Concepts

Faire de la programmation générique une programmation… normale

Concepts

- Si vous avez suivi des formations avec moi par le passé, vous avez peut-être souvenir du problème suivant :
  - « Écrire une fonction close_enough(a,b) permettant de déterminer si a et b sont assez proches pour être considérés équivalents »
  - Si a et b sont des entiers, alors a==b résout ce problème
  - Si a et b sont des nombres à virgule flottante, alors on souhaite savoir s'ils sont assez proches pour que la valeur absolue de leur différence soit sous le seuil de notre tolérance à l'erreur

Concepts

```
// ceci fonctionne, mais ne couvre pas
// tous les types pertinents
bool close_enough(int a, int b) {
    return a == b;
}
bool close_enough(float a, float b) {
    return abs(a - b) <= 0.000001f;
}
```

Concepts

```
// ceci ne fonctionne pas, car un appel à
// close_enough(a,b) avec a et b du même
// type serait ambigu
template <class T>
bool close_enough(T a, T b) {
    return a == b;
}
template <class T>
bool close_enough(T a, T b) {
    return abs(a - b) <= 0.000001f;
}
```

Concepts

```
#include <cmath>
class exact {};
class floating {};
template <class T> bool close_enough(T a, T b, exact) { return a == b; }
template <class T> bool close_enough(T a, T b, floating) { return std::abs(a - b) <= static_cast<T>(0.000001); }
struct false_type { enum { value = false }; };
struct true_type { enum { value = true }; };
template <class> struct is_floating : false_type { };
template <> struct is_floating<float> : true_type { };
template <> struct is_floating<double> : true_type { };
template <> struct is_floating<long double> : true_type { };
template <bool, class T, class F> struct static_if_else;
template <class T, class F> struct static_if_else<true, T, F> { typedef T type; };
template <class T, class F> struct static_if_else<false, T, F> { typedef F type; };
template <class T>
  bool close_enough(T a, T b) {
    return close_enough(a, b, typename static_if_else<is_floating<T>::value, floating, exact>::type{});
   }
```

C++98/03

https://godbolt.org/z/EM7j5fhnM

Concepts

```
#include <cmath>
#include <type_traits>
class exact {};
class floating {};
template <class T> bool close_enough(T a, T b, exact) {
  return a == b;
}
template <class T> bool close_enough(T a, T b, floating) {
  return std::abs(a - b) <= static_cast<T>(0.000001);
}
template <class T>
  bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<
        std::is_floating_point<T>::value, floating, exact
     >::type{});
  }
```

C++11

https://godbolt.org/z/jb5sa95ra

Concepts

```
#include <type_traits>
class exact {};
class floating {};
template <class T> constexpr bool close_enough(T a, T b, exact) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr bool close_enough(T a, T b, floating) {
  return absolute(a - b) <= static_cast<T>(0.000001);
}
```

```
template <class T>
  constexpr bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<
        std::is_floating_point<T>::value, floating, exact
     >::type{});
  }
```

C++11 avec constexpr

https://godbolt.org/z/Mz48jq84T

Concepts

```
#include <type_traits>
template <class T>
  constexpr typename
    std::enable_if<!std::is_floating_point<T>::value, bool>::type close_enough(T a, T b) {
    return a == b;
  }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T>
  constexpr typename
    std::enable_if<std::is_floating_point<T>::value, bool>::type close_enough(T a, T b) {
    return absolute(a - b) <= static_cast<T>(0.000001);
  }
```

C++11 avec enable_if

https://godbolt.org/z/qsrhEPnKn

Concepts

```
#include <type_traits>
template <class T>
  constexpr
    std::enable_if_t<!std::is_floating_point<T>::value, bool> close_enough(T a, T b) {
    return a == b;
  }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T>
  constexpr
    std::enable_if_t<std::is_floating_point<T>::value, bool> close_enough(T a, T b) {
    return absolute(a - b) <= static_cast<T>(0.000001);
  }
```

C++14

https://godbolt.org/z/GWvar11zv

Concepts

C++14 avec constantes génériques

https://godbolt.org/z/9jTr7d5rG

```
#include <type_traits>
template <class T>
   constexpr std::enable_if_t<!std::is_floating_point<T>::value, bool>
      close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) {
   return val < 0? -val : val;
}
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T>
   constexpr std::enable_if_t<std::is_floating_point<T>::value, bool>
      close_enough(T a, T b) {
          return absolute(a - b) <= threshold<T>;
      }
```

Concepts

C++17

https://godbolt.org/z/vx5fGeqd5

```
#include <type_traits>
template <class T>
  constexpr std::enable_if_t<!std::is_floating_point_v<T>, bool>
    close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T>
  constexpr std::enable_if_t<std::is_floating_point_v<T>, bool>
    close_enough(T a, T b) {
      return absolute(a - b) <= threshold<T>;
    }
```

Concepts

```
#include <type_traits>
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T> constexpr auto
  threshold = static_cast<T>(0.000001);
template <class T>
  constexpr bool close_enough(T a, T b) {
    if constexpr(std::is_floating_point_v<T>)
      return absolute(a - b) <= threshold<T>;
    else
      return a == b;
  }
```

C++17

https://godbolt.org/z/bze17jovo

Concepts

- Il est donc possible depuis longtemps de résoudre ce type de problème efficacement
  - Avec l'évolution du langage, les solutions sont devenues graduellement plus simples, plus élégantes, plus fécondes
- Avec C++20, il est possible d'exprimer une solution élégante à ce problème à l'aide de concepts
  - Un concept exprime des requis sur un type dans du code générique

Concepts

- Le concept le plus général est **auto**, qui est toujours satisfait

```
// C++17
template <class T, class U> void f(T a, U b) {
  // ...
}
// C++20 (équivalent)
void f(auto a, auto b) { /* ... */ }
```

Concepts

- Le concept le plus général est **auto**, qui est toujours satisfait

```
// C++17
template <class T, class U> void f(T a, U b) {
  // ...
}
// C++20 (équivalent)
void f(auto a, auto b) { /* ... */ }
```

Chaque paramètre peut avoir un type distinct de ceux des autres

Concepts

- Un concept s’exprime sous la forme d’un prédicat statique (au sens de « résolu à la compilation ») sur un type

```
// concepts pas particulièrement utiles,
// mais pour explorer la syntaxe...
// satisfait par tous les types
#include <type_traits>
template <class> concept tautologie = true;
```

Concepts

- Un concept s'exprime sous la forme d'un prédicat statique (au sens de « résolu à la compilation ») sur un type

```
// concepts pas particulièrement utiles,
// mais pour explorer la syntaxe...
// satisfait par tous les types
#include <type_traits>
template <class> concept tautologie = true;
// satisfait par aucun type
template <class> concept contradiction = false;
```

Concepts

- Un concept s'exprime sous la forme d'un prédicat statique (au sens de « résolu à la compilation ») sur un type

```
// concepts pas particulièrement utiles,
// mais pour explorer la syntaxe...
// satisfait par tous les types
#include <type_traits>
template <class> concept tautologie = true;
// satisfait par aucun type
template <class> concept contradiction = false;
// satisfait par bool et rien d'autre
template <class T> concept booleen =
  std::is_same_v<T,bool>;
```

Concepts

```
#include <type_traits>
template <class> concept tautologie = true; // satisfait par tous les types
template <class> concept contradiction = false; // satisfait par aucun type
template <class T> concept booleen = std::is_same_v<T,bool>;
template <class T> requires tautologie<T> void f(T) { }
template <class T> requires contradiction<T> void g(T) { }
template <class T> requires booleen<T> void h(T) { }
int main() {
   f(3); // ok, peu importe le type
   // g(3); // non, peu importe le type
   h(true); // ok
   // h(3); // non
}
```

Concepts

```
#include <type_traits>
template <class> concept tautologie = true; // satisfait par tous les types
template <class> concept contradiction = false; // satisfait par aucun type
template <class T> concept booleen = std::is_same_v<T,bool>;
template <class T> requires tautologie<T> void f(T) { }
template <class T> requires contradiction<T> void g(T) { }
template <class T> requires booleen<T> void h(T) { }
int main() {
   f(3); // ok, peu importe le type
   // g(3); // non, peu importe le ty
   h(true); // ok
   // h(3); // non
}
```

https://wandbox.org/permlink/99WIp6f7cQf1vbH6

Concepts

- On dit d'un type pour lequel un concept est applicable que ce type **satisfait** le concept

```
template <class T> concept petit = sizeof(T) <= 2;
template <class T> requires petit<T> void f(T) {
}
int main() {
  static_assert(sizeof(int)==4);
  f(false); // ok (bool satisfait petit)
  // f(3); // non (int ne satisfait pas petit)
}
```

Concepts

- Le mot clé **requires** exprime un requis
  - `template <class T> requires booleen<T> void h(T){}`
- Il est possible de combiner les requis avec && et ||

Concepts

```
#include <type_traits>
template <class T> concept entier = std::is_integral_v<T>;
template <class T> concept petit = sizeof(T) <= 2;
template <class T>
   requires entier<T> && petit<T> void f(T) {}
int main() {
   static_assert(sizeof(short) == 2);
   static_assert(sizeof(int) == 4);
   f(short{ 3 }); // Ok
   // f(3); // non
}
```

https://wandbox.org/permlink/5CQO9URwgGbhbKYw

Concepts

```
#include <type_traits>
template <class T> concept entier = std::is_integral_v<T>;
template <class T> concept petit = sizeof(T) <= 2;
template <class T> concept flottant = std::is_floating_point_v<T>;
template <class T>
   requires entier<T> && petit<T> || flottant<T> void f(T) {}
int main() {
   static_assert(sizeof(short) == 2);
   static_assert(sizeof(int) == 4);
   f(short{ 3 }); // Ok
   // f(3); // non
   f(3.0); // Ok
}
```

https://wandbox.org/permlink/dAUGy0TfM0grXD04
(notez le respect de l'ordre de priorité des opérateurs)

Concepts

```
template <class T>
  concept comparable = requires(T a, T b) {
    a == b;
    a != b;
  };
struct X {};
template <class T>
  requires comparable<T> void f(T) {}
int main() {
  f(3); // Ok
  // f(X{}); // non
}
```

La clause requires ici indique que pour a et b de type T, il faut être capable d'exprimer a==b et a!=b

Concepts

```
template <class T>
  concept comparable = requires(T a, T b) {
    a == b;
    a != b;
  };
struct X {};
template <class T>
  requires comparable<T> void f(T) {}
int main() {
  f(3); // Ok
  // f(X{}); // non
}
```

https://wandbox.org/permlink/t7sgaqoxEHy4VVo2

Concepts

```
#include <concepts>
template <class T>
  concept comparable = requires(T a, T b) {
    { a == b } -> std::convertible_to<bool>;
    { a != b } -> std::convertible_to<bool>;
  };
struct X {};
template <class T>
  requires comparable<T> void f(T) {
int main() {
  f(3); // Ok
  // f(X{}); // non
}
```

Une clause requires peut exprimer des contraintes sémantiques, par exemple sur le type des expression évaluées :
https://wandbox.org/permlink/n70NjJTq7q8ql5Ex

Concepts

- Le mot clé requires permet d'exprimer aisément l'idée de « ce type peut-il être utilisé comme suit? »
- Par exemple :

```
#include <concepts>
// pas un bon concept, mais pour explorer la syntaxe
template <class T> concept incrementable = requires(T n) {
   { ++n } -> std::same_as<T&>;
   { n++ } -> std::convertible_to<T>;
};
template <class T> requires incrementable<T> void f(T) {
}
int main() {
   f(3);
}
```

Concepts

- Le mot clé requires permet d'ex…ut-il être utilisé comme suit? »
- Par exemple :

Remarquez l'en-tête <concepts> qui définit plusieurs concepts essentiels (ici : std::same_as, std::convertible_to)

```
#include <concepts>
// pas un bon concept, mais pour explorer la syntaxe
template <class T> concept incrementable = requires(T n) {
    { ++n } -> std::same_as<T&>;
    { n++ } -> std::convertible_to<T>;
};
template <class T> requires incrementable<T> void f(T) {
}
int main() {
    f(3);
}
```

Concepts

- Le mot clé requires permet d'ex -il être utilisé comme suit? »
- Par exemple :

À titre d'exemple, le concept std::same_as<T,U> peut être exprimé à l'aide du trait std::is_same_v<T,U>. Notez que le concept est défini sur deux types…

```
#include <concepts>
// pas un bon concept, mais pour explorer la syntaxe
template <class T> concept incrementable = requires(T n) {
   { ++n } -> std::same_as<T&>;
   { n++ } -> std::convertible_to<T>;
};
template <class T> requires incrementable<T> void f(T) {
}
int main() {
   f(3);
}
```

Concepts

- Le mot clé requires permet d'exprimer aisément l'idée de « ce type peut-il être utilisé comme suit? »
- Par exemple :

```
#include <concepts>
// pas un bon concept, mais pour explorer la syntaxe
template <class T> concept incrementable = requires(T n) {
    { ++n } -> std::same_as<T&>;
    { n++ } -> std::convertible_to<T>;
};
template <class T> req      incrementable<T> void f(T) {
}
int main() {
    f(3);
}
```

La clause requires ici indique que ++n doit retourner un T& alors que n++ doit retourner quelque chose qui soit convertible en T. Le premier type de same_as ou de convertible_to est celui de l'expression placée entre accolades

Concepts

- L’idée « d’être incrémentable » est un peu… mince, disons, pour mériter un nom de concept
- Pour ces requis qui ne méritent pas vraiment le statut de concept, il est possible d’utiliser une clause requires directement
  - Cela mène à l’étrange forme grammaticale **requires requires**

Concepts

```
#include <concepts>
template <class T> requires requires(T n) {
    { ++n } -> std::same_as<T&>;
    { n++ } -> std::convertible_to<T>;
} void f(T) {
}
int main() {
    f(3);
}
```

Concepts

- Il est possible d’exprimer des contraintes reposant sur les noms des paramètres d’une fonction en utilisant une clause requires placée après le symbole -> dans une signature avec type de retour tardif (*Trailing Return Type*)

Concepts

```
#include <concepts>
template <class T>
   auto f(T arg) noexcept -> void
      requires requires { { ++arg } -> std::same_as<T&>; } {
      }
struct X {
   X& operator++() noexcept { return *this; }
};
int main() {
   f(3);
   f(X{});
}
```

Concepts

- Les concepts peuvent aussi être exprimés sur la base d’autres concepts ou d’une combinaison de concepts et de clauses requires

Concepts

```
template <class T>
   concept Decrementable =
      Incrementable<T> && requires(T obj) {
         { --obj } -> same_as<T&>;
         { obj-- } -> convertible_to<T>;
   };
```

Concepts

Utiliser un concept comme contrainte sur un autre concept n'est malheureusement pas permis avec les concepts de C++20. C'est parfois un irritant

```
/*
template <Incrementable T> // NON
    concept Decrementable =
        requires(T obj) {
            { --obj } -> same_as<T&>;
            { obj-- } -> convertible_to<T>;
    };
*/
```

Concepts

- Lorsqu’une fonction se limite à un seul requis, il est possible d’utiliser une syntaxe plus légère

Concepts

```
template <class T> concept petit = sizeof(T) <= 2;
// forme générale
template <class T> requires petit<T> void f(T) {
}
int main() {
  static_assert(sizeof(int)==4);
  f(false); // ok (bool satisfait petit)
  // f(3); // non (int ne satisfait pas petit)
}
```

Concepts

```
template <class T> concept petit = sizeof(T) <= 2;
// contrainte sur le type (un seul requis)
template <petit T> void f(T) {
}
int main() {
  static_assert(sizeof(int)==4);
  f(false); // ok (bool satisfait petit)
  // f(3); // non (int ne satisfait pas petit)
}
```

Concepts

```
template <class T> concept petit = sizeof(T) <= 2;
// forme abrégée (un seul requis)...!
void f([[maybe_unused]] petit auto arg) {
}
int main() {
  static_assert(sizeof(int)==4);
  f(false); // ok (bool satisfait petit)
  // f(3); // non (int ne satisfait pas petit)
}
```

Concepts

```
template <class T> concept petit = sizeof(T) <= 2;
// forme abrégée (un seul requis)...!
void f([[maybe_unused]] petit
}
int main() {
  static_assert(sizeof(int)==4);
  f(false); // ok (bool satisfait petit)
  // f(3); // non (int ne satisfait pas petit)
}
```

Notez l'absence du mot template… Le concept est une contrainte exprimée sous la forme d'un adjectif. La programmation générique devient de la programmation « normale »

Concepts

- Voyons voir comment appliquer les concepts à notre problème original, soit celui de déterminer si deux valeurs sont « assez proches » l'une de l'autre
  - Nous prendrons pour base la version « enable_if » de C++17

Concepts

```
#include <type_traits>
template <class T>
  constexpr std::enable_if_t<!std::is_floating_point_v<T>, bool>
    close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T>
  constexpr std::enable_if_t<std::is_floating_point_v<T>, bool>
    close_enough(T a, T b) {
      return absolute(a - b) <= threshold<T>;
    }
```

Rappel (C++ 17)

Concepts

- Parmi les concepts standards logés dans <concepts>, on trouve entre autres std::integral (concept satisfait pour tout type entier) et std::floating_point (concept satisfait pour tout type à virgule flottante)
  - Il est donc possible d'arriver au même résultat qu'avec enable_if, mais en évitant cet étrange outil

Concepts

```
#include <concepts>
template <class T> requires std::integral<T>
  constexpr bool close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T> constexpr auto
  threshold = static_cast<T>(0.000001);
template <class T> requires std::floating_point<T>
  constexpr bool close_enough(T a, T b) {
    return absolute(a - b) <= threshold<T>;
  }
```

Concepts

close_enough(3,3) appellera la version applicable à des entiers alors que close_enough(3.0,3.0) appellera celle applicable à des nombres à virgule flottante

```
#include <concepts>
template <class T> requires std::integral<T>
  constexpr bool close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T> constexpr auto
  threshold = static_cast<T>(0.000001);
template <class T> requires std::floating_point<T>
  constexpr bool close_enough(T a, T b) {
    return absolute(a - b) <= threshold<T>;
  }
```

Concepts

close_enough(3,3) appellera la version applicable à des entiers alors que close_enough(3.0,3.0) appellera celle applicable à des nombres à virgule flottante

```
#include <concepts>
template <class T> requires std::integral<T>
  constexpr bool close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T> constexpr auto
  threshold = static_cast<T>(0.0000
template <class T> requires std::floating_point<T>
  constexpr bool close_enough(T a, T b) {
    return absolute(a - b) <= threshold<T>;
  }
```

L'appel close_enough(3,3.0) ne compilera pas (un entier, un nombre à virgule flottante), pas plus que close_enough(3.0,3.0f) (deux types distincts)

Concepts

- Puisqu’il s’agit de fonctions pour lesquelles il n’y a qu’un seul requis, les formes plus concises sont possibles

Concepts

```
#include <concepts>
template <std::integral T>
  constexpr bool close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T> constexpr auto
  threshold = static_cast<T>(0.000001);
template <std::floating_point T>
  constexpr bool close_enough(T a, T b) {
    return absolute(a - b) <= threshold<T>;
  }
```

Concepts

Notez qu'avec cette écriture, a et b doivent être du même type (deux int, deux long, deux double, etc.)

```
#include <concepts>
template <std::integral T>
  constexpr bool close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T> constexpr auto
  threshold = static_cast<T>(0.000001);
template <std::floating_point T>
  constexpr bool close_enough(T a, T b) {
    return absolute(a - b) <= threshold<T>;
  }
```

Concepts

```
#include <concepts>
#include <type_traits>
constexpr bool
  close_enough(std::integral auto a, std::integral auto b) {
    return a == b;
  }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T> constexpr auto
  threshold = static_cast<T>(0.000001);
constexpr bool
  close_enough(std::floating_point auto a, std::floating_point auto b) {
    return absolute(a - b) <= threshold<std::common_type_t<decltype(a), decltype(b)>>;
  }
```

Concepts

Avec cette écriture, a et b peuvent être de même type ou de types différents (int et long, double et long double, etc.) dans la mesure où les deux types satisfont le même concept

```
#include <concepts>
#include <type_traits>
constexpr bool
  close_enough(std::integral auto a, std::integral auto b) {
    return a == b;
  }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T> constexpr auto
  threshold = static_cast<T>(0.000001);
constexpr bool
  close_enough(std::floating_point auto a, std::floating_point auto b) {
    return absolute(a - b) <= threshold<std::common_type_t<decltype(a), decltype(b)>>;
  }
```

Concepts

```
#include <concepts>
#include <type_traits>
constexpr bool
  close_enough(std::integral auto a, std::integral auto b) {
    return a == b;
  }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <class T> constexpr auto
  threshold = static_cast<T>(0.000001);
constexpr bool
  close_enough(std::floating_point auto a, std::floating_point auto b) {
    return absolute(a - b) <= threshold<std::common_type_t<decltype(a), decltype(b)>>;
  }
```

Cela explique d'ailleurs la complexité du type du seuil (threshold) ici : il s'agit du type commun entre le type de a et le type de b

Concepts

```
#include <concepts>
#include <type_traits>
constexpr bool
  close_enough(std::integral auto a, std::integral auto b) {
    return a == b;
  }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <std::floating_point T> constexpr auto
  threshold = static_cast<T>(0.000001);
constexpr bool
  close_enough(std::floating_point auto a, std::floating_point auto b) {
    return absolute(a - b) <= threshold<std::common_type_t<decltype(a), decltype(b)>>;
  }
```

Concepts

```
#include <concepts>
#include <type_traits>
constexpr bool
  close_enough(std::integral auto a, std::in
    return a == b;
  }
template <class T> constexpr T absolute(T val) {
  return val < 0? -val : val;
}
template <std::floating_point T> constexpr auto
  threshold = static_cast<T>(0.000001);
constexpr bool
  close_enough(std::floating_point auto a, std::floating_point auto b) {
    return absolute(a - b) <= threshold<std::common_type_t<decltype(a), decltype(b)>>;
  }
```

Évidemment, pour la constante générique threshold<T>, seul un type T qui soit à virgule flottante serait raisonnable, alors permettons-nous de nous en assurer. Vive les concepts!

Concepts

- Je me permets ici de citer Jonathan Wakely, avec sa permission
- « *People use C++ in many different ways, and abbreviated functions using concepts are going to be mysterious magic for a lot of developers, for a long time*. » (Jonathan Wakely, 2022)
  - Les concepts apportent un changement radical à nos pratiques de programmation

Concepts

- L’intérêt des concepts est considérable :
  - Du code source aux intentions plus claires, mieux documenté

Concepts

- L’intérêt des concepts est considérable :
  - Du code source aux intentions plus claires, mieux documenté
  - De meilleurs diagnostics en cas d’erreurs dans du code générique

Concepts

- L’intérêt des concepts est considérable :
  - Du code source aux intentions plus claires, mieux documenté
  - De meilleurs diagnostics en cas d’erreurs dans du code générique
  - Toute la puissance des interfaces, sans les coûts (pas intrusif, pas d’indirection polymorphique à l’exécution)

Concepts

- L’intérêt des concepts est considérable :
  - Du code source aux intentions plus claires, mieux documenté
  - De meilleurs diagnostics en cas d’erreurs dans du code générique
  - Toute la puissance des interfaces, sans les coûts (pas intrusif, pas d’indirection polymorphique à l’exécution)

Cette dernière puce mérite nuance : s’il est possible d’exprimer un vector<Itf*> pour un type Itf donné et d’y placer des pointeurs sur des instances de diverses classes implémentant cette interface, il n’est pas possible de faire de même pour des types implémentant un même concept

Concepts

- La bibliothèque standard, avec C++20, est enrichie de concepts
- Les concepts de la bibliothèque standard sont décrits dans <concepts> et dans https://en.cppreference.com/w/cpp/concepts

Pour un index à jour des concepts de la bibliothèque standard, voir https://timsong-cpp.github.io/cppwp/n4861/conceptindex

Concepts

- L’expérience a montré qu’il est préférable d’avoir des concepts de granularité grossière
  - Ne suivez pas les exemples simplistes de ce document en faisant un concept pour le fait d’être « incrémentable » par exemple
  - En retour, un concept exprimant la gamme d’opérations attendues d’un type utilisé à titre de forward_iterator est plus fécond
  - En général, les concepts sont associés aux requis des algorithmes et sont définis sur cette base

Concepts

- Qu'il existe un nombre restreint de cas pour lesquels un concept ne s'applique pas dans un type donné n'invalide pas le concept
  - Par exemple, double satisfait std::totally_ordered même si le type double admet des NaN
  - De même, on peut définir T* comme étant « déréférençable » même si un T* peut pointer à une adresse invalide ou être nullptr
  - Le mécanisme pour discuter de la validité des **valeurs** d'un certain type sera les contrats, qui n'ont pas « passé la barre » de C++20

Concepts

- Cas d’espèce : nous souhaitons exprimer une classe Array<T,N> représentant N éléments de type T contigus en mémoire
- Nous aimerions que son initialisation soit simple

Concepts

```
template <class T, int N>
  class Array {
    T elems[N];
  public:
    template <class ... Ts>
      Array(Ts ... elems) : elems{ elems... } {
      }
  };
int main() {
  Array arr{ 2,3,5,7,11 };
}
```

Concepts

Ceci ne compile pas : https://wandbox.org/permlink/yWpjQ5WiP1FxiLlH car le compilateur ne sait pas comment faire la correspondance entre la construction de arr et le constructeur de Array<T,N>

```
template <class T, int N>
  class Array {
    T elems[N];
  public:
    template <class ... Ts>
      Array(Ts ... elems) : elems{ elems... } {
      }
  };
int main() {
  Array arr{ 2,3,5,7,11 };
}
```

Concepts

```
#include <concepts>
using namespace std;
template <class T, int N>
  class Array {
    T elems[N];
  public:
    template <class ... Ts>
      Array(Ts ... elems) : elems{ elems... } {
      }
  };
template <class T, same_as<T>...Ts>
  Array(T, Ts...) -> Array<T, 1 + sizeof...(Ts)>;
int main() {
  Array arr{ 2,3,5,7,11 };
}
```

Concepts

Ceci compile sans peine : https://wandbox.org/permlink/ZYPLXsJcll1uyfza (notez l'application variadique de same_as aux types dans le *Deduction Guide*)

```
#include <concepts>
using namespace std;
template <class T, int N>
  class Array {
    T elems[N];
  public:
    template <class ... Ts>
      Array(Ts ... elems) : elems{ elems... } {
      }
  };
template <class T, same_as<T>...Ts>
  Array(T, Ts...) -> Array<T, 1 + sizeof...(Ts)>;
int main() {
  Array arr{ 2,3,5,7,11 };
}
```