

Blocs et portée des variables

Blocs

En C++, un bloc est identifié par des délimiteurs de début et de fin : { et } . Précédemment, nous avons vu que les structures de contrôles (i.e., les instructions qui permettent de choisir, de répéter des traitements : if-else, for, do-while, while, ...) utilisaient des blocs ({ et }). Par exemple :

```
for (int i = 0; i < 7; i++){ ... }
```

On peut l'écrire aussi comme suit :

```
for (int i = 0; i < 7; i++)
{
    ...
}
```

En fait, en C++, les instructions peuvent être regroupées en blocs indépendamment de toute structure de contrôle (if-else, for, do-while, while, ...). Pour cela, il suffit juste d'avoir une accolade ouvrante et une accolade fermante qui entourent une séquence d'instructions. Par exemple :

```
int main(){
    int j = 2;

    {
        int i = 2;
        double x = 5.7;
        cout << "i = " << i << endl;
    }
}
```

Les blocs peuvent contenir leurs propres déclarations et initialisations de variables, comme par exemple les variables `i` et `x` dans le programme ci-dessus.

Variables locales

Les blocs ont un impact sur l'utilisation des variables. Les variables déclarées à l'intérieur d'un bloc sont appelées variables locales (au bloc). C'est-à-dire, elles ne sont accessibles qu'à l'intérieur du bloc (elles ne sont pas définies à l'extérieur du bloc). Par exemple :

```
int main(){
    int i;
    cout << "i = "; cin >> i;
    if (i != 0){
        int j = 0;
        j = 2 * i;
    }
    // à partir d'ici, on ne peut plus utiliser j
}
```

`j` est une variable locale au bloc `if`. Une fois on est sorti de ce bloc, on ne peut plus utiliser la variable `j`.

Variables globales

Les variables déclarées en dehors de tout bloc (même du bloc `main()`) sont appelées variables globales (au programme). C'est-à-dire, elles sont accessibles dans l'ensemble du programme. Par exemple :

```
#include<iostream>

using namespace std;

int x;

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

`x` est une variable globale au programme.

Bonnes pratiques

1. Ne jamais utiliser des variables globales (sauf peut être pour certaines constantes).

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

int x;

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

Pourquoi ne jamais utiliser des variables globales? Parce qu'une variable globale est accessible à l'ensemble du programme, ce qui veut dire que n'importe quelle instruction peut modifier cette variable, donc, il devient extrêmement difficile à suivre la valeur de cette variable.

2. Déclarer les variables au plus près de leur utilisation (mettre la ligne de déclaration le plus proche possible de la ligne où on va la première fois utiliser la variable).

Exemple 1 : Si la variable `j` n'est pas utilisée après le `if` alors :

<pre>// Écrivez : int main(){ int i; cout << "i = "; cin >> i; if (i != 0){ int j = 0; j = 2 * i; } }</pre>	<pre>// Plutôt que : int main(){ int i; cout << "i = "; cin >> i; int j = 0; if (i != 0){ j = 2 * i; } }</pre>
--	---

Exemple 2 : Dans le programme suivant, on peut déplacer la déclaration de la variable `j` au plus près de son utilisation :

```

int j;
x = 2;
t = 5;
j = 2 * i;

```

Ainsi, on obtiendra :

```

x = 2;
t = 5;
int j;
j = 2 * i;

```

ou voire même :

```

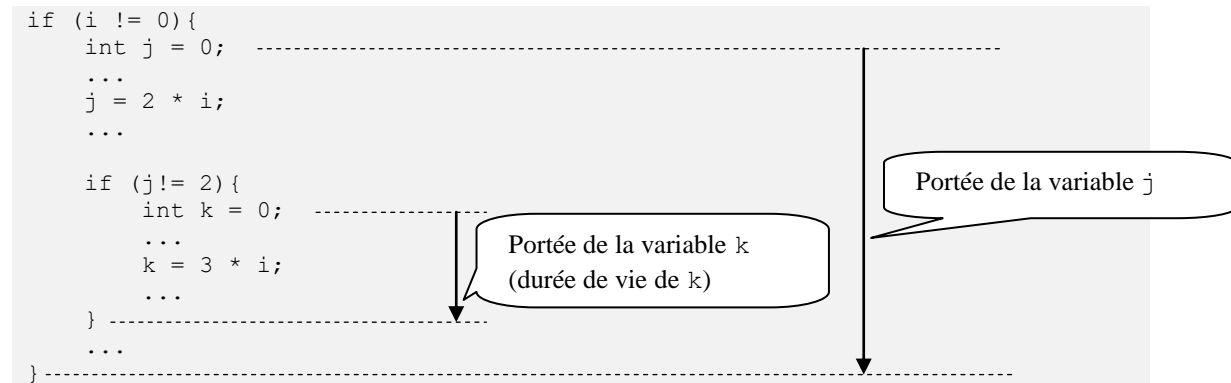
x = 2;
t = 5;
int j = 2 * i;

```

Notion de portée

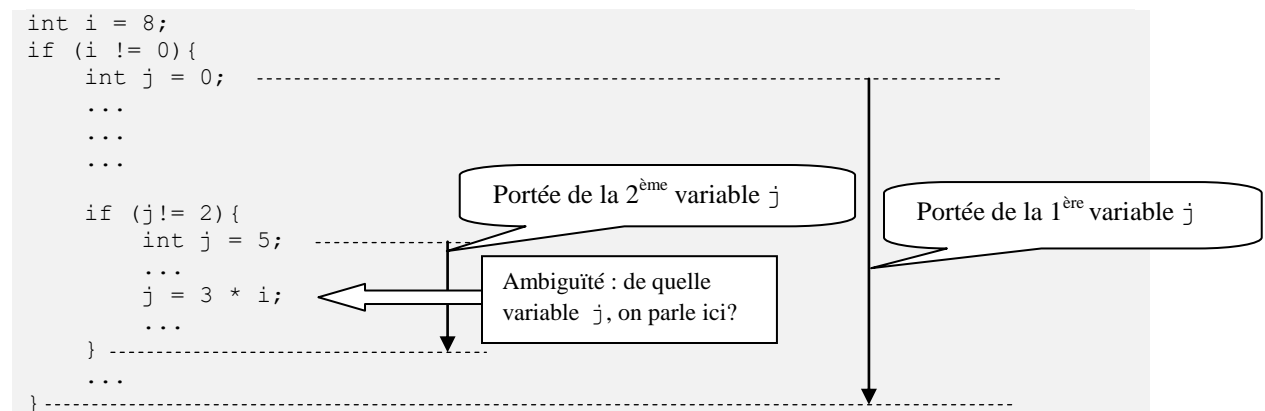
La portée d'une variable est l'ensemble des lignes de code où cette variable est accessible, autrement dit, où elle est définie (existe, a un sens, ...).

Exemple



Règle de résolution de portée

En C++, on peut avoir des variables de même nom mais de portées différentes, ce qui crée, parfois, des ambiguïtés. Voici un exemple :



Comme vous avez remarqué, dans ce programme, nous avons deux variables `j`. Alors, une instruction `comm (j = 3 * i;)` se réfère à quelle variable?

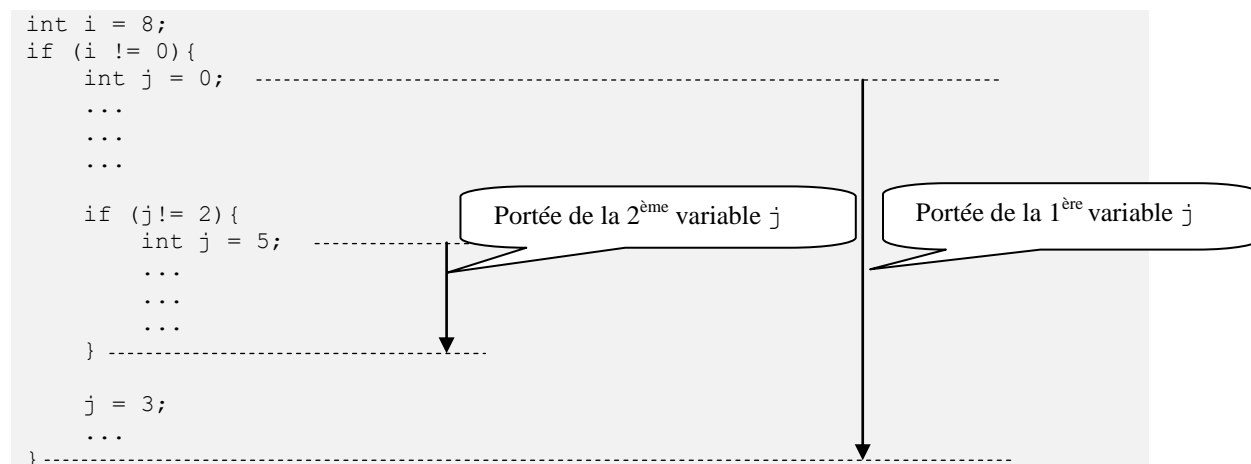
	...
j	0
j	5

Règle : En cas d'ambiguïté, on peut déterminer de quelle variable on parle, en suivant la règle suivante : *la localité masque la globalité* (i.e., la variable la plus proche est utilisée). En d'autres termes, les variables globales au bloc seront masquées et les variables locales au bloc seront utilisées.

Dans notre exemple, l'instruction `(j = 3 * i;)` va, alors, *modifier la variable locale* au bloc. L'autre variable `j` est masquée à l'intérieur du bloc (elle ne sera pas considérée).

	...
j	0
j	5 24

Considérons, maintenant, le programme suivant :



Dans ce programme, la dernière instruction `(j = 3;)` va modifier la valeur de la 1^{ère} variable `j`, parce que la deuxième variable `j` n'existe pas à l'extérieur de son bloc. Voilà ce que nous obtenons après exécution de l'instruction `(j = 3;)` :

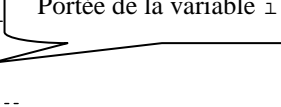
	...
j	0 3

Bonnes Pratiques

Les exemples ci-dessus sont donnés juste pour illustrer le propos. Il est conseillé d'éviter de déclarer plusieurs variables avec le même nom, sauf peut être pour des variables locales aux boucles comme : `i`, `j`, ... Pourquoi? Parce qu'on n'écrit pas des programmes pour les machines, mais pour les humains. Donc, il faut que vos programmes soient compréhensibles, faciles à corriger, faciles à transmettre à d'autres programmeurs. Alors, évitez toute ambiguïté dans vos programmes et essayez d'être au maximum le plus clair possible.

Portée : cas des itérations

```
for (int i = 0; i < 5; i++){  
    cout << "i = " << i << endl;  
}  
// à partir d'ici, on ne peut plus utiliser la variable i
```



La déclaration d'une variable à l'intérieur d'une boucle `for` est une déclaration locale au bloc et aux deux instructions de test (`i < 5`) et incrémentation (`i++`).

Exemple (à ne pas suivre) : Qu'affiche le programme suivant?

```
int main(){  
    int i = 120;  
  
    for (int i = 0; i < 5; i++){  
        cout << "i = " << i << endl;  
    }  
  
    cout << "i = " << i << endl;  
  
    return 0;  
}
```

Le programme affichera :

```
0  
1  
2  
3  
4  
120
```