UNIVERSITÉ PARIS-XII — FACULTÉ DE SCIENCES ÉCONOMIQUES ET DE GESTION
Master Professionnel *Méthodes Appliquées de la Statistique et de l'Économétrie pour la Recherche, l'Analyse et le Traitement de l'Information*
Année universitaire 2006-2007 — *François LEGENDRE*
F.Legendre@univ-paris12.fr — http://fj.legendre.free.fr

UNIC

UNIC — pour UNIté CentraleCentrale — est le nom donné à une machine seulement destinée à enseigner les bases de la programmation.

1 Présentation d'UNIC

A la différence des micro-processeurs du commerce, cette machine manipule des grandeurs en base 10.

Les adresses sont codées sur deux positions, de 00 à 99 : l'on peut donc désigner 100 emplacements différents en mémoire. Sur les micro-processeurs du commerce, les adresses sont codées sur deux octets (machines dites 16 bits), quatre octets (machines 32 bits) ou sur huit octets (machine 64 bits) : elles peuvent donc désigner un beaucoup plus grand nombre d'emplacements.

Ces emplacements sont des chiffres, de 0 à 9 : l'on dispose donc de 10 codes différents pour distinguer les grandeurs élémentaires manipulées. Sur les micro-processeurs du commerce, les grandeurs élémentaires sont des octets (un groupe de huit bits) qui permettent de disposer de 256 codes différents. Ces 256 nombres sont utilisés par exemple pour coder les caractères de l'alphabet.

UNIC dispose de trois registres : un registre général, un registre qui contient l'adresse de la base de la pile et un registre qui contient l'adresse de l'instruction en cours d'exécution. UNIC ne manipule que des entiers naturels compris entre 0 et 9. Le registre général est destiné à traiter ces entiers ; il est donc de taille 1. Les deux autres registres contiennent des adresses ; ils sont donc de taille 2.

Le jeu d'instruction d'UNIC est particulièrement simple. Il est présenté dans le tableau 1. Lors de l'appel du système d'exploitation (instruction dont le code est 0), le code qui figure juste après est l'un des quatre suivants :

Code	Explication
0	Fin du programme ⋆
1	Lecture d'un entier depuis ***std::cin*** ¶
2	Ecriture d'un entier vers ***std::cout*** §
3	Ecriture d'un entier vers ***std::cerr*** §

⋆ Le code de retour est placé dans le registre général.
¶ L'entier à lire sera disponible dans le registre général.
§ L'entier à écrire est placé dans le registre général.

TAB. 1 – Le jeu d'instructions d'UNIC

Instruction	Explication
0 ***c*** ⋆	Appeler le système d'exploitation, ***c*** étant le code de la fonction appelée
1 ***aa*** ¶	Charger le registre général avec le contenu situé à l'adresse ***aa***
2 ***aa*** ¶	Stocker le contenu du registre général à l'emplacement d'adresse ***aa***
3 ***aa*** ¶	Soustraire, l'opérande étant situé à l'adresse ***aa***
4 ***aa*** ¶	Exécuter l'instruction située à l'adresse ***aa*** (branchement inconditionnel)
5 ***aa*** ¶	Exécuter l'instruction située à l'adresse ***aa*** si le registre général n'est pas égal à zéro (branchement conditionnel)
6 ***d*** §	Charger le registre général depuis la pile, avec le déplacement ***d***
7 ***d*** §	Stocker le contenu du registre général vers la pile, avec le déplacement ***d***
8 ***aa*** ¶	Appeler une fonction@, la première instruction de la fonction étant à l'adresse ***aa***
9	Retourner (depuis une fonction)

⋆ ***c*** est un code de 0 à 3 qui indique la fonction appelée.
¶ ***aa*** est une adresse (de 00 à 99) qui désigne l'opérande.
§ ***d*** est un déplacement (de 0 à 9) qui désigne l'opérande dans la pile.
@ Le registre général contient le nombre de données locales à ne pas écraser.

2 Compilation en UNIC d'un premier programme C++

On se propose de traduire le programme suivant (en C++) en le jeu d'instructions d'UNIC.

```
1  #include <iostream> // 'cin', 'cout' et 'cerr'.
2
3  int n = 0 ; // Une variable globale entière initialisée à 0.
4
5  int
6  main() // Fonction principale du programme.
7  {
8    n = 1 ;            // La valeur entière '1' est affectée à la variable 'n'.
9    std::cout << n ; // La variable 'n' est affichée sur la sortie standard.
10   return 0 ;         // Fin du programme avec un code de retour égal à 0.
11 }
```

Ce programme est codé comme suit :

Adresse	Code	Explication
—	—	Début du segment des instructions.
00	1 13	Charger le registre général.
		13 est l'adresse de la constante ***1***.
03	2 15	Stocker le contenu du registre général.
		15 est l'adresse de la variable globale ***n***.
06	0 2	Appeler le système d'exploitation.
		2 est le code « Ecriture d'un entier vers ***std::cout*** ».
08	1 14	Charger le registre général.
		14 est l'adresse de la constante ***0***.
11	0 0	Appeler le système d'exploitation.
		0 est le code « Fin du programme ».
13	1	Constante ***1***.
14	0	Constante ***0***.
—	—	Début du segment des données globales.
15	0	Variable globale ***n*** initialisée à 0.

Notons les quelques remarques suivantes.

1. Par convention, le programme démarre à l'adresse 00.
2. Le programme est divisé en deux segments. D'une part, le segment des instructions ; d'autre part, le segment des variables globales. Le segment des instructions est normalement en « lecture seulement » : la machine n'a pas *a priori* à écrire dans ce segment. Le segment des variables globales est en lecture/écriture ; les valeurs qui figurent dans ce segment sont les valeurs données initialement aux variables. Dans notre programme, l'instruction ***int n = 0 ;*** n'a pas été traduite par une instruction exécutable ; elle a été traduite d'un côté par un emplacement réservé pour la variable ***n*** dans le segment des données globales et, d'un autre côté, par la valeur initiale ***0*** dans ce segment.
3. Les constantes utilisées dans le programme figurent dans le segment des instructions et non dans le segment des données globales. Cette solution est justifiée parce que les constantes ne sont accédées qu'en lecture.
4. Le compilateur travaille en deux passes. Dans une première passe, il engendre le code des instructions mais il ne peut renseigner ni l'adresse des variables globales ni l'adresse des constantes puisqu'il ne connaît pas encore la taille totale du programme. Il laisse alors, dans le code engendré, des emplacements vides. Dans notre exemple, ces emplacements sont situés aux adresses 01, 04 et 09. A la fin de cette première passe, il est en mesure de déduire que l'adresse de la constante ***1*** est 13, que l'adresse de la constante ***0*** est 14 et que l'adresse de la variable globale ***n*** est 15. Il peut donc finir d'engendrer le code en remplissant les emplacements laissés précédemment vides.

En exécutant ce programme pas à pas, on obtient les étapes suivantes :

1	1 13	Charger le registre général	1	03
2	2 15	Stocker le contenu du registre général	1	06
3	0 2	Appeler le système d'exploitation	1	08
4	1 14	Charger le registre général	0	11
5	0 0	Appeler le système d'exploitation	0	13

La colonne ***RG*** (pour Registre général) donne le contenu de ce registre *après* l'exécution de l'instruction. La colonne ***PC*** (pour Program counter) donne le contenu du registre qui sert à désigner l'adresse de l'instruction suivante à exécuter.

On constate notamment, puisque les instructions sont de longueur variable (elles prennent soit 2 positions soit 3 positions), que le registre qui désigne l'instruction à exécuter est incrémenté en conséquence.

3 Variables locales

Les variables locales sont allouées sur la pile. On se propose de traduire le programme suivant pour montrer de quelle façon UNIC gère la pile.

```
1  #include <iostream>
2
3  int
4  main()
5  {
6    int n = 1 ;  // Une première variable locale entière.
7    int m = 2 ; // Une seconde variable locale entière.
8    std::cout << n << m ;
9    return 0 ; // Retour au système d'exploitation avec le code '0'.
10 }
```

Ce programme est codé comme suit :

Adresse	Code	Explication
—	—	Début du segment des instructions.
00	1 23	Charger le registre général.
		23 est l'adresse de la constante ***1***.
03	7 0	Stocker le contenu du registre général vers la pile.
		0 est le déplacement qui désigne la variable locale ***n***.
05	1 24	Charger le registre général.
		24 est l'adresse de la constante ***2***.
08	7 1	Stocker le contenu du registre général vers la pile.
		1 est le déplacement qui désigne la variable locale ***m***.

Suite du programme sur la page suivante…

Adresse	Code	Explication
10	6 0	Charger le registre général depuis la pile.
		0 est le déplacement qui désigne la variable locale ***n***.
12	0 2	Appeler le système d'exploitation.
		2 est le code « Ecriture d'un entier vers ***std::cout*** ».
14	6 1	Charger le registre général depuis la pile.
		1 est le déplacement qui désigne la variable locale ***m***.
16	0 2	Appeler le système d'exploitation.
		2 est le code « Ecriture d'un entier vers ***std::cout*** ».
18	1 25	Charger le registre général.
		25 est l'adresse de la constante ***0***.
21	0 0	Appeler le système d'exploitation.
		0 est le code « Fin du programme ».
23	1	Constante ***1***.
24	2	Constante ***2***.
25	0	Constante ***0***.
—	—	Début du segment des données globales.
—	—	Début du segment de la pile.

Le recours à des variables locales se fait par l'intermédiaire de la pile. Cette pile constitue un troisième segment dans l'espace d'adressage du programme. Au démarrage d'un programme, le registre d'adresse de la base de la pile — ce registre est nommé habituellement ***SP*** (Stack pointer) — est renseigné par l'adresse du début de ce segment. Dans notre exemple, le contenu de ce registre est égal à 26 au démarrage. Notons que le segment des données globales est vide — en l'absence de variables globales dans le programme.

Les deux variables locales ***n*** et ***m*** sont placées dans la pile. Leur emplacement est donc relatif à la base de la pile ; en revanche, le déplacement par rapport à cette base est connu au moment de la compilation. Pour la variable locale ***n***, par exemple, le déplacement est égal à 0 ; son adresse absolue est égale à 26 puisque le contenu du ***SP*** est 26.

Une optimisation mineure aurait pu être adoptée dans le précédent programme. L'instruction 1 24 à l'adresse 05 qui permet de charger le registre général avec la valeur ***2*** pourrait être remplacée par l'instruction 3 23 (Additionner ; Adresse de la constante ***1***). Le registre général est ainsi incrémenter pour valoir ***2*** ; il n'est plus nécessaire de disposer de la constante ***2*** dans le segment des instructions : la taille de ce segment est un peu plus petite.

L'exécution pas à pas de ce programme conduit aux résultats suivants où la colonne ***SP*** donne le contenu du registre de la base de la pile et les colonnes 26 et 27 le contenu des emplacements de la mémoire d'adresse respectivement 26 et 27.

Pas	Code	Explication	*RG*	*PC*	*SP*	26	27
1	1 23	Charger le *RG*⋆	1	03	26	—	—
2	7 0	Stocker le *RG* vers la pile	1	05	26	1	—
3	1 24	Charger le *RG*	2	08	26	1	—
4	7 1	Stocker le *RG* vers la pile	2	10	26	1	2
5	6 0	Charger le *RG* depuis la pile	1	12	26	1	2
6	0 2	Appeler le *SdE*†	1	14	26	1	2
7	6 1	Charger le *RG* depuis la pile	2	16	26	1	2
8	0 2	Appeler le *SdE*	2	18	26	1	2
9	1 25	Charger le *RG*	0	21	26	1	2
10	0 0	Appeler le *SdE*	0	23	26	1	2

⋆ Registre général.

† Système d'exploitation.

4 Arithmétique et itération

On se propose maintenant d'examiner de quelle façon notre machine est capable de calculer et d'effectuer des opérations répétitives. Plus précisément, nous allons traduire le programme C++ suivant.

```
1  #include <iostream>
2
3  int
4  main()
5  {
6    for ( int i = 2 ; i != 0 ; -- i ) { // 'i' = 2 puis 'i' = 1.
7      std::cout << i ; }
8    return 0 ; // Retour au système d'exploitation avec le code '0'.
9  }
```

Ce programme est codé comme suit :

Adresse	Code	Explication
—	—	Début du segment des instructions.
00	1 19	Charger le registre général.
		19 est l'adresse de la constante ***2***.
—	—	Test du début de la boucle.
03	5 11	Se brancher si le régistre général n'est pas égal à 0.
		11 est l'adresse de l'instruction à exécuter.
06	1 20	Charger le registre général.
		20 est l'adresse de la constante ***0***.

Suite du programme sur la page suivante...

Adresse	Code	Explication
09	0 0	Appeler le système d'exploitation.
		0 est le code « Fin du programme ».
—	—	Début du corps de la boucle.
11	0 2	Appeler le système d'exploitation.
		2 est le code « Ecriture d'un entier vers ***std::cout*** ».
13	3 21	Soustraire.
		21 est l'adresse de la constante ***1***.
16	4 03	Se brancher inconditionnellement.
		03 est l'adresse de l'instruction à exécuter.
19	2	Constante ***2***.
20	0	Constante ***0***.
21	1	Constante ***1***.
—	—	Début du segment des données globales.
—	—	Début du segment de la pile.

La variable ***i*** est une variable locale ; elle devrait être allouée sur la pile. Le compilateur se rend compte qu'il n'est pas utile de lui affecter un emplacement en mémoire ; il suffit de garde cette variable dans le registre général.

L'exécution du programme pas à pas donne :

Pas	Code	Explication	*RG*	*PC*	*SP*
1	1 19	Charger le *RG*	2	03	22
2	5 11	Branchement si $RG \neq 0$	2	11	22
3	0 2	Appeler le *SdE*	2	13	22
4	3 21	Soustraire	1	16	22
5	4 03	Branchement inconditionnel	1	03	22
6	5 11	Branchement si $RG \neq 0$	1	11	22
7	0 2	Appeler le *SdE*	1	13	22
8	3 21	Soustraire	0	16	22
9	4 03	Branchement inconditionnel	0	03	22
10	5 11	Branchement si $RG \neq 0$	0	06	22
11	1 20	Charger le *RG*	0	09	22
12	0 0	Appeler le *SdE*	0	11	22

5 Appel des fonctions et récursion

Le jeu d'instructions de notre machine est très pauvre aussi les fonctions ne comportent-elles qu'un seul argument. La convention est la suivante. Lors de l'appel d'une fonction, l'argument est placé sur la pile, après les éventuelles variables locales. La fonction, de son côté, s'attend à trouver l'argument dans la pile en première position ; elle est libre d'utiliser la suite de la pile. L'appel d'une fonction conduit ainsi à ajuster le pointeur de pile. Plus précisément, le registre général, lors de l'appel d'une fonction, contient le nombre de variables locales à ne pas écraser. La machine, premièrement, sauvegarde l'argument ; deuxièmement, place sur la pile l'adresse de retour ; troisièmement, place sur la pile le nombre de variables locales ; quatrièmement, place sur la pile l'argument qui avait été sauvegardé ; cinquièmement, ajuste le pointeur de pile ; sixièmement, exécute la première instruction de la fonction.

Au retour d'une fonction, ce qui avait été fait est dénoué. Le pointeur de pile est ajusté, au moyen de l'information qui a été stockée dans la pile — le nombre de variables locales à ne pas écraser. Le programme reprend son cours en utilisant l'adresse de retour qui avait été sauvegardée, elle aussi, dans la pile.

Le programme suivant illustre le mécanisme, au cas particulier d'une fonction récursive — ici, cette fonction s'appelle elle-même directement.

```
1  #include <iostream>
2
3  void
4  ma_fonction(int n)
5  {
6    if ( n == 0 ) {
7      return ; } // Fin de la récursion si 'n' = 0.
8    std::cout << n ;
9    ma_fonction(n-1) ; // Appel récursif de la fonction.
10   return ;
11 }
12 int
13 main()
14 {
15   ma_fonction(3) ;
16   return 0 ; // Retour au système d'exploitation avec le code '0'.
17 }
```

Ce programme est codé comme suit :

Adresse	Code	Explication
—	—	Début du segment des instructions.
00	1 36	Charger le registre général.
		36 est l'adresse de la constante ***3***.
03	7 0	Stocker le contenu du registre général vers la pile.
		0 est le déplacement qui désigne l'argument effectif ***3***.
05	1 37	Charger le registre général.
		37 est l'adresse de la constante ***0***.

Suite du programme sur la page suivante…

Adresse	Code	Explication
08	8 16	Appeler une fonction. 16 est l'adresse de la première instruction de ***ma_fonction***.
11	1 37	Charger le registre général. 37 est l'adresse de la constante ***0***.
14	0 0	Appeler le système d'exploitation. 0 est le code « Fin du programme ».
—	—	Première instruction de la fonction ***ma_fonction***.
16	6 0	Charger le registre général depuis la pile. 0 est le déplacement qui désigne l'argument formel ***n***.
18	5 22	Sauter si le registre général n'est pas égal à zéro. 22 est l'adresse de la suite du programme.
21	9	Retourner.
22	0 2	Appeler le système d'exploitation. 2 est le code « Ecriture d'un entier vers ***std::cout*** ».
24	3 38	Soustraire. 38 est l'adresse de la constante ***1***.
27	7 1	Stocker le contenu du registre général vers la pile. 1 est le déplacement qui désigne l'argument effectif ***n-1***.
29	1 38	Charger le registre général. 38 est l'adresse de la constante ***1***.
32	8 16	Appeler une fonction. 16 est l'adresse de la première instruction de ***ma_fonction***.
35	9	Retourner.
36	3	Constante ***3***.
37	0	Constante ***0***.
38	1	Constante ***1***.
—	—	Début du segment des données globales.
—	—	Début du segment de la pile.

En mode pas à pas, l'exécution du programme précédent conduit à :

Pas	Code	Explication	*RG*	*PC*	*SP*	3444444444455555 9012345678901234
1	1 36	Charger le *RG*	3	03	39	----------
2	7 0	Stocker le *RG* vers la pile	3	05	39	3----------
3	1 37	Charger le *RG*	0	08	39	3----------
4	8 16	Appeler une fonction	0	16	42	0903----------
5	6 0	Charger le *RG* depuis la pile	3	18	42	0903----------
6	5 22	Branchement si $RG \neq 0$	3	22	42	0903----------
7	0 2	Appeler le *SdE*	3	24	42	0903----------
8	3 38	Soustraire	2	27	42	0903----------

Suite du mode pas à pas sur la page suivante…

Pas	Code	Explication	*RG*	*PC*	*SP*	3444444444455555 9012345678901234
9	7 1	Stocker le *RG* vers la pile	2	29	42	09032----------
10	1 38	Charger le *RG*	1	32	42	09032----------
11	8 16	Appeler une fonction	1	16	46	09033312--------
12	6 0	Charger le *RG* depuis la pile	2	18	46	09033312--------
13	5 22	Branchement si $RG \neq 0$	2	22	46	09033312--------
14	0 2	Appeler le *SdE*	2	24	46	09033312--------
15	3 38	Soustraire	1	27	46	09033312--------
16	7 1	Stocker le *RG* vers la pile	1	29	46	090333121-------
17	1 38	Charger le *RG*	1	32	46	090333121-------
18	8 16	Appeler une fonction	1	16	50	090333123311----
19	6 0	Charger le *RG* depuis la pile	1	18	50	090333123311----
20	5 22	Branchement si $RG \neq 0$	1	22	50	090333123311----
21	0 2	Appeler le *SdE*	1	24	50	090333123311----
22	3 38	Soustraire	0	27	50	090333123311----
23	7 1	Stocker le *RG* vers la pile	0	29	50	0903331233110---
24	1 38	Charger le *RG*	1	32	50	0903331233110---
25	8 16	Appeler une fonction	1	16	54	0903331233113310
26	6 0	Charger le *RG* depuis la pile	0	18	54	0903331233113310
27	5 22	Branchement si $RG \neq 0$	0	21	54	0903331233113310
28	9	Retourner	0	35	50	0903331233113310
29	9	Retourner	0	35	46	0903331233113310
30	9	Retourner	0	35	42	0903331233113310
31	9	Retourner	0	11	39	0903331233113310
32	1 37	Charger le *RG*	0	14	39	0903331233113310
33	0 0	Appeler le *SdE*	0	16	39	0903331233113310

6 Un chargeur d'un programme UNIC

On se propose maintenant d'écrire en C++ un chargeur de programmes UNIC ; c'est-à-dire un programme qui exécute un programme exécutable UNIC.

```
1  #include <iostream> // cin, cout et cerr
2  #include <fstream>  // ifstream
3  #include <sstream>  // istringstream
4  #include <iomanip>  // setw(...) et setfill(...)
5  #include <string>   // Chaînes de caractères
6
7  using namespace std ;
8
9  // Conversion d'une chaîne de caractères en un entier naturel.
10 inline size_t
11 txt2size_t(const string & texte)
```

```
{
  istringstream lecture(texte) ;
  size_t i ; lecture >> i ; // Il faudrait vérifier qu'il n'y a pas d'erreurs.
  return i ;
}
// Erreur fatale : le message est affiché et le programme se termine avec un code de
// retour égal à 1.
inline void
fatal(const string & message)
{
  cerr << "Erreur fatale, message : " << message << '.' << endl ;
  exit(1) ;
}

// Les variables suivantes sont globales, pour être vues depuis la fonction 'afficher(...)'.
size_t rg ;              // Registre général.
size_t sp ;              // Stack pointer,  initialisé à 0.
size_t pc ;              // Program counter, initialisé à 0.
size_t memoire[100] ; // Représentation de la mémoire, initialisée à 0.

int mem_debut = -1 ; // Adresse du premier emplacement de la mémoire à afficher.
size_t mem_fin ;     // Adresse du dernier emplacement de la mémoire à afficher.

// Cette fonction permet d'obtenir les sorties du mode pas à pas.
void
afficher(
 size_t pas,                    // Pas du programme.
 const string & explication, // Libellé pour l'instruction en cours d'exécution.
 size_t code,                   // Code de base de l'instruction.
 int SdE,                       // Code du service appelé si appel du système d'exploitation.
 int adresse,                   // Adresse de l'opérande si l'instruction comporte un tel opérande.
 int deplacement)               // Déplacement pour les instructions qui utilisent la pile.
{
  cerr << pas << '\t' << code ;
  // SdE, adresse et deplacement sont égales à -1 si elles ne sont pas pertinentes.
  if ( SdE != -1 ) {
    cerr << ' ' << SdE ; }
  else if ( adresse != -1 ) {
    cerr << ' ' << setw(2) << setfill('0') << adresse ; }
  else if ( deplacement != -1 ) {
    cerr << ' ' << deplacement ; }
  cerr << '\t' << explication << '\t' << rg << '\t' <<
    setw(2) << setfill('0') << pc << '\t' << setw(2) << setfill('0') << sp ;
  // mem_debut est égale à -1 s'il n'est pas nécessaire d'afficher le contenu d'une région
  // de la mémoire.
  if ( mem_debut != -1 ) {
    for ( size_t i = mem_debut ; i <= mem_fin ; ++ i ) {
      cerr << '\t' << memoire[i] ; } }
  cerr << endl ;
}
int
main(int argc, char *argv[])
{
  if ( (argc != 2) && (argc != 4) ) {
    fatal("Appel invalide du programme '" + string(argv[0]) + "'") ; }

  // Lecture du fichier qui contient le programme exécutable.
  ifstream fichier(argv[1]) ;
  if ( ! fichier ) {
    fatal("Echec de l'ouverture en lecture du fichier '" + string(argv[1]) + "'") ; }
  while ( sp < 100 )
  {
    char tmp ;
    // Lecture d'un caractère dans le fichier ; l'expression est égale à non si l'opé-
    // ration n'a pas abouti (fin de fichier ou erreur de lecture).
    if ( fichier >> tmp ) {
      // Un chiffre, OK ; le codage des chiffres, a priori, satisfait toujours la
      // règle selon laquelle ils sont codés consécutivement.
      if ( ('0' <= tmp) && (tmp <= '9') ) {
        memoire[sp++] = tmp - '0' ; }
      // Une fin de ligne ou un espace, à ignorer.
      else if ( (tmp == '\n') || (tmp == ' ') ) {
        ; }
      else {
        fatal("Programme, contenant un caractère étrange, invalide") ; } }
    else {
      break ; }
  }
  if ( ! fichier.eof() ) {
    fatal("Programme, de trop grande taille, invalide") ; }
  if ( sp == 0 ) {
    fatal("Programme, de taille nulle, invalide") ; }

  // Lecture des deux paramètres facultatifs qui permettent d'afficher le contenu
  // d'une région de la mémoire.
  if ( argc == 4 ) {
    mem_debut = txt2size_t(argv[2]) ; mem_fin = txt2size_t(argv[3]) ;
    if ( (mem_fin < mem_debut) || (mem_fin <= 0) || (mem_debut >= 99) ) {
      fatal("Valeurs invalides des paramètres de la région à afficher") ; } }

```

```
// Première ligne des sorties du mode pas à pas.
cerr << "Pas\tCode\tExplication\tRG\tPC\tSP" ;
if ( mem_debut != -1 ) {
  for ( size_t i = mem_debut ; i <= mem_fin ; ++ i ) {
    cerr << '\t' << i ; } }
cerr << endl ;

size_t pas = 1 ;
while ( true )
{
  size_t code = memoire[pc] ;
  int SdE = -1 ;
  int adresse = -1 ;
  int deplacement = -1 ;
  string explication ;
  size_t tmp ;
  switch ( memoire[pc++] )
  {
    // Appeler le système d'exploitation.
    case 0 : explication = "Appeler le SdE" ;
    switch ( SdE = memoire[pc++] ) {

      // Fin du programme.
      case 0 : afficher(pas, explication, code, SdE, adresse, deplacement) ;
      exit(rg) ;

      // Lecture d'un entier naturel depuis std::cin.
      case 1 : cin >> rg ;
      if ( rg > 9 ) {
        fatal("Lecture d'un nombre invalide") ; }
      break ;

      // Ecriture d'un entier naturel vers std::cout.
      case 2 : cout << rg ;
      break ;

      // Ecriture d'un entier naturel vers std::cerr.
      case 3 : cerr << rg ;
      break ;

      default : fatal("Appel d'un service invalide du système d'exploitation") ;
      }
    break ;

    // Charger le registre général.
    case 1 : explication = "Charger le RG" ;
    adresse = memoire[pc]*10 + memoire[pc+1] ;
    rg = memoire[adresse] ;
    pc += 2 ;
    break ;

    // Stocker le contenu du registre général.
    case 2 : explication = "Stocker le RG" ;
    adresse = memoire[pc]*10 + memoire[pc+1] ;
    memoire[adresse] = rg ;
    pc += 2 ;
    break ;

    // Soustraire.
    case 3 : explication = "Soustraire" ;
    adresse = memoire[pc]*10 + memoire[pc+1] ;
    if ( rg < memoire[adresse] ) {
      fatal("Débordement inférieur de capacité du registre général") ; }
    rg -= memoire[adresse] ;
    pc += 2 ;
    break ;

    // Exécuter l'instruction d'adresse 'adresse'.
    case 4 : explication = "Branchement inconditionnel" ;
    pc = adresse = memoire[pc]*10 + memoire[pc+1] ;
    break ;

    // Exécuter l'instruction d'adresse 'adresse' si le registre général n'est pas
    // égal à zéro.
    case 5 : explication = "Branchement si RG != 0" ;
    adresse = memoire[pc]*10 + memoire[pc+1] ;
    pc = (rg != 0) ? adresse : pc+2 ;
    break ;

    // Charger le registre général depuis la pile.
    case 6 : explication = "Charger le RG depuis la pile" ;
    deplacement = memoire[pc++] ;
    if ( sp + deplacement > 99 ) {
      fatal("Adresse, trop grande, invalide") ; }
    rg = memoire[sp + deplacement] ;
    break ;

    // Stocker le contenu du registre général vers la pile.
    case 7 : explication = "Stocker le RG vers la pile" ;
    deplacement = memoire[pc++] ;
```

```
192         if ( sp + deplacement > 99 ) {
193           fatal("Adresse, trop grande, invalide -- la pile déborde") ; }
194         memoire[sp + deplacement] = rg ;
195         break ;
196
197         // Appeler une fonction (elle ne comporte qu'un seul argument).
198         case 8 : explication = "Appeler une fonction" ;
199         if ( sp + rg + 3 > 99 ) {
200           fatal("Débordement de la pile lors de l'appel d'une fonction") ; }
201         // Sauvegarde de l'argument.
202         tmp = memoire[sp + rg] ;
203         // Ecriture dans la pile de l'adresse de retour.
204         memoire[sp + rg] = pc / 10 ; memoire[sp + rg + 1] = pc % 10 ;
205         // Ecriture dans la pile du nombre de locaux à ne pas écraser.
206         memoire[sp + rg + 2] = rg ;
207         // Ajustement du pointeur de la pile.
208         sp += rg + 3 ;
209         // Ecriture dans la pile de l'argument.
210         memoire[sp] = tmp ;
211         // Exécution de la première instruction de la fontion.
212         pc = adresse = memoire[pc]*10 + memoire[pc+1] ;
213         break ;
214
215         // Retourner (depuis une fonction).
216         case 9 : explication = "Retourner" ;
217         tmp = memoire[sp-1] ;
218         // Restauration de l'ancienne valeur du pointeur de pile.
219         sp -= tmp + 3 ;
220         if ( sp < 0 ) {
221           fatal("Corruption de la mémoire, le retour est invalide") ; }
222         // Reprise du flux d'instruction.
223         pc = memoire[sp + tmp]*10 + memoire[sp + tmp + 1] + 2 ;
224         break ;
225       } // Fin du 'switch ( memoire[pc++] )'.
226       afficher(pas, explication, code, SdE, adresse, deplacement) ;
227       ++ pas ;
228     }           // Fin du 'while ( true )'.
229     return 0 ; // Retour au système d'exploitation avec le code '0'.
230 }               // Fin de 'main(...)'
```