Les bases du C++ (moderne)

Joel Falcou

NumScale

10 mai 2015

Un peu d'histoire ...

C++ à travers les âges

- 1978 Bjarne Stroustrup travaille sur Simula67 qui s'avére peu efficace
- 1980 Démarrage du projet *C with Classes*, du C orienté objet compilé via `CFront`.
- 1983 *C with Classes* devient C++
- 1985 Parution de *The C++ Programming Language*
- 1998 Première standarisation de C++ : ISO/IEC 14882 :1998
- 2005 Parution du *C++ Technical Report 1* qui deviendra C++0x
- 2011 Ratification de C++0x sous le nom C++11
- 2014 Ratification de C++14
- 2017,2020,... Prochaines *milestones* du langage

C++ et C – Une histoire de famille

C++ comme héritier du C

- C++ est un sur-ensemble de C
- C++ tente de minimiser les discordances
- Possibilité de conserver une compatibilité binaire

C++, ce fils rebel

- C++ évolue afin de délaisser les éléments fondamentaux de C
- Changement drastique du modéle de programmation
- Vers un langage quasi-fonctionnel plus qu'objet et impératif

Pourquoi C++ est il d'actualité ?

Make simple things simple - Stroustrup 2014

- Les tâches simples doivent s'exprimer simplement
- Les tâches complexes ne doivent pas être outrageusement complexes
- Rien ne doit être impossible
- Ne pas sacrifier les performances !

Pourquoi évoluer

- C++ est perçu comme un langage d'expert
- C++11/14 réduit ce gap d'apprentissage
- Moins de travail pour les *Language Lawyers*

Base du langage

Types fondamentaux

Types numériques

- Entiers : `(unsigned) char`,`(unsigned) short`,`(unsigned) int`
- Entiers portables : `std::ptrdiff_t`, `std::uintptr_t`, `std::size_t`
- Réels IEEE754 : `float`, `double`
- Booléen : `bool` de valeur *true* ou *false*

```
char c = 'e';
short s = -32000;
int i = 154515;

unsigned char uc = 0xFF;
unsigned short us = 65535;
unsigned int ui = 2563489542;
```

Types fondamentaux

Types numériques

- Entiers : `(unsigned) char`,`(unsigned) short`,`(unsigned) int`
- Entiers portables : `std::ptrdiff_t`, `std::uintptr_t`, `std::size_t`
- Réels IEEE754 : `float`, `double`
- Booléen : `bool` de valeur *true* ou *false*

```
std::int8_t c = 'e';
std::uint16_t s = -32000;
std::uint32_t i = 0xDEADBEEF;
std::int64_t l = 0xC01DBABEF0011337L;

std::ptrdiff_t p = 5;
std::size_t sz = sizeof(l);
std::uintptr_t a = reinterpret_cast<std::uintptr_t>(&p);
```

Types fondamentaux

Types numériques

- Entiers : `(unsigned) char`,`(unsigned) short`,`(unsigned) int`
- Entiers portables : `std::ptrdiff_t`, `std::uintptr_t`, `std::size_t`
- Réels IEEE754 : `float`, `double`
- Booléen : `bool` de valeur *true* ou *false*

```
float f = 1.45f;
double d = 1.45;

float sf = 3.6e-4f;
double sd = -9e-25;
```

Types fondamentaux

Types numériques

- Entiers : `(unsigned) char`,`(unsigned) short`,`(unsigned) int`
- Entiers portables : `std::ptrdiff_t`, `std::uintptr_t`, `std::size_t`
- Réels IEEE754 : `float`, `double`
- Booléen : `bool` de valeur *true* ou *false*

```
bool t = true;
bool f = false;

bool x = t || (!f && t);

if(x) std::cout << "x est vrai" << std::endl;
```

Types fondamentaux

Qualificateurs de Types

- `const` : exprime la constance d'une valeur
- `*` : indique un type **pointeur**
- `&` : indique un type **référence**

```
float f = 8;
float const cf = 7;

f = 9;
cf = 10; // ERREUR : assignment of read-only variable 'cf'
```

Types fondamentaux

Qualificateurs de Types

- `const` : exprime la constance d'une valeur
- `*` : indique un type **pointeur**
- `&` : indique un type **référence**

```
int i = 1337, j = 42;
int *pn = nullptr, *pi = &i;

*pi = j;
 pn = pi;
```

Types fondamentaux

Qualificateurs de Types

- `const` : exprime la constance d'une valeur
- `*` : indique un type **pointeur**
- `&` : indique un type **référence**

```
// Exercice : Donnez le type des variables ci-dessous

int v1 = 1;
int* v2 = nullptr;
int const v3 = 1;
int const* v4 = nullptr;
int* const v5 = nullptr;
int const * const v6 = nullptr;
```

Types fondamentaux

Qualificateurs de Types

- `const` : exprime la constance d'une valeur
- `*` : indique un type **pointeur**
- `&` : indique un type **référence**

```
int& rn; // ERREUR : declared as reference but not initialized
int& pi = i, pj = j;

pi = -42;
pj = pi;
```

Types composites

Tableau style C

- Représente *N* éléments contigus de type T
- Stockage par lignes à partir de l'index 0
- Composable pour gérer tableaux multi-dimensionnels

Exemples

```
int data[5];
int code[] = {1, 58, 98, 97, 36};

for(int i=0;i<4;++i)
  data[i] = code[i];
```

Types composites

Tableau style C

- Représente *N* éléments contigus de type T
- Stockage par lignes à partir de l'index 0
- Composable pour gérer tableaux multi-dimensionnels

Exemples

```
  int matrix[3][4] = { {1,0,0,-1}, {0,1,0,0}, {0,0,1,+2} };

  for(int i=0;i<3;++i)
  {
    for(int j=0;j<4;++j)
      std::cout << matrix[i][j] << " ";
    std::cout << "\n";
  }
}
```

Types composites

Tableau style C

- Représente *N* éléments contigus de type T
- Stockage par lignes à partir de l'index 0
- Composable pour gérer tableaux multi-dimensionnels

Limitations

- Objet non-copiable
- Passage en paramètres complexe
- Attention au *pointer decay*

Types et variables

Chaîne de caractères

- Représentée par le type standard `std::string`
- Accessible via `#include <string>`
- Comparable, copiable, concatenable, redimensionnable
- Compatible avec les chaînes C

```
std::string empty;
std::string cstyle = "some text";
std::string repeat(9,'*');
std::string copy = cstyle;

repeat[5] = 'X';
copy[copy.size()-1] = '!';
```

Types et variables

Chaîne de caractères

- Représentée par le type standard `std::string`
- Accessible via `#include <string>`
- Comparable, copiable, concatenable, redimensionnable
- Compatible avec les chaînes C

```
 if(copy == cstyle)
  std::cout << "s1 et s2 sont identiques\n";
else
  std::cout << "s1 et s2 sont differentes\n";
```

Types et variables

Chaîne de caractères

- Représentée par le type standard `std::string`
- Accessible via `#include <string>`
- Comparable, copiable, concatenable, redimensionnable
- Compatible avec les chaînes C

```
x = repeat + repeat
std::cout << x << "\n";
```

Types et variables

Tableau dynamique

- Représenté par le type standard `std::vector<T>`
- Accessible via `#include <vector>`
- Comparable, copiable, redimensionnable
- Optimisé pour gérer tout type de contenu

```
std::vector<int> empty;
std::vector<int> data(3);
std::vector<int> values = {1,2,3,4,5,6,7};

for(std::size_t i=0;i<values.size();++i)
  data.push_back(values[i]);

data.resize(15);
```

Types et variables

Tableau statique

- Représenté par le type standard `std::array<T,N>`
- Accessible via `#include <array>`
- Propose une sémantique de premier ordre
- Binairement équivalent à `T[N]`

```
std::array<float,3> data;
std::array<float,7> values = {1,2,3,4,5,6,7};

for(std::size_t i=0;i<values.size();++i)
  data[i] = 3.f * values[i];

values = data;
```

Types et variables

Tableau statique

- Représenté par le type standard `std::array<T,N>`
- Accessible via `#include <array>`
- Propose une sémantique de premier ordre
- Binairement équivalent à `T[N]`

```
std::array<float,3> translate(std::array<float,3> const& p,std::array<float,3> const& v)
{
  std::array<float,3> that = p;
  for(int i=0;i<that.size();++i)
    that[i] += v[i];

  return that;
}
```

Types et variables

Inférence de type

- Renseigner le type d'une variable est parfois complexe
- Renseigner le type d'une variable est parfois redondant
- Recyclage du mot-clé `auto`
- Nouveau mot-clé `decltype`

```
auto f = 3.f;
auto s = "some C string";
auto* p = &f;
```

Types et variables

Inférence de type

- Renseigner le type d'une variable est parfois complexe
- Renseigner le type d'une variable est parfois redondant
- Recyclage du mot-clé `auto`
- Nouveau mot-clé `decltype`

```
std::vector<double> dd = {1.,2e-1,1e-2,1e-3};
auto deci = dd[1];
auto& rdeci = dd[1];
rdeci = 1e-1;

auto b = dd.begin();
```

Types et variables

Inférence de type

- Renseigner le type d'une variable est parfois complexe
- Renseigner le type d'une variable est parfois redondant
- Recyclage du mot-clé `auto`
- Nouveau mot-clé `decltype`

```
decltype(1/f + *b) z;

declype( dd[0] ) rd;
```

Flux console

La famille `cin/cout/cerr`

- Accessible via `#include <iostream>`
- `std::cout` : sortie console standard
- `std::cin` : entrée console standard
- `std::cerr` : sortie d'erreur standard

```
#include <iostream>

int main()
{
  int n;
  char c;
  std::cin >> n >> c;

  std::cout << std::string(n,c) << std::endl;
  std::cout << std::hex << 47086 << "\n";

  std::cerr << "log d'erreur\n";

}
```

Structure de boucle

Boucle for

- Effectue un nombre fini, déterministe d'itération
- Variante "automatique" sur les intervalles standards

```
auto ref = 1.;
std::vector<decltype(ref)> dd(10);

for(std::size_t i = 0;i<10;++i)
{
  ref /= 10.;
  dd[i] = ref;
}
```

Structure de boucle

Boucle for

- Effectue un nombre fini, déterministe d'itération
- Variante "automatique" sur les intervalles standards

```
auto ref = 1.;
std::vector<decltype(ref)> dd(10);

for(auto it = dd.begin();it != dd.end();++it)
{
  ref /= 10.;
  *it = ref;
}
```

Structure de boucle

Boucle for

- Effectue un nombre fini, déterministe d'itération
- Variante "automatique" sur les intervalles standards

```
auto ref = 1.;
std::vector<decltype(ref)> dd(10);

for(double& e : dd)
{
  ref /= 10.;
  e = ref;
}
```

Types de données abstraits

Énumération

- Encapsulation d'un ensemble discret de valeurs partageant une sémantique
- C++ permet le typage des `enum`
- C++ permet la spécification du support de type

```
#define NORTH_WIND 0
#define SOUTH_WIND 1
#define EAST_WIND 2
#define WEST_WIND 3
#define NO_WIND 4

int wind_direction = NO_WIND;
```

Types de données abstraits

Énumération

- Encapsulation d'un ensemble discret de valeurs partageant une sémantique
- C++ permet le typage des `enum`
- C++ permet la spécification du support de type

```
enum wind_directions { NO_WIND, NORTH_WIND, SOUTH_WIND, EAST_WIND, WEST_WIND};

wind_directions w = NO_WIND;
wind_directions e = 453; // ERREUR
```

Types de données abstraits

Énumération

- Encapsulation d'un ensemble discret de valeurs partageant une sémantique
- C++ permet le typage des `enum`
- C++ permet la spécification du support de type

```
enum Color { RED, GREEN, BLUE };
enum Feelings { EXCITED, MOODY, BLUE };
```

Types de données abstraits

Énumération

- Encapsulation d'un ensemble discret de valeurs partageant une sémantique
- C++ permet le typage des enum
- C++ permet la spécification du support de type

```
// Typage fort C++11
enum class Color { RED, GREEN, BLUE };
enum class Feelings { EXCITED, MOODY, BLUE };

Color color = Color::GREEN;
```

Types de données abstraits

Énumération

- Encapsulation d'un ensemble discret de valeurs partageant une sémantique
- C++ permet le typage des `enum`
- C++ permet la spécification du support de type

```
// un entier 8 bits est suffisant ici
enum class Colors : unsigned char { RED = 1, GREEN = 2, BLUE = 3 };
```

Types de données abstraits

La liste d'initialiseurs

- Notation générique pour un groupe d'éléments
- `std::initializer_list<T>`
- Utilisable en paramètre de fonction
- Utilisable dans une boucle `for`

```
auto l = {1,2,3,4,5};

for(auto e : l)
  std::cout << e << " ";
```

Types de données abstraits

La liste d'initialiseurs

- Notation générique pour un groupe d'éléments
- `std::initializer_list<T>`
- Utilisable en paramètre de fonction
- Utilisable dans une boucle `for`

```
double sum( std::initializer_list<double> x )
{
  double r = 0.;

  std::cout << "Somme de " << x.size() << " valeurs.\n";

  for(auto& e : x)
    r += e;

  return r;
}
```

Types de données abstraits

La liste d'initialiseurs

- Notation générique pour un groupe d'éléments
- `std::initializer_list<T>`
- Utilisable en paramètre de fonction
- Utilisable dans une boucle `for`

```
auto y = sum( {1.,2.,3.,4.,5.} );
```

Types de données abstraits

Paire et Tuple

- `std::pair` : structure contenant deux membres de deux types quelconques
- `std::tuple` : généralisation de `pair` à *N* types
- Copiable, assignable, intropectable

```
std::pair<float,int> chu(3.f,5);
float f = chu.first;
int i = chu.second;
```

Types de données abstraits

Paire et Tuple

- `std::pair` : structure contenant deux membres de deux types quelconques
- `std::tuple` : généralisation de `pair` à *N* types
- Copiable, assignable, intropectable

```
std::tuple<int,char> foo (10,'x');
auto bar = std::make_tuple ("test", 3.1, 14, 'y');

std::get<2>(bar) = 100;

int myint;
char mychar;

std::tie (myint, mychar) = foo;
std::tie (std::ignore, std::ignore, myint, mychar) = bar;
```

Types de données abstraits

Paire et Tuple

- `std::pair` : structure contenant deux membres de deux types quelconques
- `std::tuple` : généralisation de `pair` à *N* types
- Copiable, assignable, intropectable

```
mychar = std::get<3>(bar);

std::get<0>(foo) = std::get<2>(bar);
std::get<1>(foo) = mychar;
```

Types de données abstraits

Paire et Tuple

- `std::pair` : structure contenant deux membres de deux types quelconques
- `std::tuple` : généralisation de `pair` à *N* types
- Copiable, assignable, intropectable

```
std::size_t sz = std::tuple_size<decltype(foo)>::value;
```

Types de données abstraits

Structure

- Agrégat de valeur de type arbitraire
- On parle de membre (ou de champs)
- Type de données abstrait le plus simple

```
struct point3D
{
  float x,y,z;
};

int main()
{
  point3D p;
  point3D q = {1.f,2.f,3.f};

  p.x = q.y;
  p = q;
}
```

Aspect Impératif

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

Déclaration d'une fonction

```
type name( parameter1, parameter2, ...) { statements }
```

- `type` : Type de la valeur retournée par la fonction
- `name` : Identifiant de la fonction
- `parameter*` : Transfert d'information du point d'appel à la fonction
- `statements` : Corps de la fonction, *i.e* le code effectif de la fonction

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

```
double addition( double a, double b )
{
  return a + b;
}
```

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

```
void decimate( double* a )
{
  *a /= 10.;
}
```

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

```
void decimate( double& a )
{
  a /= 10.;
}
```

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

```
// C++11
auto confuzzle( double a, int& b, float c )
     -> decltype(c/b - b/a)
{
  b = static_cast<int>(c/a);
  return c/b - b/a;
}
```

Fonctions et surcharge

Notion de fonction

- Groupe nommé de `statements` appelable à volonté
- Élément fondamental de l'encapsulation en C et en C++
- Notion de paramètres et de valeur de retour

```
// C++14
auto confuzzle( double a, int& b, float c )
{
  b = static_cast<int>(c/a);
  return c/b - b/a;
}
```

Fonctions et surcharge

Définition

- Forme de polymorphisme ad-hoc
- En C : une fonction = un symbole
- En C++ : une fonction = une signature
- Une signature = symbole + type des paramètres + qualificateur

Exemples :

- `double f()`
- `double f(int)` - OK
- `double f(double,int)` - OK
- `double f(...)` - OK
- `int f()` - KO

Fonctions et surcharge

Fonctions génériques

- Généralisation de la surcharge de fonction
- Généralisation/abstraction des types des paramètres
- Déduction automatique des types

```
template<typename T> T min(T const& a, T const& b)
{
  return a<b ? a : b;
}
```

Fonctions et surcharge

Fonctions génériques

- Généralisation de la surcharge de fonction
- Généralisation/abstraction des types des paramètres
- Déduction automatique des types

```
auto a = min(13.,37.);
auto b = min(4.f,3.f);
auto c = min(4,3);
auto d = min('e','z');
```

Fonctions et surcharge

Fonctions variadiques

- Remplacement type-safe du . . . du C
- Notion de *paramaters pack*
- Déduction automatique des types

```
int sum()
{
  return 0;
}
```

Fonctions et surcharge

Fonctions variadiques

- Remplacement type-safe du . . . du C
- Notion de *paramaters pack*
- Déduction automatique des types

```
int sum()
{
  return 0;
}

template<typename T0, typename... Ts> auto sum(T0 v0, Ts... vs)
{
  return v0+sum(vs...);
}
```

Fonctions et surcharge

Fonctions variadiques

- Remplacement type-safe du . . . du C
- Notion de *paramaters pack*
- Déduction automatique des types

```
int sum()
{
  return 0;
}

template<typename T0, typename... Ts> auto sum(T0 v0, Ts... vs)
{
  return v0+sum(vs...);
}

int main(int argc, char** )
{
  auto a = sum(argc,argc,argc,argc);
  std::cout << a << "\n";

  return 0;
}
```

Fonctions et inférence de type

Impact sur le retour des fonctions

- `auto` et `decltype` simplifient l'écriture du prototype des fonctions
- Notion de *trailing return type*

```
template<typename T1, typename T2>
/* ????? */
add(T1 const& a, T2 const& b)
{
  return a+b;
}
```

Fonctions et inférence de type

Impact sur le retour des fonctions

- `auto` et `decltype` simplifient l'écriture du prototype des fonctions
- Notion de *trailing return type*

```
// C++ 11
template<typename T1, typename T2>
auto add(T1 const& a, T2 const& b) -> decltype(a+b)
{
  return a+b;
}
```

Fonctions et inférence de type

Impact sur le retour des fonctions

- `auto` et `decltype` simplifient l'écriture du prototype des fonctions
- Notion de *trailing return type*

```
// C++ 14
template<typename T1, typename T2>
auto add(T1 const& a, T2 const& b)
{
  return a+b;
}
```

Surcharge des opérateurs

Objectifs

- Rendre une classe similaire à un type de base
- Renforcer la sémantique et simplifier la syntaxe
- Attention aux abus !

Syntaxe

- Operateur unaire membre : `type operator!()`
- Operateur binaire membre : `type operator+(type)`
- Operateur unaire : `type operator-(type)`
- Operateur binaire : `type operator+(type,type)`

Surcharge des opérateurs

```
struct rational
{
  int numerator() const { return num; }
  int denumerator() const { return denum; }

  rational operator-() const { return {-num,denum}; }

  rational& operator*=(rational const& rhs)
  {
    num *= rhs.num;
    denum *= rhs.denum;
    return *this;
  }

  int num,denum;
};

rational operator*(rational const& lhs, rational const& rhs)
{
  rational that = lhs;
  return that *= rhs;
}
```

Fonction anonyme

Objectifs

- Augmenter la localité du code
- Simplifier le design de fonctions utilitaires
- Notion de *closure* et de fonctions d'ordre supérieur

Principes

- Bloc de code fonctionnel sans identité
- Syntaxe :

```
auto f = [ capture ] (parametres ... ) -> retour
         {
           corps de fonction;
         };
```

Fonction anonyme

Type de retour

- C++11 : Automatique si la fonction n'est qu'un `return`
- C++11 : Autre cas, à spécifier via ->
- C++14 : Déduction automatique

Fonction anonyme

Paramètres

- C++11 : types concrets, pas de variadiques

```
auto somme = [](int a, int b) { return a+b }
```

- C++14 : types génériques et variadiques

```
auto somme = [](auto a, auto b) { return a+b; }
auto as_tuple = [](auto... args) { return std::make_tuple(args...); }
```

Fonction anonyme

Capture de l'environnement

- [] : environnement vide
- [a] : capture a par copie
- [&a] : capture a par référence
- [=] : tout par copie
- [&] : tout par référence

```
int x,n;

auto f = [x](int a, int b) { return a*x+b; }
auto g = [&x]() -> void { x++; }
auto h = [&x,n]() -> void { x *=n; }
auto s = [&]() { x = n; n = 0; return x; }
```

Règles de résolution

Processus Général [1]

- Les variantes du symbole sont recherchées pour créer l'*Overload Set* (Ω).
- Toute variante n'ayant pas le nombre de paramètres adéquat est éliminée pour obtenir le *Viable Set*.
- On recherche dans cet ensemble la *Best Viable Function*.
- On vérifie l'accessibilité et l'unicité de la sélection.

Que faire de tout ça ?

- Comment définir (Ω) ?
- Quels critères pour la *Best Viable Function* ?

[1] *C++ Templates : The Complete Guide* – David Vandevoorde, Nicolai M. Josuttis

Règles de résolution

Construction de Ω

- Ajouter toutes les fonctions non-template avec le bon nom
- Ajouter les variantes templates une fois la substitution des paramètres templates est effectuée avec succés
- Ω est un treillis : les fonctions non-templates dominent les fonctions template

Sélection de la *Best Viable Function*

- Chaque argument est associé à un Séquence de Conversion Implicite (ICS)
- Chaque argument est trié par rapport à son ICS
- Si un argument n'est pas triable, le compilateur boude.

Règles de résolution

Les Séquence de Conversion Implicite

- Conversions standards (SCS)
  - Correspondance exacte
  - Promotion
  - Conversion
- Conversion utilisateur (UDCS) , composée comme :
  - une séquence standard
  - une conversion définie explicitement
  - une deuxiéme séquence standard
  - Un UDCS est meilleur qu'un autre si sa seconde SCS est meilleure que l'autre
- Séquence de conversion par ellipse C

Règles de résolution

```
#include <iostream>

void f(int) { cout << "void f(int)\n"; }
void f(char const*) { cout << "void f(char const*)\n"; }
void f(double) { cout << "void f(double)\n"; }

int main()
{
  f(1); f(1.); f("1"); f(1.f); f('1');
}
```

Résultats

- f(1) → void f(int)
- f(1.) → void f(double)
- f("1") → void f(char const*)
- f(1.f) → void f(double)
- f('1') → void f(int)

Règles de résolution

```
#include <iostream>

void f(int) { cout << "void f(int)\n"; }
void f(char const*) { cout << "void f(char const*)\n"; }
void f(double) { cout << "void f(double)\n"; }
template<class T> void f(T) { cout << "void f(T)\n"; }

int main()
{
  f(1); f(1.); f("1"); f(1.f); f('1');
}
```

Résultats

- `f(1)` → `void f(int)`
- `f(1.)` → `void f(double)`
- `f("1")` → `void f(char const*)`
- `f(1.f)` → `void f(T)`
- `f('1')` → `void f(T)`

Gestion des erreurs

Assertion

- Une assertion violée termine le programme
- Utile pour arrêter les fonctions s'apprétant à s'appliquer dans un cadre invalide
- Permet de capturer des problèmes logiques de développement

```
void f(int i)
{
  assert( i != 0 )

  std::cout << 1.f/i << "\n";
}
```

Gestion des erreurs

Exceptions

- Types de données structurés émis lors d'une erreur et potentiellement récupérable
- Utiles pour valider l'API publique d'un composant
- Utiles pour notifier une erreur externe.

```
void f(int i)
{
  if( i != 0 )
    throw std::domain_error("i is 0");

  std::cout << 1.f/i << "\n";
}
```

Gestion des erreurs

Exceptions

- Types de données structurés émis lors d'une erreur et potentiellement récupérable
- Utiles pour valider l'API publique d'un composant
- Utiles pour notifier une erreur externe.

```
try
{
  f(0);
}
catch( std::exception& e )
{
  std::cout << e.what() << "\n";
}
```

Entrés / Sorties

Notions de flux

Objectif

- Remplace et étend `FILE*`
- Interface homogène quel que soit le flux
- Support pour la mise en forme de sortie
- Tous les flux héritent de `ostream` ou `istream`

Type de flux

- Flux console
- Flux fichier
- Flux chaîne

Flux fichier

Les types `fstream`

- Accessibles via `#include <fstream>`
- `std::ofstream` : fichier ouvert en sortie
- `std::ifstream` : fichier ouvert en entrée
- `std::fstream` : fichier ouvert en entrée/sortie

```
int main()
{
  std::vector<float> data = {1,2,3,4,5,6,7};

  std::ofstream txtfile("data.txt" );
  std::ofstream binfile("data.bin", std::ofstream::binary );

  for(auto& e : data)
    txtfile << e;

  binfile.write ( reinterpret_cast<const char*>(data.data())
                , data.size()*sizeof(float)
                );
}
```

Flux de chaîne

Les types `strstream`

- Accessibles via `#include <sstream>`
- Permettent l'insertion/extraction au sein d'une chaîne
- `std::osstream` : chaîne en mode insertion
- `std::isstream` : chaîne en mode extraction

```
std::string values = "125 320 512 750 333";
std::istringstream iss(values);

for(int n = 0; n < 5; n++)
{
  int val;
  iss >> val;
  std::cout << val << std::endl;
}
```

Flux de chaîne

Les types `strstream`

- Accessibles via `#include <sstream>`
- Permettent l'insertion/extraction au sein d'une chaîne
- `std::osstream` : chaîne en mode insertion
- `std::isstream` : chaîne en mode extraction

```
std::ostringstream oss;

oss << "a= " << 13.7 << "\tb= "<< 255 << "\tc=" << '#' << "\n";

std::cout << oss.str();
```

Application

Lecture de fichier texte

- Lire ligne par ligne via `std::getline`
- Traîter chaque ligne par un `std::istringstream`

```
struct stat { int age; float poids, taille; };

int main()
{
  vector<stat> data;
  string ligne;

  ifstream fichier("data.txt" );

  while(getline(fichier, ligne))
  {
    int a;
    float p,t;

    istringstream src(ligne);
    src >> a >> p >> t;
    data.push_back( {a,p,t} );
  }
}
```

Interaction flux/Type utilisateur

```
struct stat { int age; float poids, taille; };

std::ostream& operator<<(std::ostream& os, stat const& s)
{
  os << s.age << s.poids << s.taille;
  return os
}

std::istream& operator>>(std::istream& is, stat& s)
{
  is >> s.age >> s.poids >> s.taille;
  return is
}

int main()
{
  string ligne;
  vector<stat> data;
  ifstream fichier("data.txt" );

  while(getline(fichier, ligne))
  {
    istringstream src(ligne);
    stat s;
    src >> s;
    data.push_back( s );
  }
}
```

La Bibliothèque Standard

La sainte trinité du standard

Conteneurs

- Encapsulation des TDA classiques
- Paramétrables au niveau type et mémoire
- Parcourables via des ...

La sainte trinité du standard

Conteneurs

- Encapsulation des TDA classiques
- Paramétrables au niveau type et mémoire
- Parcourables via des ...

Iterateurs

- Abstraction du pointeur
- Utilisables dans des ...

La sainte trinité du standard

Conteneurs

- Encapsulation des TDA classiques
- Paramétrables au niveau type et mémoire
- Parcourables via des ...

Iterateurs

- Abstraction du pointeur
- Utilisables dans des ...

Algorithmes

- Parcours décorrélé du Conteneur
- Garantie de complexité et de correction
- Paramétrables via des fonctions utilisateurs

Conteneurs standards

Conteneurs Séquentiels

- `vector`,`array`
- `list`,`forward_list`
- `deque`

Conteneurs Associatifs

- `set`,`map`
- `multi_set`,`multi_map`
- `unordered_set`,`unordered_map`
- `unordered_multi_set`,`unordered_multi_map`

Algorithmes standards

- all_of,any_of,none_of
- for_each
- count, count_if
- mismatch, equal
- find, find_if
- find_end, find_first_of
- search, search_n
- nth_element
- max_element,min_element
- transform
- copy,copy_if
- remove,remove_if
- replace,replace_if
- reverse,rotate,shuffle
- sort, stable_sort
- fill, iota
- accumulate
- inner_product

Algorithmes en action

```
bool match_pattern(MemoryBuffer const& mem)
{
  return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}

bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
  std::vector<MemoryBuffer>::const_iterator cit = mems.cbegin();

  for( ; cit != v.cend(); ++cit)
  {
    if (match_pattern(*cit))
      return true;
  }

  return false;
}
```

Algorithmes en action

```
bool match_pattern(MemoryBuffer const& mem)
{
  return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}

bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
  for(auto cit = mems.cbegin(); cit != v.cend(); ++cit)
  {
    if (match_pattern(*cit))
      return true;
  }

  return false;
}
```

Algorithmes en action

```
bool match_pattern(MemoryBuffer const& mem)
{
  return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}

bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
  for(auto const& mem : mems)
  {
    if (match_pattern(mem))
      return true;
  }

  return false;
}
```

Algorithmes en action

```
bool match_pattern(MemoryBuffer const& mem)
{
  return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
}

bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
  return find_if(std::cbegin(mems), std::cend(mems), match_pattern) != std::cend(mems);
}
```

Algorithmes en action

```
bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
  return find_if( cbegin(mems), cend(mems)
                , [](MemoryBuffer const& mem)
                  {
                    return mem.size() > 2 && mem[0] == 'E' && mem[1] == 'Z';
                  }
                ) != cend(mems) ;
}
```

Programmation Orientée Objet

Service, Interface, Contrat ?

Un objet - vision logique

- Un objet encapsule un état
- Un objet propose un service
- Un objet définit une interface

Un objet - vision physique

- un état = données membres
- un comportement = fonctions membres
- le tout définit dans une `classe`

Structure de base d'une classe C++

```
class my_little_class
{
  public:

   my_little_class() { fill(); }

  ~my_little_class() {}

   my_little_class( my_little_class const& that)
                : a_(that.a_), data_(that.data_)
   {}

   my_little_class& operator=(my_little_class const& that)
   {
      a_ = that.a_;
      data_ = that.data_;

      return *this;
   }

  private:
  int a_;
  std::vector<double> data_;
```

Structure de base d'une classe C++

```
  public:

  int value_of_a() const { return a_; }

  std::vector<double>& data() { return data_; }
  std::vector<double>const& data() const { return data_; }

  protected:
  void fill()
  {
    std::fill(data_.begin(),data_.end(), 1.);
  }
};
```

Régle du Zéro

```
class my_little_class
{
  public:

   my_little_class() { fill(); }

  private:
  int a_;
  std::vector<double> data_;

  public:

  int value_of_a() const { return a_; }

  std::vector<double>& data() { return data_; }
  std::vector<double>const& data() const { return data_; }

  protected:
  void fill()
  {
    std::fill(data_.begin(),data_.end(), 1.);
  }
};
```

Héritage public

Définition

- Une classe hérite d'une autre afin de spécialiser son comportement
- La classe "fille" accéde aux données e à l'interface publique de sa "mére"
- Notion de sous-classage

```
#include <iostream>

class base
{
  public:
  virtual void behavior() { std::cout << __PRETTY_FUNCTION__ << "\n"; }
};

class derived : public base
{
  public:
```

Héritage public

Définition

- Une classe hérite d'une autre afin de spécialiser son comportement
- La classe "fille" accéde aux données e à l'interface publique de sa "mére"
- Notion de sous-classage

```
  void derived_behavior() { std::cout << __PRETTY_FUNCTION__ << "\n"; }
};

void process(base& b)
```

Héritage public

Définition

- Une classe hérite d'une autre afin de spécialiser son comportement
- La classe "fille" accéde aux données e à l'interface publique de sa "mére"
- Notion de sous-classage

```
  std::cout << __PRETTY_FUNCTION__ << "\n";
  b.behavior();
}

int main()
{
  derived d;

  d.behavior();
```

Gestion du polymorphisme

```
#include <iostream>

class base
{
  public:
  virtual void behavior() { std::cout << ”b\n”; }
};

class derived : public base
{
  public:
  virtual void behavior() { std::cout << ”d\n”; }
  void derived_behavior() {}
};

void process(base& b) { b.behavior(); }

int main()
{
  derived d;
  process(d);
}
```

Gestion du polymorphisme

```
#include <iostream>

class base
{
  public:
  virtual void behavior() override { std::cout << ”b\n”; }
  virtual void foo() final {}
};

class derived final : public base
{
  public:
  virtual void behavior() override { std::cout << ”d\n”; }
  void derived_behavior() {}
};

void process(base& b) { b.behavior(); }

int main()
{
  derived d;
  process(d);
}
```

Principe de substitution de Liskov

Énoncé

Partout où un objet x de type T est attendu, on doit pouvoir passer un objet y type U , avec U héritant de T .

Traduction

- une classe = une interface = un **contrat**
- Les pré-conditions ne peuvent être qu'affaiblies
- Les post-conditions ne peuvent être que renforcées

Principe de substitution de Liskov

```
class rectangle
{
  protected:
  double width;
  double height;

  public:
  rectangle() : width(0), height(0) {}
  virtual void set_width(double x){width=x;}
  virtual void set_height(double x){height=x;}
  double area() const {return width*height;}
};
```

Principe de substitution de Liskov

```
class square : public rectangle
{
  public:
  void set_width(double x)
  {
    rectangle::set_width(x);
    rectangle::set_height(x);
  }

  void set_height(double x)
  {
    rectangle::set_width(x);
    rectangle::set_height(x);
 }
};
```

Principe de substitution de Liskov

```
void foo(rectangle& r)
{
  r.set_height(4);
  r.set_width(5);

  if( r.area() !=20 )
    std::cout << ”ERROR ” << r.area() << ” != 20\n”;
}

int main()
{
  rectangle r;
  square s;

  foo(r);
  foo(s);
}
```

Héritage privé

Définition

- Résout le problème de la factorisation de code
- Permet la réutilisation des composants logiciels
- Pas de relation de sous-classe

Héritage privé

```
#include <vector>

class stack : private std::vector<double>
{
  using parent = std::vector<double>;

  public:
  using parent::size;

  void push(double v)
  {
    parent::push_back(v);
  }

  double pop()
  {
    double v = parent::back();
    parent::pop_back();
    return v;
  }

  double top()
  {
    return parent::back();
  }
};
```

Sémantique de Valeur

Définition

Une classe possède une sémantique de valeur ssi deux instances de cette classe situées à des adresses différentes, mais au contenu identique, sont considérées égales.

Structure classique

- Peut redéfinir des opérateurs (+,-,*,...)
- Posséde un opérateur d'affectation
- Est comparable via < et ==
- Ne peut être utilisé comme classe de base

Sémantique d'Entité

Définition

Une classe a une sémantique d'entité si toutes les instances de cette classe sont nécessairement deux à deux distinctes, même si tous les champs de ces instances sont égaux. Elle modélise un concept d'identité : chaque objet représente un individu unique.

Structure classique

- Ne redéfinit pas d'opérateur
- Ne posséde pas d'opérateur d'affectation
- N'est pas comparable via < et ==
- Les copies sont explicites via un fonction adéquat (clone)

Que faire ?

Utilisez préférentiellement la sémantique de valeur

- Code clair et concis
- Bonne performance car pas d'allocation dynamique supplémentaire
- NRVO et élision de copie sont de la partie

Ne délaissez pas les entités

- Extrêmement utile pour le data-driven code
- Bonne base pour des bibliothèques d'infrastructure
- Se marie élégamment avec les pointeurs à sémantique riche

Gestion des ressources

Principe de RAII

Objectifs

- Assurer la sureté de la gestion des ressources
- Minimiser la gestion manuelle de la mémoire
- Simplifier la gestion des exceptions

Resource Acquisition Is Initialisation

Mise en œuvre

- Constructeurs = prise de ressource
- Destructeur = libération de ressource
- Gestion de la ressource au niveau du bloc

shared_ptr

Principes

- Pointeur à compteur de références
- Libère la mémoire lorsque aucune référence pointe sur lui
- Cycles gérés par weak_ptr

```
#include <memory>
#include <iostream>

int main()
{
  std::shared_ptr<int> p1 = std::make_shared<int>(5);
  std::shared_ptr<int> p2 = p1;

  std::cout << *p1 << "\n";
  *p2 = 42;
  std::cout << *p1 << "\n";
  p1.reset();
  std::cout << *p2 << "\n";
  p2.reset();
}
```

shared_ptr

Principes

- Pointeur à compteur de références
- Libère la mémoire lorsque aucune référence pointe sur lui
- Cycles gérés par weak_ptr

```
int main()
{
  std::shared_ptr<int> p1 = std::make_shared<int>(5);
  std::weak_ptr<int> wp1 = p1;

  {
    std::shared_ptr<int> p2 = wp1.lock();
    if(p2) std::cout << *p2 << "\n";
  }

  p1.reset();
  std::shared_ptr<int> p3 = wp1.lock();

  if(p3) std::cout << "nope :(\n";
}
```

unique_ptr

Principes

- Pointeur à propriétaire unique
- Ne peut être copié mis seulement transféré
- Transfert = Transfert de propriété

Mise en œuvre

```
#include <memory>

int main()
{
  std::unique_ptr<int> p1 = std::make_unique<int>(5);
  std::unique_ptr<int> p2 = p1;
  std::unique_ptr<int> p3 = std::move(p1);

  p3.reset();
  p1.reset();
}
```

Principe de RAII

Exemple

```
#include <string>
#include <mutex>
#include <iostream>
#include <fstream>
#include <stdexcept>

void write_to_file (const std::string & message)
{
  static std::mutex mutex;

  std::lock_guard<std::mutex> lock(mutex);

  std::ofstream file("example.txt");
  if (!file.is_open())
    throw std::runtime_error("unable to open file");

  file << message << std::endl;
}
```

Gestion des Temporaires

lvalue vs *rvalue*

- lvalue : objet avec une identité, un nom
- rvalue : objet sans identité
- La durée de vie d'une rvalue est en général bornée au `statement`
- Une rvalue peut survivre dans une référence vers une lvalue constante

```
int a = 42; // lvalue
int b = 43; // lvalue

a = b; // ok
b = a; // ok
a = a * b; // ok
int c = a * b; // ok
a * b = 42; // erreur

int getx() { return 17;}

const int& lcr = getx(); // ok
int& lr = getx(); // erreur
```

Référence vers rvalue

Objectifs

- Discriminer via un qualificateur lvalue et rvalue
- Expliciter les opportunités d'optimisation
- Simplifier la définition d'interfaces

Notation

- T& : référence vers lvalue
- T const& : référence vers lvalue constante
- T&& : référence vers rvalue

Référence vers rvalue

Exemple

```
#include <iostream>
#include <typeinfo>

void foo(int const&) { std::cout << ”lvalue\n”; }
void foo(int&& x) { std::cout << ”rvalue\n”; }

int bar() { return 1337; }

int main()
{
  int x = 3;
  int& y = x;
  int const& z = bar();

  foo(x);
  foo(y);
  foo(z);

  foo(4);
  foo(bar());
}
```

Référence vers rvalue

Le problème du forwarding

```
#include <iostream>

void foo(int const&) { std::cout << ”lvalue\n”; }
void foo(int&&) { std::cout << ”rvalue\n”; }

void chu(int&& x) { foo(x); }

int bar() { return 1337; }


int main()
{
  foo(bar());

  chu(bar());
}
```

Référence vers rvalue

Le problème du forwarding

```
#include <iostream>

void foo(int const&) { std::cout << ”lvalue\n”; }
void foo(int&&) { std::cout << ”rvalue\n”; }

void chu(int&& x) { foo( std::forward<int>(x) ); }

int bar() { return 1337; }


int main()
{
  foo(bar());

  chu(bar());
}
```

Sémantique de transfert

Problèmatique

- Copier un objet contenant des ressources est coûteux
- Copier depuis un temporaire est doublement coûteux (allocation+deallocation)
- Limite l'expressivité de certaines interfaces
- Pourquoi ne pas recycler le temporaire ?

Solution

- Utiliser les rvalue-reference pour détecter un temporaire
- Extraire son contenu et le transférer dans un objet pérenne
- Stratégie généralisée à tout le langage et à la bibliothèque standard

Sémantique de transfert

Code original

```
#include <iostream>
#include <vector>
#include <algorithm>

// Copie multiples
std::vector<int> sort(std::vector<int> const& v)
{
  std::vector<int> that{v};

  std::sort( that.begin(), that.end() );

  return that;
}

// Interface détournée
void sort(std::vector<int> const& v, std::vector<int>& that)
{
  that = v;
  std::sort( that.begin(), that.end() );
}
```

Sémantique de transfert

Version à base de transfert

```
#include <iostream>
#include <vector>
#include <algorithm>

// Cas général
std::vector<int> sort(std::vector<int> const& v)
{
  std::vector<int> that{v};

  std::sort( that.begin(), that.end() );

  return that;
}

// Cas du temporaire
std::vector<int> sort(std::vector<int>&& v)
{
  std::vector<int> that{std::move(v)};

  std::sort( that.begin(), that.end() );

  return that;
}
```

Programmation Générique

Programmation Générique

Objectifs

- Ecrire des structures et fonctions indépendantes des types manipulés

```
template<typename T, typename U> auto min(T a, U b)
{
  return a < b ? a : b;
}

template<typename T> struct print;

int main()
{
  auto x = min(0,'c');
  print<decltype(x)> p;
}
```

Applications

- Générer du code paramétré par des types
- *Design Patterns Policy*, *Strategy* ou *State*

Les `templates`

Définition

- Entités à part en C++ différentes des types et des valeurs
- Une ”recette de cuisine” pour construire un type ou une fonction

Notion d’instanciation

- ”Construire” un `template` : **instancier**
- Une instance de `template` est un vrai type ou une vrai fonction
- Les ”recettes” peuvent varier en fonctions des ”ingrédients”

Notion de spécialisation

- Spécification des ”recettes” en fonction des paramétres
- Gestion de la variabilité et des branchements
- Par ”accident”, les `template` sont Turing complet

Que faire avec des `templates`

`template` de classe

- `template<typename T> struct foo ;` est un `template`
- `foo<int>` est un type

`template` de fonction

- `template<typename U> int foo(U u) ;` est un `template`
- `foo<double>` est une fonction
- `foo(0.)` déduit U automatiquement

Paramétrages des templates

Paramétre de types

- `template<typename T> struct foo;`
- `foo<int>`

Paramétre de constantes entiéres

- `template<int N> struct bar;`
- `bar<42>`

Paramétre templates

- `template < template<int> T > struct chu;`
- `chu < bar >`

Les Template Alias

Objectifs

- Améliorer le design des bibliothèques
- Améliorer la pratique de la programmation générique
- Simplifier l'utilisation des templates en général

Exemples

```
#include <array>
#include <iostream>

template<typename T>
using point3D = std::array<t,3>;

int main()
{
  point3D<int> x{1,2,3};

  std::cout << x.size() << "\n";
}
```

Les `Template` Alias

Objectifs

- Améliorer le design des bibliothèques
- Améliorer la pratique de la programmation générique
- Simplifier l'utilisation des `templates` en général

Exemples

```
template<typename T> class allocator
{
  //...

  template<typename U> struct rebind { typedef allocator<U> other; };
};

// utilisation
allocator<T>::rebind<U>::other x;
```

Les `Template` Alias

Objectifs

- Améliorer le design des bibliothèques
- Améliorer la pratique de la programmation générique
- Simplifier l'utilisation des `templates` en général

Exemples

```
template<typename T> class allocator
{
  //...

  template<typename U> using rebind = allocator<U>;
};

// utilisation
allocator<T>::rebind<U> x;
```

Les `Template` Alias

Objectifs

- Améliorer le design des bibliothèques
- Améliorer la pratique de la programmation générique
- Simplifier l'utilisation des `templates` en général

Exemples

```
// utilisation ”a la typedef”

template<typename T> struct as_vector
{
  using type = std::vector<T>;
};
```

Les `Traits`

Objectifs

- Introspection limitée sur les propriétés des types
- Génération de nouveaux types
- Outils pour la spécialisation avancée
- Notion de **Méta-fonction** : fonction manipulant et retournant un type

Les Traits

Traits d'introspection

- Classification des types selon leur sémantique
- Vérification d'existence d'une interface donnée
- Récupération d'informations structurelles

Exemple

```
#include <type_traits>

int main()
{
  std::cout << std::is_same<float,int>::value << "\n";
  std::cout << std::is_convertible<float,int>::value << "\n";
  std::cout << std::is_base_of<istream,ifstream>::value << "\n";
  std::cout << std::is_class<vector<int>>::value << "\n";
  std::cout << std::is_constructible<string,char*>::value << "\n";
  std::cout << std::is_polymorphic<istream>::value << "\n";
  std::cout << std::is_pointer<void*>::value << "\n";
}
```

Les Traits

Générateur de type

- Manipulation sûre des qualificateurs
- Création de types vérifiant certaines propriétés

Exemple

```
#include <type_traits>

int main()
{
  int i;
  std::add_pointer<int>::type pi = &i;

  std::add_rvalue_reference<int>::type rri = std::forward<int>(i);
}
```

Les Traits - Application

```
#include <type_traits>

template<bool B> using bool_ = std::integral_constant<bool,B>;

template<typename T> inline void copy(T* b, T* e, T const* src, bool_<true> const&)
{
  std::memcpy(b,src,(e-b)*sizeof(T));
}

template<typename T> inline void copy(T* b, T* e, T const* src, bool_<false> const&)
{
  while(b != e) *b++ = *src++;
}

template<typename T>
using is_trivially_copiable_t = typename std::is_trivially_copiable<T>::type;

template<typename T> void copy(T* b, T* e, T const* src)
{
  is_trivially_copiable_t<T> select;
  copy(b,e,src,select);
}
```

Les `static_assert`

Objectifs

- `assert` : vérifie l'état logique d'un programme au runtime
- Comment vérifier l'état logique à la compilation ?
- Émission de messages d'erreur customisés
- Interaction avec les *Traits*

Exemple

```
#include <type_traits>

template<typename T> T factorial(T n)
{
  static_assert( std::is_integral<T>::value, "factorial requires integral parameter");
  return n<2 ? 1 : n*factorial(n-1);
}
```

Les Expressions Constantes

Objectifs

- Simplifier le développement de méta-fonctions numériques
- Syntaxe homogéne aux fonctions classiques
- Utilisable dans les contextes requérant une constante

Syntaxe et Exemples

```
#include <iostream>

int factorial(int n)
{
  return n<2 ? 1 : n*factorial(n-1);
}

int main()
{
  std::cout << factorial(8) << "\n";
}
```

Les Expressions Constantes

Objectifs

- Simplifier le développement de méta-fonctions numériques
- Syntaxe homogéne aux fonctions classiques
- Utilisable dans les contextes requérant une constante

Syntaxe et Exemples

```
#include <iostream>

constexpr int factorial(int n) { return n<2 ? 1 : n*factorial(n-1); }
template<int N> void display() { std::cout << N << "\n"; }

int main()
{
  display<factorial(5)>();

  int x;
  std::cin >> x;
  std::cout << factorial(x) << "\n";
}
```

Les Expressions Constantes

Objectifs

- Simplifier le développement de méta-fonctions numériques
- Syntaxe homogéne aux fonctions classiques
- Utilisable dans les contextes requérant une constante

Syntaxe et Exemples

```
#include <iostream>

constexpr int factorial(int n)
{
  return n>=0 ? (n<2 ? 1 : n*factorial(n-1)) : throw std::out_of_range("");
}

int main()
{
  static constexpr int f = factorial(-1);
}
```

Maitrise de la SFINAE

Qu'est-ce que la SFINAE ?

- Lors de la résolution de la surcharge de fonctions, il se peut qu'une surcharge instancie une fonction `template`
- Cette instanciation peut échouer
- Au lieu d'émettre une erreur, la surcharge est ignorée
- SFINAE = Substitution Failure Is Not An Error

```
#include <vector>

template<typename T>
typename T::size_type size(T const& t) { return t.size(); }

int main()
{
  size(std::vector<int>(8)); // OK

  // no matching function for call to 'size(int')
  size(8);
}
```

Maitrise de la SFINAE

Qu'est-ce que la SFINAE ?

- Lors de la résolution de la surcharge de fonctions, il se peut qu'une surcharge instancie une fonction `template`
- Cette instanciation peut échouer
- Au lieu d'émettre une erreur, la surcharge est ignorée
- SFINAE = Substitution Failure Is Not An Error

Intérêts

- Controler ces erreurs permet d'éliminer des fonctions sur des critères arbitraires
- Customisation de la surcharge de fonctions ou de classes `templates`
- Interaction avec les *Traits*
- `std::enable_if`

Maitrise de la SFINAE

Mise en œuvre

```
#include <tuple>
#include <iostream>

template <size_t n, typename... T>
typename std::enable_if<(n >= sizeof...(T))>::type
print_tuple(std::ostream&, const std::tuple<T...>&) {}

template <size_t n, typename... T>
typename std::enable_if<(n < sizeof...(T))>::type
print_tuple(std::ostream& os, const std::tuple<T...>& tup)
{
  if (n != 0) os << ", ";
  os << std::get<n>(tup);
  print_tuple<n+1>(os, tup);
}

template <typename... T>
std::ostream& operator<<(std::ostream& os, const std::tuple<T...>& tup)
{
  os << "[";
  print_tuple<0>(os, tup);
  return os << "]";
}
```