Ch. 3 - Fonctions

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

7 octobre 2020

Table des matières

Introduction

Utilité

- « Ensemble d'instructions qui effectue une tâche »
- Peut être *appelé* au sein d'un programme

Avantages

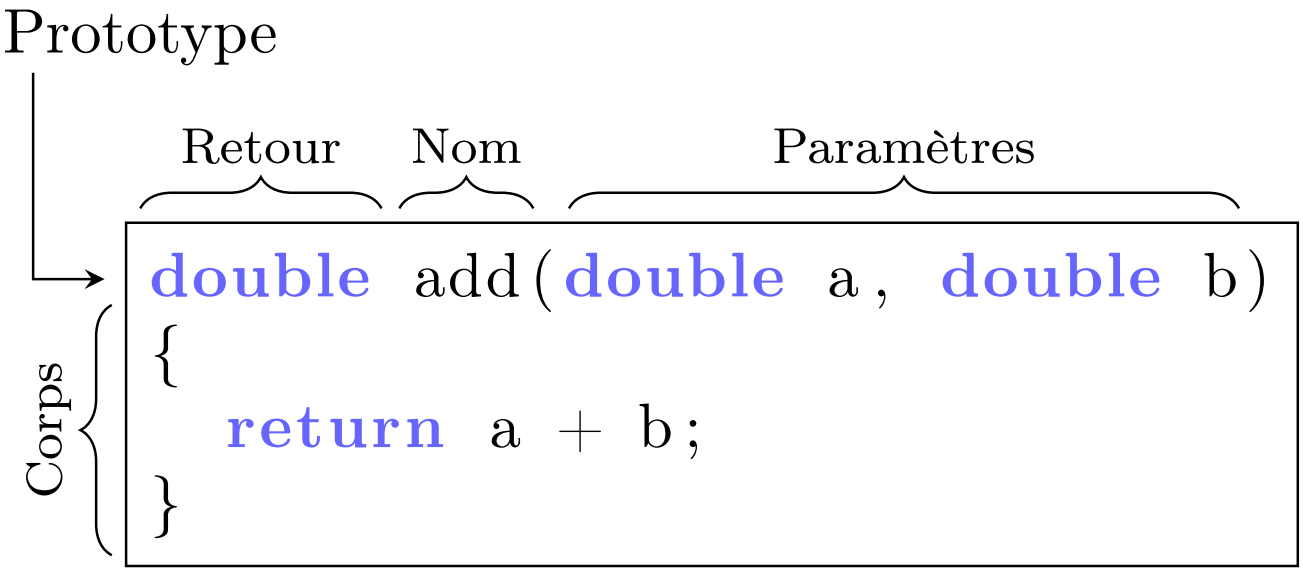
- Permet de découper le travail en parties indépendantes
- Permet de réutiliser du code
- Limite la redondance
  - Moins de « copier / coller »
  - Maintenabilité augmentée
- Augmente la lisibilité

Caractéristiques

1. Possède des paramètres et un retour
   - `sqrt` prend en paramètre un flottant et retourne un flottant
2. Identifiées par leur nom et leurs paramètres
   - Les règles d'appel sont appliquées sur ces caractéristiques
   - *Pas* le type de retour
3. Concept indépendant de la POO
   - Fonctions *membres* (méthodes : `C++`)
   - Fonctions indépendantes
4. Plus qu'une fonction mathématique
   - Effectue un travail
   - Possibilité de modifier les paramètres
   - Peut ne rien retourner (`void`)

Déclaration et définition

- Toute fonction doit être déclarée et définie
  - Possibilité de séparer la déclaration de la définition
  - Parfois nécessaire



- Seul les types des paramètres sont nécessaires dans le prototype
- Les fonctions *doivent* être déclarées avant d'être utilisées
- Déclaration possible au sein d'un bloc

Exemple

- Fichier `before.c`

```
1  int main()
2  {
3    print("Hello");
4  }
5
6  void print(const char* s)
7  {
8    printf("%s\n", s);
9  }
```

- Déclaration anticipée

```
1  void hello(const char*);
2
3  int main()
4  {
5    print("Hello");
6  }
7
8  void print(const char* s)
9  {
10   printf("%s\n", s);
11 }
```

- Même principe en `C++`

Les fonctions sans arguments en C

- En C uniquement, si on veut qu'une fonction n'accepte aucun argument, il faut écrire `void` dans la liste des paramètres
- Si on déclare `void f();`, la fonction `f` accepte un nombre arbitraire d'arguments (et les ignore)

```
1  void f() {}
2  void g(void) {}
3
4  int main()
5  {
6      f();
7      f(1); //ok
8      f(1,2); //ok
9
10     g();
11     //g(1); //ko
12 }
```

Passage d'argument

Passage par valeur

- Par défaut, à chaque appel d'une fonction, une *copie* des paramètres est envoyée à la fonction
  - Dans le cas d'un type de base, on copie la valeur
  - Dans le cas d'une `struct` (C), on copie les attributs
  - Dans le cas d'un objet (C++), on appelle le constructeur de recopie (cf. Ch. 4)
- « Ne permet pas » de modifier les paramètres
- La valeur de retour est également transmise par valeur

Avantages

- Pas d'effet de bord

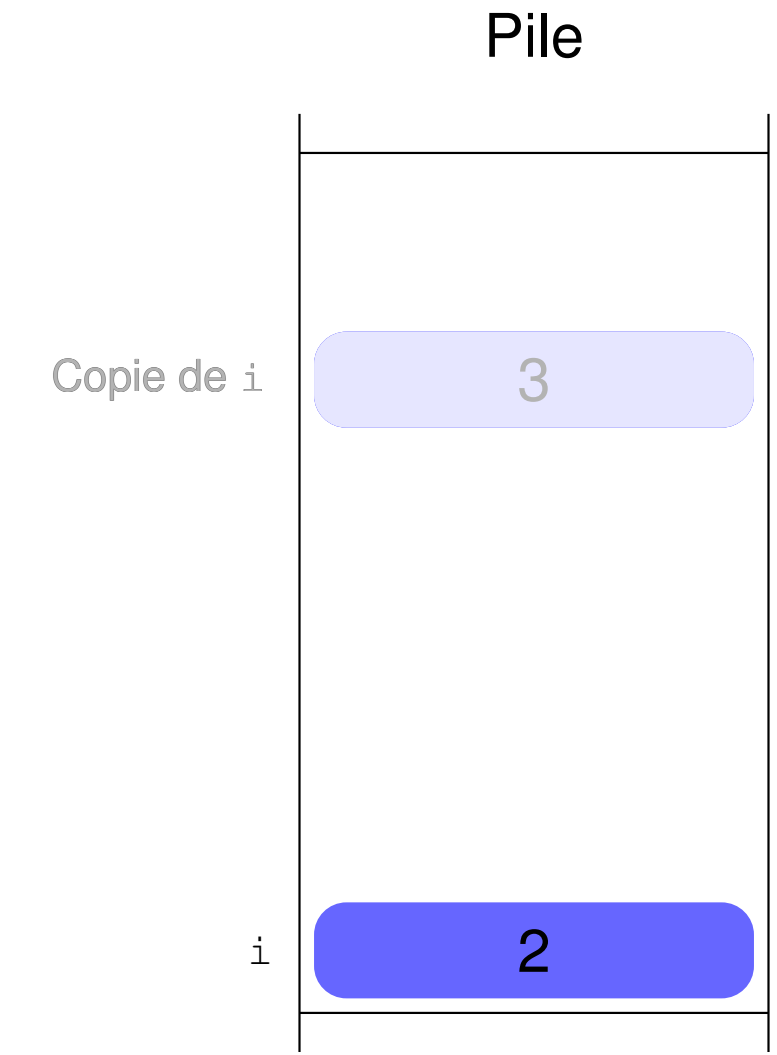
Inconvénients

- Performances réduites

Mécanisme

```
void f(int i)
{
   printf("%d\n", i);
   i++;
   printf("%d\n", i);
}

int main()
{
   int i = 2;
   f(i);
   printf("%d\n", i);
}
```



Mauvais swap

- Fichier `swap-value.cpp`

```
1  void swap(int x, int y)
2  {
3    cout << "Entering swap : " << x << " " << y << endl;
4
5    int tmp = y;
6    y = x;
7    x = tmp;
8
9    cout << "Exiting swap : " << x << " " << y << endl;
10 }
11
12 int main()
13 {
14   int i = 1;
15   int j = 2;
16
17   cout << "Before call : " << i << " " << j << endl;
18   swap(i, j);
19   cout << "After call : " << i << " " << j << endl;
20 }
```

Exemple

- Fichier `pass-value.cpp`

```
1  void countDown(int i)
2  {
3    while(i > 0)
4    {
5      cout << i << endl;
6      i--;
7    }
8    cout << "BOOM" << endl;
9  }
10
11 int main()
12 {
13   for(int i = 5; i >= 0; i--)
14   {
15     countDown(i);
16     cout << endl;
17   }
18 }
```

Passage par adresse

- On ne transmet pas une copie de l'objet, mais son adresse
  - « Comme en `Java` »
- Permet d'émuler un passage par référence
  - En `C` pur, pas d'autre solution

Inconvénients par rapport aux références

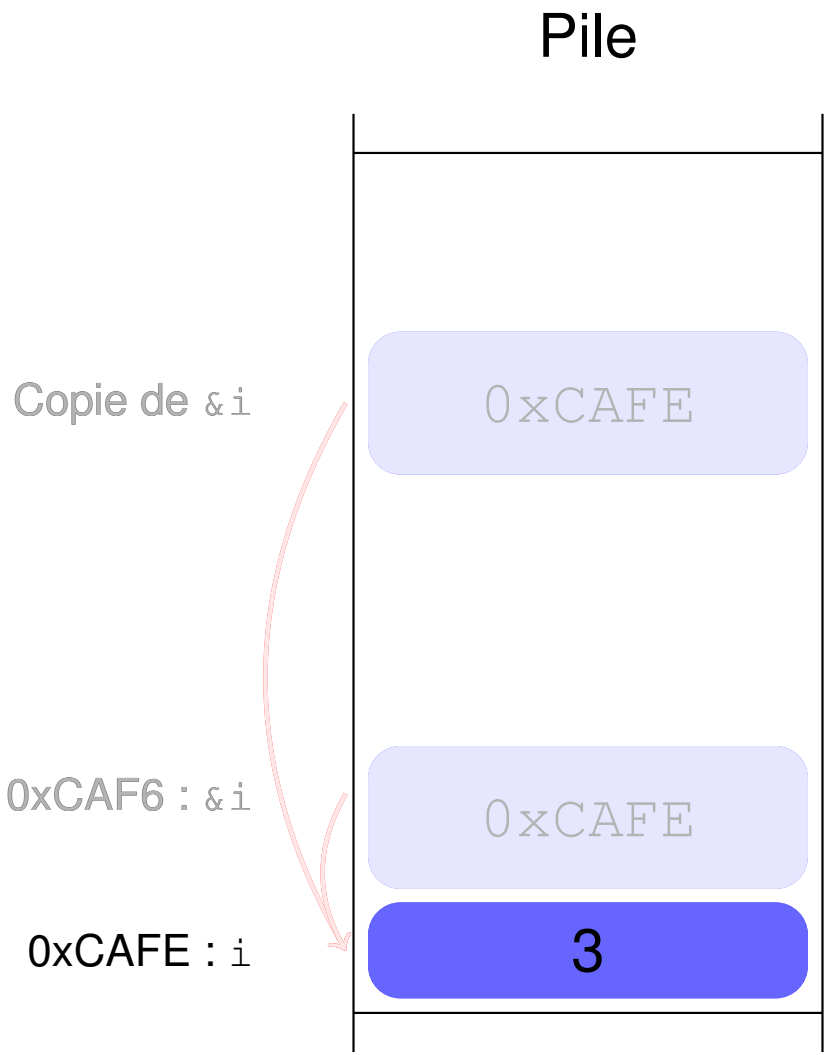
- Plus « risqué »
  - Hygiène de programmation plus stricte
- Syntaxe « moins transparente »

- Parfois (rarement) pas d'autre choix en `C++`

Mécanisme

```
void f(int * i)
{
   printf("%d\n", *i);
   (*i)++;
   printf("%d\n", *i);
}

int main()
{
   int i = 2;
   f(&i);
   printf("%d\n", i);
}
```



Exemple

- Fichier `swap-addr-wrong.cpp`

```
1  void swap(int * x, int * y)
2  {
3    cout << "Entering swap : " << *x << " " << *y << endl;
4
5    int* tmp = y;
6    y = x;
7    x = tmp;
8
9    cout << "Exiting swap : " << *x << " " << *y << endl;
10 }
11
12 int main()
13 {
14   int i = 1;
15   int j = 2;
16
17   cout << "Before call : " << i << " " << j << endl;
18   swap(&i, &j);
19   cout << "After call : " << i << " " << j << endl;
20 }
```

Exemple

- Fichier `swap-addr.cpp`

```
1  void swap(int * x, int * y)
2  {
3    cout << "Entering swap : " << *x << " " << *y << endl;
4
5    int tmp = *y;
6    *y = *x;
7    *x = tmp;
8
9    cout << "Exiting swap : " << *x << " " << *y << endl;
10 }
11
12 int main()
13 {
14   int i = 1;
15   int j = 2;
16
17   cout << "Before call : " << i << " " << j << endl;
18   swap(&i, &j);
19   cout << "After call : " << i << " " << j << endl;
20 }
```

Exemple

- Fichier `pass-addr.cpp`

```
1  void countDown(int * i)
2  {
3    while(*i > 0)
4    {
5      cout << *i << endl;
6      *i--;
7    }
8    cout << "BOOM" << endl;
9  }
10
11 int main()
12 {
13   for(int i = 5; i >= 0; i--)
14   {
15     countDown(&i);
16     cout << endl;
17   }
18 }
```

Passage par référence (C++)

- On ne transmet pas une copie de l'objet, mais l'objet lui-même
- Utilisation du caractère `&` après le type
  - Ce paramètre est transmis par référence
  - Le standard ne spécifie pas leur implémentation (souvent des pointeurs constants)
- Offre des gains de performances
- Diverses conséquences
  - Synchronisation
  - Immédiats
  - Pas de conversions possibles à l'appel

Exemple

```
void swap(int&, int&);
```

Exemple

- Fichier `swap-ref.cpp`

```
1  void swap(int& x, int& y)
2  {
3    cout << "Entering swap : " << x << " " << y << endl;
4
5    int tmp = y;
6    y = x;
7    x = tmp;
8
9    cout << "Exiting swap : " << x << " " << y << endl;
10 }
11
12 int main()
13 {
14   int i = 1;
15   int j = 2;
16
17   cout << "Before call : " << i << " " << j << endl;
18   swap(i, j);
19   cout << "After call : " << i << " " << j << endl;
20 }
```

Exemple

- Fichier `pass-ref.cpp`

```
1  void countDown(int& i)
2  {
3    while(i > 0)
4    {
5      cout << i << endl;
6      i--;
7    }
8    cout << "BOOM" << endl;
9  }
10
11 int main()
12 {
13   for(int i = 5; i >= 0; i--)
14   {
15     countDown(i);
16     cout << endl;
17   }
18 }
```

Retour d'une fonction

- Comme pour le passage de paramètre, le retour d'une fonction peut être effectué
  - par valeur (par défaut) : `int f();`
  - par adresse : `int f();`
  - par référence (C++) : `int& f();`

Attention

- Ne créez pas de pointeurs / références vers des temporaires
- Ils vont « pendouiller » (dangling)

Illustration

- Fichier `return.cpp`

```
1  string f1()
2  {
3      string s = "Hello World!";
4      return s; //returns a copy of s
5  }
6
7  string& f2() {
8      string s = "Hello World!"; string & rs = s;
9      return rs;
10 }
11
12 string* f3() {
13     string s = "Hello World!"; string * rs = &s;
14     return rs;
15 }
16
17 int main() {
18     cout << f1() << endl;
19     cout << f2() << endl; //undefined behaviour
20     cout << *(f3()) << endl; //undefined behaviour
21 }
```

Arguments par défaut

- Jusqu'à présent, une fonction était appelée avec le même nombre d'arguments que son prototype en requérait
- C++ permet de spécifier la valeur de certains paramètres s'ils sont omis
  - `double f(int x = 0) ...`
  - `double d = f(); //same as f(0)`
- Les valeurs par défaut des paramètres sont spécifiés dans la déclaration de la fonction
  - Si séparation déclaration / implémentation et spécification dans les deux cas : erreur
- Très pratique pour les constructeurs de classe
  - Cf. Ch. 4

Contraintes

Règles

1. Les arguments par défaut ne peuvent utiliser des variables locales
   - En particulier les autres paramètres
2. Les arguments par défaut ne peuvent pas utiliser `this`
3. Les arguments par défaut sont les derniers de la liste de paramètres

- `void f(int i = 5, long l, int j = 3);`
- Appel de `f(10, 20)`
- Exécute `f(5, 10, 20)` ou `f(10, 20, 3)` ?

Fichier `param-def.cpp`

```
1  int k = 2;
2
3  //void f(int n, int m = n * 2) {}
4
5  void g(int n, int m = k * 2, int p = 3)
6  {
7    cout << n << "␣" << m << p << endl;
8  }
9
10 int main()
11 {
12   //f(2);
13   g(2);
14 }
```

Fonctions lambda

Les lambdas

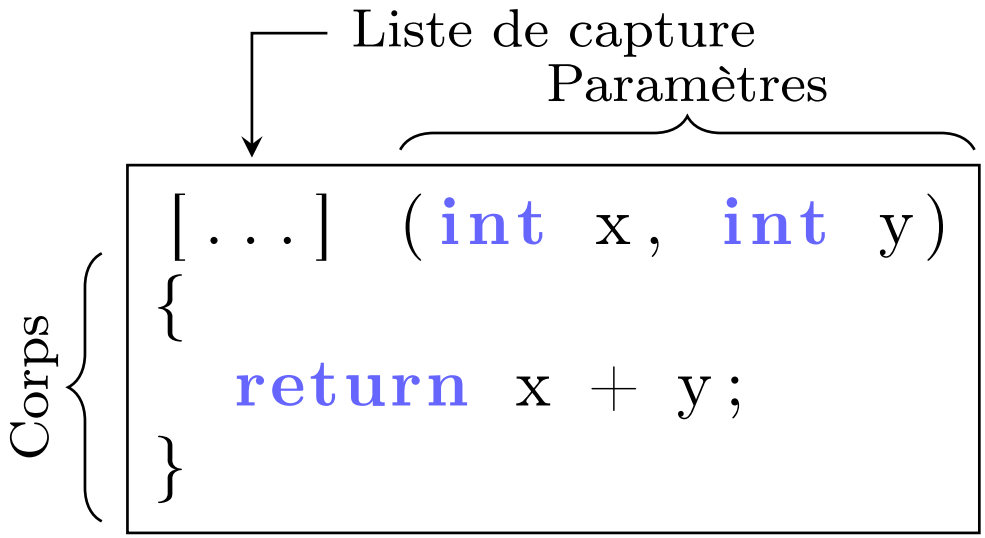
- Concept `C++` uniquement

Idée de base

- Écrire des fonctions (locales) à la volée

- Motivation : écrire une fonction indépendante est « trop verbeux » quand les fonctions sont destinées à un usage unique
- Compilé comme un objet fonction avec une surcharge de l'opérateur `()`
  - Cf. Ch. 8
- On peut construire des lambdas
  - à la volée et les passer comme paramètres d'une fonction
  - en les affectant dans une variable pour les utiliser plusieurs fois au sein d'un bloc
    - Le type de la variable est systématiquement déterminé par `auto`

Syntaxe



- Les paramètres et le corps du fonction sont spécifiés comme ceux d'une fonction « habituelle »
- La liste de capture possède une syntaxe particulière, mais peut être vide

Exemple

- Fichier `lambda.cpp`
- `std::for_each` (`algorithm.h`) est une fonction appliquant une fonction donnée à tous les éléments d'un conteneur itérable

```
1  int main()
2  {
3    vector<int> v = {1, 2, 3, 4, 5};
4    for_each(v.begin(), v.end(), [](int& i) { i++; });
5    for_each(v.begin(), v.end(), [](int i) { cout << i << endl; });
6  }
```

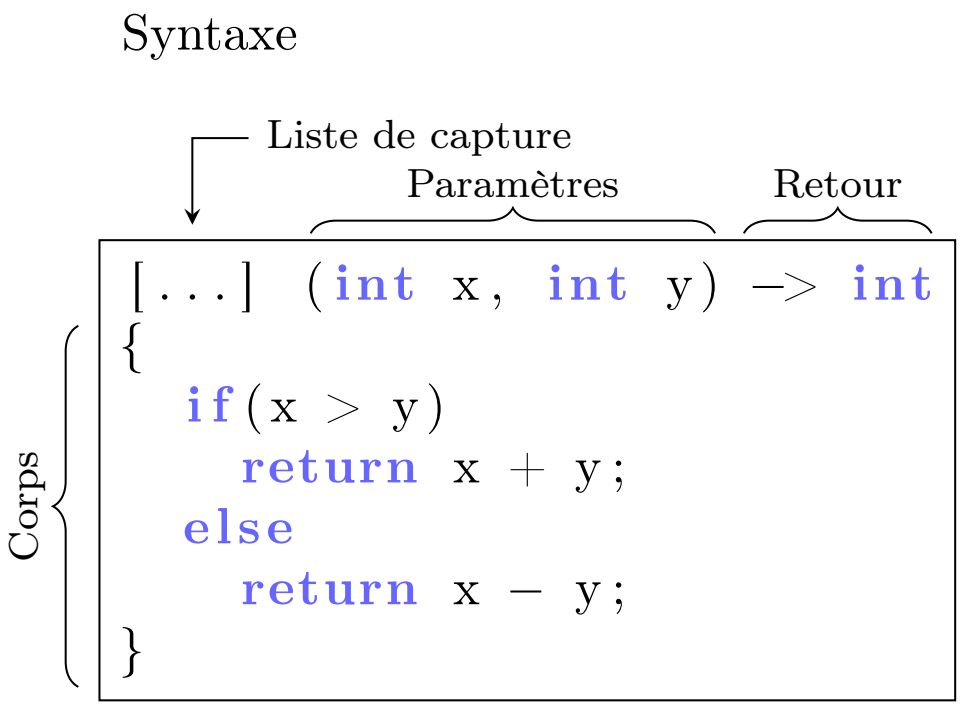
- C'est « court »

Remarque

- Avoir un code lisible est primordial
- Lambda courtes

Déduction du type de retour

- Dans l'exemple précédent, on n'a pas dû préciser le type de retour
- Il est « déduit » par le compilateur
- Si ce n'est pas possible (si `auto` ne le permet pas), il faut le préciser



Liste de capture

- Par défaut, rien en dehors des paramètres de la lambda ne peut être utilisé dans son corps
- La liste de capture permet d'inclure des éléments « extérieurs »

Syntaxe

- `[x]` : la variable `x` est passée par valeur
- `[&x]` : la variable `x` est passée par référence
- `[=]` : toutes les variables du bloc de déclaration sont passées par valeur
- `[&]` : toutes les variables du bloc de déclaration sont passées par référence

- Possibilité de combinaison
  - Par exemple : `[x, &y]`

Exemple

- Fichier `lambda-ret.cpp`

```
1  struct A { int i; };
2
3  int main()
4  {
5    A a; a.i = 1;
6    A b; b.i = 2;
7
8    auto f = [&a, b] (int i) //generic lambda
9      {
10       int k = a.i + b.i + i;
11       a.i += 3;
12       //b.i += 3; //error, b is read-only
13
14       return k;
15     };
16
17   cout << f(4) << endl;
18   cout << a.i << "␣" << b.i << endl;
19 }
```

Initialisation dans la liste de capture

- En C++14, un élément de la liste de capture peut être initialisé

```
1  int main()
2  {
3    int x = 4;
4    auto f = [&r = x, x = x + 1]()->int
5         {//r is a x-reference, x is incremented
6             r += 2;
7             return x + 2;
8         };//();
9
10   int k = f(); //comment that and uncomment stuff before
11
12   cout << x << endl; //6
13   cout << k << endl; //7
14 }
```

Rappel

- Avoir un code lisible est primordial

Constantes et inline

CV-Qualifiers

- Pour chaque type, incluant les types incomplets, il existe trois autres « sous-types »
  1. `const` : type constant, accédé en lecture seule
  2. `volatile` : type volatile, peut être modifié par un processus extérieur
  3. `const volatile` : les deux en même temps
- Motivation : optimisations compilatoires
  - Un `const` peut parfois être passé par référence
  - Les instructions comprenant des volatiles ne peuvent être réordonnées
- Applications
  - Optimisation de code
  - Multithreading

Constantes

- Valeur ne changeant pas au cours de l'exécution du programme
- Mot-clé `const`
- Réaffectation impossible
- Utilisable sur des fonctions membres (C++)
  - Ne modifie pas `this`
  - Cf. Ch. 4
- Objets (C++)
  - Modification des attributs impossible
  - Appel d'une fonction non constante impossible

Expressions `constexpr`

- Concept `C++` uniquement
  - En `C`, il faut utiliser des macros
- Variable, fonction ou constructeur *évaluable à la compilation*
- Implique `const`
- Utilisation du mot-clé `constexpr`
- Offre de grandes performances *à l'exécution*
  - Certains calculs sont effectués *une fois* à la compilation

Variable `constexpr`

Contraintes

1. Doit être un littéral
2. Doit être immédiatement assigné ou construit
   - Pas de déclaration sans assignation
   - Les paramètres doivent être des littéraux, des constantes ou fonctions `constexpr`
   - Le constructeur doit être `constexpr`

- Les contraintes ci-dessus offrent une possibilité d'évaluation et d'assignation de la variable à la compilation

Fonction `constexpr`

Contraintes

1. Ne peut pas être polymorphique
2. Son type de retour doit être un littéral
3. Les paramètres doivent être des littéraux, des constantes ou fonctions `constexpr`
4. Le corps ne peut pas contenir d'instruction non `constexpr`
5. Pas de `try` / `catch`
6. Une seule instruction `return` (pré C++14)
7. Pas de définition de variable non littérale
8. Etc.

Exemple

- Fichier `constexpr.cpp`
- Ne prêtez pas attention à la `struct constN`
  - Affichage en compile-time

```
1  constexpr double PI = atan(1) * 4;
2
3  constexpr int factorial(int n)//c++11 : recursion, one return statement
4  {
5      return n <= 1 ? 1 : n * factorial(n - 1);
6  }
7
8  constexpr long long int test(long long int n)//c++14
9  {
10   int i = n;
11   while(i >= 0)
12     i--;
13   return i;
14 }
```

- Un appel `test(9999999)` prend du temps à compiler

Fonctions inline

- Fonction dont le corps est substitué à l'appel

Avantages

- Gain de temps (pas de `call`)

Inconvénients

- Exécutable grossit (copier / coller)

- Non contraignant : « demande courtoise »
- Ces fonctions n'ont *pas* d'adresse

Contraintes

- Une fonction `inline` est soit
  - déclarée avec le mot-clé `inline`
  - implémentée dans le prototype de la classe (C++)
  - une fonction `constexpr` (C++)

Remarque

- Doit être déclarée et implémentée au sein du même fichier
- Ne peut pas être utilisée là où l'adresse d'une fonction est attendue

Exemple

- Fichier `inline.h`

```
1  struct A
2  {
3    void f() //inline
4    {
5      cout << "Brol::f" << endl;
6    }
7  };
8
9  struct B
10 {
11   inline void f();//inline
12 };
13
14 void B::f()
15 {
16   cout << "Foo::f" << endl;//defined in same file
17 }
18
19 inline double sum(double a, double b) { return a + b; } //inline
20
21 struct C
22 {
23   void f();//not inline
24 };
```

Exemple

- Fichiers `inline.cpp`, `inline-main.cpp`

```
1  void C::f()
2  {
3    cout << "C::f" << endl;
4  }
```

```
1  int main()
2  {
3    A a;
4    a.f();
5
6    B b;
7    b.f();
8
9    C c;
10   c.f();
11
12   cout << sum(2,3) << endl;
13 }
```

Les lvalue

Intuition

lvalue

- « Valeur qui peut apparaître à gauche d'un opérateur d'affectation »

- Définition insuffisante

Remarque

- `const int a = 2; //ok`
- `a = 3; //ko`

Notion de lvalue

Intuition

```
double x, y, a, b;
y = a*x + b; //ok
a*x + b = y; //ko
(x + 1) = 4; //ko
```

- Le premier opérande *doit* référencer un emplacement mémoire non temporaire
  - Autres langages : variable, en `C` / `C++`, pas assez précis
- Contrainte nécessaire à plusieurs endroits dans le langage
- Propriété très importante pour les *expressions*
- Ce qui n'est pas une `lvalue` est une *rvalue* (immédiats, temporaires, anonymes, etc.)

Conversions entre lvalues et rvalues

- Souvent, les opérateurs (et fonctions) requièrent des arguments rvalues

Exemple

- `int i = 1;`
- `int j = 2;`
- `int k = i + j;`

1. `i` et `j` sont des lvalue
2. `+` requiert des rvalue
3. `i` et `j` sont convertis en rvalue
4. Une rvalue est retournée

Règles de conversions

Conversion lvalue vers rvalue

- Toutes les lvalues qui ne sont pas des tableaux, des fonctions et des types incomplets peuvent être convertis en rvalues

Conversion rvalue vers lvalue

- Impossible implicitement
- Le résultat d'un opérateur (rvalue) peut être explicitement affecté en une lvalue

- On peut produire des lvalue à partir de rvalue explicitement
  - Le déférencement prend une rvalue et produit une lvalue
  - L'opérateur `&` prend une lvalue et produit une rvalue

Exemple

- Fichier `rvalue-conv.cpp`

```
1  int main()
2  {
3    int a[] = {1, 2};
4    int* pt = &a[0];
5    *(pt + 1) = 10;   //OK : p + 1 is an rvalue, but *(p + 1) is an lvalue
6
7    //taking adress
8    int i = 10;
9    //int* pti = &(i + 1);    // KO : lvalue required
10   int* pti = &i;            // OK: i is an lvalue
11   //&i = 20;                // KO : lvalue required
12
13   //reference making
14   //std::string& sref = std::string(); //KO : non const-ref init from rvalue
15 }
```

Références de `lvalue`

- Fichier `lvalue-ref.cpp`

```
1  int n = 5;
2  int& truc = n;
3
4  int& brol() { return n; }
5
6  int main() {
7    cout << brol() << endl;
8    brol() = 10;
9    cout << brol() << endl;
10
11   truc = 15;
12   cout << brol() << endl;
13
14   //int & i = 2;
15 }
```

- `n` est une `lvalue`, `brol()` aussi
- Les références de `lvalue` sont des `lvalue`
  - Utile pour l'opérateur `[]`
  - `v[10] = 42;`

La motivation des contraintes

- On ne veut pas pouvoir réaffecter un temporaire / immédiat

```
1  int a = 42;
2  int b = 43;
3
4   // a and b are both lvalues:
5  a = b; // ok
6  b = a; // ok
7  a = a * b; // ok
8
9  // a * b is an rvalue:
10 int c = a * b; // ok, rvalue on right hand side of assignment
11 a * b = 42; // error, rvalue on left hand side of assignment
12
13 //2 is a rvalue
14 int d = 2;
15 int & d = 2; //error
16 const int& e = 2; //ok : you are allowed to bind a const lvalue to a rvalue
```

- Il existe néanmoins des références de rvalue (Cf. Ch. 9)
- Le fait qu'une expression soit une rvalue ou une lvalue est appelé la *value category*

Règles d'appel

Surdéfinition

- On parle de surdéfinition (overloading) quand un même symbole possède plusieurs significations
- Le choix du symbole dépend du contexte

Exemple

- `a + b`
- Addition entière
- Addition flottante

- En C++, on peut redéfinir des fonctions
  - Ce qui inclut la plupart des opérateurs
- Des règles d'appel sont mises en œuvre pour le choix de la fonction à appeler en cas « d'ambiguïté »

Étapes dans l'appel d'une fonction

1. Name lookup
   - On recherche la définition d'un symbole
   - Pour des fonctions, cette recherche dépend du type des arguments
   - Pour des templates, il faut déduire le type des arguments (cf. Ch. 13)
2. Si ces étapes produisent plus d'une correspondance (surdéfinition), il faut résoudre l'ambiguïté
   - Overload resolution
   - En pratique, on cherche la « meilleure correspondance »
   - Les conversions ont un « score » (rank)
     - Les conversions de meilleur score sont privilégiées
     - Si on a le choix entre deux conversions de même score : erreur
3. Si elle ne peut pas être résolue : erreur
   - Aucune définition meilleure qu'une autre

Choix de fonction à appeler

- Idée : le compilateur cherche « la meilleure » correspondance possible

Règles d'appel

1. Correspondance exacte
   - Tous les types sont distingués
   - `const` intervient uniquement dans le cas de pointeurs et références
2. Correspondance de type « promotion »
   - Promotion entière, promotion flottante
3. Autres conversions
   - Conversion entière, conversion flottante, conversion `void*`, conversion définie par l'utilisateur, etc.

- Cas des références de rvalue : on privilégie un prototype explicite
- Fichier `surdef.cpp`

```
1  int f(int i)
2  {
3    cout << "Integer " << i << endl;
4    return 0;
5  }
6
7  //double f(int i) {} //return type matters not
8
9  //int f(const int i) {} //cv-qualifier lost
10
11 int f(double d)
12 {
13   cout << "Double " << d << endl;
14   return 0;
15 }
16
17 int f(int i, int j)
18 {
19   cout << "Integers " << i << " et " << j << endl;
20 }
21
22 int main()
23 {
24   int k = 1;
25   f(k);
26   double d = 2.1;
27   f(d);
28   f(k,d);
29 }
```

rvalue et cv-qualifiers

Conversions de rvalue cv-qualifiée

- Une lvalue de type `T` qui n'est ni une fonction ni un tableau peut être convertie en rvalue.
  1. Si ce n'est pas une classe, le type de la rvalue est la version cv non qualifiée de `T`
  2. Sinon, le type de la rvalue est `T`

Conséquence

- Les cv-qualifiers sont perdus sur les arguments explicites
- Pas sur `this`

Exemple

- Fichier `const-call.cpp`
- Ne vous souciez pas de `struct`
  - « Comme une classe » (cf. Ch. 4)

```
1  struct A
2  {
3      void brol() const { cout << "A::brol() const" << endl; }
4      void brol() { cout << "A::brol()" << endl; }
5
6      void brol2(A) { cout << "A::brol2(A)" << endl; }
7
8      //ERROR : cv-qualifier lost
9      //void brol2(const A) { cout << "A::brol2(const A)" << endl; }
10 };
11
12 int main()
13 {
14     A a = A();//creates an A (rvalue to lvalue conv)
15     const A ca = A();//creates an A (rvalue to lvalue conv)
16
17     a.brol();  //brol
18     ca.brol(); //brol const
19
20     a.brol2(a);
21     a.brol2(ca);
22 }
```

Contraintes références

- Une fonction `void f(int&);` est appelée avec une référence
- *Doit* être une lvalue

Conséquences

- Pas d'immédiat
- Pas d'expression
- Pas de conversion ou tronquage

- Idée : sinon, on pourrait changer la valeur d'un temporaire, écrire `a*x+b = y;`, etc.
- Différent avec `const`

Exemple

- Fichier `ref-cstr.cpp`

```
1  void f(int&){}
2
3  int main()
4  {
5    const int n = 15;
6    int q;
7    f(q);
8    //f(2*q+3);//not a lvalue
9    //f(3);
10   //f(n);//wrong cv-qualifier
11
12   float x;
13   //f(x);//no truncation
14
15   short k;
16   //f(k);//no conversion
17 }
```

Exceptions

- Le prototype d'une fonction peut prendre par référence des paramètres constants

Exemple

- `void f(const int& n);`

- Ici, il est prévu que les paramètres soient constants, et donc non modifiables
- Permet un appel de `f` sur *toute* expression entière
  - `f(2)`, `f(3 * n)`, etc.
  - Conversions placées dans des variables temporaires transmises par référence
  - Risque de modification de temporaire supprimé

Exemple

- Fichier `surdef-ref.cpp`

```
1  void f(int & i)
2  {
3    cout << "Ref " << i << endl;
4  }
5
6  void f(const int & i)
7  {
8    cout << "Ref cst " << i << endl;
9  }
10
11 int main()
12 {
13   int n = 3;
14   const int m = 5;
15   f(n);
16   f(3);
17   f(int{4});
18   f(4 * n);
19   f(4 * m);
20   f(m);
21 }
```

Le cas des pointeurs

- Fichier `surdef-ptr.cpp`

```
1  //void brol(char*) { cout << "Brol char" << endl; }
2  void brol(double*) { cout << "Brol double" << endl; }
3  void brol(void*) { cout << "Brol void" << endl; }
4  void brol(int*) { cout << "Brol int" << endl; }
5  //void brol(int* const) { cout << "int const" << endl; } //try to uncomment that
6  void brol(const int*) { cout << "Const int" << endl; }
7
8  int main()
9  {
10   char * ptc;
11   double * ptd;
12   void* ptv;
13
14   brol(ptc);// char, try when remove brol(char*) -> void
15   brol(ptd);// double
16   brol(ptv);//remove brol(void*) for this call : ERROR (no conv)
17
18   int n = 3;
19   const int p = 5;
20
21   brol(&n);
22   brol(&p);
23 }
```