Information, Calcul, Communication
FICHES RÉSUMÉ DE C++

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

EPFL

Variables

En C++, une **valeur** à conserver est stockée dans une variable caractérisée par :

- son **type**
- et son **identificateur** ;

(définis lors de la **déclaration**)

La *valeur* peut être définie une première fois lors de l'**initialisation**, puis éventuellement modifiée par la suite.

Rappels de syntaxe :

type nom ;	(déclaration)
type nom(valeur);	(initialisation)
nom = expression ;	(affectation)

Types élémentaires :

```
int
double
char
bool
```

Exemples :

```
int val(2) ;
const double z(x+2.0*y);
```

C++11

```
constexpr double pi(3.141592653);
i = j + 3;
```


EPFL

Opérateurs

Opérateurs arithmétiques

`*`	multiplication	
`/`	division	
`%`	modulo	
`+`	addition	
`-`	soustraction	
`++`	incrément	*(1 opérande)*
`--`	décrément	*(1 opérande)*

Opérateurs de comparaison

`==`	teste l'égalité logique
`!=`	non égalité
`<`	inférieur
`>`	supérieur
`<=`	inférieur ou égal
`>=`	supérieur ou égal

Opérateurs logiques

`and`	`&&`	« et » logique	
`or`	`||`	ou	
`not`	`!`	négation	*(1 opérande)*

Priorités (par ordre décroissant, tous les opérateurs d'un même groupe sont de priorité égale) :
`not ++ --`, `* / %`, `+ -`, `< <= > >=`, `== !=`, `and`, `or`

Les structures de contrôle

les branchements conditionnels : *si ... alors ...*

```
if (condition)
    instructions
.....................
if (condition 1)
    instructions 1
...
else if (condition N)
    instructions N
else
    instructions N+1
```

```
switch (expression) {
   case valeur:
       instructions;
       break;
   ...
   default:
       instructions;
}
```

les boucles conditionnelles : *tant que ...*

```
while (condition) {
    Instructions
}
```

```
do {
    Instructions
} while (condition);
```

les itérations : *pour ... allant de ... à ...* , *pour ... parmi ...*

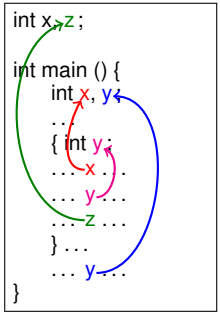
```
for (initialisation ; condition ; increment)
    instructions
```

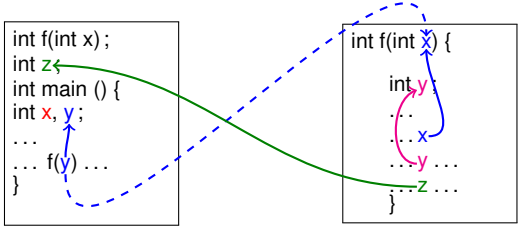
```
for (déclaration : valeurs)
    instructions
```

les sauts : `break;` et `continue;`

Note : `instructions` représente une instruction élémentaire ou un bloc.
`instructions;` représente une suite d'instructions élémentaires.

Portée / Appel





Les fonctions

Prototype (à mettre **avant** toute utilisation de la fonction) :

```
type nom ( type1 arg1, ..., typeN argN [ = valN ] );
```

type est `void` si la fonction ne retourne aucune valeur.

Définition :

```
type nom ( type1 arg1, ..., typeN argN )
{
    corps
    return valeur;
}
```

Passage par **valeur** :
type `f(type2 arg);`
arg ne peut pas être modifié par *f*

Passage par **référence** :
type `f(type2& arg);`
arg peut **être modifié** par *f*

Surcharge (exemple) :

```
void affiche (int arg);
void affiche (double arg);
void affiche (int arg1, int arg2);
```


Jean-Cédric Chappelier
& Jamila Sam
EPFL

Les chaînes de caractères


```
#include <string>
```

déclaration/initialisation : `string identificateur("valeur");`

Affectation :
```
chaine1 = chaine2;
chaine1 = "valeur";
chaine1 = 'c';
```

Concaténation :
```
chaine1 = chaine2 + chaine3;
chaine1 = chaine2 + "valeur";
chaine1 = chaine2 + 'c';
```

Accès au (i+1)-ème caractère : `chaine[i]` ;

Fonctions spécifiques :

taille :	`chaine.size()`
insertion :	`chaine.insert(position, chaine2)`
replacement :	`chaine.replace(position, longueur, chaine2)`
suppression :	`chaine.replace(position, longueur, "")`
sous-chaîne :	`chaine.substr(position, longueur)`
recherche :	`chaine.find(souschaine)`
	`chaine.rfind(souschaine)`

valeur « pas trouvé » d'une recherche : `string::npos`

Les tableaux de taille fixe


```
#include <array>
```

Déclaration : `array<type, taille> identificateur;`
Déclaration/Initialisation :

`array<type, taille> identificateur = {val`$_1$`, ... , val`$_{taille}$`};`

Accès aux éléments : `tab[i]` `i` entre **0** et **taille-1**

Fonctions spécifiques :
`size_t tab.size()` : renvoie la taille

Tableau multidimentionnel :

```
array<array<type, nb_colonnes>, nb_lignes> identificateur;
tab[i][j] = ...;
```

Les tableaux dynamiques


```
#include <vector>
```

Déclaration : `vector<type> identificateur;`
Déclaration/Initialisation :

```
vector<type> identificateur({ ... });
vector<type> identificateur = { ... };
vector<type> identificateur(taille);
vector<type> identificateur(taille, valeur);
```

Accès au (`i`+1)-ème élément (quand il existe !) : `tab[i]`

Fonctions spécifiques :
`tab.size()` : renvoie la taille (type `size_t`)
`tab.empty()` : détermine s'il est vide ou non (type `bool`)
`tab.clear()` : supprime tous les éléments
`tab.pop_back()` : supprime le dernier élément
`tab.push_back(valeur)` : ajoute un nouvel élément à la fin

Les tableaux à la C

déclaration : *type identificateur*[*taille*];
déclaration/initialisation :

type identificateur[*taille*] = { val_1, ... , val_{taille} };

Accès aux éléments : `tab[i]` `i` entre **0** et **taille-1**

Le passage *type1* `f(`*type2* `tab[]);` d'un tableau `tab` à une fonction `f` se fait automatiquement **par référence**.
Pour éviter les effets de bord :

type1 `f(const` *type2* `tab[]);`

tableau multidimentionnel :
type identificateur[*taille1*][*taille2*];
`tab[i][j];`

Les structures

Déclaration du type correspondant :

```
struct Nom_du_type {
      type1 champ1 ;
      type2 champ2 ;
      ...
};
```

Déclaration d'une variable :

```
Nom_du_type identificateur;
```

Déclaration/Initialisation d'une variable :

```
Nom_du_type identificateur = { val1, val2, ...};
```

Accès à un champ donné de la structure :

```
identificateur.champ
```

Affectation globale de structures :

```
identificateur1 = identificateur2
```

Pointeurs & références

Déclaration : `type* pointeur;`

Déclaration/Initialisation :

```
type* pointeur(adresse);
unique_ptr<type>(new type(valeur));
type& reference(objet);
```

Adresse d'une variable : `&variable`

Accès au contenu pointé par un pointeur : `*pointeur`

Allocation mémoire :

```
pointeur = new type;
pointeur = new type(valeur);
unique_ptr<type>(new type(valeur));
```

Libération de la zone mémoire allouée :

`delete pointeur` (pour les « pointeurs à la C », obligatoire)

`pointeur.reset()` (pour les « pointeurs intelligents », pas nécessaire)

EPFL

Pointeurs (avancés)


Pointeur sur une constante : `type const* ptr;`

Pointeur constant : `type* const ptr(adresse);`

Pointeur sur une fonction :

type_retour (**ptrfct*)(*paramètres...*)

C++11 `function<`*type_retour*(*paramètres...*)`>` *ptrfct*

Les entrées/sorties

Clavier / Terminal : `cin` / `cout` et `cerr`

Fichier de définitions : `#include <iostream>`

Utilisation :

- écriture : `cout << expr1 << expr2 << ... ;`
- lecture : `cin >> var1 >> var2 >> ... ;`

Saut à la ligne : `endl`

Lecture d'une ligne entière : `getline(cin, string);`

.....

Formatage :

Manipulateurs	
`#include <iomanip>`	
`cout << manip << expr << ...`	
`dec`, `oct`, `hex`	changement de base
`setprecision(int)`	nombre de chiffres à afficher
`setw(int)`	largeur de colonne nombre de caractères à lire
`setfill(char)`	caractère utilisé dans l'alignement
`cin >> ws`	saute les blancs

Options	
`setf(ios::option)`	
`unsetf(ios::option)`	
`ios::left`	alignement à gauche
`ios::showbase`	afficher la base
`ios::showpoint`	afficher toujours la virgule
`ios::fixed`	notation fixe
`ios::scientific`	notation scientifique

Les entrées/sorties (2)

Fichiers : `#include <fstream>`

Flot d'**entrée** (similaire à `cin`) : `ifstream`
Flot de **sortie** (similaire à `cout`) : `ofstream`

Création : *type_flot nom_de_flot*;

Lien (ouverture) : *flot*.open("*fichier*");


ouverture en binaire :

pour lecture : ifstream *flot*("*fichier*", ios::in|ios::binary);
pour écriture : ofstream *flot*("*fichier*", ios::out|ios::binary);

Utilisation : comme `cin` et `cout` :

flot << *expression* << ... ;
flot >> *variable_lue* >> ...;

Test d'échec de ouverture/lecture/écriture sur le flot : *flot*.fail()

Fermeture du fichier : *flot*.close()

Test de fin de fichier : *flot*.eof()

Déverminage (avec gdb)

Pour utiliser un programme de déverminage, compiler avec l'option `-g`

```
c++ -g -o monprogramme monprogramme.cc
```

Lancer le dévermineur : `gdb monprogramme` ; puis : `layout src` (dans gdb)

Démarrer mon programme dans `gdb` : `run` ou `run` *arguments*

Suspendre l'exécution du programme à des endroits précis : breakpoints :
`break` *line*
`break` *function*

Exécuter pas à pas : `next` ou `step`

Regarder le contenu d'une variable :

- soit `print` *nom_variable*
- soit `display` *nom_variable*
  La valeur de la variable est alors affichée à chaque pas de programme.

Exceptions

`throw` *`expression`*`;` lance l'exception définie par l'expression

`try { ... }` introduit un bloc sensible aux exceptions

`catch (`*`type nom`*`) { ... }` bloc de gestion de l'exception

Tout bloc `try` doit toujours être suivit d'un bloc `catch` gérant les exceptions pouvant être lancées dans ce bloc `try`.

Si une exception est lancée mais n'est pas interceptée par le `catch` correspondant, le programme s'arrête (« `Aborted` »).

Divers

Commentaires :

```
// Commentaire jusqu'à la fin de la ligne
/* Autre commentaire, sur
 * plusieurs lignes */
```

Prototype le plus général de `main` :

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

`argc` : nombre d'arguments, taille du tableau `argv`

`argv` : tableau de pointeur sur des caractères : tableau des arguments. `argv[0]` est le nom du programme.

Espaces de noms (namespaces) :

Nommage d'un espace de noms :

```
namespace nom { ... corps de l'espace de noms ...}
```

Utilisation d'un/des objet(s) d'un espace de noms :

```
nom::objet;
using namespace nom;
using nom::objet;
```