INFO-F105 – Langages de programmation 1
De Python à C++ (avancé)

1 Typage

1.1 Nombres entiers

Il existe des entiers de taille fixe définis dans le header `<cstdint>`.

Signé	Non signé	#bits
`int8_t`	`uint8_t`	8
`int16_t`	`uint16_t`	16
`int32_t`	`uint32_t`	32
`int64_t`	`uint64_t`	64

1.2 Nombres flottants

Il existe des flottants de taille fixe définis dans le header `<stdfloat>`[1].

Type	#bits
`std::float16_t`	16
`std::float32_t`	32
`std::float64_t`	64
`std::float128_t`	128

Ces types ne sont définis qu'à partir de C++23.

2 Fonction `main`

Tout comme en Python, il est possible de passer des arguments en ligne de commande.

```
./program argument1 argument2
```

```python
# Python
import sys

if __name__ == "__main__":
  if (len(sys.argv) == 2):
    print(sys.argv[1])
```

```cpp
// C++
#include <iostream>

int main(int argc, char *argv[]) {
  if (argc == 2) {
    std::cout << argv[1] << std::endl;
  }
  return 0;
}
```

Pour récupérer ces arguments, il faut définir deux paramètres au `main` :

- `argc` contient le nombre d'arguments + 1 (le chemin vers le programme)
- `argv` est un tableau de *C strings* (`char*`), un type que nous verrons plus tard

3 Variables

Une *expression constante* est une constante dont la valeur est connue à la compilation.

```cpp
constexpr double PI = 3.14;
```

Utilisez `constexpr` autant que possible pour les constantes globales pour indiquer au compilateur qu'il peut substituer la constante par sa valeur et ainsi éviter un accès mémoire, ce qui réduira le temps d'exécution.

```cpp
// Avant compilation
constexpr double PI = 3.14;
double area(double radius) {
  return PI * pow(radius, 2);
}
```

```cpp
// Après compilation
double area(double radius) {
  return 3.14 * pow(radius, 2);
}
```

4 Opérations

Les opérateurs retournent une valeur de même type que les opérandes.

```cpp
5 / 2 == 2
5.0 / 2.0 == 2.5
```

Dans certains cas ce n'est pas ce que l'on veut et il est alors possible de convertir les opérandes en un autre type avant l'opération pour changer le type retour.

```cpp
double(5) / double(2) == 2.5
int(5.0) / int(2.0) == 2
```

5 Blocs

5.1 Conditions

Si vous avez de nombreuses conditions sur la valeur d'une variable entière ou une `enum`, vous pouvez remplacer le bloc `if...else` par un `switch`.

```
if (data == 0) {
  // ...
} else if (data == 1) {
  // ...
} else if (data == 2) {
  // ...
} else {
  // ...
}
```

```
switch (data) {
  case 0:
    // ...
    break;
  case 1:
    // ...
    break;
  case 2:
    // ...
    break;
  default:
    // ...
    break;
}
```

Un `switch` est plus performant qu'un `if`, en particulier s'il y a de nombreuses conditions :

- Un `if` doit évaluer toutes les conditions dans l'ordre jusqu'à trouver le bon `else if`
- Un `switch` crée une *lookup table* pour arriver immédiatement dans le bon `case`

Il est indispensable de mettre un `break` à la fin de chaque `case` pour sortir du `switch`, sinon le programme commencera au bon `case` puis continuera à exécuter les `case` suivants.

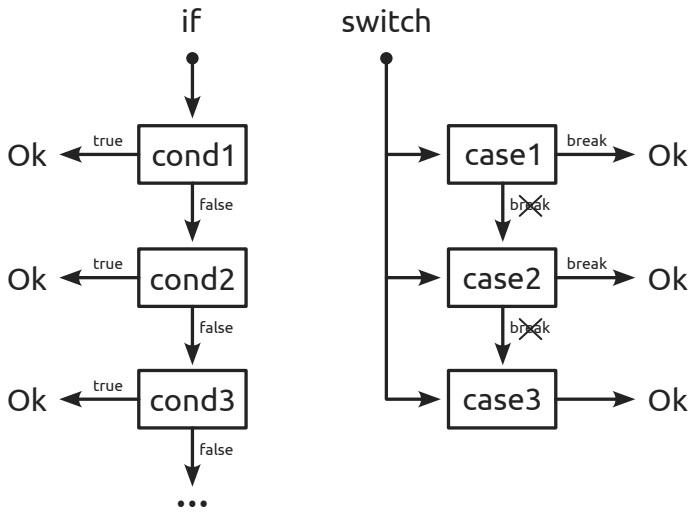

Figure 1: Exécution d'un `if` et d'un `switch`

5.2 Boucles

Une boucle `do..while` est similaire à une boucle `while`, sauf que la condition est évaluée à la fin du corps de la boucle.

```cpp
std::string input;

do {
  std::cout << "Continue [y/n]: ";
  std::cin >> input;
} while (input == "y");
```

```cpp
std::string input = "y";

while (input == "y") {
  std::cout << "Continue [y/n]: ";
  std::cin >> input;
}
```

C'est utile lorsque la condition dépend d'une valeur obtenue dans le corps de la boucle.

5.3 Fonctions

Paramètres

Il existe 3 modes de passage de paramètres qu'on utilise dans différents cas.

```cpp
void foo(int n);           // Copie (types numériques)
void foo(std::string& n);  // Référence (types non numériques)
void foo(int* arr);        // Pointeur (principalement tableaux)
```

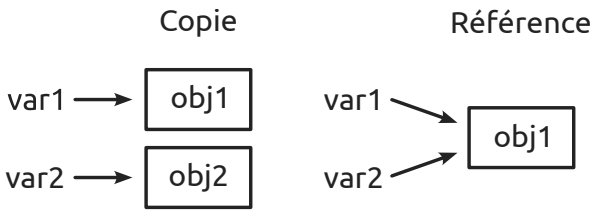

Figure 2: Passage de paramètres

Inline

Le compilateur remplace les appels aux fonctions `inline` par leur corps.

```cpp
// Avant compilation
inline int add(int a, int b) {
  return a + b;
}

int main() {
  int sum = add(5, 2);
}
```

```cpp
// Après compilation
int main() {
  int sum = 5 + 2;
}
```

Utilisez `inline` autant que possible sur les fonctions avec très peu de code ou qui sont appelées peu souvent pour accélérer les performances en évitant les appels. Utiliser `inline` sur une grande fonction appelée très souvent risque d'augmenter la taille de l'exécutable.

Constexpr

Une fonction `constexpr` est évaluée à la compilation pour former une valeur `constexpr`.

```cpp
// Avant compilation
constexpr double PI = 3.14;
constexpr double area(int radius) {
  return PI * pow(radius, 2);
}

int main() {
  std::cout << area(5) << std::endl;
}
```

```cpp
// Après compilation
int main() {
  std::cout << 78.5 << std::endl;
}
```

Utilisez `constexpr` autant que possible sur les fonctions ne nécessitant que des valeurs connues à la compilations. Cela permettra d'améliorer grandement les performances lors de l'exécution du programme car la fonction aura été évaluée une seule fois au moment de la compilation.

Lambda

En Python comme en C++, les fonctions sont elles-mêmes des variables que l'on peut assigner.

```python
# Python
def is_greater(a, b):
  return a > b

f = is_greater
print(f(2, 5))
```

```cpp
// C++
bool is_greater(int a, int b) {
  return a > b;
}

int main() {
  auto f = is_greater;
  std::cout << f(2, 5) << std::endl;
}
```

Il existe une syntaxe spéciale pour définir une fonction directement dans une variable[2].

```python
# Python
arr = [3, 6, 1, 8, 4, 6]
f = lambda a, b : a > b
sort(arr, key=f)
```

```cpp
// C++
#include <algorithm>

int main() {
  int arr[] = { 3, 6, 1, 8, 4, 6 };
  auto f = [](int a, int b) {
    return a > b;
  };

  std::sort(arr, arr+6, f);
}
```

Cette syntaxe est particulièrement utile lorsque l'on veut passer une fonction comme paramètre à une autre fonction. On la retrouve couramment en Python mais assez rarement en C++.

5.4 Scope

Static

Une variable `static` existe pour toute la durée de l'exécution du programme.

```
unsigned foo() {
  static unsigned n = 0;
  return n++;
}
```

Le premier appel de fonction `foo` renverra `0`, le deuxième renverra `1`, et ainsi de suite car la variable `n` est `static` et n'est donc pas redéfinie à chaque appel.

Extern

Une variable ou fonction marquée comme `extern` indique au compilateur qu'elle est définie dans une autre unité de compilation (un autre fichier).

```
extern int x;
extern bool is_prime(unsigned);
```

5.5 Namespace

Un *namespace* est un bloc dont les variables sont accessibles depuis l'extérieur.

```
namespace my_lib {
  void hello() {
    std::cout << "Hello, World!" << std::endl;
  }
}

int main() {
  my_lib::hello();
}
```

Les namespaces sont principalement utilisés dans des *librairies*, des bibliothèques de code qui contiennent des fonctions et structures utilisables dans plusieurs projets. C'est le cas par exemple de la librairie standard de C++, dont le namespace est `std`.

Ils permettent de ne pas *polluer* l'espace global. Imaginez que vous souhaitiez créer votre propre fonction `hello` mais tout en ayant toujours accès à la fonction `my_lib::hello`.

Références

[1] "Fixed width floating-point types - cppreference.com." Available: https://en.cppreference.com/w/cpp/types/floating-point

[2] "Lambda expressions - cppreference.com." Available: https://en.cppreference.com/w/cpp/language/lambda