CALCUL PERFORMANT ET PRATIQUE AVEC C++20

14 − 17 NOVEMBRE 2022 14E JOURNÉES INFORMATIQUES IN2P3/IRFU

Sylvain JOUBE

Joël FALCOU, Hadrien GRASLAND, David CHAMONT.

LISN LABORATOIRE INTERDISCIPLINAIRE DES SCIENCES DU NUMÉRIQUE

UCLab Irène Joliot-Curie Laboratoire de Physique des 2 Infinis

Problématiques de calcul pour la HEP : Ergonomie vs Performance

- Comment diminuer le temps de dév. des non experts en calcul (ex : physiciens) ?
- Comment garantir une maintenance simple par les scientifiques ?
- Comment s'abstraire de la complexité du matériel ?
- Python : bonne ergonomie mais seulement runtime donc mauvaises performances

Une semi-solution: C++

- *Zero Cost Abstraction* via plusieurs mécanismes dont les *templates* (l'objet n'est pas performant)
- Les *templates* ont plusieurs problèmes :
  - Difficiles à prendre en main et à écrire
  - Binaires énormes
  - Très long à compiler
  - Messages d'erreurs longs et cryptiques
- Choix entre Ergonomie et Performances ?

Somme de deux instances de types ayant un membre nommé `value`

Avant substitution

```
struct S1 { int value; S1(int v) : value(v) {} };
struct S2 { int value; S2(int v) : value(v) {} };


template<typename A, typename B>
auto sum(A a, B b) -> decltype(a.value+b.value)
{
  return a.value + b.value;
}

int main()
{
  S1 s1{1};
  S2 s2{2};
  auto x = sum(s1, s2);
}
```

Somme de deux instances de types ayant un membre nommé `value`

Types substitués à la compilation

```
struct S1 { int value; S1(int v) : value(v) {} };
struct S2 { int value; S2(int v) : value(v) {} };

// Instanciation de la fonction avec les types S1 et S2
/*template<typename A, typename B>*/
int  sum(S1 a, S2 b) /*-> decltype(a.value+b.value)*/
{
  return a.value + b.value;
}

int main()
{
  S1 s1{1};
  S2 s2{2};
  auto x = sum(s1, s2);
}
```

Template attendant un conteneur

Utilisation avec std::array

```
// Retourne la valeur de l'élément à la position n
template <typename T>
auto get_nth(T const& c, int n)
{
  assert(n < std::size(c));
  auto b = std::begin(c);
  for(int i = 0; i<n; ++i) b++;
  return *b;
};

int main()
{
    std::array<float, 8> a;
    auto x1 = get_nth(a, 2); // Compile
    auto x2 = get_nth(1, 2); // Grosse erreur sur le terminal
}
```

Template attendant un conteneur : Message d'erreur long et peu clair

```
main.cpp:22:14: error: no matching function for call to 'size'
  assert(n < std::size(c));
             ^~~~~~~~
/usr/include/assert.h:93:27: note: expanded from macro 'assert'
     (static_cast <bool> (expr)                                         \
                          ^~~~
main.cpp:36:14: note: in instantiation of function template specialization 'get_nth<int>' requested here
    auto x = get_nth(1,2);
             ^
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:264:5: note: candidate template ignored: su
    size(const _Container& __cont) noexcept(noexcept(__cont.size()))
    ^
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:274:5: note: candidate template ignored: co
    size(const _Tp (&)[_Nm]) noexcept
    ^
main.cpp:24:12: error: no matching function for call to 'begin'
  auto b = std::begin(c);
           ^~~~~~~~~
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/initializer_list:90:5: note: candidate template ignored: could
    begin(initializer_list<_Tp> __ils) noexcept
    ^
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:52:5: note: candidate template ignored: sub
    begin(_Container& __cont) -> decltype(__cont.begin())
    ^                                     ~
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:63:5: note: candidate template ignored: sub
    begin(const _Container& __cont) -> decltype(__cont.begin())
    ^                                           ~
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:95:5: note: candidate template ignored: cou
    begin(_Tp (&__arr)[_Nm]) noexcept
    ^
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:113:31: note: candidate template ignored: c
  template<typename _Tp> _Tp* begin(valarray<_Tp>&) noexcept;
                              ^
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:114:37: note: candidate template ignored: c
  template<typename _Tp> const _Tp* begin(const valarray<_Tp>&) noexcept;
                                    ^
2 errors generated.
```

Problématiques de calcul pour la HEP

Une solution performante et plus ergonomique : C++20

- C++20 introduit :
  - **Concepts** : généricité contrainte
  - **Calculs à la compilation** via `constexpr`
- Potentiel pour améliorer l'Ergonomie et les Performances ?

Concepts - généricité contrainte, explication

Principes généraux

- Idée : contraindre les paramètres template (sans avoir recours aux méta-monstruosités)
- Utilisable pour optimiser/spécialiser les fonctions selon les types

Bénéfices attendus

- Code plus simple à comprendre et à écrire
- Gains en temps de compilation
- Messages d'erreur plus clairs

Concepts - message d'erreur

```
main.cpp:29:14: error: no matching function for call to 'get_nth'
    auto x = get_nth(1,2);
             ^~~~~~
main.cpp:13:6: note: candidate template ignored: constraints not satisfied [with c:auto = int]
auto get_nth(container auto const& c, int n)
     ^
main.cpp:13:14: note: because 'int' does not satisfy 'container'
auto get_nth(container auto const& c, int n)
             ^
main.cpp:8:5: note: because 'std::begin(c)' would be invalid: no matching function for call to 'begin'
  { std::begin(c) } -> std::forward_iterator;
    ^
1 error generated.
```

- Ne parle que du code écrit par l'utilisateur
- Plus clair et concis

Concepts - Une écriture qui reste complexe

Un concept vérifie certaines contraintes à la compilation

- Validité d'une expression arbitraire
- Propriétés des valeurs de retour
- Présence d'un type interne

```
template<typename T>
concept streamable = requires(T const& t, std::ostream& os)
{
  { os << t }; // Expression à vérifier
};

template<typename P, typename... Args>
concept predicate = requires(P pred, Args&&... args)
{
  { pred(std::forward<Args>(args)...)  } -> std::same_as<bool>;
};
```

Concepts - Une écriture qui reste complexe

Combinaison et Raffinement

- Un concept peut se définir à partir d'un autre
- Sémantique proche de l'héritage d'interface

```
template<typename T>
concept container = std::regular<T> && requires(T const& c)
{
  { std::begin(c) } -> std::forward_iterator;
  { std::end(c)   } -> std::forward_iterator;
  { std::size(c)  } -> std::integral;
};

template<typename T>
concept random_access_container = container<T> && requires(T const& c, int i)
{
  { c[i] };
};
```

Concepts - Une utilisation très simple

Spécialisation de fonction avec syntaxe compacte (surcharge)

```
// Cas général
auto get_nth(container auto const& c, int n)
{
  assert(n < std::size(c));
  auto b = std::begin(c);
  for(int i = 0;i<n;++i) b++;
  return *b;
};

// Spécialisation
auto get_nth(random_access_container auto const& c, int n)
{
  assert(n < std::size(c));
  return c[n];
};
```

- Contrainte la plus forte choisie

Constexpr

Objectif

- Être plus ergonomique que des macros
- Calcul élaboré de constante de compilation (exemple : taille de tableau)
- Ressembler à du C++ classique

Mise en oeuvre

- Fonctions constexpr
- `if constexpr`

Constexpr - Fonctions constexpr

Avant `constexpr`

```
template<int N> struct factorial    { static const int value = N * factorial<N-1>::value; };
template<>      struct factorial<0> { static const int value = 1; };

std::array<float, factorial<4>::value>> x; // appel dans un contexte compile-time
```

Stratégie `constexpr`

```
constexpr int factorial(int n)
{
  int r = 1;
  for(int i=2;i<=n;++i) r *= i;
  return r;
}

std::array<float, factorial(4)> x; // appel dans un contexte compile-time
auto y = factorial(7);             // appel dans un contexte runtime
```

Constexpr - `if constexpr`

Sélection de code à la compilation : avant if constexpr

```
#include <type_traits>

template<typename T, typename Enable = void>
struct has_size : std::false_type {};

template<typename T>
struct has_size<T, std::void_t<decltype(std::declval<T>().size())>> : std::true_type {};

template<typename T> auto taille(T const& t, std::false_type) { return 1; }

template<typename T> auto taille(T const& t, std::true_type) { return t.size(); }

template<typename T> auto taille(T const& t){ return taille(t,typename has_size<T>::type{}); }

auto n = taille(std::vector<int>(58));  //  n == 58
auto m = taille(8);                     //  m == 1
```

Constexpr - `if constexpr`

Sélection de code à la compilation : avec if constexpr

```
template<typename T>
auto taille(T const& t)
{
  if constexpr(requires(T const& t) { t.size(); }) return t.size();
  else                                             return 1;
}

auto n = taille(std::vector<int>(58));  //  n == 58
auto m = taille(8);                     //  m == 1
```

- `if constexpr` permet la sélection de code à la compilation ergonomique
- Combinaison avec des contraintes sur les types
- Une seule fonction est nécessaire

Illustration : Kiwaku

Objectif et composants de base

- *(yet another)* Bibliothèque de tableaux multidimensionnels
- Vues *(non owning)* et tables *(owning)*
- Algorithmes de parcours optimisés et ergonomiques
- Support de multiples contextes d'exécution (GPU, CPU, vectorisation...)
- Non pris en charge : algèbre linéaire, expression templates, calculs

Principes d'implémentation

- **Utilise C++20** : meilleure ergonomie à performances égales
- Programmation générative via `templates`, `concepts`, `constexpr`
- Tableaux (conteneurs) définissables via des paramètres de haut niveau
- Squelettes algorithmiques

Exemples avec Kiwaku

Création et manipulation de vue avec concepts

```
#include <kwk/kwk.hpp>

// Attend uniquement une vue 1D de float : contrainte via les concepts C++20
void square_each(  kwk::concepts::view<kwk::_1D, kwk::as<float>>   auto&&  v  )
{
  // Pour tous les index i                :   élever au carré à la position i
  for(std::ptrdiff_t i=0; i<v.numel(); ++i)   v(i) *= v(i);
}

// Construction d'une vue 1D de float à partir d'un pointeur et élévation au carré
void demo_kiwaku_view(float* data, int view_size)
{
  kwk::view v { kwk::size = view_size, kwk::source = data };
  square_each(v);
}
```

Exemples avec Kiwaku

Création et manipulation de vue avec concepts : équivalent compact

```
#include <kwk/kwk.hpp>

void demo_kiwaku_view(float* data, int view_size)
{
  kwk::view v { kwk::size = view_size, kwk::source = data };

  // Élévation au carré
  kwk::for_each([](auto& e) { e*= e; }, v);
}
```

Exemples avec Kiwaku

Cas d'une vue 2D

```
#include <kwk/kwk.hpp>

void demo_kiwaku_view(float* data, int d1, int d2)
{
  kwk::view v { kwk::size = kwk::of_size(d1, d2), kwk::source = data };

  // Élévation au carré
  kwk::for_each([](auto& e) { e*= e; }, v);
}
```

Exemples avec Kiwaku

Peu importe l'ordre des options !

```
#include <kwk/kwk.hpp>

void demo_kiwaku_view(float* data, int d1, int d2)
{
  kwk::view v { kwk::source = data, kwk::size = kwk::of_size(d1, d2) };

  // Élévation au carré
  kwk::for_each([](auto& e) { e*= e; }, v);
}
```

Conclusion

Nouveautés C++20 : meilleure ergonomie, mêmes performances

- Concepts :
  - Généricité contrainte via prédicat compile-time
  - Combinaisons possibles et spécialisation (surcharge) de fonctions aisée
- Constexpr :
  - Similaire à du C++ classique, plus ergonomique que les macros
  - Permet de passer des valeurs plus complexes en paramètres de template

Kiwaku et Future work

- Stockage et traitement performants pour tableaux multidimensionnels en C++20
- Ergonomique pour les développeurs et les utilisateurs
- Algorithmes de parcours optimisés
- Futur support des contextes d'exécution GPU via SYCL

Merci !

- Kiwaku : `https://github.com/jfalcou/kiwaku`