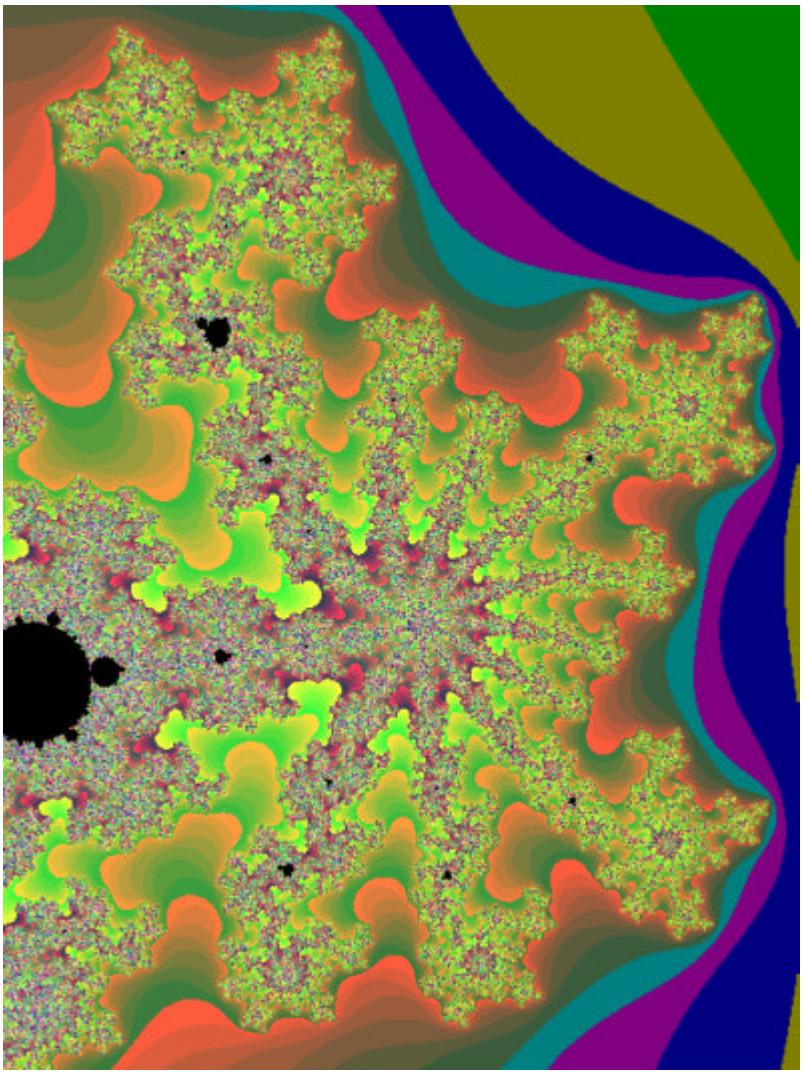
CHAPITRE 10

Fonctions



Une fonction est une opération définie par l'utilisateur. En général, on définit une fonction par un nom plutôt que par un opérateur. Les opérandes d'une fonction, appelés ses arguments, sont spécifiés par l'intermédiaire d'une liste fermée par des parenthèses et dont les éléments sont séparés par des virgules. Une fonction a également une valeur de retour d'un certain type: si aucune valeur n'est retournée par la fonction, cette valeur est dite de type void. Pour des raisons historiques (compatibilité avec C), ne pas spécifier de type pour une fonction est équivalent à la spécifier de type `int` (et non pas void !).

La fonction suivante calcule le PGCD de deux nombres entiers (`int gcd(int, int)`).

```
#include<iostream.h>
int       gcd(int v1, int v2)

     {
     int       tmp;
     while (v2)
          {
          temp = v2;
          v2 = v1 % v2;
          v1 = temp;
          }
     return v1;
     }

main(char*, int)

     {
     cout<<"Le plus grand dénominateur commun de 100 et 30 est "<<
          gcd(100,30)<<endl;
     }
```

Une fonction est évaluée chaque fois que l'opérateur d'appel "()" est appliqué au nom de la fonction. La définition d'une fonction, accompagnée de la liste des types d'arguments passés à la fonction, sert également de déclaration. Mais il n'est pas possible de définir plus d'une fois la même fonction dans un programme. Une fonction est définie par son nom et la liste de types d'arguments passés à la fonction, mais pas par le type retourné (la raison n'en est pas claire).

```
void      Foo(int, int);          // fonction Foo
void      Foo(float, int);        // Autre fonction Foo : les paramètres
                                  // ont des types différents, il s'agit
                                  // donc d'une autre fonction que la
                                  // précédente
int       Foo(float, int);        // Faux ! même définition que la pré-
                                  // cédente. Le type de retour ne fait
                                  // pas partie de la définition pro-
                                  // prement dite.
```

Généralement, une fonction est définie dans un fichier séparé de ceux où elle va être utilisée. Dans les fichiers où elle sera utilisée, cette fonction doit être déclarée séparément. Ceci se fait à l'aide d'un **prototype** :

```
int gcd(int, int);
```

Notons que seuls les types des arguments sont définis, non les noms des arguments. Il est néanmoins possible de donner des noms à ces arguments, et cette possibilité devrait être utilisée à des fins de documentation. Il est possible de faire apparaître un prototype plusieurs fois dans un même fichier source sans problèmes.

Certains compilateurs (Metrowerks Code Warrior, par exemple) interprètent la norme ANSI de manière très stricte, et demandent l'établissement d'un prototype pour toute fonction exportable, alors que d'autres ne requièrent pas obligatoirement cette définition. La norme, interprétée de manière stricte, tend à rendre mandatoire la définition d'un fichier en-tête (.h) pour tout module de programmation définissant des fonctions réutilisables. Cette habitude ne pouvant être qu'encouragée, l'interprétation stricte de la norme est donc une excellente chose...

10.1 *Paramètres*

10.1.1 *Syntaxe*

En C K&R, le passage de paramètres obéit à une syntaxe requérant la définition du type des paramètres passés entre la ligne d'en-tête et l'accolade ouvrante du corps de la fonction :

```
int  uneFonction(param1, param2)
int       param1;   /* type du paramètre 1 */
float     param2;   /* type du paramètre 2 */
     {
          /* Corps de la fonction */
     }
```

La norme ANSI permet la définition des types des paramètres sous une autre forme, plus appréciée par les habitués de PASCAL, par exemple :

```
int  uneFonction(int param1, float param2)

     {
          /* Corps de la fonction */
     }
```

Cette forme remplace de plus en plus l'ancienne forme dans les programmes récents. L'ancienne notation est toutefois encore fréquemment rencontrée dans les bibliothèques de code existantes, très nombreuses.

En C++, seule la nouvelle forme syntactique est tolérée[1], tant il est vrai que C++ s'appuie sur ANSI C en le consolidant. C++ offre des possibilités de contrôle supplémentaire quant à l'utilisation des paramètres par la fonction :

```
int uneFonction(int param1, float param2)

     {
          // param2 pas utilisé !
     return 2 * param1;
     }
```

produit un warning de la part du compilateur C++ :

```
warning : param2 defined but not used
```

ce qui peut servir à détecter certaines omissions. Si en revanche, cet "oubli" est volontaire, et que l'on n'a déclaré param2 que dans le but de satisfaire un interface spécifié dans un autre cadre, il reste possible de dire au compilateur que l'on ne veut pas utiliser param2, de la manière suivante :

1. A l'exception de portions de code explicitement déclarées comme étant de nature C. (Voir *Fonctions C*, page147.)

```
int uneFonction(int param1, float )
                  // param2 n'est pas défini, seul son
                  // type est déclaré, pour définir correctement
                  // la pile. Le compilateur ne produit aucun
                  // warning.

     {
     return 2 * param1;
     }
```

10.1.2 *Passage par valeur*

Les arguments à une fonction sont en principe toujours passés par valeur. Avant l'appel de la fonction, les arguments sont copiés dans la pile, et ce sont ces copies qui seront manipulées par la procédure, sans que ces manipulations se réflètent sur les valeurs originales des arguments.

En C K&R et en C ANSI, il n'existe, du fait de ce passage par valeur, qu'une seule façon pour une fonction de modifier la valeur d'un paramètre de manière permanente, et c'est de passer un pointeur sur le paramètre à modifier.

```
void      uneFonction(int    *param)

     {
          // Elever le parametre au carre :
     *param = *param * *param;        // Aimez-vous les pointeurs ?
                                      // moi non plus...
     }

void      main()

     {
     int       x = 3;
     cout<<"Avant appel de uneFonction x = "<<x<<endl;
     uneFonction(&x);                 // Calcul de l'adresse de x
     cout<<"Après appel de uneFonction x = "<<x<<endl;
     }
```

Exécution du programme:

```
Avant appel de uneFonction x = 3
Après appel de uneFonction x = 9
```

10.1.3 *Passage par référence*

Le passage d'arguments par référence est une nouveauté de C++, et fait exception à la règle de passage par valeur. Il correspond au passage de paramètres par référence en PAS-

CAL, bien connu des habitués de ce langage :

```
FUNCTION passByRef(VAR refVar : INTEGER) : INTEGER;
              (* Pass by reference example *)
```

La déclaration correspondante, en C++, serait :

```
int           passByRef(int& refVar)
              // Pass by reference example
```

Noter à nouveau la syntaxe de la référence, qui peut “embrouiller” les idées. Attention à ne pas confondre l’opération de calculer une adresse d’une variable avec la notation correspondant à une référence.

Quand faut-il utiliser le passage par référence? Chaque fois que l'on désire que les modifications apportées à un argument soient réflétés dans le code appelant la procédure. Une autre utilisation est pour optimiser le passage de gros objets à des procédures, ce qui évite des copies lourdes dans la pile. Dans ce dernier cas, s'assurer néanmoins qu'il est sûr de passer une référence plutôt que la copie. L'implémentateur peut signaler qu'il est sûr d'agir ainsi en insérant la clause `const` de manière judicieuse... Au fait, où placeriez-vous cette clause dans l'exemple çi-dessous ?

```
typedef struct big_arr { int  poubelle[10000]; } bigTruc;

int           max(bigTruc& machin)
     {
     // Trouver le plus grand entier de machin.poubelle
     }
```

10.1.4 *Paramètres de type tableau*

Passer des tableaux à une procédure nécessite de connaître la manière dont C et C++ implémentent les tableaux. Il n'y a jamais de copie effectuée pour un tableau. En lieu et place, on passe à la procédure un pointeur sur l'élément zéro du tableau. La taille du tableau n'est pas transmise, même si on la spécifie dans le prototype de la fonction. Du point de vue de la procédure, les trois déclarations çi-dessous sont identiques :

```
void      functionWithArrayParameter( int* );
void      functionWithArrayParameter( int[] );
void      functionWithArrayParameter( int[ 10 ] );
```

Cette implémentation implique deux choses pour le programmeur :

- Une modification à un élément d'un tableau est répercutée vers le tableau original. Si il s'avère nécessaire de conserver un tableau inchangé, la copie devra être effectuée par l'appelant, et non par la fonction appelée, qui s'attend à pouvoir modifier un paramètre qu'on lui passe.

- La fonction qui manipule le tableau n'a aucune connaissance de la dimension du tableau. Si cette connaissance est nécessaire, il faudra la passer explicitement. Une exception à cette règle est constituée par les tableaux de caractères (chaînes de caractères, strings). Par convention, une chaîne de caractères en C++ est terminée par un caractère nul ('\0'): la longueur de la chaîne n'a donc pas besoin d'être passée explicitement. Cette convention n'est respectée que dans le cadre de procédures C: lors de l'importation de textes provenant d'autres langages, il peut y avoir quelques surprises.

10.1.5 *Paramètres de type fonction*

On peut passer, comme en PASCAL, une fonction comme paramètre. Pour passer la fonction, on passera un ***pointeur*** sur cette fonction en paramètre.

```
void f1(int i)
     {
     cout<<"f1"<<endl;
     }

void f2(void(*f)(int))  // Pointeur sur la fonction

     {
     cout<<"f2 ...";
     f(4);
     }

int     main()

     {
     f2(f1);            // Appel de f2 avec f1 comme paramètre
     return 0;
     }
```

10.2 *Valeur de retour d'une fonction*

10.2.1 *Différences entre C et C++*

Dans le langage C K&R, une fonction retourne toujours une valeur. Si on ne spécifie pas de valeur, alors le type par défaut sera `int`. Les deux déclarations çi-dessous sont équivalentes:

```
uneFonction(float unParametre)
int       uneFonction(float unParametre)
```

Cette règle de valeur de retour par défaut a été conservée en C++ pour des raisons de compatibilité avec les masses de code C existantes. Mais combiné aux contrôles plus stricts qu'effectue le compilateur C++, cette caractéristique cause parfois des messages d'erreurs ou des avertissements pouvant surprendre :

```
eleverAuCarre(int unParametre)
     {
     cout<<"le carre de "<<unParametre<<" vaut "<<
          unParametre*unParametre<<endl;
     }
```

```
warning : function does not return any value (int expected)
```

Pour eliminer cet avertissement, on peut :

- retourner explicitement une valeur, par exemple en modifiant le code çi-dessus de la manière suivante :

```
eleverAuCarre(int unParametre)
     {
     cout<<"le carre de "<<unParametre<<" vaut "<<
          unParametre*unParametre<<endl;
     return(unParametre*unParametre);
     }
```

- ou, si on ne veut pas retourner de valeur, déclarer explicitement que cette fonction n'en retourne pas. Ceci peut se faire de la manière suivante :

```
void      eleverAuCarre(int unParametre)
     {
     cout<<"le carre de "<<unParametre<<" vaut "<<
          unParametre*unParametre<<endl;
     }
```

le mot reservé `void` signifie littéralement "**rien**".

Lorsqu'une fonction retourne une valeur, certains compilateurs C++ s'attendent à ce que cette valeur soit également utilisée; il se peut donc que le code suivant provoque un avertissement :

```
int       eleverAuCarre(int unParametre)
     {
     cout<<"le carre de "<<unParametre<<" vaut "<<
          unParametre*unParametre<<endl;
          // retourner une valeur
     return(unParametre*unParametre);
     }

void      main()

     {
     int       uneValeur = 10;
     eleverAuCarre(uneValeur);
     }
```

warning : result of eleverAuCarre not used.

La morale de ces quelques exemples est qu'il vaut mieux effectuer des déclarations correctes, et oublier certaines habitudes parfois un peu laxistes du langage C. Déclarer void ce qui l'est effectivement, et utiliser les valeurs retournées par les fonctions (après tout, si une fonction retourne une valeur, c'est sans doute pour une raison ou une autre, n'est-ce-pas ?).

10.2.2 *Utilisation de return*

`return` peut être utilisé n'importe où dans une fonction, et éventuellement plus d'une fois. Cette instruction met fin à l'exécution de la fonction en cours, et peut éventuellement retourner une valeur (`return(unParametre*unParametre);`). Si la fonction a été déclarée comme devant retourner une valeur, l'utilisation de return tout seul (sans mention de la valeur à retourner) sera signalée comme une erreur. A l'inverse, une fonction de type void comprenant une instruction `return(uneValeur)` sera également signalée comme erreur.

Dans certains cas, une erreur potentielle peut ne pas être signalée par certains compilateurs. Ainsi le code suivant sera-t-il signalé comme erroné par certains compilateurs, et comme douteux (*warning*) par certains autres :

```
int  uneFonction(char* unParametre)

     {
     if (unParametre != NULL)
          return (strlen(unParametre));
     }
```

Si `unParametre` est un pointeur NULL, la fonction ne retourne pas de valeur, d'où la protestation du compilateur.

Les parenthèses encadrant l'expression après return ne sont pas obligatoires, mais correspondent plutôt à une habitude. Certains compilateurs n'acceptent pourtant pas que l'on utilise des parenthèses sans expression à l'intérieur, comme dans :

```
return ();
```

10.3 *Variables locales*

Il est possible, comme dans les langages similaires, de déclarer des variables à l'intérieur d'une fonction. Ce sont des variables appelées **locales**. La durée de vie de ces variables n'excède pas la durée de vie de la fonction. A chaque appel de la fonction, les valeurs précédentes de ces variables sont donc perdues.

En C++, et en C ANSI, cette notion de variable locale s'étend à l'intérieur d'un bloc d'instructions. On peut définir un identificateur dans le contexte d'un bloc d'instructions, et sa durée de vie (ainsi que sa visibilité, bien sûr) se limitera à ce bloc d'instructions.

```
void      uneProcedure(int& unParametre)

     {
     int  uneVariableLocale;
          // uneVariableLocale est definie uniquement
          // dans le cadre de uneProcedure
     uneVariableLocale = unParametre;
     while (uneVariableLocale--)
          {
          int  localAuWhile = uneVariableLocale;
               // localAuWhile est definie uniquement
               // dans le cadre de l'instruction while
          while (localAuWhile--) unParametre *= uneVariableLocale;
          }
     }
```

Il est néanmoins possible de définir une variable dont la durée de vie excède celle d'un bloc d'exécution, voire celle d'une fonction, en la faisant précéder du mot reservé `static`.

```
int  aProc()

     {
     static    int  aStaticValue = 1;
          // aStaticValue n'est initialisé qu'une fois,
          // mais garde sa valeur lors des exécutions successives
     return aStaticValue++;
     }

void      main()

     {
     for (int i = 0; i < 5; i++) cout<<aProc()<<" "; cout<<endl;
     }
```

produit pour résultat :

```
1 2 3 4 5
```

10.4 *Fonctions inline*

Soit la fonction suivante :

```
int       min( int v1, int v2 )
     {    return (v1 <= v2 ? v1 : v2);}
```

Cette fonction retourne la plus petite des deux valeurs v1 et v2. Pourquoi l'implémenter comme une fonction, puisque l'appel de la fonction est presqu'aussi compliqué que le corps de la fonction, mais que l'appel a pour inconvénient de nécessiter un passage de paramètres par le stack ? Les avantages de la fonction sont nombreux :

- min(a, b) se lit plus facilement, sans doute, que (v1 <= v2 ? v1 : v2);
- Il est plus facile de modifier une ligne que 300, en cas d'erreur
- La sémantique est uniforme
- D'éventuelles erreurs de type sont détectées au moment de la compilation.
- La fonction est réutilisable

Néanmoins, l'implémentation au travers d'une fonction est tout de même moins efficient qu'un codage en ligne :

```
int a = min( v1, v2 );
int a = v1 <= v2 ? v1 : v2;
```

La deuxième écriture est nettement plus rapide: la première nécessite la copie de deux arguments dans la pile, le sauvetage des registres, et le saut à une sous-routine avec le sauvetage de contexte que cela implique. Les fonctions dites inline résolvent ce problème, en effectuant une expansion en ligne au moment de la compilation :

```
inline intmin( int v1, int v2 )
     {    return (v1 <= v2 ? v1 : v2);}

int a = min( v1, v2 ); // équivalent à int a = v1 <= v2 ? v1 : v2;
```

Le mot-clé inline indique au compilateur que l'utilisateur souhaite voir le code de la fonction réécrit à chaque invocation, plutôt que de générer un appel. Notons bien qu'il s'agit d'un souhait! La fonction suivante ne peut pas être étendue inline :

```
inline    int       RecursiveGCD(int v1, int v2)

     {
     if (v2 == 0) return v1;
     return RecursiveGCD(v2, v1%v2);
     }
```

Une fonction récursive ne peut pas être étendue inline, à part sa première invocation. Une fonction de 1200 lignes, par exemple, ne sera vraisemblablement pas non plus étendue inline. La directive inline est à considérer comme une indication pour le compilateur, similai-

rement à l'indication `register`. Elle ne signifie en aucune façon que le code sera effectivement regénéré à chaque appel de fonction, mais que le compilateur est prié de générer un code inline pour autant que cela soit judicieux et possible.

Il y a plusieurs inconvénients aux fonctions inline. Tout d'abord, on ne peut pas les déboguer avec un débogueur (Comment mettre un breakpoint sur du code qui n'existe pas ?). Il est impossible de calculer l'adresse d'une fonction inline, pour les mêmes raisons. Si la logique du programme requiert de passer l'adresse d'une fonction inline à une autre composante logicielle, le compilateur devra générer une instance de la fonction, bien qu'elle ait été déclarée inline.

En principe, on évite d'utiliser la directive `inline` à toutes les sauces. Une méthode utilisée fréquemment est de ne pas définir de fonctions `inline` (sauf les fonctions triviales) dans un premier temps, et de déclarer des fonctions choisies comme `inline` lors de la phase d'optimisation du programme, lorsque la logique du programme est stable. D'autre part, lorsqu'un compilateur ne parvient pas, ou refuse, d'implémenter une fonction comme `inline`, il émet généralement un avertissement. Cet avertissement devrait être considéré comme une invitation expresse à ne pas déclarer cette fonction comme inline, et de l'implémenter comme une fonction normale. L'implémentation non-inline que le compilateur générerait par lui-même peut ne pas correspondre à ce que le programmeur a considéré comme implicite, et présenter des effets de bord désagréables.

Il faut se souvenir qu'un programme passe en moyenne 80% du temps d'exécution dans 20% du code. Optimiser (par l'utilisation de `inline` ou autres) dans les 80% de code non significatifs n'apporte virtuellement rien, sinon un risque plus élevé d'avoir des ennuis. Il est préférable de se donner la peine d'identifier tout d'abord les 20% de code critique, et de se pencher sur l'optimisation de ces 20%.

10.5 *Fonctions surchargées*

En C, ou en PASCAL, une fonction est entièrement identifiée par son nom (identificateur). De ce fait, il est impossible de la surcharger. Si je désire écrire une fonction min pour des entiers, je pourrai la déclarer de la façon suivante :

```
inline    int       min( int v1, int v2 );
```

De manière similaire, en PASCAL (rappelons que la clause inline n'est pas définie en PASCAL standard, bien qu'elle soit implémentée dans diverses moutures de compilateurs PASCAL) :

```
FUNCTION      min(INTEGER v1, INTEGER v2) : INTEGER;
```

Déclarer la fonction min pour des valeurs réelles pourrait théoriquement se faire de la même manière:

```
inline    float     min( float v1, float v2 );
FUNCTION      min(REAL r1, REAL r2) : REAL;
```

Mais si la fonction min a déjà été définie pour des entiers, comme çi-dessus, le compilateur réclamera, disant que l'on définit plusieurs fois le même identificateur (multiple definition).

Il en va de même en PASCAL: redéfinir cette fonction pour des valeurs en virgule flottante nous aurait obligé à définir une fonction minFloat de la manière suivante:

```
FUNCTION      minFloat(REAL r1, REAL r2) : REAL;
```

En C conventionnel, on aurait eu recours au même artifice, et on pourrait aussi écrire, une fonction similaire en C++:

```
inline    float     minFloat( float r1, float r2 );
```

De manière regrettable, on a défini deux fonctions, portant des identificateurs différents, mais qui réalisent en fait la même chose. Du point de vue de la documentation, ce n'est pas très favorable. Et nous pourrions avoir le cas où il est nécessaire d'implémenter également minDouble, minChar, minShort, minComplex, et que sais-je encore. Cette pléthore d'identificateurs ne peut que nuire à la lisibilité du programme : du point de vue du lecteur, il suffit de savoir que la fonction min retourne la quantité la plus petite de deux quantités qui lui sont fournies comme argument.

C++ permet de redéfinir min avec d'autres arguments. Le segment de code suivant est parfaitement licite en C++ :

```
inline    int       min( int v1, int v2 )
```

```
     {    return (v1 <= v2 ? v1 : v2);}

inline    float     min( float r1, float r2 )
     {    return (r1 <= r2 ? r1 : r2);}

int       main(int argc, char *argv[])

     {
     int       a = 10, b = 2;
     floatr1 = 1.0, r2 = 1.4;
     cout<<"Minimum, int function : "<<min(a, b)<<endl;
     cout<<"Minimum, float function : "<<min(r1, r2)<<endl;
     return 0;
     }
```

Comme déjà mentionné, en PASCAL ou en C, une fonction est entièrement définie par son identificateur. En PASCAL en particulier, du moins pour les implémentations qui supportent des unités de compilation séparée, il suffit de définir une fonction externe pour pouvoir violer les règles de contrôle du passage de paramètres défini par PASCAL, car la fonction externe est en-dehors du contrôle du compilateur (Notons que certaines approches de PASCAL ont retenu, pour résoudre ce problème, la notion de module chère à MODULA-2; ces approches ont néanmoins l'inconvénient de souvent nécessiter un éditeur de liens spécifique).

C++ ajoute à l'identificateur la notion de signature. Une signature comprend non seulement l'identificateur de la fonction, mais également l'information permettant de connaître le type des arguments passés. Les deux déclarations suivantes, en C++, sont légales, et sont correctement résolues par le compilateur :

```
int       min( int v1, int v2 );

float     min( float r1, float r2 );
```

La signature se compose de l'identificateur, et des types des arguments passés, mais pas du type de la fonction lui-même. Ainsi :

```
int       functionA(int, int);
float     functionA(int, int);
```

ont la même signature, et vont provoquer une erreur soit à la compilation, soit à l'édition de liens.

La surcharge de fonctions introduit la notion d'ambiguité liée à la conversion de types automatique de C++. Dans l'exemple suivant, il est à priori malaisé de savoir ce qui va se passer :

```
extern    void      print( int  pp );
extern    void      print( unsigned int  pp );
extern    void      print( char *c );
```

```
print(10);                  // print int
print("Hello world");       // print *c
print('A');                 // ?
```

Bien qu'aucune fonction print(char c) soit définie, le segment de code çi-dessus compile correctement, et tourne sans problèmes. Le résultat serait l'impression des valeurs suivantes :

10

Hello world

33

L'appel d'une fonction surchargée peut conduire à trois résultats possibles :

1. Une identification univoque. L'appel correspond univoquement à une définition de la fonction surchargée.
2. Pas d'identification. Le compilateur ne trouve pas de définition de fonction qui corresponde à l'appel.
3. Une ambiguité. Le compilateur trouve plus d'une définition de fonction pouvant correspondre à l'appel.

Cette situation provient de la possibilité de conversion implicite de types par le compilateur. Si le compilateur ne trouve pas de correspondance exacte, il cherche à déterminer, par conversion implicite, une instance de la fonction qui conduise à l'identification; s'il trouve une identification, il l'utilisera sans plus en avertir le programmeur. S'il trouve plus d'une instance, ou s'il ne trouve aucune instance possible, il générera un message d'erreur.

La stratégie suivie par le compilateur pour essayer de trouver une fonction correspondante à un appel est la suivante :

1.Rechercher tout d'abord une correspondance exacte.

```
extern   void     f( int );
extern   void     f( char* );

f(0);    // f ( int ), 0 est un int
```

2.Rechercher une correspondance par promotion de types.

```
extern   void     f( int );
extern   void     f( char* );

f('A');  // f ( int ), A est un char, et un char peut être
                  // promu en int.
```

3.Rechercher une correspondance en appliquant une conversion standard.

```
typedef  X;
extern   void     f( X& );
extern   void     f( char* );

f(0);         // f ( char* ), 0 peut être converti en un pointeur
              // sur char* (NULL pointer)
```

4. Rechercher une correspondance en appliquant des conversions définies par l'utilisateur. Cette stratégie n'est disponible qu'avec la construction class, que nous verrons plus loin dans ce cours ("Classes", page167).

```
class     Byte
     {
     public :
          operator int();
               // Définition d'une conversion de Byte en entier
          ....
     };

Byte      bt;
extern    void      f( int );
extern    void      f( char* );

f(bt);         // f ( int ), car la classe Byte a défini une conversion
               // de byte en int.
```

De surcroît, l'utilisateur peut toujours forcer une correspondance en recourant à la force brute : il peut imposer une conversion de type, ou un *casting*.

En résumé, il faut se méfier des possibilités offertes par la surcharge de fonctions, ou d'opérateurs (ce que nous avons discuté pour des fonctions ici vaut aussi bien pour les opérateurs, bien sûr). La surcharge (*overloading*) doit être appliquée avec parcimonie, et seulement dans les cas où cette solution s'impose d'elle-même, et ajoute quelque chose à la lisibilité du programme. Dans tous les autres cas, on risque non seulement de diminuer la lisibilité du programme, mais encore, en cas d'usage réellement abusif de la surcharge, de se trouver confronté avec des problèmes obscurs, difficiles à localiser.

En tous cas, l'utilisation de castings (conversions) explicites, bien qu'il alourdisse l'écriture, permet de résoudre d'éventuelles ambiguités pour le lecteur. Il peut être très avantageux d'utiliser, même abusivement, des conversions explicites, afin d'éviter de se faire piéger par un appel du compilateur à une fonction que l'on n'avait pas prévue dans ce cas.

La notion de signature apporte à C++ une autre caractéristique importante: le contrôle des types des paramètres au travers de différents modules de compilation. Dans le cas où une fonction située dans un autre module de compilation (un autre fichier) est appelée avec des paramètres incorrects, le programme appelant compilera sans doute correctement (pour peu que le protoype forcément déclaré dans le programme soit incorrect également), mais lors de l'édition de liens, la signature de la fonction appelée ne correspondra pas à la signature définie effectivement par la fonction, et l'édition de liens échouera avec une erreur du type "Unresolved external".

10.6 *Fonctions C*

On peut définir en C++ des fonctions C. La syntaxe est la suivante :

```
extern   C
     {
     // Définition des fonctions C appelées
     }
```

Cette directive permet l'utilisation des librairies existantes pour le langage C. Lors de l'utilisation d'un interface utilisateur comme MOTIF, par exemple, les fichiers en-tête de la librairie MOTIF incluent cette directive à l'usage du compilateur C++. Généralement, on rencontre la séquence suivante dans le fichier en-tête :

```
#ifdef   cplusplus
extern   C
     {
#endif
/*
....
Déclarations C
....
*/
#ifdef   cplusplus
     }
#endif
```

Sous UNIX, la variable d'environnement cplusplus (ou _cplusplus) est définie par le compilateur C++. D'autres environnements (Symantec C/C++, Visual C++) disposent d'outils similaires pour distinguer entre du code C et du code C++. La directive "extern C" implique que les déclarations qui vont suivre ne se conforment plus aux conventions de C++, et demandent au compilateur de ne pas leur appliquer les tests normalement applicables aux déclarations C++.

Il existe un autre cas particulier où la directive extern C est indispensable. Lorsque l'on veut appeler du code C++ depuis un fragment de programme écrit en C, on va normalement se heurter au mécanisme de signature de C++. Il n'est pas possible de déclarer un identificateur C++ dans un morceau de code C, car le compilateur ne va considérer que l'identificateur, alors que le compilateur C++ a généré une signature, qui en plus de l'identificateur, comprend des informations sur les types des arguments passés. Tant le code C++ que le code C compileront correctement, mais à l'édition de liens, l'identificateur C++ déclaré comme externe dans le code C sera déclaré non résolu.

Ce problème ne peut être résolu que si le rédacteur de la procédure C++ a prévu son utilisation par du code C. Par l'utilisation de la directive extern C qualifiant sa procédure, il va empêcher le compilateur C++ de convertir l'identificateur en une signature, et son problème s'en trouvera résolu. Ce problème se pose fréquemment en conjonction avec des outils gé-

nérant du code C, comme par exemple les générateurs d'interface. Ces outils font des appels au code utilisateur (callbacks) que le programmeur doit implémenter lui-même.

L'utilisation de la directive extern C empêche la génération de signature par le compilateur : la fonction est dès lors entièrement définie par son identificateur. En conséquence, la surcharge n’est plus possible.

10.7 *La directive* `asm`

Beaucoup de compilateurs acceptent la directive de compilation `asm`, qui permet d'inclure du code assembleur dans le code C++. La directive `asm` est supportée par la référence de USL Cfront 3.5, mais n'est pas encore supportée par la norme ANSI.

Soit une fonction faisant la somme de deux entiers passés comme arguments, voici une manière possible de l'implémenter :

```
int       anAssemblyFunction(int aParam, int anotherParam)

     {
     int  retVal;
     asm
          {
          push r0;
          ld   r0, aParam;
          add  r0, anotherParam;
          ld   retVal, r0;
          pop  r0;
          }
     return retVal;
     }
```

Noter la syntaxe de asm, qui englobe les instructions d'assembleur dans un bloc d'instructions, et la possibilité d'accès à des variables C++ de manière confortable.

Le compilateur C de Microsoft, Visual C++, supporte la directive `__asm { }` (underline - underline asm { }). Certains compilateurs autorisent un argument après le mot-clé asm pour préciser éventuellement le type de CPU à utiliser comme cible, ainsi, on pourrait rencontrer :

```
asm  68020     { };          // Cible = Motorola 68020, défaut 68000
asm  386       { };          // Cible = 80386, défaut 8086
```

Il est en tous cas fortement conseillé de limiter l'utilisation de code assembleur inline à des fonctions parfaitement délimitées. Le fait de pouvoir conserver l'interface de la fonction en C++ permet de s'abstraire des problèmes de passage de paramètres de C++ à l'assembleur.

10.8 *Visibilité des identificateurs*

Nous avons vu qu'il était possible de redéfinir un identificateur pour autant que la signature des deux identificateurs soit différente. Il existe des cas où l'on peut redéfinir un identificateur même si sa signature est identique: il s'agit de tous les cas où la discrimination peut s'effectuer à l'aide du contexte. Il y a trois types de contextes :

- le contexte fichier. Il s'agit de la portion de code qui n'est pas comprise dans une définition de fonction ou de classe. Ce contexte inclut les contextes locaux et de classe.
- le contexte local. C'est le code contenu à l'intérieur d'une fonction, généralement. Plus exactement, il s'agit du code contenu à l'intérieur d'un bloc { } (Voir *Variables locales*, page140.). Ces blocs peuvent bien sûr être imbriqués. La liste d'arguments passés à une procédure n'est pas considérée comme faisant partie du contexte local à la procédure.
- le contexte de classe. Chaque classe définit un contexte distinct, et des fonctions membre sont considérées comme faisant partie du contexte de la classe.

Il est possible de contrôler la visibilité d'un identificateur au moyen des mécanismes suivants :

L' opérateur de contexte (::) permet à tout moment de spécifier le contexte global, ou un contexte donné, pour un identificateur.

Dans l'exemple suivant :

```
int       max = 10000;

void someProc(int max)

     {
     if (max > ::max)
          // Test si le max paramètre est plus grand que le max global
          {
          ...
          }
     ...
     }
```

on compare la valeur du paramètre, qui est au niveau local de la procédure, à la valeur définie globalement. L'opérateur :: lève l'ambiguité. Il en va de même dans l'exemple suivant, où on a redéfini localement un identificateur global :

```
int       someGlobalInt = 4;

void aProc()

     {
     int  someGlobalInt = 2;// Cache la variable globale
     cout<<"Access to global and local instances : "<<
          someGlobalInt + ::someGlobalInt<<end;
     }
```

```
Access to global and local instances : 6
```

Cette manière de faire est déconseillée, car peu lisible. Il vaut mieux essayer de définir des identificateurs ne conduisant à aucun risque d'ambiguité.

Les variables externes permettent à deux unités de compilation d'utiliser une même variable, sans qu'elle doive être définie à deux endroits.

Un identificateur externe est précédé du mot extern suivi de l'identificateur. Ce dernier peut être également un prototype de fonction.

```
extern    int       someVar;// someVar est défini ailleurs
extern    void aProc( int, int );// aProc est défini ailleurs
extern    int       anotherVar = 512;
                                  // anotherVar est défini dans le
                                  // module courant !
```

Le sens du mot réservé extern est un peu différent de celui que l'on trouve dans diverses implémentations de PASCAL supportant les modules de compilation séparés. Dans les implémentations de PASCAL, extern signifie que l'identificateur est effectivement défini dans un autre module de compilation. En C et en C++, ceci signifie que l'identificateur est visible à l'extérieur du module de compilation.

Les variables globales statiques permettent de définir des variables au niveau global (fichier) sans qu'elles soient visibles de l'extérieur de l'unité de compilation.

Des identificateurs `const` et `inline` sont par défaut static (ce qui est différent du langage C). D'une manière générale, il est préférable de déclarer statique tout identificateur n'ayant pas de raison d'être déclaré global. On rappellera en effet qu'à part pour les cas cités çi-dessus, un identificateur global est par défaut, extern.

Les variables locales ne sont en principe définies que temporairement, lorsque le contexte dans lequel elles sont définies se trouve activé (par exemple par un appel de procédure).

La place mémoire qu'elles occupent se trouve libérée dès qu'elles sortent du contexte courant. Il est indispensable de garder ce fait en mémoire lorsque l'on utilise des variables locales. L'exemple qui suit conduira tôt ou tard à une erreur :

```
char      *someString;
char      *badFunction()

     {
     char localString[100];
     // fill up the local String
     return localString;
     }

main()    { someString = badFunction(); ...}
```

Inutile, je pense d'expliquer ce qui ne va pas dans ce code?...

Une variable locale peut redéfinir une variable globale. La variable locale est alors ca-

chée par l'instance locale :

```
int       someGlobalInt;

void aProc()

     {
     int  someGlobalInt;// Cache la variable globale
     ...
     }
```

10.9 *Paramètres avec valeurs par défaut*

C++ introduit la possibilité de déclarer des paramètres avec une valeur par défaut, comme dans l'exemple çi-dessous :

```
double    exp(double x, int y = 0)
     {
     if (y < 0) y = 0;
     double r = 1.0,
           z = x;
     while (y > 0)
          {
          while (!(y % 2))
               {
               y /= 2;
               z *=;
               }
          y--;
          r = x * z;
          }
     return r;
     };

void main()
     {
     cout<<exp(2.0, 2)<<endl;// == 4.0
     cout<<exp(2.0)<<endl;// == 2.0**0 == 1
     }
```

Si le programmeur omet la valeur de y dans l'appel de la procédure exp(), le compilateur C++ va automatiquement compléter le paramètre manquant par la valeur indiquée dans la déclaration de la procédure comme valeur par défaut. Une limitation cependant à cette utilisation : seuls les paramètres de droite dans la liste peuvent avoir des valeurs par défaut. Ainsi, la déclaration ci-dessous est refusée par le compilateur :

```
double    exp(double x = 0.0, int y);
```

Mais la suivante, par contre est légale :

```
double    exp(double x = 0.0, int y = 0)
```

Les concepteurs de C++ ont renoncé à l'introduction dans le langage de la possibilité de signaler des paramètres manquants, ainsi que cela se fait dans de nombreux langages de commande (*ITU-TS MML Man-Machine Language Specification*, par exemple), et qui aurait donné la possibilité, dans le premier cas, d'appeler la fonction avec
`exp(, 2);`

ce qui, il faut bien l'avouer, n'ajoute rien à la clarté déjà discutable de C.

10.10 *Remarques additionnelles*

10.10.1 *Récursivité*

Les fonctions C peuvent être rendues récursives. Une fonction peut s'appeler elle-même.

```
long      factorial(long n)

     {
     if (n <= 1) return 1;
     return n*factorial(n - 1);
     }
```

Dans le cas de fonctions mutuellement récursives, il faut déclarer l'identificateur de la fonction avant de l'implémenter, afin de permettre au compilateur de connaître ce dernier :

```
int       f(int);
int       g(int i)
     {
     ...
     return f(i);
     }

int       f(int k)
     {
     ...
     return g(k);
     }
```

10.10.2 *Fonctions imbriquées*

Il n'est pas possible d'imbriquer des fonctions, comme en PASCAL, par exemple.

10.11 *Test*

1. Le code suivant va-t-il passer sans problèmes quelconques à la compilation ? Si non, le corriger.

```
int f(int x, int y)
     {
     return 2 * x;
     }
```

2. Redéfinir la fonction `min()` pour des segments de droite définis par leurs extrémités. `min()` retournera le segment de longueur la plus petite (on essaiera d'optimiser le passage des paramètres, si faire se peut) :

```
typedef   struct
     {
     double    xorg, yorg;             // Coordonnées origine
     double    xextrem, yextrem;       // Coordonnées extrémité
     }    SegLigne;

inline    double    min( ????? )
     {    return ( ????? );}
```

3. Ecrire la fonction factorielle (Voir *Récursivité*, page154.) de manière non récursive.
4. Le code ci-dessous est-il correct ?

```
const int& f()

     {
     int x = 1024;
     return x;
     }
```