Algorithmique
Cours 6

IUT Informatique de Lens, 1ère Année
Université d'Artois

Frédéric Koriche
koriche@cril.fr
2011 - Semestre 1

Sommaire

L'objectif de ce cours est d'étudier les **fonctions** et **modules** en algorithmique et programmation C++.

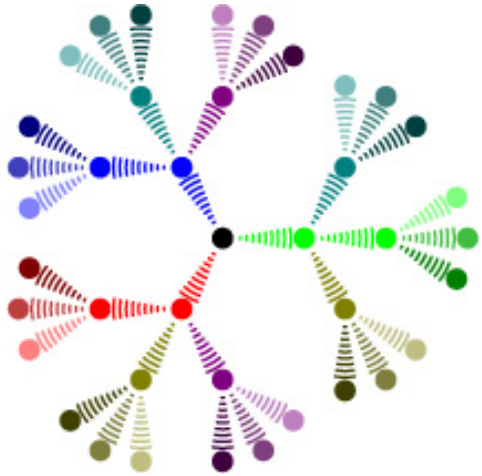
Sommaire

L'objectif de ce cours est d'étudier les **fonctions** et **modules** en algorithmique et programmation C++.

Programmation procédurale

La programmation procédurale vise à décomposer un problème en sous-problèmes de plus en plus simples. Chaque sous-problème est traité par une fonction avec des valeurs d'entrée qu'elle peut recevoir et des valeurs de sortie qu'elle peut transmettre.

Comment résoudre mon problème ? . . .



. . . en le décomposant en sous problèmes de plus en plus simples !

Fonction

En algorithmique, une fonction est définie par deux parties :

- Une **en-tête** : elle spécifie le type de la fonction, c'est à dire, le type de ses données d'entrées et le type de sa valeur de sortie.
- Un **corps** : il spécifie l'algorithme permettant de passer des données d'entrée à la valeur de sortie.

Fonction

En algorithmique, une fonction est définie par deux parties :

- Une **en-tête** : elle spécifie le type de la fonction, c'est à dire, le type de ses données d'entrées et le type de sa valeur de sortie.
- Un **corps** : il spécifie l'algorithme permettant de passer des données d'entrée à la valeur de sortie.

Déclaration

La **déclaration** d'une fonction consiste à spécifier seulement l'en-tête de la fonction.

Fonction

En algorithmique, une fonction est définie par deux parties :

- Une **en-tête** : elle spécifie le type de la fonction, c'est à dire, le type de ses données d'entrées et le type de sa valeur de sortie.
- Un **corps** : il spécifie l'algorithme permettant de passer des données d'entrée à la valeur de sortie.

Déclaration

La **déclaration** d'une fonction consiste à spécifier seulement l'en-tête de la fonction.

Définition

La **définition** d'une fonction consiste à spécifier à la fois l'en-tête et le corps de la fonction.

Exemple

J'aimerais un algorithme qui permette de calculer mes chances de gagner au loto, sachant que :

- je dois choisir k numéros
- parmi n numéros possibles

Exemple

J'aimerais un algorithme qui permette de calculer mes chances de gagner au loto, sachant que :

- je dois choisir k numéros
- parmi n numéros possibles

Rappel : le nombre de tirages possibles est égal au nombre de combinaisons de k parmi n

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Déclaration d'une fonction en pseudo-code

Mes chances de gagner au loto

```
réel fact(entier p)

variables
    entier n,k
    réel possibilités

début
    lire n
    lire k
     // Calculer le nombre de tirages possibles
    possibilités ← fact(n)/(fact(k) * fact(n - k))
     // Afficher la probabilité de gagner
    afficher 1/possibilités
fin
```

Déclaration d'une fonction en pseudo-code

Mes chances de gagner au loto

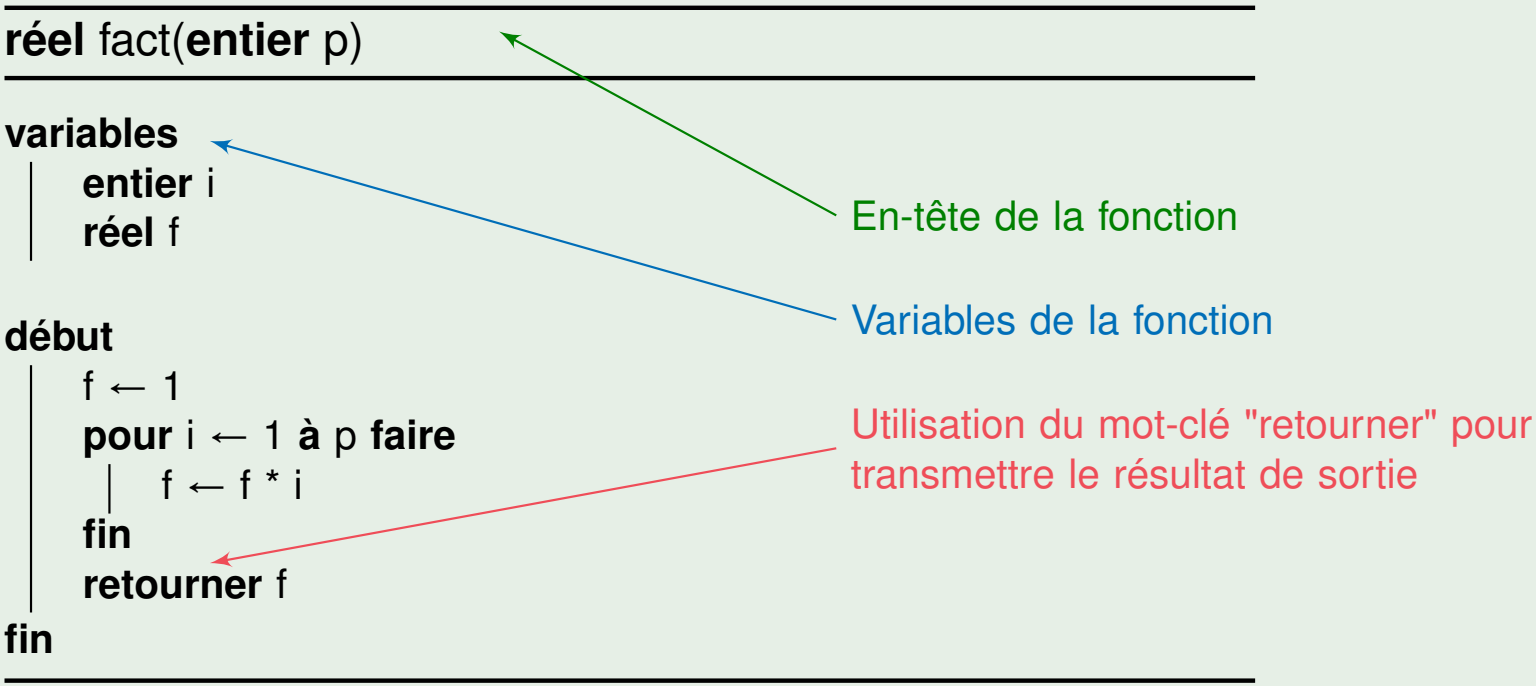
```
réel fact(entier p)

variables
    entier n,k
    réel possibilités

début
    lire n
    lire k
     // Calculer le nombre de tirages possibles
    possibilités ← fact(n)/(fact(k) * fact(n - k))
     // Afficher la probabilité de gagner
    afficher 1/possibilités
fin
```

La déclaration de fonctions figure avant la déclaration de variables

Définition d'une fonction en pseudo-code

```
réel fact(entier p)

variables
    entier i
    réel f

début
    f ← 1
    pour i ← 1 à p faire
        f ← f * i
    fin
    retourner f
fin
```

Définition d'une fonction en pseudo-code



Déclaration d'une fonction en C++

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


double fact (int p);


int main()
{
int k, n;
double possibilites;
cin >> n;
cin >> k;
// Calculer le nombre de tirages possibles
possibilites = fact(n)/(fact(k) * fact(n-k));
// Afficher la probabilité de gagner
cout << 1/possibilites;
}
```

loto.cpp

Déclaration d'une fonction en C++

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

double fact (int p);

int main()
{
int k, n;
double possibilites;
cin >> n;
cin >> k;
// Calculer le nombre de tirages possibles
possibilites = fact(n)/(fact(k) * fact(n-k));
// Afficher la probabilité de gagner
cout << 1/possibilites;
}
```

Ne pas oublier le point virgule !

loto.cpp

Définition d'une fonction en C++

```
using namespace std;


double fact (int p)
{
int i;
double f;
f = 1;
for(i = 1; i <= p; i++)
     f = f * i;
return f;
}
```

fact.cpp

Définition d'une fonction en C++

```
using namespace std;


double fact (int p)
{
int i;
double f;
f = 1;
for(i = 1; i <= p; i++)
    f = f * i;
return f;
}
```

fact.cpp

En-tête de la fonction

Variables de la fonction

Le mot-clé `return` indique une sortie de fonction

Procédure

En algorithmique, une procédure est une fonction **sans valeur de sortie**.

Déclaration en pseudo-code

Jeu des allumettes

afficherAllumettes(**entier** n)

variables

| ...

début

| ...

fin

Procédure

En algorithmique, une procédure est une fonction **sans valeur de sortie**.

Déclaration en pseudo-code

Jeu des allumettes

afficherAllumettes(**entier** n)

variables

| ...

début

| ...

fin

L'en-tête de la procédure ne possède pas de type de sortie

Procédure

En algorithmique, une procédure est une fonction **sans valeur de sortie**.

Déclaration en pseudo-code

Jeu des allumettes

afficherAllumettes(**entier** n)

variables

| ...

début

| ...

fin

L'en-tête de la procédure ne possède pas de type de sortie

Déclaration en C++

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


void afficherAllumettes (int n);


int main()
{


...


}
```

Procédure

En algorithmique, une procédure est une fonction **sans valeur de sortie**.

Déclaration en pseudo-code

Jeu des allumettes

afficherAllumettes(**entier** n)

variables

| ...

début

| ...

fin

L'en-tête de la procédure ne possède pas de type de sortie

Déclaration en C++

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

void afficherAllumettes (int n);

int main()
{

...

}
```

Le type de sortie de la procédure est `void`

Définition d'une procédure en C++

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


void afficheAllumettes (int n)
{
int i;
for(i = 1; i <= n; i++)
     cout << 'I';
cout << endl;
}
```

afficheAllumettes.cpp

Définition d'une procédure en C++

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


void afficheAllumettes (int n)
{
int i;
for(i = 1; i <= n; i++)
     cout ≪ 'I';
cout ≪ endl;
}
```

afficheAllumettes.cpp

En-tête de la procédure

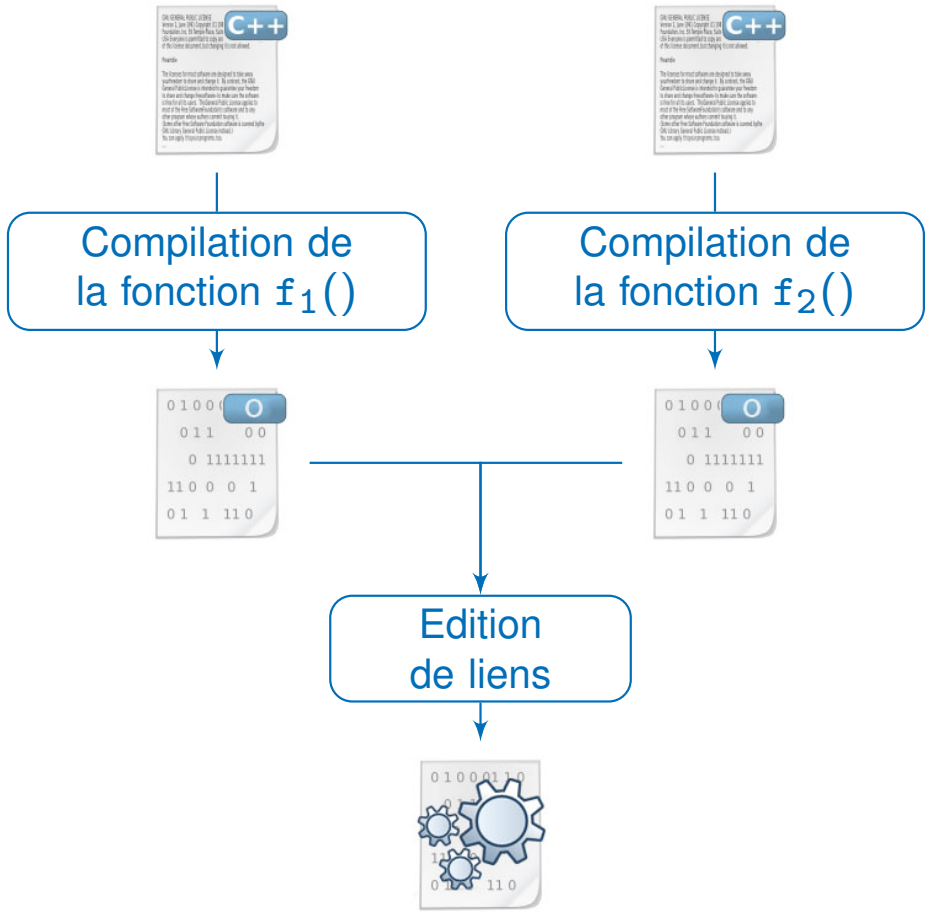
Variable de la procédure

La procédure n'a aucune sortie

Compilation

L'intérêt de la programmation procédurale est de pouvoir compiler **séparément** les différentes fonctions d'un programme.

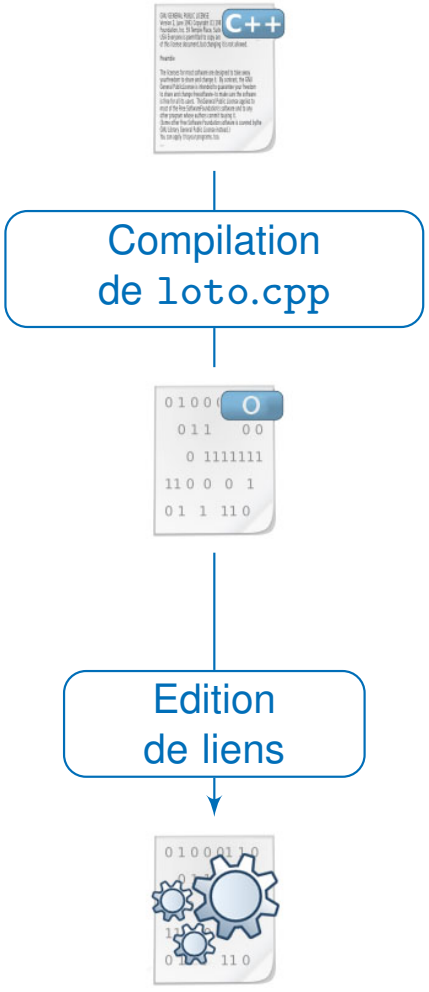
- Le code de la définition d'une fonction `f()` est compilable s'il contient la déclaration des fonctions que `f()` utilise.
- Le code du programme complet est exécutable s'il contient le code objet de toutes les fonctions qu'il utilise.



Un seul fichier pour le loto

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


double fact (int p)
{
int i;
double f;
f = 1;
for(i = 1; i <= p; i++)
    f = f * i;
return f;
}


int main()
{
int k, n;
double possibilites;
cin >> n;
cin >> k;
// Nombre de tirages
possibilites = fact(n)/(fact(k) * fact(n-k));
// Probabilité de gagner
cout << 1/possibilites;
}
```

loto.cpp



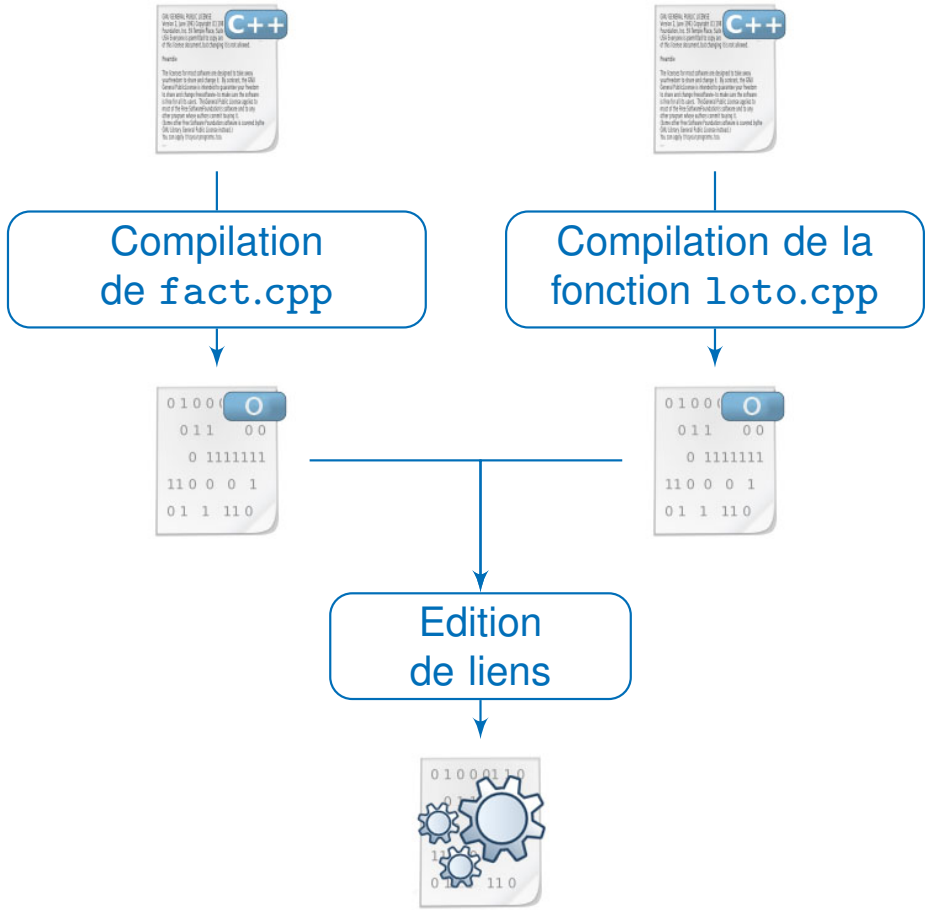
Deux fichiers pour le loto

```
using namespace std;


double fact (int p)
{
int i;
double f;
f = 1;
for(i = 1; i <= p; i++)
    f = f * i;
return f;
}
```

fact.cpp

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


double fact (int p);


int main()
{
int k, n;
double possibilites;
cin >> n;
cin >> k;
// Nombre de tirages
possibilites = fact(n)/(fact(k) * fact(n-k));
// Probabilité de gagner
cout << 1/possibilites;
}
```

loto.cpp



Portée des données

La **portée** d'une donnée (variable ou constante) est définie par le bloc d'instructions dans lequel elle est déclarée. Ce bloc est le **domaine de visibilité** de la variable.

Portée des données

La **portée** d'une donnée (variable ou constante) est définie par le bloc d'instructions dans lequel elle est déclarée. Ce bloc est le **domaine de visibilité** de la variable.

Donnée locale

Lorsqu'une donnée (variable ou constante) est déclarée au début d'une fonction, sa portée se limite au bloc de la fonction. Elle est dite **locale** à la fonction.

Portée des données

La **portée** d'une donnée (variable ou constante) est définie par le bloc d'instructions dans lequel elle est déclarée. Ce bloc est le **domaine de visibilité** de la variable.

Donnée locale

Lorsqu'une donnée (variable ou constante) est déclarée au début d'une fonction, sa portée se limite au bloc de la fonction. Elle est dite **locale** à la fonction.

Donnée globale

Une donnée (variable ou constante) est dite **globale** si sa déclaration figure en dehors de toute fonction. Sa portée s'étend à tout le fichier dans lequel elle est déclarée.

Portée des données

La **portée** d'une donnée (variable ou constante) est définie par le bloc d'instructions dans lequel elle est déclarée. Ce bloc est le **domaine de visibilité** de la variable.

Donnée locale

Lorsqu'une donnée (variable ou constante) est déclarée au début d'une fonction, sa portée se limite au bloc de la fonction. Elle est dite **locale** à la fonction.

Donnée globale

Une donnée (variable ou constante) est dite **globale** si sa déclaration figure en dehors de toute fonction. Sa portée s'étend à tout le fichier dans lequel elle est déclarée.

Variables locales versus globales

- A cause de leur portée illimitée, les variables globales peuvent être modifiées n'importe où dans un fichier, leur utilisation est donc à éviter !
- Grâce à leur portée limitée, les variables locales sont modifiées à des endroits précis, leur utilisation est donc recommandée afin de contrôler le programme.

Portée des variables

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

void f();

int a = 0;

int main()
{
int b = 1;
b++;
a = b;
f();
}

void f()
{
int c = 5;
c++;
a = a * c;
}
```

Déclaration de la procédure `f()`

Portée des variables

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

void f();

int a = 0;

int main()
{
int b = 1;
b++;
a = b;

f();
}

void f()
{
int c = 5;
c++;
a = a * c;
}
```

Déclaration de la procédure `f()`

Portée de b

Portée de c

Portée des variables

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

void f();

int a = 0;

int main()
{
int b = 1;
b++;
a = b;
f();
}

void f()
{
int c = 5;
c++;
a = a * c;
}
```

Déclaration de la procédure `f()`

Portée de b

Portée de c

Portée de a

Homonymie

Le compilateur sanctionne par une erreur la déclaration de deux variables de **même nom** ayant la **même portée**.

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


int main()
{
int a = 0;
a++;
int a = 1;

...

}
```

Homonymie

Le compilateur sanctionne par une erreur la déclaration de deux variables de **même nom** ayant la **même portée**.

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


int main()
{
int a = 0;
a++;
int a = 1;

...


}
```

Erreur de compilation ! La variable ”a” est déclarée deux fois dans le même bloc

Homonymie

Le compilateur autorise la déclaration de deux variables de **même nom** ayant des **portées disjointes**.

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

void f()
{
int n = 5;
n++;
cout << n;
}

void g()
{
int n = 1;
n++;
cout << n;
}

int main()
{
f();
g();
}
```

Homonymie

Le compilateur autorise la déclaration de deux variables de **même nom** ayant des **portées disjointes**.

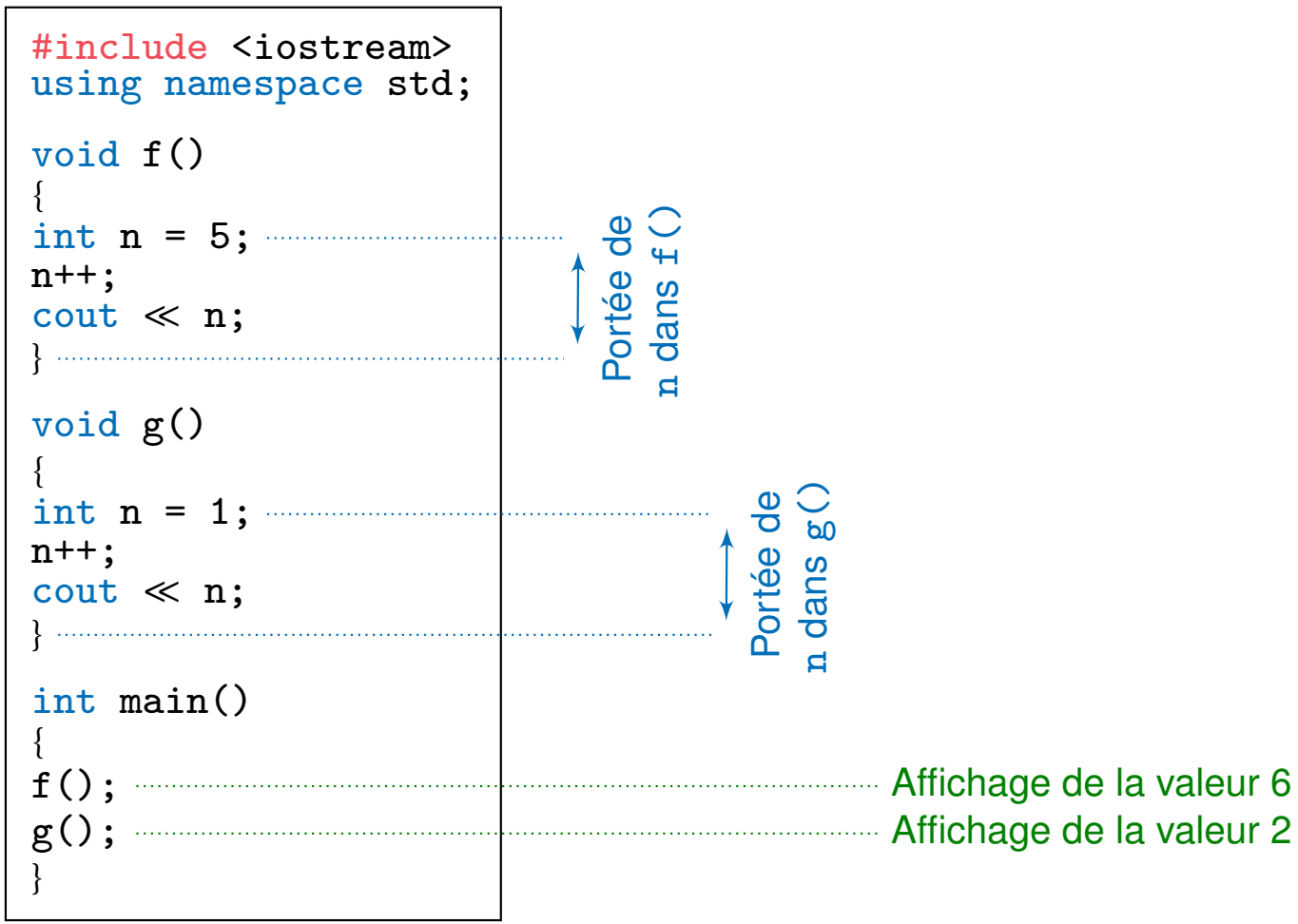
```
#include <iostream>
using namespace std;

void f()
{
int n = 5;
n++;
cout << n;
}

void g()
{
int n = 1;
n++;
cout << n;
}

int main()
{
f();
g();
}
```

Portée de n dans f()

Portée de n dans g()

Homonymie

Le compilateur autorise la déclaration de deux variables de **même nom** ayant des **portées disjointes**.



Masquage

Le compilateur peut autoriser la déclaration de deux variables de **même nom** ayant des **portées différentes** (mais pas forcément disjointes). Dans ce cas, la variable du domaine interne **masque** la variable du domaine externe.

Masquage

Le compilateur peut autoriser la déclaration de deux variables de **même nom** ayant des **portées différentes** (mais pas forcément disjointes). Dans ce cas, la variable du domaine interne **masque** la variable du domaine externe.

Recommandation

Le masquage peut souvent entraîner des comportements imprévisibles, et il est recommandé de déclarer les variables locales au début de la fonction.

Masquage

Le compilateur peut autoriser la déclaration de deux variables de **même nom** ayant des **portées différentes** (mais pas forcément disjointes). Dans ce cas, la variable du domaine interne **masque** la variable du domaine externe.

Recommandation

Le masquage peut souvent entraîner des comportements imprévisibles, et il est recommandé de déclarer les variables locales au début de la fonction.

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

int main()
{
int i = 0;
for (int j = 1; j <= 10; j++)
   {
   i++;
   cout ≪ i * j;
   }
}
```

Masquage

Le compilateur peut autoriser la déclaration de deux variables de **même nom** ayant des **portées différentes** (mais pas forcément disjointes). Dans ce cas, la variable du domaine interne **masque** la variable du domaine externe.

Recommandation

Le masquage peut souvent entraîner des comportements imprévisibles, et il est recommandé de déclarer les variables locales au début de la fonction.

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

int main()
{
int i = 0;
for (int j = 1; j <= 10; j++)
   {
   i++;
   cout << i * j;
   }
}
```

Masquage

Le compilateur peut autoriser la déclaration de deux variables de **même nom** ayant des **portées différentes** (mais pas forcément disjointes). Dans ce cas, la variable du domaine interne **masque** la variable du domaine externe.

Recommandation

Le masquage peut souvent entraîner des comportements imprévisibles, et il est recommandé de déclarer les variables locales au début de la fonction.

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

int main()
{
int i = 0;
for (int j = 1; j <= 10; j++)
   {
   i++;
   cout << i * j;
   }
}
```

Affichage des valeurs $1, 4, 9, 16, \ldots, 100$

Masquage

Le compilateur peut autoriser la déclaration de deux variables de **même nom** ayant des **portées différentes** (mais pas forcément disjointes). Dans ce cas, la variable du domaine interne **masque** la variable du domaine externe.

Recommandation

Le masquage peut souvent entraîner des comportements imprévisibles, et il est recommandé de déclarer les variables locales au début de la fonction.

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

int main()
{
int i = 1;
cout << i;
for (int j = 1; j <= 10; j++)
   {
   int i = 0;
   cout << i * j;
   }
}
```

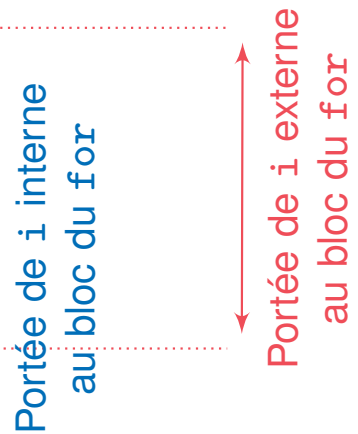
Masquage

Le compilateur peut autoriser la déclaration de deux variables de **même nom** ayant des **portées différentes** (mais pas forcément disjointes). Dans ce cas, la variable du domaine interne **masque** la variable du domaine externe.

Recommandation

Le masquage peut souvent entraîner des comportements imprévisibles, et il est recommandé de déclarer les variables locales au début de la fonction.

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

int main()
{
int i = 1;
cout ≪ i;
for (int j = 1; j <= 10; j++)
   {
   int i = 0;
   cout ≪ i * j;
   }
}
```



Masquage

Le compilateur peut autoriser la déclaration de deux variables de **même nom** ayant des **portées différentes** (mais pas forcément disjointes). Dans ce cas, la variable du domaine interne **masque** la variable du domaine externe.

Recommandation

Le masquage peut souvent entraîner des comportements imprévisibles, et il est recommandé de déclarer les variables locales au début de la fonction.

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

int main()
{
int i = 1;
cout << i;
for (int j = 1; j <= 10; j++)
   {
   int i = 0;
   cout << i * j;
   }
}
```

Affichage de la valeur 1

Affichage de la valeur 0

Transmission des données

Lorsqu'une fonction `f` est appelée par une autre fonction (ex : `main`) il faut que

- la liste des données transmises en entrée de `f` corresponde bien au type d'entrée de `f`,
- la variable affectée à la sortie de `f` corresponde bien au au type de sortie de `f`.

Transmission des données

Lorsqu'une fonction `f` est appelée par une autre fonction (ex : `main`) il faut que

- la liste des données transmises en entrée de `f` corresponde bien au type d'entrée de `f`,
- la variable affectée à la sortie de `f` corresponde bien au au type de sortie de `f`.

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


long long combinaisons(int k, int n);


int main()
{
char k = 'a';
int n = 10;
long long c = combinaisons(k,n);
}
```

Déclaration de la fonction retournant le nombre de combinaisons de k parmi n

Erreur ! L'entrée n'est pas de type `(int,int)`

Transmission des données

Lorsqu'une fonction `f` est appelée par une autre fonction (ex : `main`) il faut que

- la liste des données transmises en entrée de `f` corresponde bien au type d'entrée de `f`,
- la variable affectée à la sortie de `f` corresponde bien au au type de sortie de `f`.

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


long long combinaisons(int k, int n);


int main()
{
int k = 3;
int n = 10;
int c = combinaisons(k,n);
}
```

Déclaration de la fonction retournant le nombre de combinaisons de *k* parmi *n*

Erreur ! La sortie n'est pas affectée à une variable de type `long long`

Transmission des données

Lorsqu'une fonction `f` est appelée par une autre fonction (ex : `main`) il faut que

- la liste des données transmises en entrée de `f` corresponde bien au type d'entrée de `f`,
- la variable affectée à la sortie de `f` corresponde bien au au type de sortie de `f`.

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


long long combinaisons(int k, int n);


int main()
{
int k = 3;
int n = 10;
long long c = combinaisons(k,n);
}
```

Déclaration de la fonction retournant le nombre de combinaisons de k parmi n

Correct !

Transmission des données

Lorsqu'une fonction `f` est appelée par une autre fonction (ex : `main`) il faut que

- la liste des données transmises en entrée de `f` corresponde bien au type d'entrée de `f`,
- la variable affectée à la sortie de `f` corresponde bien au au type de sortie de `f`.

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


long long combinaisons(int k, int n);


int main()
{
int k = 3;
int n = 10;
double c = (double)combinaisons(k,n);
}
```

Déclaration de la fonction retournant le nombre de combinaisons de k parmi n

Correct aussi par conversion de type

Transmission et mémoire

Au moment de l'appel d'une fonction, une plage de mémoire est réservée pour chacun de ses arguments.

Transmission par valeur

Par défaut, la transmission des données à une fonction s'effectue par valeur : pour chaque donnée, une copie de sa valeur est réalisée et cette copie est transmise au corps de la fonction.

Transmission et mémoire

Au moment de l'appel d'une fonction, une plage de mémoire est réservée pour chacun de ses arguments.

Transmission par valeur

Par défaut, la transmission des données à une fonction s'effectue par valeur : pour chaque donnée, une copie de sa valeur est réalisée et cette copie est transmise au corps de la fonction.

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


long long combinaisons(int k, int n);


int main()
{
int k = 3;


int n = 10;


long long c = combinaisons(k,n);
}
```

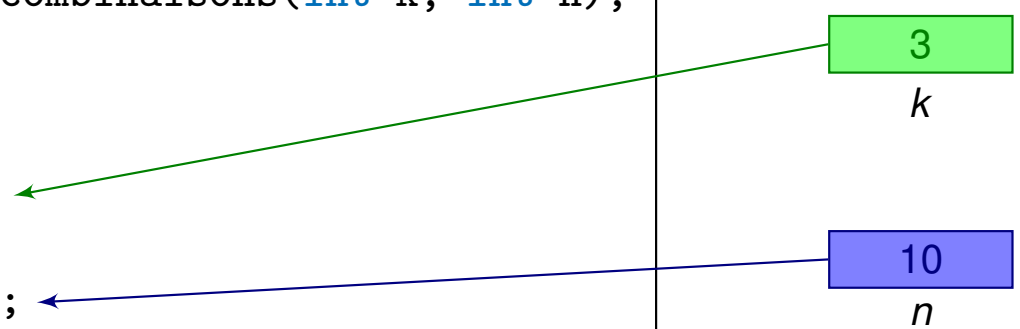
Transmission et mémoire

Au moment de l'appel d'une fonction, une plage de mémoire est réservée pour chacun de ses arguments.

Transmission par valeur

Par défaut, la transmission des données à une fonction s'effectue par valeur : pour chaque donnée, une copie de sa valeur est réalisée et cette copie est transmise au corps de la fonction.

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

long long combinaisons(int k, int n);

int main()
{
int k = 3;

int n = 10;

long long c = combinaisons(k,n);
}
```



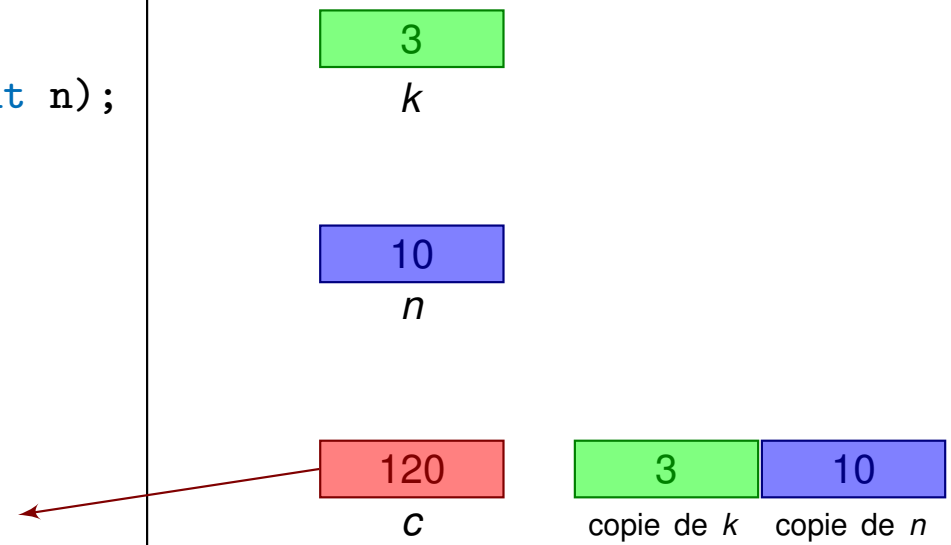
Transmission et mémoire

Au moment de l'appel d'une fonction, une plage de mémoire est réservée pour chacun de ses arguments.

Transmission par valeur

Par défaut, la transmission des données à une fonction s'effectue par valeur : pour chaque donnée, une copie de sa valeur est réalisée et cette copie est transmise au corps de la fonction.

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

long long combinaisons(int k, int n);

int main()
{
int k = 3;

int n = 10;

long long c = combinaisons(k,n);
}
```



Transmission par valeur

En transmission par valeur, il est impossible de modifier les valeurs d'entrée d'une fonction.

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


void permuter(int a, int b)
{
int temp;
temp = a;
a = b;
b = temp;
}
```

Procédure réalisant la permutation de deux variables

a et *b* restent inchangées : seules les **copies** de *a* et de *b* sont permutées !

Transmission par valeur

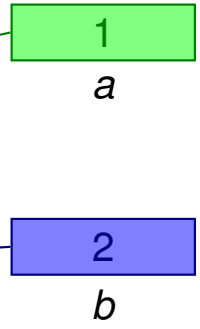
En transmission par valeur, il est impossible de modifier les valeurs d'entrée d'une fonction.

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


void permuter(int & a, int & b);


int main()
{
int a = 1;


int b = 2;


permuter(a,b);
}
```

Transmission par valeur

En transmission par valeur, il est impossible de modifier les valeurs d'entrée d'une fonction.

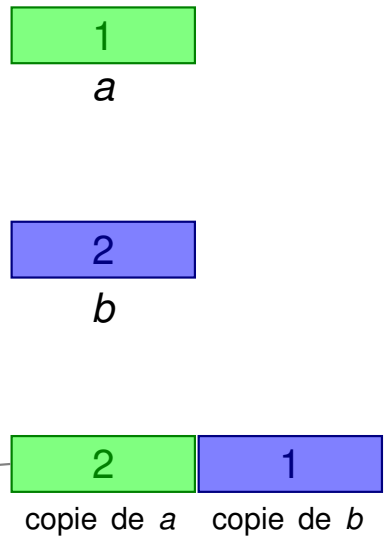
```
#include <iostream>
using namespace std;

void permuter(int & a, int & b);

int main()
{
int a = 1;

int b = 2;

permuter(a,b);
}
```



Transmission par valeur

En transmission par valeur, il est impossible de modifier les valeurs d'entrée d'une fonction.

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


void permuter(int & a, int & b);


int main()
{
int a = 1;


int b = 2;


permuter(a,b);
}
```



Transmission par adresse

Lorsqu'une variable x est transmise par adresse, sa valeur n'est pas copiée en mémoire, mais l'adresse de x est transmise à la fonction qui peut donc directement modifier la valeur de x.

L'opérateur &

La spécification d'une transmission de donnée par adresse s'effectue par l'opérateur &.

Transmission par adresse

Lorsqu'une variable x est transmise par adresse, sa valeur n'est pas copiée en mémoire, mais l'adresse de x est transmise à la fonction qui peut donc directement modifier la valeur de x.

L'opérateur &

La spécification d'une transmission de donnée par adresse s'effectue par l'opérateur &.

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


void permuter(int & a, int & b)
{
int temp;
temp = a;
a = b;
b = temp;
}
```

Procédure réalisant la permutation de deux variables

Les arguments de la procédure sont transmis par adresse

Transmission par adresse

Lorsqu'une variable x est transmise par adresse, sa valeur n'est pas copiée en mémoire, mais l'adresse de x est transmise à la fonction qui peut donc directement modifier la valeur de x.

L'opérateur &

La spécification d'une transmission de donnée par adresse s'effectue par l'opérateur &.

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


void permuter(int & a, int & b);


int main()
{
int a = 1;


int b = 2;


permuter(a,b);
}
```

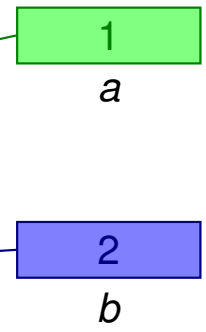
Transmission par adresse

Lorsqu'une variable *x* est transmise par adresse, sa valeur n'est pas copiée en mémoire, mais l'adresse de *x* est transmise à la fonction qui peut donc directement modifier la valeur de *x*.

L'opérateur &

La spécification d'une transmission de donnée par adresse s'effectue par l'opérateur &.

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


void permuter(int & a, int & b);


int main()
{
int a = 1;


int b = 2;


permuter(a,b);
}
```



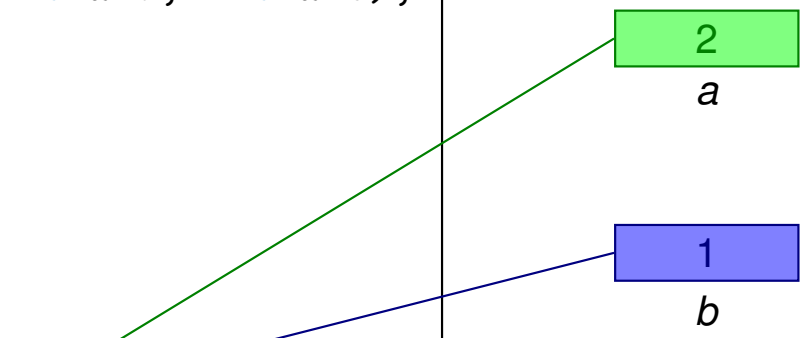
Transmission par adresse

Lorsqu'une variable *x* est transmise par adresse, sa valeur n'est pas copiée en mémoire, mais l'adresse de *x* est transmise à la fonction qui peut donc directement modifier la valeur de *x*.

L'opérateur &

La spécification d'une transmission de donnée par adresse s'effectue par l'opérateur &.

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


void permuter(int & a, int & b);


int main()
{
int a = 1;


int b = 2;


permuter(a,b);
}
```



Transmission par adresse

La transmission par adresse permet :

- de modifier directement la valeur d'une variable (vue comme une **entrée-sortie**),
- d'accéder à une donnée structurée sans avoir à copier toute ses valeurs.

Le préfixe `const`

Lorsque l'on souhaite accéder à une donnée structurée sans la modifier, le préfixe `const` garantit que la fonction doit traite la donnée comme une constante.

Transmission par adresse

La transmission par adresse permet :

- de modifier directement la valeur d'une variable (vue comme une **entrée-sortie**),
- d'accéder à une donnée structurée sans avoir à copier toute ses valeurs.

Le préfixe `const`

Lorsque l'on souhaite accéder à une donnée structurée sans la modifier, le préfixe `const` garantit que la fonction doit traite la donnée comme une constante.

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


void afficher(const Carte & c)
{
cout ≪ c.face;
cout ≪ " de ";
cout ≪ c.couleur ≪ endl;
}
```

Procédure affichant les valeurs (face et couleur) d'une carte

La structure est transmise par adresse

Le préfixe `const` garantit que la valeur ne peut pas être modifiée par la fonction

Transmission par adresse

La transmission par adresse permet :

- de modifier directement la valeur d'une variable (vue comme une **entrée-sortie**),
- d'accéder à une donnée structurée sans avoir à copier toute ses valeurs.

Le préfixe `const`

Lorsque l'on souhaite accéder à une donnée structurée sans la modifier, le préfixe `const` garantit que la fonction doit traite la donnée comme une constante.

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


void afficher(const Carte & c)
{
c.couleur = coeur;
}
```

Erreur de compilation ! Le préfixe `const` interdit la modification de la valeur

Opérateur `typedef`

Les tableaux peuvent être transmis par adresse en construisant d'abord un type avec l'opérateur `typedef`.

```
using namespace std;

typedef float Prix[100];

void init(Prix& p)
{
for(int i = 0; i < 100; i++)
    p[i] = 0;
}
```

Le type `Prix` est un tableau de 100 réels

La procédure `init` remplit le tableau avec des zéros

Opérateur `typedef`

Les tableaux peuvent être transmis par adresse en construisant d'abord un type avec l'opérateur `typedef`.

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

typedef float Point[3];

float dist(const Point& x, const Point& y)
{
float carre = 0;
for(int i = 0; i < 3; i++)
    carre += (x[i] - y[i])*(x[i] - y[i]);
return sqrt(carre);
}
```

Le type `Point` est un tableau avec trois réels (l'abscisse, l'ordonnée et la cote)

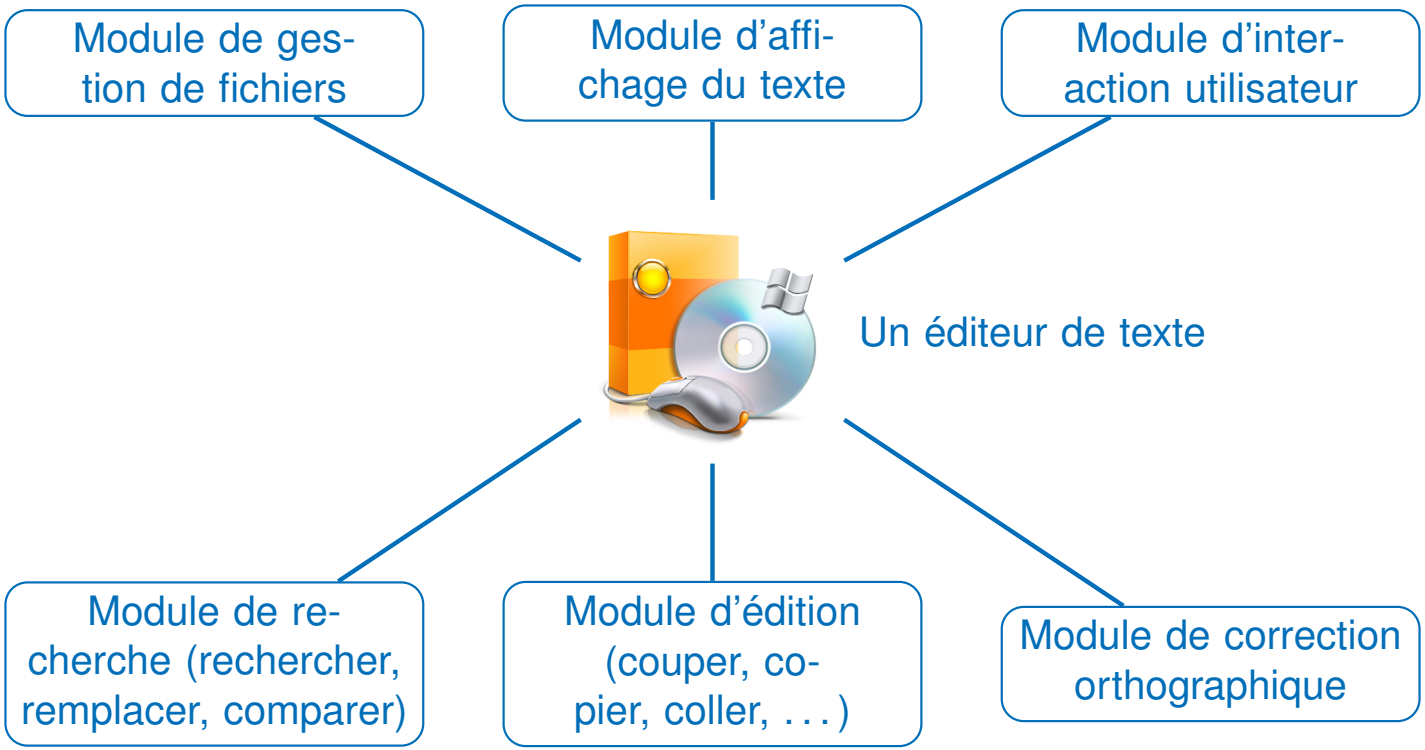
La fonction `dist` retourne la distance entre deux points transmis par leur adresse et vus comme des constantes

Programmation modulaire

La programmation modulaire est une généralisation de la programmation procédurale. Elle vise à décomposer une application en **modules**, chacun correspondant au traitement d'une tâche précise et associé avec ses fonctions correspondantes.

Programmation modulaire

La programmation modulaire est une généralisation de la programmation procédurale. Elle vise à décomposer une application en **modules**, chacun correspondant au traitement d'une tâche précise et associé avec ses fonctions correspondantes.

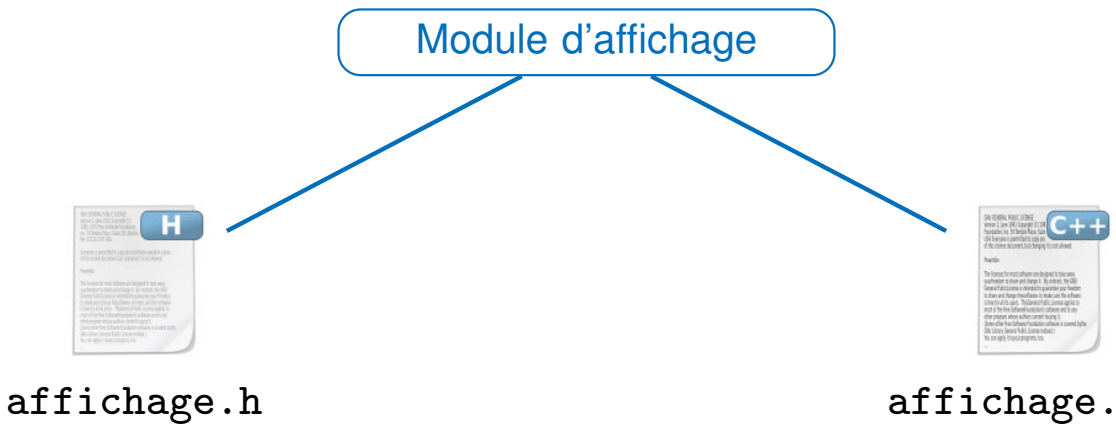


Programmation modulaire en C++

En C++, un module est composé de deux parties :

- Un fichier d'entête (`.h`) qui comprend la déclaration des fonctions du module
- Un fichier source (`.cpp`) qui comprend la définition des fonctions du module

Le fichier d'entête peut être utilisé par divers programmes afin d'accéder aux fonctions du module



Programmation modulaire en C++

La programmation modulaire permet la compilation séparée et la réutilisation des modules.