Comment écrire un algorithme « standard »

Table des matières

- I. Introduction
- Étape 1 : Poser clairement le problème
- Étape 2 : Essayer d'éviter tout travail
- Étape 3 : Définir l'interface
  - Étape 3-a : Déterminer le nom de la fonction
  - Étape 3-b : Déterminer les arguments
  - Étape 3-c : Déterminer le type de la valeur de sortie
  - Étape 3-d : Déterminer les contrats de l'interface (les exigences sur les valeurs des arguments et le résultat)
  - Étape 3-e : Déterminer les concepts (les exigences sur les types d'arguments et le résultat)
- Étape 4 : Écrire les tests
- Étape 5 : Écrire l'algorithme
- Étape facultative 6 : Optimiser
- II. Résumé
- III. Remerciements

Cet article explique comment écrire un algorithme similaire à ceux présents dans la bibliothèque standard du C++.

Les deux auteur et traducteur

Explicit C++

Traducteur : Eric GERARD

L'article

Publié le 28 avril 2015

Version PDF Version hors-ligne

ePub, Azw et Mobi

I. Introduction▲

Un de mes dadas à propos de la plupart des cours, tutoriels et livres sur le C++ est de dire qu'ils semblent ostensiblement ignorer les algorithmes. Je ne parle pas d'algorithmes d'un point de vue théorique, mais des algorithmes de la bibliothèque standard. Je suis étonné du nombre de professeurs de C++ demandant à leurs élèves de ne pas utiliser d'algorithmes dans leurs travaux. Et, à part les algorithmes standards actuels, la majorité des livres, tutoriels et cours ne font aucun effort pour discuter des algorithmes qui suivent le même schéma général. À mon avis, écrire un algorithme standard devrait être un des points clés d'un cours moderne sur C++ pour débutant.

Et, bien sûr, j'ai joint mon code à ma parole : dans chaque cours de C++ que j'ai donné, un des modules décrivait un algorithme standard. C'était la fonction copy_n() avant C++11, mais depuis c'est copy_until(), qui copie une suite jusqu'à ce qu'une valeur donnée soit trouvée(1). Ce papier sera un guide à travers les étapes de création d'un algorithme standard. Il ne s'agit pas de se focaliser sur l'algorithme lui-même, mais sur son processus de création.

Étape 1 : Poser clairement le problème▲

Le problème que nous voulons traiter est de copier une suite d'éléments jusqu'à - mais sans l'inclure - un élément de terminaison donné, ou bien la totalité de la suite si l'élément recherché n'existe pas dans la suite. J'ai trouvé plusieurs situations où cela était généralement intéressant. Par exemple, pour l'analyse : lorsque vous analysez des données, il y a de nombreux cas où vous souhaitez récupérer des données jusqu'à un certain caractère - comme lire une ligne entière (le caractère de terminaison est '\n') ou bien une chaîne entre guillemets (le guillemet fermant étant le caractère de terminaison). Il est aussi fréquent de lire une liste de nombres terminés par zéro (pas '\0' mais bien le chiffre 0).

La fonction doit donc, bien évidemment, récupérer la chaîne de caractères en entrée, soit sous forme d'une paire d'itérateurs ou bien d'un ensemble… Nous allons coder les deux options parce que c'est simple. Il nous faudra également un itérateur Output pour

écrire les données copiées. Et, bien entendu, nous définirons le caractère d'arrêt. Soit quelque chose comme cela :

Sélectionnez

```
// version avec itérateurs
copy_until (first, last, out, stop_value) ;
// version avec un ensemble
copy_until (in, out, stop_value) ;
```

Nous nous intéresserons aux ordres des arguments et à leurs noms plus tard. Conceptuellement, ce que nous souhaitons faire est simplement :

Sélectionnez

```
copy (first, find (first, last, stop_value), out) ;
```

Étape 2 : Essayer d'éviter tout travail▲

Le meilleur algorithme que vous puissiez écrire est : pas d'algorithme ; si vous pouvez effectuer le travail à l'aide d'algorithmes standards, il serait stupide d'écrire le vôtre.

Donc, dans cette étape, le but est de voir si ce que nous voulons faire est déjà faisable à l'aide d'algorithmes standards. Si ça l'est, nous avons fini - le problème est résolu. Au pire, la seule chose à faire est d'emballer la solution standard dans notre fonction.

Comme évoqué ci-dessus, le problème que nous essayons de résoudre est juste :

Sélectionnez

```
copy (first, find (first, last, stop_value), out) ;
```

Cela semble vouloir dire que nous avons fini - il n'y a aucun besoin d'écrire un algorithme quand une simple ligne fait le travail.

Mais, il y a un problème. Ce code ne fonctionne pas lorsque les itérateurs de la source sont des itérateurs Input. Pour comprendre pourquoi, mettons ce code sur deux lignes :

Sélectionnez

```
auto p = find (first, last, stop_value) ;
copy (first, p, out) ;
```

Avec des itérateurs Forward, vous pouvez effectuer plusieurs passes sur une suite de données, mais avec des itérateurs Input, vous ne pouvez effectuer qu'une seule passe. La recherche s'effectue en une passe, et comme vous ne pouvez en effectuer qu'une, la copie ne fonctionne pas. Avec des itérateurs Input, la recherche et la copie ne peuvent être effectuées séparément - elles doivent être effectuées en une passe.

Il existe une surcharge de getline() qui peut lire jusqu'à un caractère spécifique - newline par défaut. Mais cela ne fonctionne que lorsque l'entrée est un flux et la sortie une chaîne (j'ignore ici, volontairement, la version utilisant un tampon de caractères), et elle peut utiliser uniquement un caractère comme valeur d'arrêt. Cela est trop rigide pour être généralement utile. Il n'existe pas d'algorithmes standards capables d'effectuer la recherche et la copie en une seule passe (du moins, pas sans faire quelque chose de particulier, comme trafiquer la fonction find_if()). Ce qui signifie que nous allons devoir écrire la nôtre.

Étape 3 : Définir l'interface▲

Lorsque vous écrivez votre propre algorithme, il est conseillé de copier le style des algorithmes standards autant que possible. Faire les choses d'une façon différente et étonnante va uniquement embrouiller et ennuyer quiconque essaiera de l'utiliser, et vous demandera plus de travail pour documenter clairement en quoi votre algorithme se différencie du standard.

Définir l'interface est une des étapes les plus importantes du processus… si vous faites les choses à moitié, vous allez pratiquement tout gâcher. Donc, prenez votre temps, et réfléchissez longtemps et profondément. Plus vous y passerez de temps et plus vous en gagnerez, non seulement en écriture, mais aussi lors de futures utilisations.

Étape 3-a : Déterminer le nom de la fonction▲

Dans le cas qui nous intéresse ici, le nom est évident, mais cela n'est pas toujours le cas. Parfois, déterminer comment nommer un algorithme est une décision très délicate.

Les personnes du comité de normalisation parlent d'abri à vélo (bike shedding). Ce terme se réfère à une observation faite en 1957 lorsqu'un comité chargé de concevoir une centrale nucléaire, passa un temps incroyable sur des détails triviaux comme : « de quelle couleur peindre l'abri à vélo du personnel ? » alors qu'il ne s'intéressa pas assez aux détails fonctionnels importants de la centrale. Autrement dit, la couleur de l'abri à vélo n'a aucune importance dès l'instant qu'il a été correctement conçu ou, en termes C++, le nom de la fonction n'a aucune importance si vous avez proprement et clairement spécifié son comportement.

Les personnes du comité de normalisation sont un peu pince-sans-rire lorsqu'ils utilisent ce terme, car la vérité est qu'ils prennent la question du nommage très sérieusement. Non seulement ils font attention à ne pas gâcher des noms qui pourraient être utilisés pour de futures fonctionnalités(2), mais encore ils s'assurent que les noms ne sont pas troublants ou trompeurs.

Cela vaut donc le coup de prendre quelques minutes à réfléchir à comment nommer votre algorithme. Vous voulez un nom simple et clair, facile à retenir et écrire, et offrant des possibilités de confusion limitée.

Par chance, comme je le disais, dans notre cas, le choix est assez évident. Vous voulez copier une suite de caractères jusqu'à ce qu'une valeur donnée soit rencontrée. Par exemple, « copier les données sources jusqu'à la valeur 42 ». Copier… jusqu'à… copy_until. D'autres options pourraient être copy_up_to ou bien copy_upto, mais je pense que copy_until définit clairement l'utilisation réelle (et aussi, j'utiliserai copy_up_to ou copy_upto pour une fonction de copie qui copierait jusqu'à N éléments, s'arrêtant plus tôt, si les données en entrée se tarissent).

L'algorithme doit être inséré dans un espace de noms - polluer l'espace de noms global est une mauvaise idée. L'espace de noms que vous choisirez sera probablement déterminé par le projet sur lequel vous travaillez, donc dans ce papier j'utiliserai l'espace de noms indi.

Finalement, l'algorithme doit être placé dans un fichier d'en-têtes. Le nom de ce fichier d'en-têtes, et la façon dont il doit être inclus, fait également partie de l'interface. À l'instar de l'espace de noms, déterminé habituellement par le projet, j'estime que le fichier d'en-têtes doit être inclus comme #include "indi/copy_until.hpp".

Étape 3-b : Déterminer les arguments▲

Lors de cette étape, vous devez déterminer quels arguments sont nécessaires pour cette fonction, ce qui n'est pas toujours évident, car vous pouvez avoir des cas où certains arguments sont optionnels ou alors vous pouvez avoir plusieurs possibilités différentes ce qui impliquera différentes surcharges. Vous devez également définir leur ordre.

Nous avons déjà noté précédemment les arguments dont nous avions besoin : soit une paire d'itérateurs ou un ensemble, un itérateur pour la sortie, et la valeur d'arrêt. Maintenant, il faut décider comment prendre ces éléments : par valeur ou par référence ? Et dans quel ordre ?

Je conseille toujours de parcourir les algorithmes standards pour rechercher ceux qui sont similaires à celui que vous voulez écrire. Dans notre cas, copy() est un candidat évident, mais copy_if() est sans doute plus intéressant.

Comme pour notre fonction envisagée copy_until(), copy_if() parcourt une suite en entrée pour la copier dans une suite en sortie, mais sans copier toute la séquence d'entrée. Le prédicat détermine quels éléments seront copiés, ce qui est analogue à la valeur d'arrêt de copy_until() effectuant la même chose, bien que ce soit fait de façon différente. Comme copy_until(), copy_if() prend quatre arguments :

Sélectionnez

```
copy_if (
  InputIterator first,
  InputIterator last,
  OutputIterator out,
  UnaryPredicate f) ;
```

Ceci nous indique dans quel ordre les arguments de copy_until() devraient être :

Sélectionnez

```
copy_until (
  InputIterator first,
  InputIterator last,
  OutputIterator out,
  ??? stop) ;
```

De quel type doit être l'argument valeur d'arrêt ? La réponse évidente est du type de la séquence d'entrée :

Sélectionnez

```
copy_until (
  InputIterator first,
  InputIterator last,
  OutputIterator out,
  typename iterator_traits < InputIterator >::value_type stop) ;
```

Comme la valeur d'arrêt est seulement utilisée pour des comparaisons, l'algorithme n'a pas de raison logique d'en faire une copie. Elle doit être prise par référence constante.

Mais attendez un peu.

Souvenez-vous de ce que je vous ai dit au début de cette étape ? Plus vous passerez de temps et d'efforts dans cette phase de définition, et moins de problèmes vous aurez ensuite. Arrêtons-nous là et réfléchissons aux conséquences de ce que nous venons de faire.

La valeur d'arrêt doit-elle être réellement du même type que la séquence d'entrée ? Non ! Elle doit seulement être compatible en égalité avec ce type.

Considérez ceci : imaginez que vous lisiez une suite de mots, et que la valeur d'arrêt soit 'end'. Naturellement, vous écrirez ceci :

Sélectionnez

```
copy_until (
  istream_iterator < string > {in},
  istream_iterator < string > {},
  out,
  "end");
```

Si vous écrivez copy_until() en forçant la valeur d'arrêt à être identique au type de valeur de l'itérateur Input, cela nécessitera de construire une chaîne à partir de 'end', ce qui requiert une allocation dynamique, et ainsi de suite(3). Mais… pourquoi ? Il n'y a aucune raison pour cela. std::string est capable de comparer une suite de char (grâce à un pointeur) directement (en fait, si la séquence de char est donnée littéralement, les compilateurs peuvent optimiser la comparaison).

Une autre option serait de laisser le type de la valeur d'arrêt non spécifié :

Sélectionnez

```
copy_until (
  InputIterator first,
  InputIterator last,
  OutputIterator out,
  T const& stop) ;
```

Non seulement cela rend cet algorithme plus flexible, mais encore cela permet des possibilités d'optimisation… et c'est plus facile à écrire !

Passer les itérateurs par valeur est une pratique standard. Les itérateurs sont définis pour être peu coûteux à copier. Ceci fait que notre interface devient :

Sélectionnez

```
template <
  typename InputIterator,
  typename OutputIterator,
  typename T >
auto copy_until (
  InputIterator first,
  InputIterator last,
  OutputIterator out,
  T const& stop) ;
```

Et maintenant, si on parlait de la version « ensemble » ?

En fait, la seule différence entre la version « itérateur » et la version « ensemble » réside dans les arguments entrés. La conversion de la paire d'itérateurs en ensemble donne :

Sélectionnez

```
copy_until (
  Range in,
  OutputIterator out,
  T const& stop) ;
```

Mais, bien entendu, cela n'a aucun sens de passer un ensemble par valeur ; le but de l'algorithme est déjà de le copier vers l'ensemble de sortie. Où serait la logique de le copier deux fois ? Alors, devons-nous prendre l'ensemble comme :

a. Une référence constante (lvalue),
b. Une référence non constante (lvalue),
c. Une nouvelle référence de stockage ?

Nous pouvons éliminer b immédiatement : il n'y aucune raison logique pour un algorithme sensé lire et copier une plage de données de devoir la modifier. Il nous reste donc a et c.

Il ne semble y avoir aucune raison pour que l'algorithme ne prenne le contrôle de l'ensemble en entrée à aucun moment. Tout ce qu'il a besoin de faire est de lire la plage (pour copier) et pas pour la changer ou en prendre le contrôle. Ce qui signifie nous avons besoin d'une référence constante à la plage de données :

Sélectionnez

```
copy_until (
  Range const& in,
  OutputIterator out,
  T const& stop) ;
```

Et ainsi, notre interface version « ensemble » devient :

Sélectionnez

```
template <
  typename Range,
  typename OutputIterator,
  typename T >
auto copy_until (
  Range const& in,
  OutputIterator out,
  T const& stop) ;
```

Tout est correct maintenant, donc nous pouvons continuer. Cependant, je veux juste attirer l'attention sur quelque chose qui pourrait arriver dans le futur du C++. À ce jour, tous les algorithmes standards qui prennent des paires d'itérateurs en arguments

imposent que les deux itérateurs soient du même type. Or, Eric Niebler travaille sur la prochaine génération de bibliothèque d'ensembles pour le C++, et un des points qu'il a soulevés est qu'il serait plus flexible et plus efficace d'autoriser que le type de l'itérateur de fin puisse être différent de celui de l'itérateur de début. Si vous prenez cette idée à cœur, vous pouvez écrire la version « itérateur » de copy_until() de la sorte :

Sélectionnez

```
template <
  typename InputIterator,
  typename Sentinel,
  typename OutputIterator,
  typename T >
auto copy_until (
  InputIterator first,
  Sentinel last,
  OutputIterator out,
  T const& stop) ;
```

Je ne vais pas le faire dans cet article, mais c'est quelque chose à considérer.

Étape 3-c : Déterminer le type de la valeur de sortie▲

Tous les algorithmes standards, qui fonctionnent avec un ensemble en entrée et en sortie, renvoient l'itérateur modifié de la séquence de sortie. C'est vrai aussi pour notre fonction modèle : copy_if().

Ceci a du sens, car vous ne pouvez pas toujours savoir où la séquence de sortie va s'arrêter à la fin de l'algorithme. Par exemple, considérez copy_if() : vous ne savez pas, avant l'opération, combien d'éléments vont satisfaire au prédicat. Cela peut être tous les éléments, la moitié ou bien aucun. Récupérer l'itérateur de sortie vous autorise à continuer depuis l'endroit où la copie s'est terminée, ou bien vous donne la possibilité de savoir combien d'éléments ont été copiés, et ainsi de suite. C'est juste très facile.

La même logique s'applique à copy_until() : vous ne savez pas avant l'opération où se trouve la valeur d'arrêt, ou même si vous trouverez cette valeur d'arrêt. Et, vous ne pouvez pas le trouver, au moins pour les ensembles en entrée. Vous ne pouvez donc pas savoir combien d'éléments seront copiés. Récupérer l'itérateur modifié de sortie vous permet de le savoir, ou de continuer à partir de l'endroit où la copie s'est arrêtée.

Nous voulons donc, définitivement, renvoyer l'itérateur modifié de la chaîne de sortie.

Cependant, à l'inverse de copy_if() et de tous les algorithmes standards qui possèdent un ensemble de données en entrée et en sortie, copy_until() ne parcourt pas nécessairement toute la séquence de données en entrée. La fonction peut s'arrêter plus tôt, dès qu'elle trouve le critère d'arrêt. Comme mentionné ci-dessus, vous ne pouvez pas savoir si ou à quel endroit cela va se produire dans la plage d'entrée. Si vous ne renvoyez pas l'itérateur de l'endroit où la copie s'est arrêtée, il y aura de nombreux cas où vous n'aurez aucune idée si la suite de données a été totalement copiée ou si ce n'est pas le cas où la copie s'est arrêtée.

Ainsi, à l'inverse de la plupart des algorithmes standards, nous voulons aussi renvoyer l'itérateur Input modifié. Cela signifie que nous voulons renvoyer deux choses. Le seul moyen d'y parvenir en C++, sans se tracasser avec un type personnalisé, est d'utiliser un tuple, ou, dans le cas particulier de deux valeurs, une paire.

OK, nous voulons retourner l'itérateur de type Input modifié et l'itérateur de type Output modifié dans une paire. Mais alors dans quel ordre ?

Malheureusement, la bibliothèque standard d'algorithmes n'est pas d'une grande aide ici. Pratiquement aucun algorithme standard ne renvoie de paires d'itérateurs, le peu d'entre eux qui le font - comme mismatch() - n'a pas véritablement la même sémantique. Le reste de la bibliothèque n'est pas d'une grande aide non plus - des fonctions, comme insert() pour les map ou les set, ne sont pas réellement pertinentes.

Nous devons donc rechercher ailleurs. Boost.Algorithm possède quelques algorithmes qui renvoient des paires, comme copy_while(). Ils retournent les itérateurs dans le même ordre que celui dans lequel ils ont été passés - en premier, l'entrée, en second la sortie. C'est aussi ce que fait mismatch(), il semble donc que ce soit le choix logique.

Cela donne comme spécification de retour :

Sélectionnez

```
auto copy_until (/*…*/) - >
  pair <InputIterator, OutputIterator>
```

Cela sera également la spécification de retour pour la version « ensemble », avec l'avertissement suivant que l'itérateur Input sera decltype(begin (in)) (avec un appel ADL-enabled à begin()).

Cela aussi longtemps que je resterai logique au fil de cet article, mais vous voudrez sans doute y penser plus profondément, je vous livre ici quelque chose pour pondérer mon propos :

Sélectionnez

```
auto src = vector < int > {1, 2, 3, 4};
auto out = vector < int > {};

// version avec itérateurs
```

```
auto result1 = copy_until (begin (src), end (src),
back_inserter (out), 3) ;
// out contient maintenant {1,2}

// Positionne l'itérateur à l'endroit où la copie s'est arrêtée
// pointe sur 3 dans le src
auto it1 = get < 0 > (result1) ;

// Maintenant essayons de le modifier
*it1 = 0 ; // pas de problème, src devient {1,2,0,4}
```

Pas de problèmes, donc avec la version « itérateurs »… mais maintenant essayons la même chose avec la version « ensemble » de copy_until() :

Sélectionnez

```
auto src = vector < int > {1, 2, 3, 4};
auto out = vector < int > {};

// version avec plage de données
auto result2 = copy_until (src, back_inserter (out), 3) ;
// out contient maintenant {1,2}

// Positionne l'itérateur à l'endroit où la copie s'est arrêtée
// pointe sur 3 dans le src
auto it2 = get < 0 > (result2) ;

// Essayons maintenant de le changer
*it2 = 0 ; // erreur de compilation
```

D'où vient l'erreur ? Et bien, la version itérateur utilise simplement les types d'itérateurs donnés - car, comme src n'est pas constant, l'itérateur est non-constant. Comme les itérateurs arguments ne sont pas des itérateurs constants, les itérateurs de la paire renvoyée par la fonction ne sont pas non plus constants et donc nous pouvons les utiliser en écriture sans souci. Alors que la version ensemble récupère les données avec une référence constante… ce qui signifie que le type d'itérateur renvoyé sera toujours un itérateur constant. Si vous voulez retourner un itérateur non constant sur l'endroit où a été rencontré le critère d'arrêt, ce n'est pas facile à faire quand bien même c'est possible.

Vous me dites : quelle est la solution dans ce cas ? Devez-vous écrire une paire de fonctions - une récupérant les données à l'aide d'une référence constante et une avec une référence non constante ? Devez-vous utiliser une référence à une adresse de stockage ? Et si vous utilisez une référence à une adresse de stockage, devez-vous refuser de renvoyer un itérateur sur les valeurs d'entrée lorsqu'elle est passée comme une rvalue (car cela signifie retourner un itérateur sur une séquence qui n'existe plus) ? Quelle est donc la bonne chose à faire ?

Voilà une chose à laquelle réfléchir si vous êtes intéressé, mais pour l'instant je vais ignorer ce cas.

Étape 3-d : Déterminer les contrats de l'interface (les exigences sur les valeurs des arguments et le résultat)▲

Les algorithmes sont des fonctions, cela implique donc deux types de contrats : les conditions initiales et les conditions finales (les invariants étant sans intérêt).

Les conditions initiales pour copy_until() sont très simples et évidentes, car ce sont les mêmes conditions initiales pour la plupart des algorithmes standards :

- les itérateurs de début et de fin doivent se référer aux mêmes données ;
- l'itérateur de début doit pointer avant l'itérateur de fin dans la séquence, ou à la même place (de telle sorte que l'itérateur de début atteigne éventuellement l'itérateur de fin) ;
- lorsque l'itérateur de début est différent de l'itérateur de fin, il doit être déréférencable ;
- l'itérateur de la séquence de sortie doit pointer sur une plage suffisamment grande pour contenir toutes les valeurs qui seront copiées (vous ne connaissez pas cette quantité de données au départ, mais elle ne sera pas plus grande que le nombre d'éléments de la séquence en entrée).

Et pour les fonctions de copie (exceptée copy_backward()):

- l'itérateur de sortie ne doit pas pointer entre les itérateurs Input de début et de fin.

C'est tout à ce sujet. En réalité, il n'est pas nécessaire de parler explicitement de ces conditions initiales, car ce sont des conditions initiales standards et quiconque connaît les algorithmes standards respectera ces conditions initiales pour tous les algorithmes par défaut (c'est d'ailleurs une autre bonne raison de faire en sorte que vos algorithmes ressemblent autant que possible aux algorithmes standards).

Les conditions finales sont pratiquement aussi simples et évidentes. Cependant, il y a quelques détails intéressants à noter à propos de la présence de la valeur d'arrêt dans les données en entrée et en sortie :

- le premier itérateur de la paire sera un itérateur sur les données en entrée entre l'itérateur de début et l'itérateur de fin fournis comme entrée, inclusivement ;
- le premier itérateur de la paire sera soit l'itérateur de fin, soit pointera sur la première ; occurrence de la valeur d'arrêt dans la séquence d'entrée
- le second itérateur de la paire sera un itérateur sur la séquence en sortie ou après ;

- le nombre maximum de données copiées sera le nombre total de valeurs en entrée ;
- la valeur d'arrêt ne sera pas dans la chaîne de sortie.

Étape 3-e : Déterminer les concepts (les exigences sur les types d'arguments et le résultat)▲

Le langage C++ ne contient actuellement pas de concepts, mais ils arriveront, presque certainement. Même s'il n'y en a pas, les concepts sont une excellente façon de décrire.

copy_until() est une fonction simple, et donc les concepts requis sont évidents. Bien entendu, j'en ai déjà utilisé la plupart.

Le concept principal pour les itérateurs d'entrées est InputIterator - par construction (s'il existait un ForwardIterator ou mieux, il n'y aurait pas besoin d'écrire cette fonction ; il suffirait d'utiliser la fonction monoligne find/copy).

Le concept principal pour l'itérateur Output est… OutputIterator. Cela coule de source.

Copier la séquence d'entrée vers la séquence de sortie nécessite que le type de valeur de la séquence d'entrée soit transférable à la séquence de sortie (elle ne doit pas être nécessairement du même type - vous pouvez copier une suite d'entiers vers une suite de doubles par exemple).

Et finalement, la valeur d'arrêt doit être comparable (en égalité) avec le type de valeurs de la séquence d'entrée.

Les concepts n'ont pas encore été standardisés, et nous avons fait tellement d'itérations que la syntaxe courante est un peu floue et la suite est donc une réflexion brute :

Sélectionnez

```
template <
    InputIterator InIt,
    OutputIterator OutIt,
    typename T>
  requires
    Assignable <ValueType<OutIt>, ValueType<InIt>> &&
    EqualityComparable <ValueType <InIt>, T >
auto copy_until (
    InIt first,
    InIt last,
    OutIt out,
    T const& t) - >
  pair <InIt, OutIt> ;
```

La version « ensemble » est pratiquement identique, bien que je doive réfléchir aux noms du concept :

Sélectionnez

```
template < InputRange Rng, typename T >
  requires
    Assignable <ValueType <OutIt>, ValueType <Rng>> &&
    EqualityComparable <ValueType<Rng>, T>
auto copy_until (
    Rng const & rng,
    OutIt out,
    T const& t) - >
  pair <Iterator <Rng const&>, OutIt> ;
```

Comme tout ceci est très spéculatif, je ne vais pas répéter tous les concepts évoqués dans le reste de l'article. Cependant, si vous devez véritablement écrire une bibliothèque avec des algorithmes comme celui-ci, vous devriez jeter un coup d'œil à Boost.ConceptCheck.

Étape 4 : Écrire les tests▲

Arrivés ici, la plupart des programmeurs vont s'en aller écrire l'algorithme (en fait, la plupart des programmeurs l'auront fait après les étapes 1 ou 2). Vous pourriez faire ça, mais je vous recommande d'écrire les tests avant plutôt qu'après. Si vous n'agissez pas de cette façon, vous n'écrirez probablement jamais les tests.

Il n'existe pas de bibliothèques de tests standards, je vais donc utiliser Boost.Test. Il existe d'autres bonnes bibliothèques de tests - j'ai eu de bonnes expériences avec celle de Google - mais celle de Boost est très riche, facile d'utilisation et… c'est Boost. Je ne rentrerai pas dans les détails de l'écriture de tests avec Boost.Test dans cet article, je vais juste vous montrer quelques exemples.

En premier lieu, il y a le test évident… La fonction fait-elle ce qu'elle est supposée faire ? Vous devrez la tester avec les deux versions (itérateur et ensemble). Voici le test de la version itérateur :

Sélectionnez

```
BOOST_AUTO_TEST_CASE (copy_until_iterator)
{
    using std::begin ;
    using std::end ;

    auto const src = std::vector <int> {1, 2, 3, 4, 5};
    auto const expected = std::vector <int> {1, 2, 0, 0};
    auto dest = std::vector <int> {0, 0, 0, 0};
```

```
    auto const stop_value = 3 ;
    auto const stop_value_index =
      std::distance (begin (src),
        std::find (begin (src), end (src), stop_value));

    auto const result = indi::copy_until (
      begin (src), end (src), begin (dest), stop_value) ;

    // Vérification que la séquence en sortie est correcte
    BOOST_CHECK_EQUAL_COLLECTIONS (
      begin (expected), end (expected),
      begin (dest), end (dest));

    auto const in_it = std::get <0> (result) ;
    auto const out_it = std::get <1> (result) ;

    // Vérification du résultat sur l'itérateur d'entrée
    BOOST_CHECK_EQUAL (stop_value, *in_it) ;
    BOOST_CHECK_EQUAL (stop_value_index,
      std::distance (begin (src), in_it));

    // Vérification du résultat sur l'itérateur de sortie
    BOOST_CHECK_EQUAL (0, *out_it) ;
    BOOST_CHECK_EQUAL (stop_value_index,
      std::distance (begin (dest), out_it));
}
```

Ceci n'est pas destiné à être un tutoriel sur la façon d'écrire des tests, c'est seulement le code que je montre. Cependant, il y a d'autres choses que vous pourriez vouloir tester :

- une séquence d'entrée vide (rien ne doit être copié),
- une séquence d'entrée ne contenant pas la valeur d'arrêt (toute la suite doit être copiée),
- une séquence d'entrée dont le premier terme est la valeur d'arrêt (rien ne doit être copié),
- une séquence source d'itérateurs de type Input,
- une séquence de sortie d'itérateurs de type Output.

Cette liste n'est pas exhaustive, bien sûr - tout ce qui peut aller de travers doit être testé.

Naturellement, tous les tests vont rater… Ils ne vont même pas compiler. Ce n'est pas surprenant, l'algorithme n'existe pas encore. C'est l'étape suivante.

Étape 5 : Écrire l'algorithme▲

Il est enfin temps d'écrire la fonction.

Beaucoup de travail a déjà été fait. L'interface a déjà été définie, nous avons donc le squelette de la fonction, l'espace de nom, et le fichier qui va recevoir l'algorithme. Tout rassemblé, nous obtenons ceci :

Sélectionnez

```
// indi/copy_until.hpp

#ifndef INDI_INC_copy_until_
#define INDI_INC_copy_until_

#include <iterator>
#include <utility>

namespace indi {

namespace copy_until_detail {

using std::begin ;

template <typename Range>
auto adl_begin (Range & r) - > decltype (begin (r));
template <typename Range>
auto adl_begin (Range const & r) - > decltype (begin (r));

} // namespace copy_until_detail

template <
  typename InputIterator,
  typename OutputIterator,
  typename T >
auto copy_until (
    InputIterator first,
    InputIterator last,
    OutputIterator out,
    T const& stop) - >
  std::pair <InputIterator, OutputIterator>
{
  // à définir
}

template <
  typename Range,
  typename T>
auto copy_until (
    Range const& r,
    OutputIterator out,
    T const& stop) - >
  std::pair <
    decltype (copy_until_detail::adl_begin (r)),
    OutputIterator >
{
```

```
  // à définir
}

} // namespace indi

#endif // include guard
```

Il n'y a aucune raison de dupliquer le code, nous pouvons simplement écrire la version « ensemble » à l'aide de la version « itérateur » (ou vice versa).

Sélectionnez

```
template <
  typename Range,
  typename T>
inline auto copy_until (
    Range const& r,
    OutputIterator out,
    T const& stop)
{
    using std::begin ;
    using std::end ;

    return copy_until (begin (r), end (r), out, stop) ;
}
```

Et voilà la moitié du travail de fait !

Le corps de la fonction avec itérateurs n'est pas trop difficile à écrire à partir des exigences. Tant que l'itérateur Input courant n'est pas égal à l'itérateur de fin et qu'il ne pointe pas sur la valeur d'arrêt, la valeur qu'il pointe est copiée-assignée à l'itérateur Output, puis les itérateurs Input et Output avancent. L'itérateur Input et l'itérateur Output sont tous les deux renvoyés à la fin de la fonction :

Sélectionnez

```
template <
  typename InputIterator,
  typename OutputIterator,
  typename T>
auto copy_until (
    InputIterator first,
    InputIterator last,
    OutputIterator out,
    T const& stop)
{
    while (bool (first ! = last) && !bool (*first == stop))
        *out ++ = *first ++;

    return std::make_pair (first, out) ;
}
```

Et voilà ! Vous pouvez maintenant passer les tests que vous avez écrits précédemment sur ces fonctions. Si tous les tests passent, vous avez fini.

Étape facultative 6 : Optimiser▲

Ainsi écrit, l'algorithme est aussi simple qu'il peut l'être, et il ne peut réellement pas être plus simple s'il doit fonctionner avec des itérateurs Input. Pour utiliser des itérateurs Forward ou mieux, vous avez l'option d'utiliser find/copy :

Sélectionnez

```
auto const p = std::find (first, last, stop) ;
return std::make_pair (p, std::copy (first, p, out));
```

Il ne semble pas y avoir beaucoup à gagner, même si c'est possible. Cela demanderait de parcourir la suite de données d'entrée deux fois - une fois pour trouver la valeur d'arrêt, la deuxième fois pour copier - tandis que la solution écrite ci-dessus parcourt une seule fois la séquence, effectuant la comparaison et la copie en même temps.

Il existe, cependant, un cas où il peut être profitable d'effectuer la recherche et la copie séparément. Lorsque vous avez des éléments dans une séquence d'entrée contiguë - comme un vecteur ou un tableau - et que la séquence de sortie est également contiguë et que le passage du type d'entrée au type de sortie est trivial, vous pouvez copier les éléments avec un memcopy() ultra rapide de la séquence totale d'un seul coup. En premier lieu, vous effectuez la recherche pour obtenir la taille du bloc à copier - qui est situé dans une zone mémoire contiguë, et d'une opération en lecture seule uniquement, puis vous effectuez la copie rapide de bloc. Il y a de fortes chances que ce soit plus rapide qu'une comparaison élément par élément suivie d'une copie (vous devez tester bien entendu, cette optimisation peut n'être valable que sur certaines plateformes seulement).

En supposant que vous vouliez faire cette optimisation, vous devez trouver une façon de déterminer si vous pouvez utiliser la version optimisée ou bien si vous devez retomber sur la version générale. Vous pouvez faire cela au moment de la compilation.

Sélectionnez

```
namespace copy_until_detail {

struct general_version {};
struct optimized_version {};

template <
```

```
  typename InputIterator,
  typename OutputIterator,
  typename T >
auto copy_until (
    InputIterator first,
    InputIterator last,
    OutputIterator out,
    T const& stop,
    general_version) - >
  std::pair < InputIterator, OutputIterator >
{
    while (bool (first ! = last) && bool (*first ! = stop))
        *out ++ = *first ++;

    return std::make_pair (first, out) ;
}

template <
  typename InputIterator,
  typename OutputIterator,
  typename T>
auto copy_until (
    InputIterator first,
    InputIterator last,
    OutputIterator out,
    T const& stop,
    optimized_version) - >
  std::pair < InputIterator, OutputIterator >
{
    auto const p = std::find (first, last, stop) ;
    return std ::make_pair (p, std ::copy (first, p, out));
}

} // namespace copy_until_detail

template <
  typename InputIterator,
  typename OutputIterator,
  typename T>
inline auto copy_until (
    InputIterator first,
    InputIterator last,
    OutputIterator out,
    T const& stop) - >
  std::pair < InputIterator, OutputIterator >
{
    return copy_until_detail ::copy_until (
      first, last, out, stop,
      std::conditional_t <
      ???,
      optimized_version,
      general_version>{});
}
```

Qu'est-ce qui vient dans le « ??? » pour sélectionner la version optimisée ? Les prérequis sont listés ci-dessous :

- la séquence d'entrée doit être contiguë ;
- la séquence de sortie doit être contiguë ;
- le passage du type de valeur de la séquence d'entrée au type de valeur de sortie doit être trivial.

Vous pouvez ajouter d'autres choses à cette liste, en fonction des spécifications d'interface de copy_until(). Par exemple, je n'ai encore rien dit au sujet des exceptions, mais qu'est-ce qui nous garantit que, si une exception est levée lors de la lecture, la comparaison ou le cast d'un élément, les éléments précédents auront été correctement copiés dans la séquence de sortie ? Ce sera le cas dans la version générale écrite dans l'étape 5, mais non pour la version optimisée - en effet si la lecture ou la comparaison plantent pendant la recherche, rien du tout ne sera mis dans la séquence de sortie, car la copie n'aura pas débuté. Vous pourriez ajouter un test : si la lecture, la comparaison ou le cast lèvent une exception… Mais je recommanderais de ne pas avoir du tout cette exigence dans la spécification d'interface. Ajouter cette exigence dans la spécification d'interface rend la fonction plus prédictive dans le cas d'une exception, mais rend ses implémentation et optimisation plus difficiles. Je dois équilibrer entre spécifier suffisamment pour rendre l'algorithme aussi pratique que possible et ne pas surspécifier. Mon opinion est que : spécifier le comportement dans le cas d'exception(4) relève de la surspécification.

Tester que l'affectation est triviale est facile :

Sélectionnez

```
using src_type = typename
  std::iterator_traits <InputIterator>::value_type ;
using dest_type = typename
  std::iterator_traits <OutputIterator>::value_type ;

std::is_trivially_assignable <dest_type, src_type>::value
```

Malheureusement, le C++ ne comporte aucun test permettant de savoir si un itérateur se réfère à une suite de données contiguës(5). Le mieux que nous pouvons faire est de tester si l'itérateur est à accès aléatoire. Il est possible pour un itérateur d'être à accès aléatoire et non séquentiel - par exemple std::deque - mais cela est très rare - par exemple parmi les six itérateurs à accès aléatoire définis explicitement dans le standard, seul std::deque n'est pas séquentiel. Ce n'est pas un gros problème si un itérateur à accès aléatoire mais non séquentiel est utilisé avec la version optimisée - il peut y avoir un problème de performances, mais le comportement restera correct. Et de toute façon, de tels itérateurs sont assez rares. Tant qu'il n'existera pas de façon de tester si un

itérateur est séquentiel, tester seulement si l'itérateur est à accès aléatoire fonctionnera (mais là encore, vous devez effectuer le test sur votre plateforme pour voir cette optimisation est digne d'intérêt).

Ainsi, si nous voulons tester si l'itérateur est à accès aléatoire, nous ferons :

Sélectionnez

```
using src_it_cat = typename
  std::iterator_traits <InputIterator>::iterator_category ;

std::is_base_of <std::random_access_iterator_tag, src_it_cat>::value
```

Et, la même chose pour l'itérateur de sortie.

En rassemblant tout cela, la fonction d'aiguillage devient :

Sélectionnez

```
template <
  typename InputIterator,
  typename OutputIterator,
  typename T>
inline auto copy_until (
    InputIterator first,
    InputIterator last,
    OutputIterator out,
    T const& stop) - >
  std::pair < InputIterator, OutputIterator >
{
    using src_type = typename
      std::iterator_traits < InputIterator >::value_type ;
    using dest_type = typename
      std::iterator_traits < OutputIterator >::value_type ;
    using src_it_cat = typename
      std::iterator_traits < InputIterator >::iterator_category ;
    using dest_it_cat = typename
      std::iterator_traits < OutputIterator >::iterator_category ;

    return copy_until_detail::copy_until (
      first, last, out, stop,
      std::conditional_t <
        std::is_trivially_assignable < dest_type,
            src_type >::value &&
          std::is_base_of <
            std::random_access_iterator_tag,
            src_it_cat >::value &&
          std::is_base_of <
            std::random_access_iterator_tag,
            dest_it_cat >::value,
        optimized_version,
        general_version > {}) ;
```

II. Résumé▲

Le but de cet article n'était pas vraiment d'écrire la fonction spécifique copy_until(), mais plutôt d'étudier le processus de création d'un nouvel algorithme conforme au standard. Cela m'étonne de voir combien de programmeurs n'ont aucune idée à propos de la bibliothèque standard d'algorithmes ou seulement de vagues notions sur une poignée d'algorithmes les plus utilisés.

Je pense qu'écrire un algorithme conforme au standard devrait être une étape nécessaire dans tous les tutoriels basiques sur le C++ - c'est si simple et ça apprend tellement. Vous pouvez choisir d'effectuer cet exercice de façon superficielle ou bien creuser profondément - cet exercice est valable aussi bien pour les débutants que pour les programmeurs expérimentés en C ++.

Je ne plaide pas l'écriture de vos propres algorithmes au lieu d'utiliser ceux de la bibliothèque standard (ou ceux de Boost). Il n'y a pas de mal à essayer d'en réimplémenter un ou deux pour s'entraîner, mais la réalité est qu'il a tellement d'algorithmes potentiellement utiles n'existant pas dans la bibliothèque standard (ou chez Boost) que cela n'est pas nécessaire. Lorsque vous programmez normalement, il y a des chances que vous rencontriez une situation dans laquelle vous avez besoin de quelque chose potentiellement réutilisable et qui n'existe pas dans la bibliothèque standard - une parfaite opportunité de mettre la théorie en pratique.

Gardez aussi à l'esprit que j'ai simplement esquissé comment vous pouviez creuser le sujet avec une simple fonction copy_until(). J'ai abordé les deux versions « itérateur » et « ensemble », mais vous pouvez avoir également des versions qui prennent un autre argument - une fonction objet qui effectue la comparaison autrement qu'avec '==' (et vous pouvez avoir les versions basiques s'y reportant en utilisant std::equal_to).

Vous avez ici les étapes basiques, que j'ai soulignées, à appliquer lorsque vous écrirez votre propre algorithme « standard » :

1. Spécifiez clairement le problème : une compréhension vague du problème vous garantit pratiquement une solution merdique. Prenez le temps de comprendre clairement ce qui est nécessaire.
2. Essayez d'en faire le moins possible. Recherchez les solutions préexistantes plutôt que d'essayer de réinventer la roue.
3. Définissez l'interface :
    a. Déterminez le nom de la fonction : ainsi que l'espace de nom et le nom du fichier d'en-têtes, etc.,

   b. Déterminez les arguments : considérez ce qui est nécessaire pour que la fonction fasse ce qu'elle doit faire, ainsi que l'ordre et le type des arguments,
   c. Déterminez le type de retour : souvent le type de retour est prédéterminé, mais quand ça ne l'est pas, prenez le temps de considérer de quelles informations utiles les appelants ont besoin de connaître sur ce que la fonction fait,
   d. Déterminez les contrats d'interface (les exigences sur les valeurs des arguments et du résultat) : spécifiez les conditions initiales et les conditions finales au minimum dans la documentation,
   e. Déterminez les concepts (les exigences sur les types d'arguments et de résultat) : les concepts ne sont pas complètement standardisés, mais essayez de spécifier les concepts utilisés par votre algorithme en des termes qui vous permettront de profiter des concepts lorsqu'ils arriveront (ou encore, pensez à Boost. ConceptCheck).
4. Écrivez les tests : testez les opérations normales puis tous les cas aux limites en essayant de penser à tout ce qui peut aller de travers.
5. Écrivez l'algorithme.
6. (Optionnel) Optimisez. Ne pas oublier de profiler !

Si vous êtes tenté de vous essayer à l'écriture d'un ou deux algorithmes pour

augmenter votre aisance en C++ et que vous ne savez pas quoi faire, voici quelques suggestions que vous pouvez considérer, et souvenez-vous, vous pouvez creuser les sujets aussi loin qu'il vous plaira :

- Pour les débutants, essayez un algorithme qui renvoie vrai si une suite de données avec itérateurs bidirectionnels est un palindrome. Ceci peut être délicat ; je l'ai utilisé dans un concours il y a plusieurs années de cela, et pratiquement personne n'y est arrivé - y compris certains qui pensaient être très bons en C++ (bien que, honnêtement, ça en dise probablement plus sur le triste état de l'expertise en C++ que sur la difficulté du défi). Essayez d'effectuer le minimum absolu d'itérations et de comparaison.

- Maintenant, un exercice légèrement plus délicat, mais toujours bon pour les débutants. La bibliothèque standard comprend min_element(), max_element() et minmax_element() - qui renvoient des itérateurs pointant vers le plus petit ou le plus grand (ou les deux) des éléments d'une suite. Essayez d'écrire un algorithme renvoyant une paire contenant la plus petite (ou plus grande, ou les deux) valeur (pas un itérateur, mais bien une valeur) d'une suite et le nombre de fois où elle apparaît, en effectuant une seule passe à travers la suite de façon à pouvoir utiliser un itérateur Input.
- Celui-ci est spécifique aux suites de caractères, mais je le trouve assez utile. Le but est de prendre n'importe quelle suite de caractères - y compris les caractères Unicode et pas seulement les char - et de les convertir en limace. Ce qui implique deux choses : convertir toutes les majuscules en minuscules et remplacer tout ce qui n'est pas alphanumérique par un tiret. La limace ne peut pas commencer par un tiret, se terminer par un tiret, ni comprendre deux tirets consécutifs. Vous devrez prendre une paire d'itérateurs Input sur la séquence en entrée, un itérateur sur la séquence de sortie, et un std::locale (que vous pourrez utiliser avec des choses comme std::tolower() et std::isalnum(). Avec l'itérateur Output, vous devrez renvoyer également un booléen à faux si la sortie débute par un chiffre ou si la sortie est vide. Ce sujet n'est pas facile, mais pas non plus du niveau expert.

- En voici un véritablement difficile. À partir d'une suite définie par des itérateurs Forward, créez une suite faite de paires, où chaque paire contient une des valeurs de la suite et le nombre de fois où elle apparaît, triée dans l'ordre des valeurs. En d'autres termes, vous générez un histogramme des données de la séquence d'entrée. Par exemple, la séquence {7, 7, 7, 9, 3, 1, 1} doit produire la séquence {(1, 2), (3, 1), (7, 3), (9, 1)} en sortie. Cela demande plusieurs passes à travers la suite entrée pour éviter tout besoin d'allocation dynamique, mais essayez de le faire de la façon la plus efficace possible.

Si vous vous essayez à l'un d'entre eux, n'hésitez pas à me le faire savoir en commentaire (ne vous embêtez pas à essayer de coller le code dans le commentaire - utilisez un espace de stockage et collez le lien dans le commentaire).

Bon codage !

III. Remerciements▲

Cet article est une traduction autorisée de l'article de paru sur Explicit C++.

Merci aussi à Ced pour sa relecture orthographique.

---

(1)
Boost.Algorithm possède un algorithme appelé copy_until(), mais, de mon point de vue, ils ont commis une erreur. Leur algorithme est plutôt ce que j'aurais appelé copy_while_not() ou copy_while() utilisé avec un prédicat inversé. Ils n'ont rien d'équivalent à mon copy_until() (bien que vous puissiez l'émuler à l'aide de copy_while() et un lambda).
(2)
Un exemple récent de leur orientation se trouve dans la façon dont ils nomment les modifications naturelles de type comme make_unsigned, ils ont tardivement réalisé qu'ils pouvaient utiliser les alias template pour les rendre plus simples d'utilisation.

Ainsi, nous devons utiliser make_unsigned_t<T> ou typename make_unsigned<T>::type à la place d'un tout simple make_unsigned<T>.
(3)
Une petite optimisation de la chaîne pourrait certainement s'appliquer dans cet exemple particulier, mais ignorons cela au bénéfice de l'argument, en général vous aurez besoin d'une construction coûteuse.
(4)
J'entends par là spécifiant un comportement spécifique, pas simplement faisant une garantie d'exception comme une exception forte ou une exception basique - et dans ce cas une exception basique seulement semble avoir du sens, car vous ne pouvez pas effacer des choses que vous avez déjà écrites dans la séquence de sortie lorsqu'une exception se produit durant la copie.
(5)
Bien que, cela ait été proposé.

Meilleurs langages en 2019 selon l'IEEE : Python leader pour la troisième année consécutive, il s'impose dans tous les domaines dans lesquels il est utilisé, du développement web à l'embarqué

Microsoft amorce le support des Concepts apportés par C++ 20 dans Visual Studio 2019 version 16.3 Preview 2, au niveau du compilateur mais aussi de la bibliothèque

Est-ce une grosse erreur de considérer la POO comme standard de l'industrie pour l'organisation des bases de code ?

Parallélisme avec Python : comment diriger la vectorisation avec NUMEXPR ? Intel Parallel Universe

Contacter le responsable de la rubrique C++

Nous contacter Participez Hébergement Informations légales Partenaire : Hébergement Web