Programmation Orientée Objet

Mise en oeuvre en C++

1 - Les Bases du C++

Julien Yves ROLLAND (julien.rolland@univ-fcomte.fr)

Laboratoire de Mathématiques de Besançon
Université de Franche-Comté

1.1

Buts et objectifs

Buts et objectifs de ce chapitre.

1. Situer le positionnement du C++ parmi les langages de programmation ;
2. Comprendre le fonctionnement et la structure d'un code C++ ;
3. Identifier et utiliser les types et opérations de base d'un code C++ ;
4. Préparer le terrain pour l'introduction des notions avancées.

1.2

Table des matières

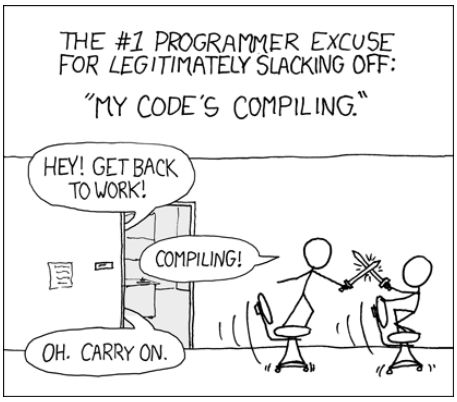
1.3

1 Généralités

La programmation

Développement, programmation
Ensemble des concepts, techniques et outils visant à exprimer des idées sous la forme de code.

Un langage de programmation est un outil. On souhaite résoudre un problème, à l'aide d'un programme, le langage n'est qu'un intermédiaire. Il n'y a pas d'outil parfait...



XKCD - https://xkcd.com/303/

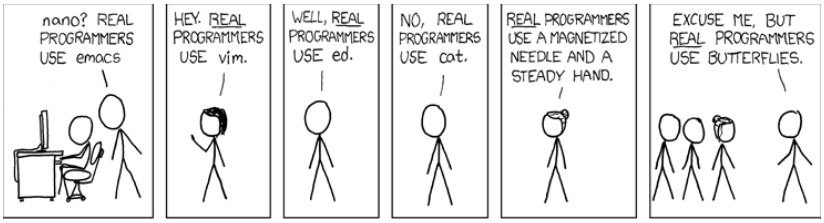
Déroulement classique

- Analyse
- Design
- Code
- Test

1.4

IDE, éditeurs,...

Environnement de développement
Ensemble d'outils logiciels regroupés en une application facilitant l'écriture, la compilation et le test de programme.



XKCD - https://xkcd.com/378/

Quelques IDE :

- CodeBlocks
- Qt Creator
- Eclipse
- Visual Studio

Quelques éditeurs :

- Emacs/Vim
- Atom
- Notepad++
- Gedit / Kate

En pratique
Un IDE gère les étapes de compilation et debugging, apporte une coloration syntaxique et une introspection du code.

1.5

Quelques conseils pour ne pas paniquer

- « *No pain, no gain* » Importance de savoir faire des erreurs.
- Corollaire : Ne laissez jamais quelqu'un faire un bout de code pour vous ! Au pire, examinez comment il fait, reproduisez puis discutez !
- On ne se résigne à écrire un programme qu'au moment où on a besoin de l'aide d'un ordinateur pour résoudre un problème ⇔ Un programme n'est simple que pour un problème trivial.
- Le "bon sens" n'existe pas du tout en informatique, il faut pouvoir décrire une tâche en entier pour la programmer. Importance de la compréhension.
- Un problème, 100 solutions : Il n'y a clairement pas unicité de la solution.
- KISS : Keep It Simple Stupid !
- Coder en anglais... sérieusement...

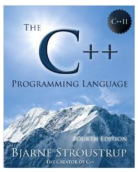
1.6

2 Introduction historique

Le langage C++

1.7

Le langage C++



Bjarne Stroustrup.

— Années 80 : Émergence de la « Programmation Orientée Objet »(POO).
— 1983 : Le « C with Classes » de Bjarne Stroustrup (AT&T) devient le C++.
— 1985 : Publication de « The C++ Programming Language ».

Normalisation

— 1985-1998 : Les éditions de « The C++ Programming Language » servent de norme.
— 1998 : Définition de la première norme ISO (C++98)
— 2017 : Cinquième et actuelle révision ISO (C++17)
— 2020 : Future révision majeure

1.8

Positionnement du C++

Le C++ est apparu comme supplément au C ajoutant principalement les spécificités nécessaires à la Programmation Orientée Objet. Ses grandes caractéristiques sont :

— Une norme (ANSI/ISO) cadrée et régulièrement révisée ;
— D'excellents résultats en terme de performance, d'efficacité et de flexibilité ;
— Une flexibilité sur le choix du paradigme de programmation (POO non obligatoire) ;
— Une relative complexité héritée de la compatibilité C

Les dernières normes s'orientent principalement vers la programmation générique.

1.9

Pourquoi apprendre le C++ ?

De par sa flexibilité le C++ est apte à remplir des tâches appartenant à différents domaines :

— traitement local ou distribué ;
— utilisation de fonctions numériques, graphiques ;
— traitement des interactions utilisateurs et accès aux bases de données.

Cette polyvalence en fait un des langages les plus utilisés dans la recherche et le calcul scientifique, de nombreuses librairies de calcul sont écrites ou portées pour ce langage.

Enfin, il a largement influencé la conception d'autres langages (C#, Java,...). Sa maîtrise est donc utile dans le cadre général de la programmation.

1.10

C++17 $\geq$ C++98 ?

Les dernières normes apportent surtout des outils de programmation générique et facilite la manipulation des données « habituelles »

— Expression régulière, support Unicode
— Nouvelle boucle-for
— Modèle de conteneur plus évolué
— Génération aléatoire robuste
— Meilleure gestion de ressource (pointeurs intelligents, initialisation uniforme)
— Outils pour la programmation générique (variadic template, lambda function)

Les normes préservant la compatibilité ascendante, le C++98 est encore très largement présent.

La majorité de ce chapitre aborde la norme C++98.

1.11

Avantages du C++

Le C++ n'est pas le plus concis ni le plus « propre » des langages mais il est :

— suffisamment « propre » pour l'enseignement des concepts de base ;
— suffisamment efficace et souple pour l'élaboration de programme exigeant ;
— suffisamment ouvert pour s'intégrer dans un écosystème hétérogène ;
— suffisamment complet pour l'enseignement et l'usage des techniques de programmation avancées.

> « C++ is a language that you can grow with. »
>
> *Bjarne Stroustrup*

> « Even I knew how to design a prettier language than C++. »
>
> *(aussi) Bjarne Stroustrup*

1.12

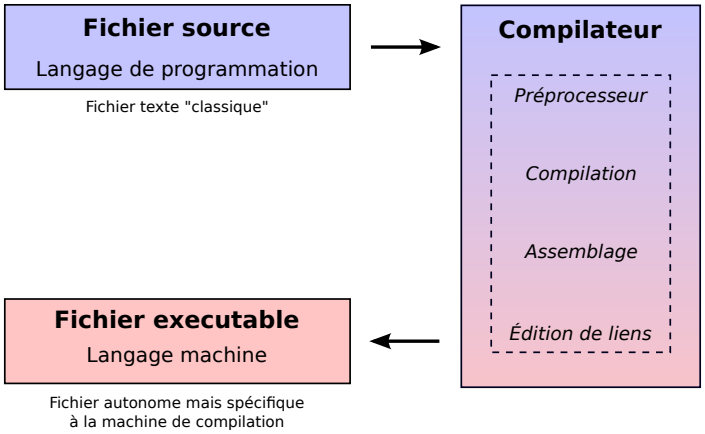
3 Concepts et structures du C++

3.1 La compilation

1.13

C++, langage compilé

Un langage compilé ?
Le C++ est un langage compilé, les fichiers d'origine doivent être « transformés » pour générer des fichiers compréhensibles par la machine. Un langage interprété (Matlab, Scilab, R, Python) est directement lisible par un « interpréteur ».



1.14

Rôle du compilateur

Le compilateur est un programme (ou une chaîne d'outils logiciels) qui a pour charge de :

- Vérifier la syntaxe du fichier source ;
- Générer des erreurs compréhensibles pour l'utilisateur ;
- Optimiser tout ou partie du code selon les paramètres de l'utilisateur ;
- Regrouper les différents sous-programmes constituant le programme final ;
- Générer un code assembleur optimisé pour la machine cible.

Le compilateur est votre meilleur ami, écoutez le !

1.15

Étapes de compilation

Pré-processeur (`gcc -E`)
Il « nettoie » le code en interprétant les fichiers sources selon les directives propres (identifiées par le symbole #).

Compilation (`gcc -S`)
Il traduit le fichier du pré-processeur en assembleur : Une suite d'instructions pour le processeur.

Assemblage (`gcc -c`)
Le code assembleur est transformé en fichier binaire, instructions en code machine (adapté à l'architecture cible). Étape souvent regroupée dans la compilation.

Édition de liens
Regroupe les différents fragments du programme (plusieurs sources, librairies externes,...). Produit le fichier exécutable.

1.16

Faune et flore locales

Les fichiers sources

- .cpp : Le fichier source (implémentation)
- .hpp ou .h : Fichier en-tête (pré-processeur, macro, ...)

Fichiers intermédiaires

— .o : Fichier objet (fichier temporaire à assembler)
— .a : Fichier archive (librairies pré-compilées)
— .s : Fichier compilé
— .c : Fichier assemblé

Fichiers de sortie

— .exe (Windows) : Fichier exécutable
— a.out : exécutable par défaut de GCC
— .a : Librairie compilée statique
— .so (Linux) ou .dll (Windows) : Librairie dynamique

1.17

3.2 Exemples de code

1.18

Exemple commenté de code C++ : Hello World

```
1  // Mon premier programme C++
2
3  #include <iostream>
4
5  using namespace std;
6
7  int main()
8  {
9    cout << "Hello World !" << endl;
10
11   return 0;
12 }
```

1.19

Décodage

Commentaires

```
// Mon premier programme C++
```

Commentaires (supprimé par le pré-processeur).

Directive au pré-processeur

```
#include <iostream>
```

Demande d'inclure les fichiers d'une librairie standard C++ (`iostream`) du système (les symboles <>) pour permettre l'utilisation de la fonction `cout`.

Instructions

```
using namespace std;
```

Les fonctions de la librairie standard (dont `iostream`) sont déclarées dans un espace de noms : le `namespace` « std ».

1.20

Plus de décodage

Déclaration

```
int main()
{
...
}
```

On déclare un bloc d'instruction (symboles `{ }`) nommée `main` sans argument (parenthèses vides) et retournant un `int` (un entier).

Instructions

```
cout << "Hello World !" << endl;
return 0;
```

Deux lignes d'instruction, elles se terminent par un `;`

`cout` affiche une chaîne de caractères (contenue dans " ... ") et ajoute un retour à la ligne (`endl` pour « end line »). `return` permet de sortir de `main` en retournant `0` (un entier).

1.21

Exemple commenté de code C++ : Racines carrés

```
1  #include <iostream>
2  #include <cmath>
3  #define NFOIS 5
4
5  using namespace std;
6
7  int main(void){
8    int i;
9    float x;
10   float racx = 0;
11
12   cout<<"Bonjour"<<endl;
13   cout<<"Je vais calculer "<<NFOIS<<" racines carrées réelles.\n";
14
15   for(i=0; i<NFOIS; i++){
16     cout<<"Donnez un nombre: ";
17     cin>>x;
18     if(x<0.0){
19       cout<<"Le nombre "<<x<<" ne possède pas de racine réelle\n";
20     }//end if
21     else{
22       racx = sqrt(x);
23       cout<<"La racine carrée de "<<x<<" est: "<<racx<<endl;
24     }//end else
25   }//end for
26   cout<<"Fin du programme\n";
27 }//end main
```

1.22

Décodage

Directive au pré-processeur

```
#define NFOIS 5
#include <cmath>
```

Le pré-processeur remplace automatiquement les occurrences de `NFOIS` par le nombre 5. On charge la bibliothèque de fonctions mathématiques (pour `sqrt()`).

En pratique
On n'utilise quasiment jamais #define pour ça.

Instruction de déclaration

```
int i;
float x;
```

On déclare des variables (`i` et `x`) qui seront respectivement un entier et un réel.

1.23

Encore du décodage

Déclaration et initialisation

```
float racx = 0.0;
```

On déclare la variable `racx` de type réel AVEC initialisation (par affectation) à la valeur 0.

Opérateurs de flux

```
cout<<"Bonjour"<<endl;
cin>>x;
```

Les symboles `<<` et `>>` sont des opérateurs de flux. Il agissent sur des flux (des données typées) qu'ils manipulent. Ici `cout` envoie la chaîne de caractères vers l'écran, `cin` envoie la saisie utilisateur vers une variable.

1.24

Toujours plus de décodage

Instruction de contrôle : boucle

```
for(i=0; i<NFOIS; i++){
}
```

Ouvre une boucle d'itération sur la variable `i`, de la valeur 0 à la valeur `NFOIS`. L'instruction `++` indique d'avancer le pas « d'une unité » (ici entière).

Instruction de contrôle : Condition

```
if(x<0.0){
}
else{
}
```

`if` ouvre un bloc d'instruction conditionnel : Si la condition est remplie le bloc est exécuté, sinon le bloc `else` est exécuté, s'il existe.

1.25

3.3 Les composantes de base

1.26

Les composantes élémentaires du C++

Un programme en C++ est constitué de 7 groupes de composantes élémentaires :

- les identificateurs
- les mots-clés
- les constantes
- les chaînes de caractères
- les opérateurs
- les signes de ponctuations
- les commentaires

1.27

Les identificateurs

L'identificateur sert à donner un nom à une entité du programme. Il peut désigner :

- un nom de variable ou de fonction ;
- un `struct`, `class`, `enum`, `union` ;
- un type défini par `typedef`.

L'identificateur est composé d'une suite de caractères parmi :

- des lettres (majuscules ou minuscules, différenciées, pas d'accent) ;
- des chiffres ;
- un blanc souligné (blanc_souligné).

L'identificateur ne peut pas commencer par un chiffre !

1.28

Les mots-clés

Certains mots sont réservés dans le langage et ne peuvent donc pas être utilisés comme identificateurs. Voici une liste non exhaustive :

`auto`	`do`	`int`	`struct`
`bool`	`double`	`long`	`switch`
`break`	`else`	`new`	`typedef`
`case`	`enum`	`register`	`union`
`char`	`extern`	`return`	`unsigned`
`class`	`float`	`short`	`using`
`const`	`for`	`signed`	`void`
`continue`	`goto`	`sizeof`	`volatile`
`default`	`if`	`static`	`while`

1.29

Les commentaires

Commentaire en ligne, bloc de commentaire
Le commentaire d'une ligne est précédé du symbole `//`. On peut créer un bloc de commentaire en le bornant des symboles `/*` et `*/`.

```
/* Ce code est un exemple ne servant
   à
   rien créé par Julien Yves Rolland.
*/
int main ()
{
  // Je ne sers à rien !
  return 0;
}
```

1.30

3.4 La structure d'un programme

1.31

Expression et instructions

Expression
Suite de composantes élémentaires syntaxiquement correcte.

```
((i >= 0) && (i < 10)) || (P[i] != 0)
```

Instruction et bloc d'instructions
Expression suivie d'un point virgule. L'accolade permet de créer des bloc d'instructions.

```
if( x != 0 ){
  z = y / x;
  t = y % x;
}
```

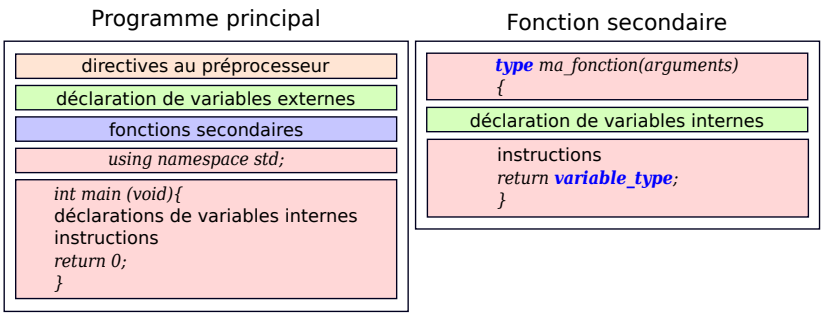
Déclaration composée

```
int a, b;
float x=1.5, y;
```

1.32

Structure classique d'un programme C++

En résumé, un programme classique et simple se présente sous la forme suivante :



1.33

Fonctions secondaires

Les fonctions secondaires sont des fonctions annexes au programme principal et qui sont souvent déclarées avant leur utilisation dans le bloc d'instruction `main` (ou la fonction secondaire l'utilisant).

```
int somme(int a, int b)
{
  int resultat;
  resultat = a+b;
  return resultat;
}

int produit(int a, int b)
{
  int resultat = 0;
  for(int i=0; i<b; i++){
    resultat = somme(resultat, a);
  }
  return resultat;
}
```

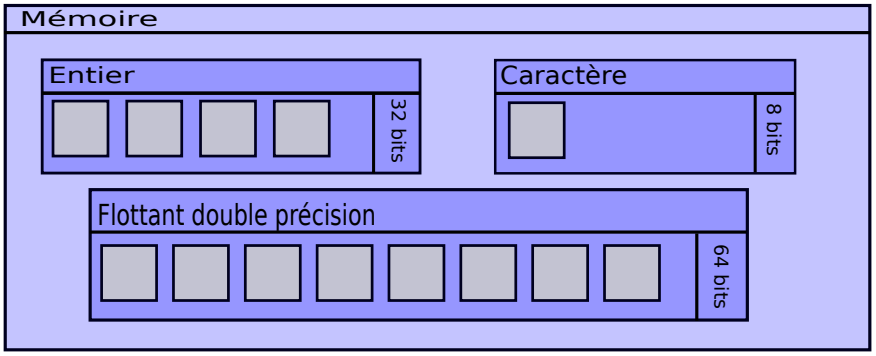
1.34

3.5 Un mot sur la mémoire

1.35

Représentation mémoire des objets

Représentation mémoire
La mémoire est une suite continue d'octets (1 octet = 8 bits). Chaque octet est identifié par son adresse qui est un entier. Deux octets contigus en mémoire ont une adresse qui diffère d'une unité. Le type définit le nombre d'octets consécutifs réservés.



1.36

Organisation de la mémoire pour un logiciel

Le langage C++ permet de gérer l'utilisation de la mémoire (langage bas niveau) avec un certain degré d'abstraction (langage haut niveau). Il permet de choisir la zone mémoire utilisée.

La pile (stack) et le tas (heap)
La totalité de la mémoire allouée à un programme est divisée en deux zones :

- La pile, mémoire statique : allocation/dés-allocation rapide car zone hiérarchisée (*first in / last out*) mais de taille limitée ;
- Le tas, mémoire dynamique : le reste, zone non hiérarchisée et donc plus lente mais de taille bien plus importante.

Par exemple, sur une machine Linux moderne, la taille par défaut de la pile, définie par l'OS, est de 8MB (commande "*ulimit -a*"), le tas « peut » occuper tout le reste.

1.37

Gestion de la mémoire en C++, vie et mort d'une entité

Sans entrer dans les détails et en simplifiant, la durée de vie d'une entité dans le code est définie par la méthode de déclaration (et donc sa nature) :

- Un type de base ou une classe d'objets existe au sein d'un bloc d'instruction (`{}`). On appelle cet espace sa « portée » (scope). Ils existent dans la pile ;
- Toute entité créée par l'instruction `new` est créée dans le tas, n'est pas limitée par un bloc d'instruction et doit être détruite par l'instruction `delete` ;
- Tout ce qui n'est pas déclaré dans un bloc ou tout ce qui n'a pas été détruit par `delete` n'est désalloué qu'à la fin du programme.

En pratique
Généralement le tas est utilisé pour stocker les instances des classes d'objet, mais pas toujours. Attention lors de l'usage des `new` à ne pas oublier de `delete` (fuite mémoire).

1.38

4 Les bases du langage C++

4.1 Les types de base

1.39

C++ : Un langage typé

Langage typé ?
Le C++ est un langage typé, c-à-d que toute variable, constante ou fonction est d'un type précis qui définit sa représentation en mémoire.

Les types de base en C/C++

- les entiers (mot-clé `int`);
- les flottants (mot-clé `float` ou `double`);
- les caractères (mot-clé `char`);
- les booléens (mot-clé `bool`).

1.40

Un mot sur les espaces de noms

Espaces de noms
Tous les objets du C++ sont distribués dans des espaces de noms (*namespace*). On peut les voir comme des boîtes nommées regroupant un jeu de fonctions, variables, classes,...

La qualification explicite, mot-clé `using` et opérateur `::`
`using` permet d'intégrer des espaces de nom dans l'espace global statique (le fichier). L'opérateur de qualification explicite `::` doit être utilisé dans le cas contraire.

```
1  using namespace std;
2  int main(){
3    cout<<"Coucou ?"<<endl;
4    return 0;
5  }
```

À la place de :

```
1  int main(){
2    std::cout<<"Coucou ?"<<std::endl;
3    return 0;
4  }
```

1.41

Le type entier : `int`

Le type `int`
Il existe en 3 versions :

- `short (int)` (au plus la taille d'un int)
- `int`
- `long (int)` (au moins la taille d'un int)

Gestion du signe
Usuellement, le bit de poids fort (premier) est réservé pour définir le signe. Les 3 versions d'`int` peuvent être préfixées du mot-clé `unsigned`, qui définit alors un entier positif. Le bit fort est alors utilisé pour représenter l'entier (on gagne 1 bits).

Mot-clé `sizeof`
`sizeof(`*type*`)` permet d'obtenir un entier correspondant aux nombre d'octets utilisés pour représenter le *type*.

1.42

Représentation en mémoire

Représentation d'un `int`

Si l'entier est signé, le bit de poids fort est réservé pour le signe. Les autres bits permettent de représenter l'entier en base 2. Le nombre de bits utilisé n'est pas exactement spécifié dans la norme mais un `int` est codé sur au moins 16bits. Exemple : Entier « 12 » codé sur 32bits *0 0...(25 fois 0)...01100*

Plage de représentation d'un `int`

Sur un ordinateur de nos jours (Unix64 - LP64 data model) :

`short int`	$[-2^{15};2^{15}[$	16 bits	2 octets
`int`	$[-2^{31};2^{31}[$	32 bits	4 octets
`long int`	$[-2^{63};2^{63}[$	64 bits	8 octets
`unsigned short int`	$[0;2^{16}[$	16 bits	2 octets
`unsigned int`	$[0;2^{32}[$	32 bits	4 octets
`unsigned long int`	$[0;2^{64}[$	64 bits	8 octets

1.43

Le type flottant : `float`

Le type `float`

Il existe en 3 versions qui diffèrent par leur précision :

- `float` (simple précision)
- `double` (double précision)
- `long double` (précision étendue)

Représentation des flottants

Ils représentent de manière **approchée** les nombres réels sous la forme :

$$(-1)^s M \cdot B^E$$

où s définit le signe, M est la mantisse, B la base (2 ou 16) et E l'exposant. Seuls signe, mantisse et exposant sont stockés dans les octets de représentation.

1.44

Le type flottant : `float` (suite)

Particularités de représentation

- Le nombre de bits utilisés n'est pas exactement spécifié dans la norme C++. Une norme existe pour les flottants en informatique (IEE 754) où le `float` du C++ est codé sur 32bits (*binary32*).
- L'exposant E est toujours codé comme positif. Un décalage de valeur $2^{\text{size}(E)-1}-1$ est utilisé pour retrouver la valeur (pouvant être négative).
- Des valeurs particulières de la mantisse M et de l'exposant E permettent de définir les infinis, le zéro exact et le `NaN` (Not a Number).

Plage de représentation des `float`

Type	Expo.	Mant.	Plage	Significatifs
`float`	8	23	$\pm 3.4 \cdot 10^{\pm 38}$	≈ 7
`double`	11	52	$\pm 1.7 \cdot 10^{\pm 308}$	≈ 16
`long double`	15+	64+	$\pm 1.2 \cdot 10^{\pm 4932}$	≈ 20

1.45

Les types `char` et `bool`

Le type `char`

Le type `char` est codé sur 1 octet.

```
char c = 'h';
cout<<"c=␣"<<c<<endl;
```

Il est possible de représenter une chaîne de caractères avec le type `string` nécessitant la directive `#include <string>`.

```
string s;
s = "L'écriture␣!";
```

Le type `bool`

Le mot-clé `bool` désigne le booléen prenant valeur `False` (0) ou `True` (entier $\neq 0$). Il est codé sur un seul octet (un bit suffirait mais n'est pas adressable).

1.46

Exemple

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string s1, s2, s;
   cout<<"Entrer une chaîne de caractères\n";
   cin>>s1;
   cout<<"Entrer une autre chaîne de caractères\n";
   cin>>s2;
   s = s1 + s2;
   cout<<s<<endl;
   return 0;
}
```

1.47

4.2 Les constantes

1.48

Définition et types

Une constante ?
Une constante est une expression littérale dans le code (« en dur ») qui servira souvent à l'initialisation d'une variable pour une partie non interactive du code.

Types de constantes
Le type de la constante dépend de la façon dont elle est écrite lors de sa définition. Il en existe de quatre types :

- entière ;
- flottante ;
- caractère ;
- chaîne de caractères.

En pratique
La constante est la partie d'une expression que l'on rencontre généralement à droite du signe égal (lors d'une affectation), ou en argument.

1.49

Constante entière

Choix de la base
Il existe 3 bases différentes définies par le préfixe utilisé :

- base 10, classique (et défaut) : `int x = 42;`
- base 8, préfixé par un `0` : `int x = 052;`
- base 16, préfixe par `0x` (ou `0X`) : `int x = 0x2a`

Choix du format
Le format de représentation est choisi à l'aide d'un suffixe :

- suffixe `u` (ou `U`) pour `unsigned` : `x = 18U + 24u;`
- suffixe `l` (ou `L`) pour `long` : `x = 15 + 27l;`
- suffixe `ll` (ou `LL`) pour `long long` : `x = 15 + 27LL;`

Ces suffixes ont leur utilité lors de passage de types implicite (« cast »).

En pratique
On utilise principalement le type fondamental `int` en base 10.

1.50

Constante flottante

Expression
La constante flottante est exprimée par l'usage d'un point et/ou d'un `e` (ou `E`) dans l'expression.

Choix du type
Le type par défaut est `double`. On peut, lui aussi, le choisir grâce à un suffixe :

- Type double : x = 3.0 + 3e1 (3 et 30 en flottant)
- Type float : x = 3f ou x = 3F
- Type long double : pi = 3.14159L

En pratique
On rencontre très souvent le type `float` en calcul si l'apport de la double précision du format `double` est inutile (gain de mémoire).

1.51

Constantes caractères

Caractère et chaîne de caractères
Un caractère s'obtient à l'aide d'apostrophes, la chaîne de caractères avec l'apostrophe double.

```
char toto = 't';
string test = "Hello !";
```

Écriture et caractères non-imprimables
Tous les caractères ont aussi un code octal ou hexa (ex : `\64` ou `\x40` pour « @ »). Quelques caractères disposent d'une notation simplifiée (appelés « escape code ») :

`\n`	newline	`\a`	alert (beep)
`\r`	carriage return	`\'`	single quote (')
`\t`	tab	`\"`	double quote (")
`\v`	vertical tab	`\?`	question mark (?)
`\b`	backspace	`\\`	backslash (\)
`\f`	form feed (page feed)		

1.52

4.3 Les opérateurs

1.53

Opérateur d'affectation, conversion implicite

L'affectation (« assignement ») ou opérateur =

```
Variable = expression;
```

Le terme de gauche peut être une variable simple, un élément de tableau mais pas une constante !

```
1  #include <iostream>
2  using namespace std;
3  int main(){
4    int i, j = 2;
5    float x = 2.5;
6    i = j + x;
7    x = x + i;
8    cout<<"x= "<<x<<endl;
9    return 0;
10 }
```

Le code affichera `x= 6.5`, une conversion implicite a lieu en ligne 6 vers le type `int` de la variable `i` :
`i = int(j + x);`

1.54

Les opérateurs arithmétiques

Liste des opérateurs standards

— l'opérateur unaire de signe : `-`

— les opérateurs binaires : `+` , `-` , `*` , `/`

Attention à la division d'`int`, elle retourne un `int` !

L'opérateur binaire modulo `%`

Il n'est valable que sur des opérations entières. Le signe du résultat est (en général) celui du dividende. L'opération suivante retournera `-1` :

```
x = -5 % 2;
```

La puissance

L'opérateur de puissance n'existe pas ! Il faut employer la fonction `pow(x,y)` nécessitant la directive `#include <cmath>`.

1.55

Les opérateurs relationnels

Syntaxe

Opérateurs binaires agissant sur 2 expressions et retournant un booléen suivant la syntaxe suivante :

```
Expression_1 (op.) Expression_2
```

Les opérateurs relationnels disponibles sont :

`>` , `>=` , `<` , `<=` , `==` , `!=`

```
1 int main(){
2   int a = 0;
3   int b = 1;
4   if(a=b)
5     cout<<"a et b sont égaux\n";
6   else
7     cout<<"a et b sont différents\n";
8   return 0;
9 }
```

La réponse sera `"a et b sont égaux"`, il ne faut pas confondre l'opérateur = et == ! Ici l'affectation étant réussie (les valeurs ne sont PAS comparées), l'opération retourne un booléen `True`.

1.56

Les opérateurs logiques

Syntaxe

```
Expr_1 (op.) Expr_2 (op.) Expr_3
```

Trois opérations logiques sont disponibles :

`&&` , `||` , `!`

Principe d'évaluation et "Short-circuit evaluation"

L'évaluation de gauche à droite. Stop dès que résultat défini.

```
if((i >= 0) && (i <= 9) && !(P[i] == 0))
```

Dernière condition ignorée si $i \notin [0,9]$

Opérations logiques

Opérateur `&&`

a	b	a && b
True	True	True
True	False	False
False	True	False
False	False	False

Opérateur `||`

a	b	a || b
True	True	True
True	False	True
False	True	True
False	False	False

1.57

Opérateurs logiques bit à bit

Définition
Permettent de manipuler les entiers (`short`, `int`, `long`) au niveau du bit.

&	ET logique	|	OU inclusif
∧	OU exclusif	∼	complément à 1
<<	décalage à gauche	>>	décalage à droite

Expression	Binaire	Décimal
a	01001101	77
b	00010111	23
a & b	00000101	5
a | b	01011111	95
a ∧ b	01011010	90
∼a	10110010	178
b<<2	01011100	92
b<<5	11100000	112
b>>1	00001011	11

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

∧	0	1
0	0	1
1	1	0

1.58

Les opérateurs d'affectation composée

Principe de fonctionnement

```
expr_1 (op.)= expr_2
```

est équivalent à :

```
expr_1 = expr_1 (op.) expr_2
```

Opérateurs disponibles :

```
+= , -= , *= , /= , %= ,
&= , ∧= , |= , <<= , >>=
```

```
1  int i = 5;
2  i+=3;
3  cout<<"i = "<<i<<endl;
```

Ce bloc d'instruction retourne `i = 8`.

1.59

Opérateurs d'incrémentation et de décrémentation

Utilisation

```
++ , --
```

Incrémentent ou décrémentent la variable associée d'une unité. S'utilisent en suffixe (i++) ou en préfixe (++i).

— Le suffixe retourne l'ancienne valeur (retourne puis incrémente) ;

— Le préfixe retourne la nouvelle valeur (incrémente puis retourne).

```
1  int i = 5, a;
2  a = i++;
3  cout<<"a = "<<a<<endl;
4  cout<<"i = "<<i<<endl;
```

Retourne `a = 5` et `i = 6`.

```
1  int i = 5, a;
2  a = ++i;
3  cout<<"a = "<<a<<endl;
4  cout<<"i = "<<i<<endl;
```

Retourne `a = 6` et `i = 6`.

1.60

Autres opérateurs

Opérateur virgule : ,
Exécution de gauche à droite. La dernière expression compte pour l'évaluation.

```
a = (b=3, b+2);
```

retourne `a = 5` et `b = 3`.

Opérateur conditionnel ternaire : `: ?`

```
x >= 0 ? x : -x;
m = ((a > b) ? a : b);
```

codent respectivement "valeur absolue" et "maximum".

Opérateur conversion de type
Appelé `cast`, il permet de modifier le type d'un objet.

```
int i=3, j=2;
cout<<(float)i/j<<endl;
```

1.61

Règle de priorité des opérateurs

::
() [] -> .
! ~ ++ -- - &(adresse) *() new/delete sizeof()
.* ->*
* / %
+ -(binaire)
<< >>
< <= > >=
== !=
& (ET bit à bit)
^
|
&&
||
? :
= += -= *= /= %= &= ^= |= <<= >>=
,

1.62

4.4 Les instructions de contrôle

1.63

`if...else`

Syntaxe

```
1  if (expression_1)
2    instruction_1
3  else if(expression_2){
4    instruction_2_1
5    instruction_2_2
6    instruction_2_3
7  }
8  else if(expression_3)
9    instruction_3
10 else
11   instruction_4
```

Imbrications de `if`
Le `else` s'applique au premier `if` précédent non attribué.

```
1 if (a <= b)
2   if(b <= c)
3     cout<<"ordonné\n";
4   else cout<<"non ordonné\n";
```

1.64

Instruction `switch`

Syntaxe

```
switch (expression_etiquette){
  case constante_1:
    instructions_1
    break;
  case constante_2:
    instructions_2
    break;
  ...
  case constante_n:
    instructions_n
    break;
  default:
    instructions_def
}
```

Explications

— Évalue `expression_etiquette` (type `int` ou `char`).

— Effectue un test logique == successivement sur les constantes énumérées par les `case`.

— Exécute `default` ssi. aucun test n'est `True`.

1.65

Exemple de switch

```
using namespace std;
int main(){
  int n;
  cout<<"Entrer un entier: ";
  cin>>n;
  switch(n){
    case 0: cout<<": nul\n"<<endl;
            break;
    case 1: cout<<": un\n"<<endl;
            break;
    case 2: cout<<": deux\n"<<endl;
            break;
    default: cout<<": trop grand\n"<<endl;
  }
}
```

On peut utiliser des `char` et les comparer à des `int`, ils sont systématiquement convertis en `int`.

1.66

4.5 Les boucles

1.67

Boucle `while`

Principe

Tant qu'une condition est vérifiée (est `True`), un bloc d'instruction est exécuté. On sort de la boucle quand la condition est `False`. Le bloc n'est jamais exécuté si la condition est `False` au départ.

Syntaxe

```
int i=1;
while (i < 10){
  cout<<"i= "<<i<<endl;
  i++;
}
```

1.68

Boucle `do...while`

Principe

Variante de la boucle `while` avec test après exécution du bloc d'instruction. L'instruction sera exécutée au moins une fois.

Syntaxe

```
int a;
do {
  cout<<"Entrer un entier : ";
  cin>>a;
  cout<<endl<<"L'entier est : "<<a<<endl;
}
while((a <= 0) || (a>10));
```

1.69

Boucle `for`

Principe

```
for(expr_1 ; expr_2 ; expr_3){
  instructions
}
```

est l'équivalent des instructions :

```
expr_1;
while(expr_2){
  instructions
  expr_3;
}
```

Exemple :

```
for (int i=0 ; i < 10 ; i++){
  cout<<"i= "<<i<<endl;
}
```

À la fin de la boucle, `i` vaut 10.

1.70

`break` non conditionnel

Principe

L'instruction `break` permet de sortir d'un structure de contrôle ou d'une boucle indépendamment de l'état du test. `break` fait sortir de la boucle la plus interne.

```
using namespace std;
int main(){
  for(int i=0 ; i < 5 ; i++){
    cout<<"i= "<<i<<endl;
    if(i == 3)
      break;
    }
    cout<<"Valeur de i en sortie : "<<i<<endl;
  }
```

1.71

`continue` non conditionnel

Principe
L'instruction `continue` permet de passer directement au tour de boucle suivant. Le reste des instructions de la boucle est ignoré pour ce tour.

```
#include <iostream>
using namespace std;
int main()
{
  for(int i=5 ; i > 0 ; i--)
    {
      if(i < 3)
        continue;
      cout<<i<<", ";
    }
  cout<<"Décollage !"<<endl;
}
```

1.72

5 Utilisation avancée

5.1 Retour sur les types

1.73

Synonyme de type

Type
Un type doit être vu comme une garantie : Il indique ce qu'est une variable et surtout, ce qu'on peut en faire.

`typedef`
On peut déclarer un "synonyme" avec l'instruction `typedef`.

```
typedef unsigned long int ulong;
ulong i = 1;
```

En pratique
Jusqu'ici, les types sont plutôt courts. Nous verrons, après l'introduction de la POO, qu'ils peuvent vite devenir... encombrants. Dans ces situations, `typedef` est utilisé, ou plutôt sa version moderne `using`.

1.74

Conversion des types

Conversion et promotion

- Implicite : Le type peut être adapté par le compilateur (danger de perte d'information)
- Explicite : Le préfixe `(type) var` permet de forcer un type `type` sur la variable `var`, on parle de « type-casting »

Initialisation universelle et uniforme
`{..}` permet d'initialiser "sans risque" de conversion

```
double taille = 5;  // 5 est un int
int l(5.2);
int l{5.2};         // erreur
int a{3};
double b = a + 2.5; // addition int + double
```

Conversion plus complexe de type
Nous verrons en POO des opérateurs de conversion de type bien plus explicites comme `static_cast<int>`.

1.75

5.2 Types composés

1.76

Les tableaux

L'opérateur de déclaration de tableau : suffixe `[]`
La plupart des variables peuvent être déclarées en tableau. Ce tableau hérité du C (« C-array ») est statique : Sa taille doit être connu à la déclaration.

```
int taille = 12;
double mois[taille];
```

"Inconvénient"
Nous verrons plus loin comment manipuler des pointeurs, ces pointeurs permettent aussi de manipuler les tableaux C.

En pratique
Héritage du langage C, ces tableaux sont très « primitifs ». Si le choix est donné, il est préférable d'utiliser des conteneurs de la STL comme `std::vector` ou `std::array` que nous présenterons plus tard.

1.77

Les tableaux (suite)

Accès aux données
L'accès aux éléments du tableau se fait par l'opérateur `[]`. Ne pas le confondre avec l'utilisation de `[]` pour créer le tableau !

Contrôle des bornes
Comme souvent en informatique, la numérotation commence à 0. Attention, `mois[12]` n'existe donc pas ! Il est cependant possible de compiler le code avec ce genre d'erreur...

```
1 int taille = 12;
2 double mois[taille];
3
4 mois[0] = 123.45;
5 mois[11] = 999.99;
6
7 mois[45] = 33.33;  // Compile mais plante
```

1.78

Les tableaux (fin)

Initialisation
Les tableaux sont déclarés non initialisés. Deux méthodes :

- Par une boucle `for`
- Par une liste d'initialisation `{ , , }`

Initialisation spéciale
La liste d'initialisation permet un raccourci : Elle donne automatiquement la taille.

```
1 // Boucle
2 int taille = 12;
3 double salaire[taille];
4 for(int i=0; i<taille; ++i)
5   salaire[i] = 2000.0;
6
7 // Liste d'initialisation
8 double prix[3] = {152.5, 33.58, 1978.33};
9 int toto[] = {2, 5, 17}; // Taille implicite
```

1.79

Adresse et indirection (dereference)

Préfixe & (d'une variable)
Cet opérateur retourne « l'adresse mémoire » d'une variable

Préfixe ⋆ (d'une variable)
Permet de suivre une adresse contenu dans une variable et d'accéder au contenu stocké.

Le pointeur : suffixe ⋆ (d'un type)
Une variable contenant une adresse est nommé un pointeur. Il est nécessaire d'indiquer le type pointé.

```
1 double a = 55.5;
2
3 double* pa = &a;
4 double *pa2 = &a;   // Possible
5 double * pa3 = &a; // Possible (aussi)
6
7 double b = *pa +1;
```

1.80

Pointeur et tableaux

Le vrai visage du tableau
La déclaration d'un C-array n'est en fait qu'un pointeur ! Il est plus « pratique » [1] d'indiquer le type des données du tableau et où les trouver. Un tableau est donc un pointeur sur le premier élément du tableau.

```
1 int a[] = {1, 2, 3, 4, 5};
2 int* pa = a; int* pa2 = &a[0];
3
4 int b = *(a+1); b++; a[2] = b+10;
5 pa = &a[3]; *pa = 5;
6 pa++; *pa = 6;
```

En pratique
On s'en sert rarement, on utilise des conteneurs de la STL (pas forcément plus simples mais moins sujets aux erreurs).

1.81

Pointeurs en folie

Pointeurs de pointeurs
Il est possible de créer des pointeurs de pointeurs.

```
char a; a = 'z';
char * b = &a; char ** c = &b;
```

Pointeur universel
`void` est un type fondamental particulier, il est possible de créer un « void pointer » qui peut stocker une adresse de type quelconque. Aucune indirection possible.

Pointeur nul
Le mot clef `nullptr` permet d'indiquer qu'un pointeur ne pointe sur rien (`NULL` dans les codes anciens).

```
int * p = 0;
if (p == nullptr) cout << "vide␣!";
```

1.82

1. Définition de « pratique » sujet à caution et n'impliquant que les informaticiens concernés. Nul n'est tenu des dommages encourus, à utiliser à vos risques et périls. Ni échangé, encore moins remboursé.

Définir un contenu constant

Définir une "constante"

— `constexpr` : La constante est connue à la compilation

— `const` : La constante est connue à l'entrée dans la portée

En pratique

— `constexpr` est apparu avec le C++11. Il est donc encore assez rare mais il est souvent le meilleur choix.

— Déclarer des variables constantes permet d'éviter de faire apparaître des « magic numbers » dans son code et les remplacer par des identificateurs explicites (ex : 3.14159, 299792458)

Et les fonctions ?

— `constexpr` peut être associé à une fonction (notion relativement avancée et peu utile aux débutants)

— On verra en POO un sens supplémentaire à `const`

1.83

Pointeur constant ou pointeur de constante ?

```
1 int x;
2 // non-const pointer to non-const int
3       int *       p1 = &x;
4 // non-const pointer to const int
5 const int *       p2 = &x;
6 // const pointer to non-const int
7       int * const p3 = &x;
8 // const pointer to const int
9 const int * const p4 = &x;
```

Le Bon, la Brute et le Truand

Le premier `const` (du type) a le même sens préfixé ou suffixé :

```
const int * p2a = &x;
int const * p2b = &x;
```

En pratique

Tout le monde s'y prend au moins à trois fois...

1.84

5.3 Les fonctions

1.85

Définitions

Construction

```
type nom (param1, param2, ...)
{ instructions }
```

— `type` : Type de ce que retourne la fonction

— `nom` : Identificateur d'appel

— `paramX` : Liste ordonnée des types des arguments

— `instructions` : Corps d'exécution de la fonction

Les fonctions sont caractérisées par l'usage de l'opérateur d'appel `operator()` (call operator) en suffixe d'une variable.

Signature, surcharge

— Le couple `nom` et `liste` des paramètres constitue la signature d'une fonction, elle doit être unique.

— Une fonction ayant même `nom` mais une `liste` d'argument différente est appelée une surcharge de la première fonction (exemple `operator+`).

1.86

Exemple

```cpp
#include <iostream>
using namespace std;

int addition(int a, int b){
  return a+b;
}

double addition(double a, double b){
  return a+b;
}

int main(void)
{
  cout << addition(2, 5) << endl;
  double a = 2.5;  double b = 6.5;
  cout << addition(a, b) << endl;

  return 0;
}
```

1.87

Conseils en vrac

Pourquoi des fonctions ?

- La fonction est l'unité algorithmique qui permet de structurer la résolution d'un problème
- Définition plus concrète : Unité de découpage exprimant une tâche simple
- Il est conseillé d'essayer de limiter ses fonctions à « un écran » de longueur et de subdiviser si ça devient trop long
- Recyclez le travail : si un bout de code apparaît plus de 3 fois, transformez le en fonction
- Le nom des arguments n'a aucune importance, leur ordre et leur type est essentiel
- Profitez de la surcharge : À tâche similaire, nom similaire !
- Le compilateur ne vérifie rien sur la tâche d'une surcharge, à vous de conserver un « sens » similaire

1.88

5.4 Passage des arguments

1.89

Copier l'argument

Passage par valeur / Pass-by-value

```cpp
int mafonction(int a){}
```

On clone le contenu de la variable donnée en argument, elle ne sera donc pas modifiée par la fonction. On parle d'une copie locale de la variable (construction par copie cf. plus tard).

```cpp
int mafonction(int premier){
  premier += 2;
  return premier; }

int main(void){
  int a = 5;
  int b = mafonction(a);
  cout << a << endl; // 5
  cout << b << endl; // 7

  return 0;
  }
```

1.90

On travaille sur l'original...

Passage par référence `&` / Pass-by-ref

```
int mafonction(int& a){}
```

On travaille directement sur la variable donnée en argument.

— Extraire plusieurs résultats d'une fonction

— Éviter de copier un argument trop lourd

```
1  void echanger(int& premier, int& second){
2    int temp = premier;
3    premier = second; second = temp;
4  }
5
6  int main(void){
7    int a = 5; int b = 10;
8    echanger(a,b);
9    cout << a << endl; // 10
10   cout << b << endl; // 5
11   }
```

1.91

Mais promis, on ne va rien casser

Passage par référence constante `const &` / Pass-by-const-ref

```
int mafonction(const int& a){}
```

On travaille avec l'original mais on garantit de ne pas modifier le contenu.

— Le compilateur refuse une modification

— On se passe aussi de faire la copie

```
1  void imprimer(const int& premier,
2                const int& second){
3    cout << "Mon premier est " << premier;
4    cout << " et mon second " << second << endl;
5  }
6
7  int main(void){
8    int a = 5; int b = 10;
9    imprimer(a,b);
10   }
```

1.92

Économie d'argument

Valeurs par défaut / Default value

```
double toto(double a, double b=2){}
```

On définit une valeur par défaut d'un argument et autorise d'appeler la fonction sans l'indiquer

— Permet d'alléger les écritures

— Ne fonctionne que sur les X derniers arguments de la liste

```
1  double diviser(const double& a=1,
2                 const double& b=2){
3    return a/b;
4  }
5
6  int main(void){
7    double a{5}, b{10};
8    cout<< diviser();      // 0.5
9    cout<< diviser(10);    // 5
10   cout<< diviser(12,3); // 4
11   }
```

1.93

Et un tableau ?

Tableau comme argument

```
void impList(int v[][2][5], int taille)
```

— On passe l'adresse et le type, la taille réelle est inconnue

— Seule la dimension externe est inconnue

```
1 void impList2D(int v[][2], int taille){
2   for(int i=0; i<taille; ++i){
3       for(int j=0; j<2; ++j)
4         cout << v[i][j]<< ' ';
5       cout << endl;
6   }
7 }
```

En pratique
On note encore une fois l'aspect archaïque du C-array. On se sert ici de l'équivalence tableau/pointeur.

1.94

Règles générales

Quel type de passage ?

— Pass-by-value : petits arguments

— Pass-by-const-ref : objets plus larges sans modification

— Utiliser la valeur de `return` plutôt qu'une référence

— Pass-by-ref si on n'a pas le choix

En pratique
Le troisième point indique qu'on préfère utiliser l'argument de `return` plutôt qu'une fonction sans retour (`void`) avec un argument par référence.

Cas où on n'a pas le choix

— Objets pouvant changer en interne (ex : mémoire pour les conteneurs comme `std::vector`)

— Fonctions modifiant plusieurs arguments (un seul `return` disponible)

1.95

5.5 Déclaration et définition

1.96

« Dis moi qui tu es, et je te dirai... »

Déclaration
Instruction qui introduit un nouvel identificateur.

— Elle spécifie le type de l'identificateur (de la variable de retour pour une fonction)

— En option, elle l'initialise (avec une valeur ou le corps pour une fonction)

Définition
Une définition spécifie complètement un objet, elle « définit » l'espace mémoire à réserver et ce qu'il contient, elle spécifie les actions qu'effectue une fonction.

En pratique

— Toutes les définitions sont des déclarations mais l'inverse n'est pas toujours vrai (ex : double sqrt(double))

— Il est possible d'avoir plusieurs déclarations mais qu'une seule définition.

1.97

Déclarations de fonctions

Notion de prototype
Une fonction doit être déclarée avant de pouvoir être utilisée, mais sa définition peut venir plus tard, voire après ses appels. On parle alors de « forward declaration », de pré-déclaration ou de prototype. Il n'est pas nécessaire d'y nommer les arguments.

```
1  // Declaration
2  double carre(double);
3
4  int main(void){
5    // Appel
6    cout << carre(25.3) << endl;
7    return 0;
8  }
9
10 // Definition (aussi declaration)
11 double carre(double a){
12   return a*a;
13 }
```

1.98

Organisation des fichiers

Fichier en-tête : `.h`
Ces fichiers sont prévus pour regrouper les déclarations

Fichier source : `.cpp`
Ces fichiers regroupent les définitions (fonctions, classes)

En pratique
On utilise la directive `#include` pour inclure les déclarations dans :

- Le fichier de définition (cahier des charges)
- Les fichiers utilisant les objets déclarés

Unité de compilation
Les fichiers en-tête permettent de distribuer les déclarations. Chaque unité de compilation saura que « les fonctions existent et sont définies quelque part ».

1.99

Un mot sur les en-tête

Programmation modulaire
Les en-tête permettent d'organiser son code en regroupant les objets par "thème" : modules ou librairies.

Include Guards
Les inclusions multiples et répétées posent un soucis de compilation quand on ajoute "quelques" définitions "thématiques". On se sert alors de macros nommées « Include Guards » composées des directives `#ifndef`, `#define` et `#endif`.

```
1  #ifndef MESMATHS_H
2  #define MESMATHS_H
3
4  double sqrt(double, int i = 2);
5  double carre(double);
6  int carre(int);
7  double addition(double, double);
8
9  #endif  // MESMATHS_H
```

1.100

Notion de portée

Portée / Scope
Une déclaration introduit un nom associé à une portée.

- Global scope : Sur le fichier
- Namespace scope : Une portée dotée d'un nom (ex : `std`)
- Class scope : Portée d'une classe (cf. POO)
- Local scope : entre une paire `{}` ou une liste d'arguments
- Statement scope : dans une instruction (ex : `for`)

En pratique

- On utilise souvent l'indentation afin de marquer visuellement les différents niveaux de portée. Un IDE l'effectue presque toujours automatiquement.
- Certaines portées sont invisibles (exemple une boucle `if` d'une ligne)

1.101

Résolution de nom

Espace de nom : `namespace`
Portée dotée d'un nom afin de regrouper des fonctions, des variables, des objets. L'opérateur de résolution de nom `::` permet de spécifier le `namespace` visé.

Directive `using`
Permet d'introduire un objet ou un espace de nom dans l'espace de nom global et de se passer de l'opérateur `::`

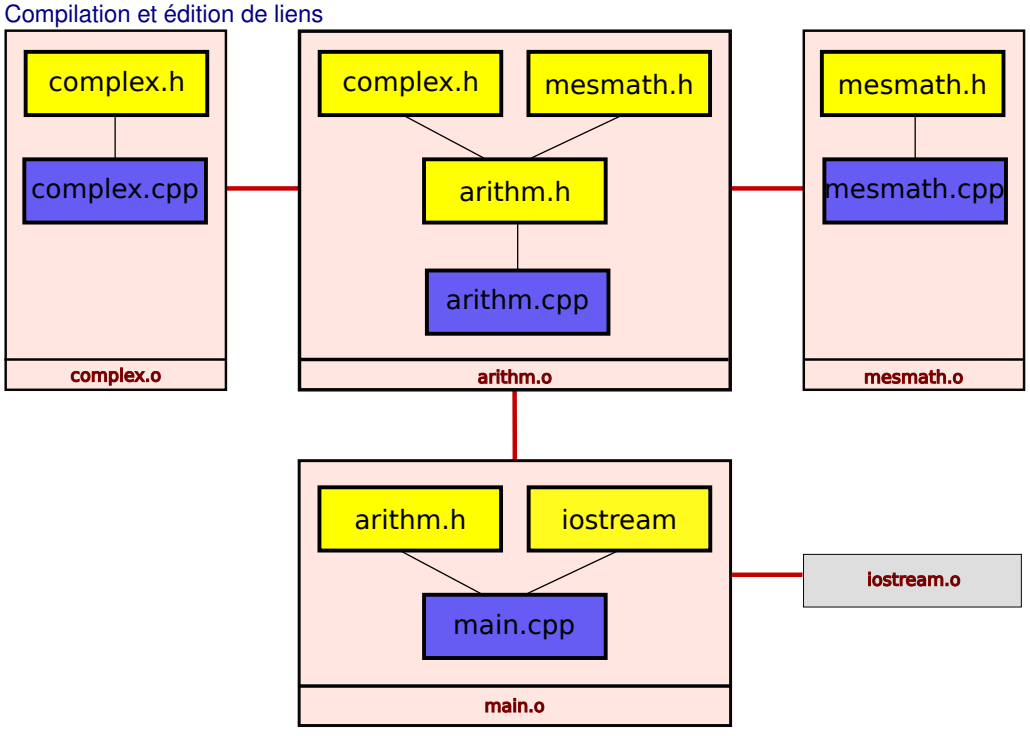
```
1  #include <iostream>
2
3  using std::cout;
4
5  int main(void){
6    double variable;
7    cout << "Entrez un nombre :" << endl;
8    std::cin >> variable;
9
10   return 0;
11 }
```

1.102

5.6 Retour sur la compilation

1.103

Compilation et édition de liens



1.104