Plan

- Rappels, généralités
- Approche impérative
- Approche fonctionnelle
- Approche objet
- Approche générique
- Compléments

Points abordés

- Variable, type de données
- Utilisation de la mémoire par les données
- Adresse d'une donnée
- Tableau
- Définition de types
- Référence

Programmation impérative

- Généralisation des concepts des langages d'assemblage
  - Mots (ou ensemble de mots) → ***variables***
  - Rangement dans un mot → ***affectation***
  - Structure de base courante : le ***tableau*** (suite de mots, indice)
  - Sous-programmes → procédures ou ***fonctions*** (retour de valeur)
- Introduction de ***structures de contrôle***
  - Boucles, instructions conditionnelles

Exemple impératif en C

```c
/*
 * Pré-condition : tab contient au moins
 *     un entier égal à 0
 */
int longueur_tableau( int tab[] ) {
   int i = 0;
   while( tab[i] != 0 ) {
      i = i + 1;
   }
   return i;
}
```

Modification de la variable i

Donnée, variable

- Une ***donnée*** manipulée par un programme :
  - est dans une ***zone mémoire*** (suite d'octets repérée par une adresse)
  - est conforme à un ***type*** (comment interpréter correctement la suite d'octets dans la zone)
  - possède une ***valeur*** (le contenu de la zone interprétée selon le type)
- Un langage de programmation accède aux données en utilisant des ***variables*** qui associent un nom à une donnée
- L'***affectation*** est l'instruction qui modifie la donnée associée à une variable

Valeur, lien

- Une variable peut :
  - être associée directement à la donnée : elle « contient » sa valeur qui ne peut donc pas être contenue dans une autre variable
  - permettre d'accéder (en général de manière non exclusive) à la donnée : elle contient un ***lien*** (souvent nommé ***référence*** ou ***pointeur***) vers la donnée
- L'affectation copie (donc duplique) la donnée dans le premier cas, duplique le lien (mais pas la donnée) dans le second
- Les langages (sauf C et C++) ne laissent en général pas le choix entre ces deux sortes de variables

Typage

- ***Typage statique explicite*** (Java, C, C++, C#)
  - Variables typées par le programmeur
  - Erreurs de typage détectées à la compilation

- ***Typage statique par inférence*** (Java, C++, C#)
  - Variables typées par le compilateur/interpréteur
  - Erreurs de typage détectées à la compilation

- ***Typage dynamique*** (Python, JavaScript)
  - Variables non typées (variables polymorphiques)
  - Erreurs de typage détectées à l'exécution

Définition de variables

- Définition sans initialisation (déconseillé)

```
int i;
```

- 2 syntaxes conseillées pour l'initialisation en C++ :
  - Typage statique ***explicite***

```
int i{ 5 };  // Initialisation à 0 si {}
auto l = long{ i + 7 };
```

  - Typage statique ***par inférence***

```
auto ul{ 2021ul };
auto f = -2.129e-3f;
```

- Seule syntaxe autorisée en C

```
char c = 'c';
```

- Autres syntaxes (*historiques*)

```
double d( 6.666 );
auto v = { 1 };    // À éviter !
```

Portée d'une variable

- La portée (lexicale) d'une variable est la partie du code source pouvant accéder à cette variable

```
int i{ 1 };       // Variable globale
void foo() {
  int j{ 2 };     // Variable locale à foo()
  int i{ 3 };     // Une locale peut masquer
                  // une globale
  {
    int j{ 4 }; // Une locale peut masquer
                  // une locale dans un sous-bloc
    i   = -2;     // Accès à la locale de foo()
    ::i = -1;     // Accès à la globale
  }
}
```

Bonne pratique

- Minimiser l'utilisation de variables globales

Types primitifs en C, C++

- Les entiers

```
char ≤ short ≤ int ≤ long ≤ long long
signed ou unsigned
25, 62L, 0xFE23U, 015UL, -53LL, 123'456'789
0b10010011 (C++14)
```

  - stdint.h
    `int8_t ... int64_t uint8_t ...`

- Les réels en virgule flottante

```
float ≤ double ≤ long double
3.14159, 5.28E-12F, 1.L
```

- Le type booléen (C++ et C23)

```
bool (_Bool en C99)
false ou true (0 ou 1 en C99)
```

- Les caractères

```
char, wchar_t (C++20 : char8_t, char16_t, char32_t)
'a', '\t', L'\\'
```

Démo

GitLab

cours/imperatif/BasicTypes.cpp

Types primitifs selon les langages

- Java, C# : entiers (uniquement signés en Java) sur 8, 16, 32 et 64 bits, flottants sur 32 et 64 bits, caractères sur 16 bits.

- JavaScript : uniquement des flottants sur 64 bits (mais les entiers inférieurs à 2^{53} sont représentés de manière exacte).

- Python : entiers sans limite de taille, flottants double précision, pas de type caractère (on utilise des chaînes de 1 caractère).

Pointeur (1)

- L'opérateur `&` donne l'adresse de la zone mémoire allouée à une variable

```
int i = 5;  // Une variable de type entier
&i;         // Son adresse (= un pointeur)
```

- Il est possible de définir des variables pouvant contenir de telles adresses (`*` n'est pas ici l'opérateur de multiplication) :

```
int *p;     // Une variable de type pointeur
            // sur un entier
p = &i;     // p "pointe" sur i
```

- Il est possible d'accéder à la variable pointée :

```
*p = -7;    // Modification de la valeur de i
```

- Les opérateurs `&` et `*` sont complémentaires :

`*&i` est la même donnée que `i`
`&*p` est la même donnée que `p`

Démo

GitLab

cours/imperatif/Pointer_1.cpp

Pointeur (2)

- ***Pointeur nul***

```
int * p{ nullptr };  // p ne pointe sur
                     // aucune donnée
```

  - Toute donnée d'un programme a une adresse distincte du pointeur nul
  - L'accès à la donnée pointée par `p` est illégal
  - En C < 23 : `int * p = NULL; // ou 0 !`
- On peut définir des pointeurs sur pointeur sur…

```
int    i =  5;
int *  p = &i;
int ** q = &p;
     ** q =  3;
```

- ***Pointeur générique***, très peu utilisé en C++

```
void * v = p;
```

Démo

GitLab

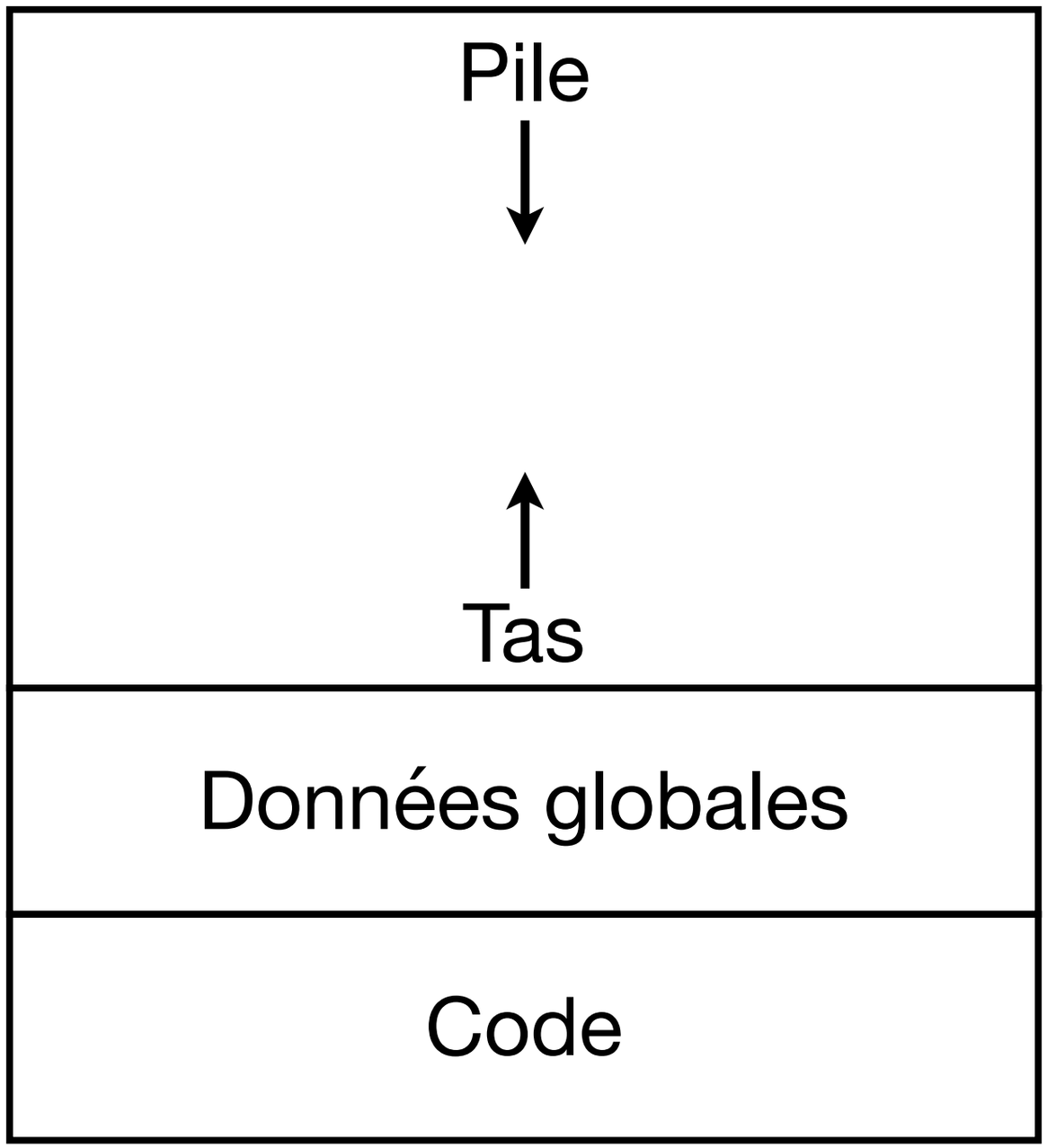
cours/imperatif/Pointer_2.cpp

Bonne pratique

- En C++ et C23, on utilise `nullptr` (et pas `NULL`, ni `0`)

- En C<23, on utilise `NULL`

Espace mémoire d'un processus



Allocation statique

- Variables définies au niveau principal en dehors de toute fonction (ou dans une fonction mais qualifiée par `static`)

- Création (et initialisation à `0` par défaut) avant l'exécution de la première instruction de `main()` (ou avant la première utilisation)

- Destruction après la dernière instruction de `main()` avant la fin du programme

- Attention : en-dehors de toute fonction, le mot-clé `static` sert à limiter une variable ou une fonction à l'unité courante, au lieu du programme entier

Déclaration de variables globales

- Syntaxe

```
extern int globale; // Initialisation interdite
```

- Pas de réservation de la zone mémoire
- Une variable doit être déclarée avant d'être utilisée
- Une définition est une déclaration
- Une variable doit être définie une et une seule fois

Bonne pratique

- Pas de définition de variable globale dans un fichier d'entête

Allocation automatique

- Variables définies dans une fonction ou dans un bloc d'instructions, non qualifiée par `static`

- Création et initialisation lors de la définition (pas d'initialisation par défaut en C/C++/Java)

- Destruction à la sortie du bloc

Allocation dynamique

- Création par le programmeur (`new` en Java et C#, implicite en Python)

- Pas d'initialisation par défaut en C/C++

- Destruction par le programmeur (C, C++) ou par l'environnement d'exécution (Java, C#, Python)

Allocation dynamique en C/C++

- Opérateurs `new` et `delete` en C++

```
int * p{ new int( 7 )};
*p = 5;
// ...
// pas obligatoirement dans la même fonction
delete p;
```

La zone mémoire pointée par p est libérée (la donnée est détruite)

- Fonctions de la bibliothèque standard en C (`stdlib.h`)

```
int * p = malloc( sizeof( int ));
*p = 5;
// ...
// pas obligatoirement dans la même fonction
free( p );
```

Bonne pratique

- Ne pas mélanger `new`/`delete` et `malloc()`/`free()`

Allocation de mémoire selon les langages

- En Java, C#
  - Allocation automatique (éventuellement statique) uniquement pour les types de base et les références (≈ pointeurs) d'objets
  - Allocation dynamique uniquement pour les tableaux et les objets

- En C, C++
  - Allocation statique, automatique ou dynamique selon le choix du programmeur, et quelque soit le type de variable créée

Ramasse-miettes
(GC : Garbage Collector, Glaneur de Cellules)

- Le problème ne se pose que pour l'allocation dynamique

- Existe en Java, C#, Python

```
{
  new Personne();
}
```

- N'existe pas en C, C++ (attention aux fuites de mémoire)

```
{
  new Personne();
}
```

Démo

GitLab

cours/imperatif/Allocation.cpp

Constante nommée

- En C++ (expression constante) et C (variable constante)

```
const int taille = 10;
```

- En C++11 (expression pouvant être calculée par le compilateur)

```
constexpr int taille{ 10 };
```

- En C (expression constante, traitement par le préprocesseur)

```
#define TAILLE 10
```

ou

```
enum { taille = 10 };
```

Bonne pratique

- En C++, ne pas utiliser `#define` pour nommer des constantes

Conversion

- Conversion d'une valeur d'un type vers une valeur d'un autre type, peut être intègre ou dégradante
- Implicites

```
int i{ 5 };
i *= 2.0;   // Erreur en Java, pas en C ou C++
```

- Ou explicites
  - Notation préfixée en C

```
i *= ( int ) 2.0;
```

  - Notation fonctionnelle C++

```
i *= int( 2.0 );
```

  - Notation avec accolades : erreur

```
// i *= int{ 2.0 };
```

Notation des conversions en C++

- Conversion de type (changement de représentation)

```
auto d{ 2.0 };
auto i = static_cast< int >( d );
```

- Suppression des qualificatifs **`const`** et **`volatile`**

```
auto const j{ 2 };
auto p{ const_cast< int * >( &j )};
```

- Conversion de pointeurs

```
int * p;
auto q{ reinterpret_cast< double * >( p )};
```

- Super-classe vers sous-classe

```
static_cast
dynamic_cast
```

Tableau

- En C (allocation statique)

```
int tab[TAILLE];  // expression constante
```

- En C (allocation automatique)

```
int tab[taille];
```

- En C (allocation dynamique)

```
int * tab = malloc( taille * sizeof( int ));
```

- En C++ (allocation dynamique)

```
auto tab{ new int[taille] };
```

- En C++ (bibliothèque standard)

```
auto tab{ std::vector< int >( taille )};
```

Bonne pratique

- Utiliser `std::vector` (ou une variante, voir plus tard) en C++

Pointeur et tableau (1)

- Pointeurs sur des éléments d'un tableau

```
int tab[10];
int * p1 = &tab[0];
int * p2 = &tab[5];
```

- Arithmétique sur les pointeurs

```
ptrdiff_t i = p2 - p1;
p2 = p1 + i;
p1 = p2 - i;
++p1;   // La valeur du pointeur (l'adresse)
        // est incrémentée de sizeof( *p1 )
```

Pointeur et tableau (2)

- Le nom d'un tableau (`tab`) est converti dans une ***expression évaluée*** en pointeur constant sur le premier élément (`&tab[0]`)
  - Exemples où `tab` n'est pas converti

```
sizeof( tab );
&tab;
```

- `tab[i]` est défini comme

```
*( tab + i ) == *( &tab[0] + i )
```

- Ceci explique pourquoi l'affectation de tableaux est interdite, et pourquoi la comparaison de tableaux ne donne pas le résultat attendu

Parcours d'un tableau

```
int tab[10];
```

- Parcours par indice

```
for( int i = 0; i < 10; ++i ) {
  tab[i] = 0;
}
```

- Parcours par pointeur

```
for( int * p = tab; p < tab + 10; ++p ) {
  *p = 0;
}
```

Démo

GitLab

cours/imperatif/Array_1.cpp
cours/imperatif/Array_2.cpp

Chaîne de caractères

- C++ : classe `std::string` de la bibliothèque standard

```
std::string s{ "Hello" };
if( s + " world!" == "Hello world!" ) {
  // ...
```

- C : Tableau de caractères avec une sentinelle (`'\0'`) à la fin

```
char s[] = "Hello world!";  // Ou char * s
```

  - Pas d'affectation par `=` ni de comparaison par `==`
  - Fonctions de la bibliothèque standard C :

```
strlen, strcpy, strcat, strcmp…
```

Démo

GitLab

cours/imperatif/String.cpp

Bonne pratique

- Utiliser `std::string` en C++

Énumération

- Une ***énumération*** est une liste de constantes nommées :

```
enum class Color { red, green, blue };
Color c{ Color::green };
```

- Par défaut, la première constante sera convertie (***conversion explicite*** vers un entier) en `0`, les suivantes la constante précédente `+1` ; il est possible d'imposer ces valeurs qui ne sont pas obligatoirement distinctes.
- Les énumérations issues du langage C sont moins strictes

```
enum Niveau { nul, moyen = 10, bon = 15 };
Niveau n{ moyen };  // Pas besoin de ::
int a{ n };         // Conversion implicite
```

Nouveau nom d'un type

- Il est possible de donner un nom à un type existant (primitif, utilisateur, dérivé) :

```
using Integer = short;
using Tab = std::vector< Integer >;
```

- En C

```
typedef short Integer;
```

Agrégat

- Un ***agrégat*** est un type permettant la description d'une donnée par une liste de ses différents ***attributs*** (nom et type) :

```
struct Personne {
  char sexe_;
  int age_;
};
struct Personne p;
p.sexe_ = 'F';
p.age_ = 20;
// En C++
Personne p2{ 'M', 82 };
```

Il faut un point-virgule !

`struct` non nécessaire en C++

- Les unions (il vaut mieux oublier !)

```
union IntOrFloat {
  int   i_;
  float f_;
};
```

Pointeur et agrégat

- Un ***type utilisateur*** est un type :

```
// ligne typedef inutile en C++
typedef struct Personne Personne;
Personne p;
Personne * pp = &p;
```

- L'accès à un attribut avec l'opérateur `.` à partir d'un pointeur impose l'utilisation de parenthèses :

```
(*pp).age_ = 20;
```

- Un autre opérateur `->` évite ce problème :

```
pp->age_ = 30;
```

Démo

GitLab

cours/imperatif/Struct.cpp

Déclaration de types utilisateurs

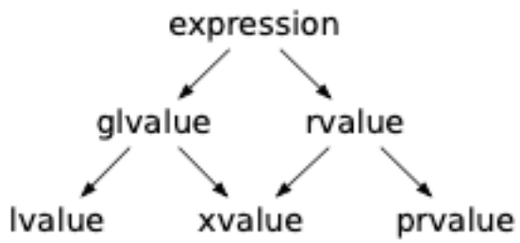
- `struct Personne;`
- Un type doit être défini avant d'être utilisé pour déclarer des variables de ce type (`Personne p;`)
- Un type doit être déclaré avant d'être utilisé pour déclarer des variables de type pointeur (ou référence : voir plus loin) sur ce type (`Personne * p;`) ou pour définir type dérivé (`using PersonnePtr = Personne *;`)
- Une définition est une déclaration
- Un type doit être défini une et une seule fois par unité de compilation, un type ne doit pas avoir plusieurs définitions dans un programme

Valeurs gauches et droites

- L'affectation `=` est l'opérateur caractéristique de l'approche impérative (on modifie l'état).
- On nomme ***valeur gauche*** (*lvalue*) une expression pouvant être à gauche de l'affectation et ***valeur droite*** (*rvalue*) une expression ne pouvant être qu'à droite de l'affectation
-
```
int i;  int *p;  Complexe c;
```
Valeur gauche ou pas ?

```
0  i  i+2  &i  ++i  i++  i=2  p  *p  c  c.real
```

- Le standard C++ a ajouté les *glvalue* (*Generalized lvalue*), *xvalue* (*eXpiring lvalue*) et *prvalue* (*Pure rvalue*)



Référence

- ***Référence*** : sens commun (lien vers une donnée), mais sémantique différente selon les langages
- En Python, Java
  - Référence sur un objet ~ pointeur en C++
  - La copie de la référence ne provoque pas la copie de l'objet référencé
- En C++
  - Une référence permet de définir une variable (un nom) associée à une donnée déjà existante
  - Ne correspond à *aucun élément* d'une architecture Von Neuman

Référence sur valeur gauche

- Une ***référence sur valeur gauche*** n'est pas utile pour la définition de variable :

```
int    i{ 3 };    // auto    i{ 3 };
// i est une valeur gauche
int & ri{ i };    // auto & ri{ i };

i  =  5;              // i == ri == 5
ri = -2;              // i == ri == -2

int j{ 7 };
ri = j;             // i == ri == 7
j = -1;             // i == ri == 7

//    int & rk{ 8 };  // 8 n'est pas une lvalue
const int & ck{ 8 };  // OK
```

Démo

GitLab

cours/imperatif/Reference.cpp