Ch. 11 - Conversions

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

9 décembre 2020

Table des matières

Introduction

C++ et les conversions

- Pour les types de base en C / C++, les conversions ont déjà été abordées
- Pour les autres types, en C++, il existe deux types de conversions
  1. Explicites : demandées par l'utilisateur, via une instruction
  2. Implicites : mises en place automatiquement par le compilateur
- Dans les deux cas, certaines conversions sont illégales

Opérateurs de conversion

- C++ : `static_cast`, `dynamic_cast`, `reinterpret_cast`, `const_cast`
- C : cast « régulier » (avec parenthèses)
  - Combinaisons de casts C++ (sans `dynamic_cast`)

Conversions explicites

- On a déjà vu plusieurs types de conversions explicites avec les types de base

Exemple

- `int n; double x;`
- `{ ... }`
- `x = double(n);`
- `n = int(x);`

- Conversions explicites, appelées avec l'opérateur de cast
  - Toujours autorisé avec des types de base
  - Tronquage possible

Conversions implicites

- « Non demandées » par l'utilisateur
- Mises en place par le compilateur en fonction du contexte

Exemple

- Affectation : conversion dans le type de la variable réceptrice
- Appel de fonction : conversion d'un paramètre dans le type déclaré dans le prototype
- Au sein d'une expression : pour chaque opérateur, un opérande peut être convertie dans le type de l'autre

- Certaines conversions ne sont pas légales
  - Celles qui impliquent des classes
  - Pointeurs, sauf vers `void*`

Conversions définies par l'utilisateur

- C++ permet de définir soi-même des conversions implicites autorisées
- Permet de faire implicitement des conversions de type
  1. Classe → Type de base
  2. Type de base → Classe
  3. Classe → Classe
- Mises en place via
  - la surcharge d'un opérateur de cast
  - l'écriture d'un constructeur « de conversion »
- Allège l'écriture de certains codes
- Des conversions en chaîne peuvent avoir lieu

Bonnes pratiques

- Les conversions définies par l'utilisateur peuvent s'avérer dangereuses
  - Le compilateur peut effectuer des conversions là où vous ne le voudriez pas

Hygiène de programmation

- Ne pas abuser
- Écrire des conversions implicites « sensées »

- Exemple
  1. Oui : `fraction` → `int`, `int` → `fraction`
  2. Non : `int` → `vector`, `vector` → `int`

Conversions explicites

Conversons explicites

- En C / C++, on a vu comment effectuer des conversions explicites entre types de base
  - Avec un appel explicite à l'opérateur de cast
- En C++, il y a plusieurs autres types de conversions explicites
  1. `static_cast` : vérification à la compilation
  2. `dynamic_cast` : vérification à l'exécution (polymotphisme)
  3. `const_cast` : supprime la cv-qualification
  4. `reinterpret_cast` : conversion de motifs binaires
- Le cast régulier effectue une combinaison des précédents, hors `dynamic_cast`
- Utiliser l'opérateur en fonction des besoins

L'opérateur `static_cast`

- Conversion avec vérification à la compilation
  - *Pas* à l'exécution
  - Rapide
- Utilisation : `static_cast<A>(b)`
  - Convertit `b` vers un type `A`
- Permet
  - « d'inverser » une conversion implicite
  - d'effectuer un « downcast » dans le cas d'héritage
  - d'effectuer des conversions tableaux → pointeurs, etc.

Hygiène de programmation

- Utiliser au maximum quand on est certain du succès
  - Vérification à l'exécution inutile

Exemple

- Fichier `static.cpp`

```
1  int plusOne(void* pv)
2  {
3    int * pi = static_cast<int*>(pv);
4    (*pi)++; //ask why () are important
5    return *pi;
6  }
7
8  struct B {}; //base class
9  struct D : B {}; //D inherits from B
10
11 int main()
12 {
13   int i = 2;
14   cout << plusOne(&i) << endl;
15   cout << i << endl;
16
17   D d;
18   B& br = d; // upcast via implicit conversion
19   D& another_d = static_cast<D&>(br); // manual downcast
20
21   D a[10];
22   B* dp = static_cast<B*>(a);//array to pointer + upcast
23 }
```

L'opérateur `dynamic_cast`

- Conversion avec vérification à l'exécution
  - Plus lent que `static_cast`
- Utilisation : `dynamic_cast<A>(b)`
- Pratique pour effectuer des conversions qui peuvent échouer
  - downcast polymorphique
- Si échec avec
  - un pointeur : retourne un `nullptr`
  - une référence : lance une exception `bad_cast`

Hygiène de programmation

- Utiliser pour conversions polymorphiques qui peuvent échouer
  - Vérification à l'exécution indispensable

Exemple

- Fichier `dynamic.cpp`

```
1  struct B
2  {
3      virtual void f() {};  // must be polymorphic to use runtime-checked dynamic_cast
4  };
5  struct D : B {}; //D inherits from A
6
7  int main()
8  {
9      D d;
10     B& b = d; //upcast
11     D& new_d = dynamic_cast<D&>(b); // downcast
12
13     B* b1 = new B;
14     D* d1 = static_cast<D*>b1; //ok but D component is invalid
15     if(D* d = dynamic_cast<D*>(b1))
16     {
17         std::cout << "downcast from b1 to d successful" << endl;
18     }
19
20     B* b2 = new D;
21     if(D* d = dynamic_cast<D*>(b2))
22     {
23         std::cout << "downcast from b2 to d successful" << endl;
24     }
25 }
```

L'opérateur `const_cast`

- Permet d'enlever la cv-qualification de pointeurs et références
  - Ne fonctionne pas sur les pointeurs de fonctions
- Utilisation : `const_cast<A>(b)`
- Pratique si une référence (ou pointeur) est constant, mais pas l'objet référencé
- Si la conversion échoue, le comportement est indéterminé

Hygiène de programmation

- Éviter d'écrire du code requérant un `const_cast`
- Peu lisible, peut induire l'utilisateur en erreur

Exemple

- Fichier `const.cpp`

```
1  struct A
2  {
3      A() : i(3) {}
4      void m1(int v) const
5      {
6          // this->i = v;                 // compile error: this is a pointer to const
7          const_cast<A*>(this)->i = v; // OK as long as the type object isn't const
8      }
9      int i;
10 };
11
12 int main()
13 {
14     int i = 3;
15     const int& cref_i = i; //const ref
16     const_cast<int&>(cref_i) = 4; // OK: modifies i
17     cout << "i = " << i << endl;
18
19     A a;
20     a.m1(4); //if a is const : undefined behaviour
21     cout << "A::i = " << a.i << endl;
22
23     const int j = 3; // j is declared const
24     int* pj = const_cast<int*>(&j);
25     *pj = 4;          // undefined behavior!
26 }
```

L'opérateur `reinterpret_cast`

- Permet de réinterpréter un motif binaire
- Ne permet *pas* de retirer la cv-qualification
- Utilisation : `reinterpret_cast<A>(b)`
- Très dépendant de l'architecture
  - Système little/big indian ?
- Souvent, peu de garanties sur le résultat

Hygiène de programmation

- Ne pas utiliser

Exemple

- Fichier `reinterpret.cpp`

```
1  struct A {  char a, b, c, d; };
2
3  int main()
4  {
5      int i = 7;
6
7      char* p2 = reinterpret_cast<char*>(&i);
8      if(p2[0] == '\x7')
9          cout << "This␣system␣is␣little-endian" << endl;
10     else
11         cout << "This␣system␣is␣big-endian" << endl;
12
13     i = 1094861636; //0x41424344
14     A &p = reinterpret_cast<A&>(i); //if little endian : D C B A
15
16     cout << p.a << "␣" << p.b << "␣" << p.c << "␣" << p.d << endl;
17 }
```

Le cast régulier

- Effectue une combinaisons des opérateurs précédents
  - *Pas* `dynamic_cast`
- Le compilateur tente, dans cet ordre, d'effectuer
  1. `const_cast`
  2. `static_cast`
  3. `static_cast` suivi de `const_cast`
  4. `reinterpret_cast`
  5. `reinterpret_cast` suivi de `const_cast`

Hygiène de programmation

- Éviter : utiliser `static_cast` autant que possible

Le mot-clé `explicit`

- Empêche qu'un constructeur de conversion ou qu'un opérateur de cast soit utilisé lors de conversions implicites
- `explicit` sur un autre prototype provoque une erreur de compilation
- Exemple dans le standard : constructeur de `vector`
  - On en peut pas écrire `vector<int> v = 2;`
- Pratique si l'on veut autoriser la conversion, mais que celle-ci est sujette à des erreurs si utilisée implicitement
  - `vector<int> v = 2;`
    - vecteur de taille 2, contenant deux zéros ?
    - vecteur de taille 1, contenant un unique 2 ?
  - C'est une bonne pratique d'avoir défini ce constructeur comme `explicit`
- Permet aussi de lever des ambiguïtés (cf. sections suivantes)

Exemple

- Fichier `explicit.cpp`

```
1  struct B
2  {
3      explicit B(int) {}
4      explicit B(int, int) {}
5      explicit operator int() const { return 0; }
6  };
7
8  int main()
9  {
10 //  B b1 = 1; // Error
11     B b2(2); // OK
12     B b3 {4,5}; // OK
13 //  B b4 = {4,5}; // Error
14 //  int nb1 = b2; // Error
15     int nb2 = static_cast<int>(b2); // OK
16     B b5 = (B)1; // OK
17 }
```

Classe→Type de base

Définition

- Mis en place via la surcharge de l'opérateur de cast

Exemple : conversions vers `int`

- `operator int() const { ... }`
- Autorise des écritures
  - `fraction f = ... ;`
  - `int n = f;`

- Toujours défini comme une fonction membre
- Le type de retour *ne doit pas* être précisé

Exemple

- Fichier `fraction.cpp`

```
1  class fraction
2  {
3    unsigned num, denom;
4    bool positive;
5
6    public:
7      fraction(int num, int denom) : ...
8      {
9        cout << "Call cstr " << num << " " << denom << endl;
10     }
11
12     fraction(const fraction& f) : ...
13     {
14       cout << "Call copy cstr" << endl;
15     }
16
17     operator int() const
18     {
19       cout << "Call cast " << num << " " << denom << " " << positive << endl;
20       return positive ? num / denom : -(num / denom); //ask why () needed
21     }
22 };
23
24 void f1(int n) { cout << "Call f1(int) " << n << endl; }
25 void f2(double x) { cout << "Call f2(double) " << x << endl; }
```

Exemple

- Fichier `fraction.cpp`

```
1  int main()
2  {
3    fraction a(5,-2), b(2,5);
4    int n1, n2;
5    double x1, x2;
6    cout << endl;
7
8    n1 = int(a); cout << "n1 = " << n1 << endl;
9    n2 = b; cout << "n2 = " << n2 << endl << endl;
10
11   f1(a); f2(a);
12   cout << endl;
13
14   n1 = a + 3; cout << "n1 = " << n1 << endl;
15   n2 = a + b; cout << "n2 = " << n2 << endl << endl;
16
17   x1 = a + 3; cout << "x1 = " << x1 << endl;
18   x2 = a + b; cout << "x2 = " << x2 << endl << endl;
19
20   n1 = a + 3.85; cout << "n1 = " << n1 << endl;
21   x1 = a + 3.85; cout << "x1 = " << x1 << endl;
22   x2 = a; cout << "x2 = " << x2 << endl << endl;
23 }
```

Debriefing

- Plusieurs choses se produisent sur ce code
  - Appel explicite et implicite de l'opérateur de cast à l'affectation
  - Appel implicite de l'opérateur de cast lors de
    - l'appel de fonction
    - l'évaluation d'une expression

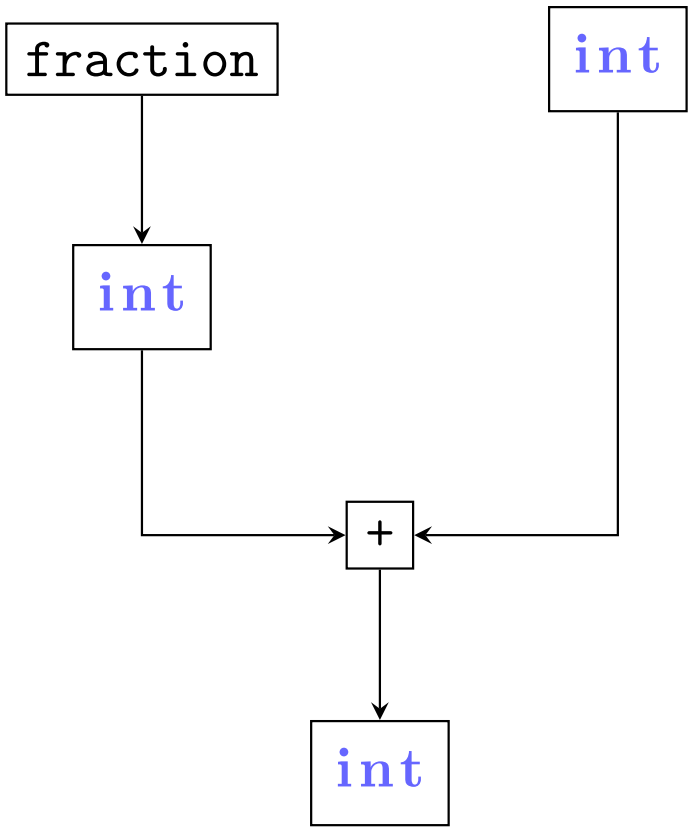
Remarque

- Aucun appel au constructeur de recopie n'est effectué

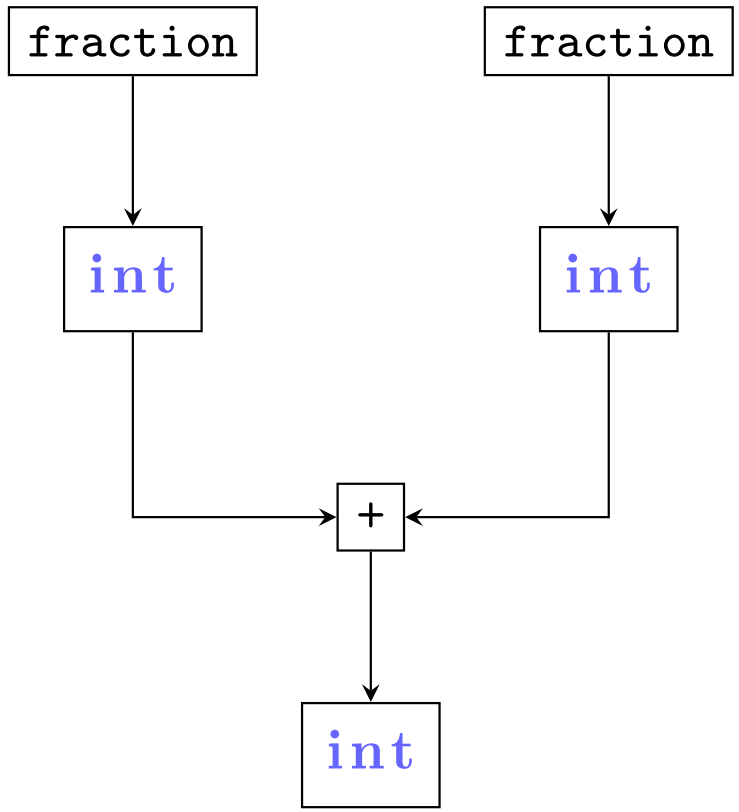
- Évaluation `a + 3`
  1. Existe-t-il un opérateur `+` entre `fraction` et `int` ?
  2. Existe-t-il des conversions possibles ?
     - Conversion de `a` en `int`
     - Application de l'opérateur `+` entre `int` et `int`

Conversions en chaîne

```
int i = a + 3;
```




```
int i = a + b;
```



- Plus de détails en fin de chapitre

Type de base→Classe

Définition

- Mis en œuvre via un constructeur à un paramètre
  - Paramètres par défaut possible
  - Si ambiguïté : rejet à la compilation

Exemple

- `fraction f = 12; //same as fraction f(12);`

1. Crée un objet temporaire de type `fraction`
2. Affecte cet objet à `f`

- Le compilateur accepte ce constructeur comme un « opérateur de conversion » de `int` vers `fraction`

Exemple

- Fichier `fraction-2.cpp`

```
1  class fraction
2  {
3    ...
4
5    public:
6      fraction(int num = 0, int denom = 1) : ...
7      {
8        cout << "Call cstr " << num << " " << denom << endl;
9      }
10
11     fraction(const fraction& f) : ...
12     {
13       cout << "Call copy cstr" << endl;
14     }
15 };
16
17 void f(fraction f) { cout << "Call f " << endl; }
18
19 int main()
20 {
21   fraction fr(1,2);
22   fr = fraction(3);
23   fr = 12;
24   f(5);
25 }
```

Rappel sur les lvalue

- Contraintes d'écriture, conversions, cv-qualifiers, règles d'appel, etc.
  - Cf. Ch. 3 - Fonctions

Conséquences

1. Si `f` avait été déclarée comme `void f(fraction&)`, l'appel à `f(5)` aurait été rejeté à la compilation
2. Si `f` avait été déclarée comme `void f(const fraction&)`, l'appel à `f(5)` aurait été accepté
   1. Conversion de `5` en `fraction`
   2. Création d'une fraction temporaire
   3. Transmission de la fraction par référence

Choix entre constructeur et opérateur d'affectation

- Fichier `fraction-3.cpp`

```
1  class fraction
2  {
3    unsigned num, denom;
4    bool positive;
5
6    public:
7      fraction(int num = 0, int denom = 1) : ...
8      {
9        cout << "Call cstr " << num << " " << denom << endl;
10     }
11
12     fraction& operator =(const fraction& f)
13     {
14       cout << "Aff. fraction " << f.num << " " << f.denom << " " << f.positive << endl;
15       ...
16       return *this;
17     }
18
19     fraction& operator =(const int n)
20     {
21       cout << "Aff. int " << n << endl:
22       ...
23       return *this;
24     }
25 };
```

Règle de conversion

Question

- Que fait une instruction `f = 12;` ?

- Deux choix possibles
  1. Conversion `int` → `fraction` suivi d'une affectation `fraction` → `fraction`
  2. Conversion `int` → `fraction`

Règle

- Les conversions définies par l'utilisateur, de type cast ou constructeur, ne sont mises en œuvre que lorsque cela est nécessaire

Exemple d'utilisation : opérateurs arithmétiques

- Objectif : définir des opérateurs arithmétiques « symétriques »

Interface publique

- `fraction::fraction(int n = 0, int num = 1);`
- `fraction fraction::operator *(const fraction& f) const;`

Problème

- On peut écrire `f * 2;` mais pas `2 * f;`
- Comme `*` est membre, le premier paramètre (`this`) est toujours de type `fraction`

Solution

1. Garder le constructeur « de conversion »
2. Définir `*` comme indépendant et amie de `fraction`

Avantages

- Symétrie
- Efficace
- Permet de faire des chaînes de conversions

- Exemple avec constructeur de conversion de `int`
  - `A * double`, `A * int`, `A * float`, `A * long`, `A * short`, `A * char` et symétriques, etc.

Illustration

- Fichier `fraction-4.cpp`

```
1  class fraction
2  {
3    unsigned num, denom;
4    bool positive;
5
6    public:
7      fraction(int num = 0, int denom = 1) : ...
8      {}
9
10         friend fraction operator * (fraction,fraction);
11 };
12
13 fraction operator *(fraction f1, fraction f2)
14 {
15     return fraction(f1.num * f2.num, f1.denom * f2.denom);
16 }
```

Classe→Classe

Définition

- Mis en œuvre soit via
  1. la surcharge d'un opérateur de cast
  2. l'écriture d'un constructeur de conversion
- Les règles d'application dans les conversions Classe → Type de bas et Type de base → Classe sont d'application
- Les ambiguïtés sont rejetées à la compilation
  - On ne peut pas utiliser les deux mécanismes
- La « qualité » de la conversion est laissée à la discrétion du programmeur

Exemple cast

- Fichier `fraction-5.cpp`

```
1  class Fraction
2  {
3    unsigned num, denom;
4    bool positive;
5
6    public:
7      Fraction(int num, int denom) : ... {}
8
9      operator ErrNbr();
10 };
11
12 class ErrNbr
13 {
14   double f, error;
15
16   public:
17     ErrNbr(double f, double error = 0) : f(f), error(error) {}
18     friend Fraction::operator ErrNbr();
19 };
20
21 Fraction::operator ErrNbr()
22 {
23   return positive ? ErrNbr((num + 0.) / denom) : ErrNbr(-((num + 0.) / denom));
24 }
```

Exemple constructeur

- Fichier `fraction-6.cpp`

```
1  class ErrNbr
2  {
3    double f, error;
4
5    public:
6      ErrNbr(double f, double error = 0) : f(f), error(error) {}
7      ErrNbr(Fraction f);
8  };
9
10 class Fraction
11 {
12   unsigned num, denom;
13   bool positive;
14
15   public:
16     Fraction(int num, int denom) : ... {}
17     friend ErrNbr::ErrNbr(Fraction);
18 };
19
20 ErrNbr::ErrNbr(Fraction f)
21   : f(f.positive ? (f.num + 0.) / f.denom : -((f.num + 0.) / f.denom)), error(0) {}
```

- Remarquez les différences dans les déclarations anticipées

Conversions en chaîne

Conversions en chaîne

Rappel

- Des conversions implicites en chaîne de types numériques sont effectuées
- Le compilateur privilégie les promotions numériques aux autres conversions

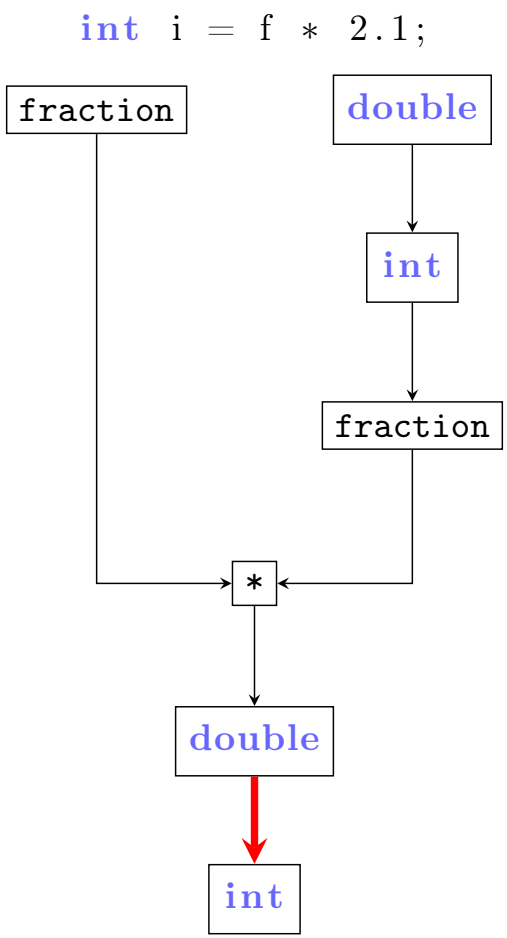
- En C++, le compilateur teste, dans cet ordre
  1. Une conversion standard
  2. Une conversion définie par l'utilisateur
  3. Une conversion standard
- Il est impossible d'effectuer en chaîne, implicitement,
  - deux conversions définies par l'utilisateur
  - les trois actions ci-dessus
- Les ambiguïtés (deux chaînes possibles) sont rejetées à la compilation

Conversion standard puis CDU

- Fichier `chain1.cpp`

```
1  class fraction
2  {
3    ...
4
5    public:
6      fraction(int num = 0, int denom = 1) : ...
7      {  }
8
9      explicit operator int() const
10     {
11       return positive ? num / denom : -(num / denom);
12     }
13
14     friend fraction operator * (fraction,fraction);
15 };
16
17 int main()
18 {
19   fraction f {1,2};
20   int i = static_cast<int>(f * 2.1);
21 }
```

Illustration

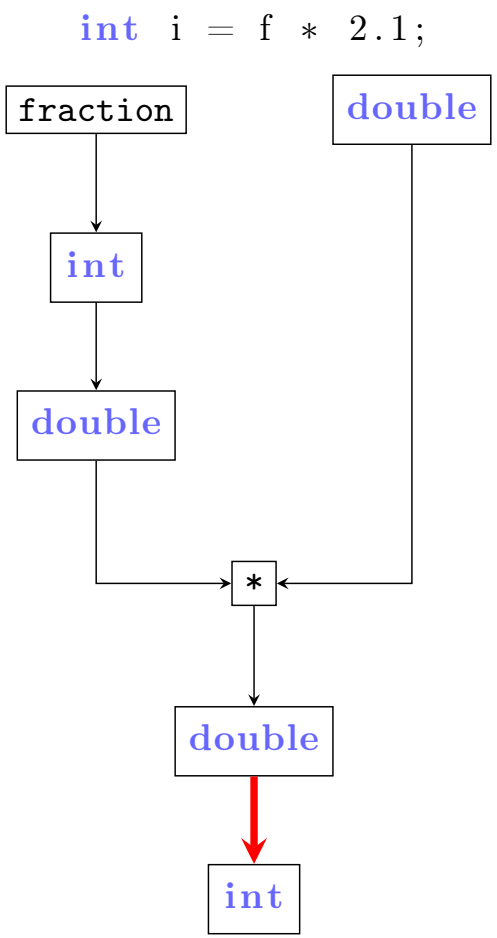


CDU puis conversion standard

- Fichier `chain2.cpp`

```
1  class fraction
2  {
3    ...
4
5    public:
6      explicit fraction(int num = 0, int denom = 1) : ...
7      {  }
8
9      operator int() const
10     {
11       return positive ? num / denom : -(num / denom);
12     }
13
14     friend fraction operator * (fraction,fraction);
15 };
16
17 int main()
18 {
19   fraction f {1,2};
20   int i = (f * 2.1);
21 }
```

Illustration

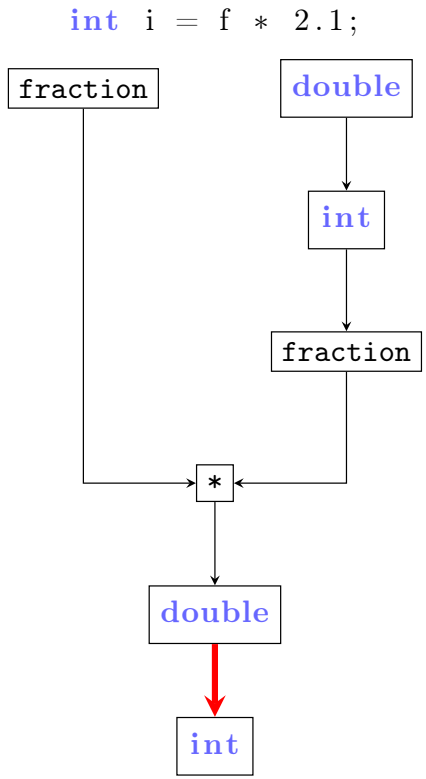


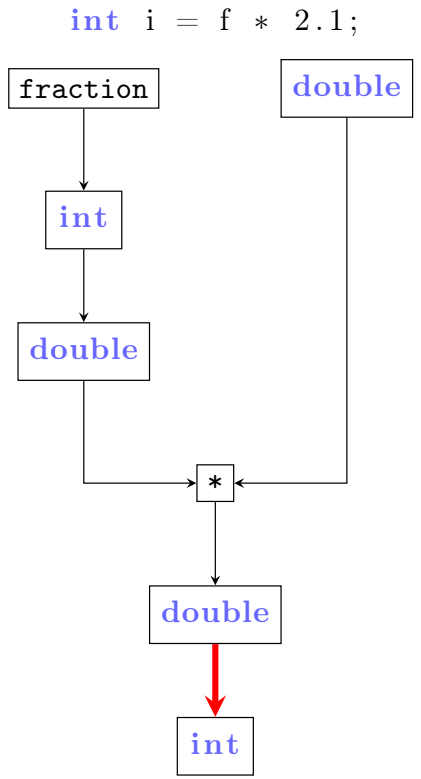
Ambiguïté

- Fichier `chain3.cpp`

```
1  class fraction
2  {
3    ...
4
5    public:
6      fraction(int num = 0, int denom = 1) : ...
7      {  }
8
9      operator int() const
10     {
11       return positive ? num / denom : -(num / denom);
12     }
13
14     friend fraction operator * (fraction,fraction);
15 };
16
17 int main()
18 {
19   fraction f {1,2};
20   int i = (f * 2.1);
21 }
```

Illustration





- Utiliser `explicit` là où, sémantiquement, on perd de la précision
  - Ici, sur l'opérateur de cast (qui fait la division entière)

Autre exemple d'ambiguïté

```
1  class fraction
2  {
3    explicit fraction(int n = 0, int denom = 1);
4    operator int();
5    operator double();
6  };
```

```
1  int i = f * 2.1;
```