Matthieu GRALL

Cours de C / C++

Partie 1

Attention : ce document est un complément aux séances de cours et de travaux pratiques.

Table des matières

1 Introduction

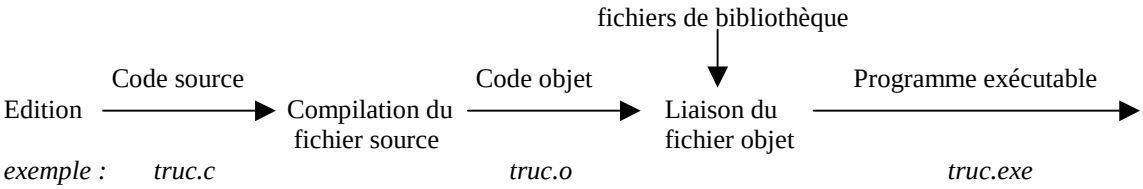
1.1 Historique

Le nom du langage C vient de son prédécesseur le langage B des laboratoires Bell. Le C a été conçu par Dennis Ritchie en 1972, dans le but de développer le système d'exploitation UNIX. Ce langage puissant et souple a connu un développement énorme et problématique puisque chacun développait son propre C. En réponse à ce problème, l'American National Standards Institute (ANSI) a formé un comité en 1983 pour établir une définition standard du C que tous les compilateurs actuels reconnaissent.

Le C++, quant à lui, est une extension de la norme ANSI de C. Ses atouts sont l'héritage (Programmation Orientée Objet) et la surcharge des fonctions et des opérateurs.

1.2 Le cycle de développement

Vous réaliserez rapidement des programmes en C++ à l'aide d'un simple éditeur de textes, cependant l'ordinateur ne comprend pas le code source C ou C++. En effet, il ne comprend que le langage machine (binaire), il faudra alors passer par des étapes intermédiaires avant de les exécuter :



L'édition est la rédaction du code source en langage C ou C++.
La compilation s'effectue à l'aide d'un des nombreux compilateurs existant sur le marché. La commande à exécuter dépend du compilateur (qui peut être sous forme d'un environnement graphique).
La liaison sert à lier au programme toutes les fonctions prédéfinies du langage (telles que celles qui servent à lire, afficher, etc.…) qui sont déjà compilées.
Compilation et liaison sont parfois exécutées en même temps, cela dit, il faut se rappeler que ce sont deux opérations distinctes.

Le suffixe des fichiers générés dépend du système de votre machine, mais le principe est toujours le même.
Si des erreurs surviennent lors de ces phases ou si le programme ne réalise pas ce qu'on attend, on modifie le code source, et on recommence le cycle de développement.

1.3 Le contenu des supports de cours

Il est fortement recommandé de s'appuyer sur des bases d'algorithmique pour comprendre ces documents, notamment en ce qui concerne les divers mécanismes de programmation et d'optimisation qui ne seront pas forcément détaillés.

Les principales notions de la programmation en C / C++ seront présentées dans ces supports de cours, mais il faut bien garder à l'esprit que leur maîtrise parfaite requiert un approfondissement important et beaucoup de pratique. Cependant, ces cours permettront de s'adapter facilement à ce que l'on n'a pas encore vu.

Les exemples étudiés seront réalisés en C++, et des spécificités du langage C seront parfois indiquées.

Il est vivement conseillé de découvrir des choses par soi-même. Le but est avant tout de rendre les programmeurs autonomes.

2 Structure d'un programme simple

2.1 Un premier exemple

```
// premier programme en C++, qui réalise l'addition de deux nombres

#include <iostream.h>          // indique au compilateur qu'il doit inclure
                               // le contenu du fichier iostream.h dans le
                               // programme (pour cout et cin)

int addition(int,int) ;        // prototype de la fonction addition

void main()                    // la fonction main est le corps du programme,
                               // elle est obligatoire, son contenu est
                               // décrit entre les accolades suivantes
       {
       int a,b,c ;             // déclaration des 3 variables (int : entiers)
       cout << endl << "Premier nombre : ";
       cin >> a ;              // saisie du premier nombre
       cout << endl << "Second nombre : ";
       cin >> b ;              // saisie du second nombre
       c = addition(a,b) ;     // appel de la fonction addition
       cout << endl <<a<<" + "<<b<<" = "<<c ;        // affichage du résultat
       }

int addition(int x, int y)     // description de la fonction addition
       {
       return(x+y) ;           // return renvoie le résultat d'une fonction
                               // au programme appelant
       }
```

2.2 Le résultat de l'exécution

```
Premier nombre : 14
Second nombre : 6
14 + 6 = 20
```

2.3 L'apprentissage par la pratique

Le programmeur débutant devra tout d'abord se familiariser avec la syntaxe et les termes du langage. Ensuite, il lui faudra rapidement mettre en pratique ces aspects théoriques afin de faciliter la compréhension. Les programmes présentés ici se complexifient peu à peu en intégrant de plus en plus de nouvelles notions. Ceci permet par conséquent un apprentissage progressif.

3 Structures de données

3.1 Types de données

Tout programme est destiné à traiter des données. Celles-ci sont désignées par un nom associé à un type. Ainsi, on pourra avoir un âge de type entier :

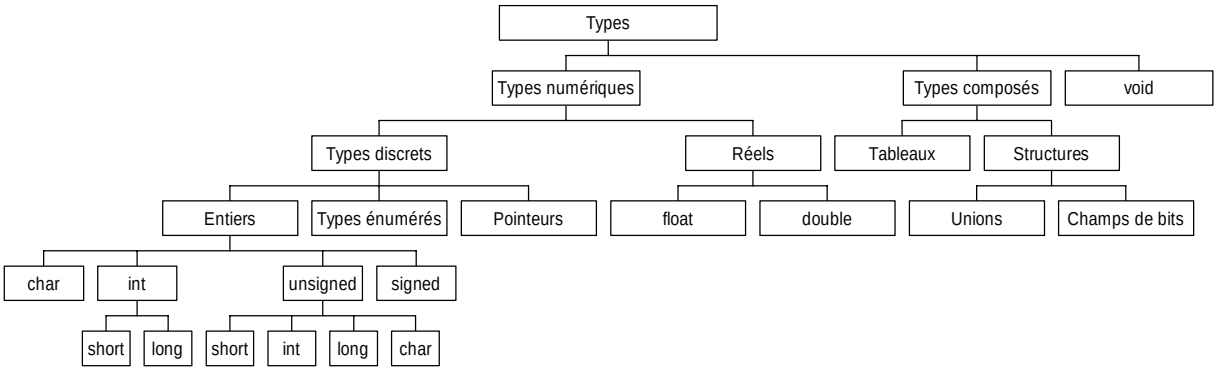
```
int age;
age = 20;
```

La déclaration de données fixe, selon le type, la mémoire occupée par celle-ci, le format de stockage interne, et les opérations applicables à un objet de ce type. Une donnée doit toujours être déclarée avant d'être utilisée ou affectée.

La plupart des types sont expliqués dans les paragraphes suivants. Néanmoins, certains types (types énumérés, unions, champs de bits, …) ne seront pas présentés.

3.1.1 Types de données de base

Le tableau suivant montre l'arborescence des types de base en C / C++. On choisit le type en fonction des données que l'on veut représenter. Les plus courants sont les caractères (*char*), les entiers (*int, long*, …), et les réels (*float, double*).



Déclaration

```
typeX nomX              // typeX est un nom de type quelconque
                        // nomX est un nom valide quelconque

char a ;
int b ;
double c, d, *e ;       // e est un pointeur sur une donnée de type double
```

Accès

```
nomX
```

Affectation

```
nomX = valeurX          // valeurX est une valeur valide quelconque

a = 'k' ;
b = 4 ;
c = 3,14 ;
d = 1,222789 ;
*e = 0,666 ;
```

3.1.2 Tableaux

On peut créer des tableaux de données de n'importe quel type (de base, structures, pointeurs…). Attention aux décalages : la première cellule d'un tableau a pour indice 0.

Déclaration
typeX nomX [taille du tableau]

```
int   table1 [10],            // tableau de 10 entiers
      table2 [10][10][100] ;  // tableau d'entiers à 3 dimensions
double table3 [50] ;          // tableau de 20 réels
```

Accès
nomX [indice de 0 à (taille du tableau –1)]

Affectation
nomX [indice de 0 à (taille du tableau –1)] = valeurX

```
table1 [4] = 256 ;            // on affecte 256 à la cellule d'indice 4
```
…

3.1.3 Pointeurs

Les pointeurs sont les adresses des données qu'elles référencent. On doit indiquer de quel type est la donnée adressée. Ainsi, on peut « pointer » n'importe quel type de données.

Déclaration
*typeX * nomX*

```
double *i ;
unsigned long * u ;
```

Accès
nomX permet d'accéder au pointeur, alors que **nomX* permet d'accéder à la donnée pointée.

Affectation
**NomX = valeurX*

```
*i = 4,001 ;
*u = 256 ;
```

3.1.4 Structures (agrégats)

Une structure étant un regroupement de plusieurs variables qui ne sont pas forcément du même type, elle peut aussi servir à définir de nouvelles variables du type de a structure.

Définition

```
struct structX{
      type1 nom1 ;      // liste des attributs de la structure
      type2 nom2 ;
      …
      typeN nomN ;
      } ;

struct etudiant{
      char * nom ;
      chat * prenom ;
      int age ;
      } ;
```

Déclaration

```
structX nomX ;

etudiant e1, e2 ;
```

Accès

nomX permet d'accéder à un objet, et *nomX.nomI* permet d'accéder à un attribut de l'objet.

Affectation

L'affectation ne s'effectue qu'attribut par attribut :

```
NomX.nomI  = valeurX

e1.age = 20 ;
e1.nom = ″SMITH″ ;
e1.prenom = ″Jack″ ;
e2.age = 25 ;
…
```

3.2 Tailles des données

Type	Mot clé	Taille (octets)	Intervalle des valeurs
Caractère	*char*	1	-128 à 127
Entier	*int*	2	-32 768 à 32 767
Entier court	*short*	2	-32 768 à 32 767
Entier long	*long*	4	-2 147 483 648 à 2 147 438 647
Caractère non signé	*unsigned char*	1	0 à 255
Entier non signé	*unsigned int*	2	0 à 65 535
Entier court non signé	*unsigned short*	2	0 à 65 535
Entier long non signé	*unsigned long*	4	0 à 4 294 967 295
Simple précision virgule flottante	*float*	4	environ $1,2^{E}$-38 à $3,4^{E}38$
Double précision virgule flottante	*double*	8	environ $2,2^{E}$-308 à $1,8^{E}308$
Tableau	**typeX***[***tailleX***]*	Taille (**typeX**)***tailleX**	
Structure	*struct* **X**	dépend des types	
Pointeur	***typeX**	4	

3.2.1 Exemple d'un programme donnant la taille des types

```
/* programme qui donne la taille de types à l'aide de la fonction sizeof, prenant un type
ou une variable en paramètre */

#include <iostream.h>

void main (){
       cout << endl << "----- Tailles de différents types de base -----";
       cout << endl << "char           : " << sizeof(char) << " octets";
       cout << endl << "int            : " << sizeof(int) << " octets";
       cout << endl << "short          : " << sizeof(short) << " octets";
       cout << endl << "long           : " << sizeof(long) << " octets";
       cout << endl << "unsigned char  : " << sizeof(unsigned char) << " octets";
       cout << endl << "unsigned int   : " << sizeof(unsigned int) << " octets";
       cout << endl << "unsigned short : " << sizeof(unsigned short) << " octets";
       cout << endl << "unsigned long  : " << sizeof(unsigned long) << " octets";
       cout << endl << "float          : " << sizeof(float) << " octets";
       cout << endl << "double         : " << sizeof(double) << " octets";
       cout << endl << "----- Tailles de quelques tableaux -----";
       cout << endl << "char [256]     : " << sizeof(char[256]) << " octets";
       cout << endl << "long [10]      : " << sizeof(long[10]) << " octets";
       cout << endl << "double [10]    : " << sizeof(double[10]) << " octets";
       cout << endl << "----- Tailles de quelques structures -----";
       cout << endl << "struct{int i1;int i2;}      : "
                                          << sizeof(struct{int i1;int i2;}) << " octets";
       cout << endl << "struct{int i;char c;}       : "
                                          << sizeof(struct{int i;char c;}) << " octets";
       cout << endl << "struct{int i[10];char c;}   : "
                                          << sizeof(struct{int i[10];char c;})<< " octets";
       cout << endl << "----- Tailles de quelques pointeurs -----";
       cout << endl << "char *                      : " << sizeof(char *) << " octets";
       cout << endl << "long *                      : " << sizeof(long *) << " octets";
       cout << endl << "double *                    : " << sizeof(double *) << " octets";
       cout << endl << "struct{int i1;int i2;}     *       : "
                                          << sizeof(struct{int i1;int i2;} *) << " octets";
       }
```

3.2.2 Résultat d'exécution

```
----- Tailles des différents types de base -----
char              : 1 octets
int               : 2 octets
short             : 2 octets
long              : 4 octets
unsigned char     : 1 octets
unsigned int      : 2 octets
unsigned short    : 2 octets
unsigned long     : 4 octets
float             : 4 octets
double            : 8 octets
----- Tailles de quelques tableaux -----
char [256]        : 256 octets
long [10]         : 40 octets
double [10]       : 80 octets
----- Tailles de quelques structures -----
struct{int i1 ;int i2 ;}      : 4 octets
struct{int i ;char c ;}       : 3 octets
struct{int i[10] ;char c ;}   : 21 octets
----- Tailles de quelques pointeurs -----
char *                        : 2 octets
long *                        : 2 octets
double *                      : 2 octets
struct{int i1 ;int i2 ;}      : 2 octets
```

3.3 Noms et statuts des données

3.3.1 Noms des données

Il est possible de donner n'importe quel nom aux données tant que l'on respecte les règles suivantes :

- Le nom peut contenir des lettres, des chiffres et le caractère « _ »
- Le premier caractère doit être une lettre ou le caractère « _ » (il n'est pas recommandé)
- Les lettres minuscules sont différentes des lettres majuscules
- Il ne faut pas utiliser les mots clés du langage C ou C++

3.3.2 Variables

Syntaxe de la déclaration des variables :

```
typeX nomX ;
```

L'initialisation des variables peut s'effectuer dès la déclaration :

```
typeX nomX ; nomX=valeurX ;
```
ou
```
typeX nomx=valeurX ;
```

La valeur des variables peut être changée dans le programme.

3.3.3 Le mot clé *typedef*

typedef permet de créer un synonyme pour un type de donnée existant. Par exemple l'instruction :

```
typedef int entier ;
```

crée le synonyme *entier* pour *int*. Vous pourrez ainsi utiliser *entier* pour définir des variables de type *int*, comme dans l'exemple suivant :

```
entier compte ;
```

typedef ne crée pas un nouveau type de donnée, il permet seulement d'utiliser un nom différent pour un type de donnée déjà défini. L'usage le plus fréquent concerne les structures (agrégats).

3.3.4 Constantes

Syntaxe de la déclaration des constantes :

```
const typeX nomX = valeurX ;
```

La valeur des constantes ne peut changer dans le programme.

4 Opérateurs

4.1 Opérateurs unaires

Il existe toute une série d'opérateurs unaires, qui s'appliquent à des expressions (E) de droite à gauche :

`++E`	incrémentation (la valeur de l'expression globale est celle de E après incrémentation)
`--E`	décrémentation (la valeur de l'expression globale est celle de E après décrémentation)
`&E`	adresse où réside la variable ou la fonction
`*E`	indirection (déréférencement) qui renvoie le contenu sur lequel pointe E
`-E`	changement de signe
`~E`	complément à 1 de E (E doit être un entier)
`!E`	non logique, renvoie 0 si E est différent de 0 et 1 sinon
`sizeof (E)`	calcul du nombre d'octets nécessités par une variable ou un type
`new typeX`	allocation de mémoire nécessaire au type, renvoie l'adresse réservée ou *NULL* en cas d'échec ; si la donnée est un objet avec un constructeur, ce dernier est appelé
`delete E`	libération de la mémoire allouée par *new* (E est un pointeur), et appel du destructeur (si objet)

4.2 Transtypage explicite

`(typeX) E` la valeur de l'expression E est convertie (si possible et selon des règles prédéfinies ou définies par le programmeur) en une valeur du type indiqué qui est renvoyée comme résultat. E reste inchangé.

4.3 Opérateurs arithmétiques

`*`	multiplication
`/`	division
`%`	modulo
`+`	addition
`-`	soustraction
`E1 > E2`	décalage binaire vers la droite de E1d'un nombre de positions E2
`E1 < E2`	décalage binaire vers la gauche de E1d'un nombre de positions E2

4.4 Opérateurs de comparaison

`>`	supérieur strictement
`<`	inférieur strictement
`>=`	supérieur ou égal
`<=`	inférieur ou égal
`==`	égal
`!=`	différent

4.5 Opérateurs logiques

`&&`	ET
`||`	OU

4.6 Opérateurs bit à bit (sur des entiers)

`&`	ET
`^`	OU exclusif
`|`	OU inclusif

4.7 Affectation

`E1 = E2` affectation de E2 à E1 s'ils sont de même type (sinon il y aura une conversion ou une erreur)

Combinaison avec des opérateurs arithmétiques (*E1 op=E2* équivaut à *E1=E1 op E2*) :

`*=` `/=` `%=` `+=` `-=` `<=` `>=` `&=` `^=` `|=`

4.8 Opérateur virgule

`E1, …, EN` regroupement de plusieurs expressions dans une même instruction

4.9 Autres opérateurs et instructions usuels

`//`	commentaires sur la suite de la ligne (C++)
`/*`	début de commentaires jusqu'à « */ »
`*/`	fin de commentaires
`{`	début de bloc d'instructions
`}`	fin de bloc d'instructions
`cout <<`	affichage
`cin >>`	saisie

5 Sous-programmes

5.1 Procédures et fonctions

Afin d'éviter les redondances, et pour des questions de lisibilité, on emploie des fonctions (qui retournent une valeur) et des procédures (fonctions qui retournent une valeur de type *void*, c'est-à-dire rien) le plus souvent possible. Une fonction (ou une procédure) est référencée par un nom, elle est indépendante, et réalise une tâche particulière.

```
…
// déclaration sous forme de prototype
typeX fonctionX (liste_des_types_des_paramètres) ;
…
void main () {
      …
      // appel de la fonction
      fonctionX (liste_des_valeurs) ;
      …
      }
…
// définition de la fonction
typeX fonctionX (liste_des_types_et_des_paramètres)
      {instruction}
…
```

On peut se passer du prototype si la définition se situe au-dessus de l'appel de la fonction (ou de la procédure).

5.2 Paramètres

Le nombre et les types des paramètres indiqués à l'appel de la fonction (ou de la procédure) doivent correspondre au nombre et aux types spécifiés dans la définition. La valeur renvoyée peut être utilisée comme opérande dans le cas d'une fonction (et être directement affectée à une variable par exemple). Les fonctions (ou les procédures) peuvent s'appeler mutuellement. A l'issue d'une fonction (ou d'une procédure), c'est l'instruction qui suit son appel qui est exécutée.

5.3 Exemple de programme utilisant un sous-programme

```
// Exemple d'une fonction simple
#include <iostream.h>   // pour les instructions de saisie (cin >> …) et
                        // d'affichage (cout << …)
// prototype de la fonction
long cube(long) ;

// programme principal
void main(){
    long valeur, reponse ;
    cout << "Entrez une valeur entière : " ;
    cin >> valeur ;
    reponse = cube (valeur) ; // appel de la fonction
    cout << endl << "Le cube de " << valeur << " est " << reponse ;
    }

// définition de la fonction
long cube(long x){
      long x_au_cube ;
      x_au_cube = x * x * x ;
      return (x_au_cube) ;
      }
```

6 Instructions de contrôle

6.1 Branchement conditionnel simple (si…sinon)

```
if (condition)
      instruction1 ;
else
      instruction2 ;
```

6.2 Branchement conditionnel multiple (selon)

```
Switch (condition){
      case constante1 : instruction1 ; break ;
      case constante2 : instruction2 ; break ;
      …
      default : instruction ;
     }
```

6.3 Boucles (pour, répéter, tant que)

Boucle « Pour » :

```
for (initialisation ; condition d'arrêt ; instruction à chaque itération)
      instruction ;
```

Boucle « Répéter » :

```
do
      instruction ;
while (condition) ;
```

Boucle « Tant que » :

```
while (condition)
      instruction ;
```

6.4 Branchements inconditionnels

6.4.1 goto

C'est un saut dans un programme. On l'utilise quelquefois pour la gestion des erreurs qui impliquent l'abandon de plusieurs structures de contrôle imbriquées. On conseille de ne l'employer qu'à l'intérieur d'une fonction pour des raisons de compréhension à la lecture du programme :

```
…
goto nom_de_label ;
…
nom_de_label :
…
```

6.4.2 continue

Liée à une boucle, cette instruction permet de mettre fin à une itération et de poursuivre l'exécution de la boucle en entamant l'itération suivante. Elle s'applique à tous les types de boucles. Exemple :

```
while (…) {
      …
      if (…)
            continue ;
      …
      }
```

6.4.3 break

Cette instruction provoque la sortie d'une boucle (et pas seulement d'une itération de la boucle). Exemple :

```
while (…) {
      …
      if (…)
            break ;
      …
      }
```

6.4.4 return

Cette instruction doit terminer toute fonction, elle précise la valeur renvoyée. L'exécution continue ensuite à l'endroit qui a appelé la fonction

```
Nom_de_fonction (…) {
      …
      return (valeur_à_renvoyer) ;
     }
```