ContainerFacade

Naomasa Matsubayashi

2011年11月5日土曜日

自己紹介

2011年11月5日土曜日

はじめまして、おひさしぶりです。松林です。★Twitterでは@fadis_とかいう名前です。

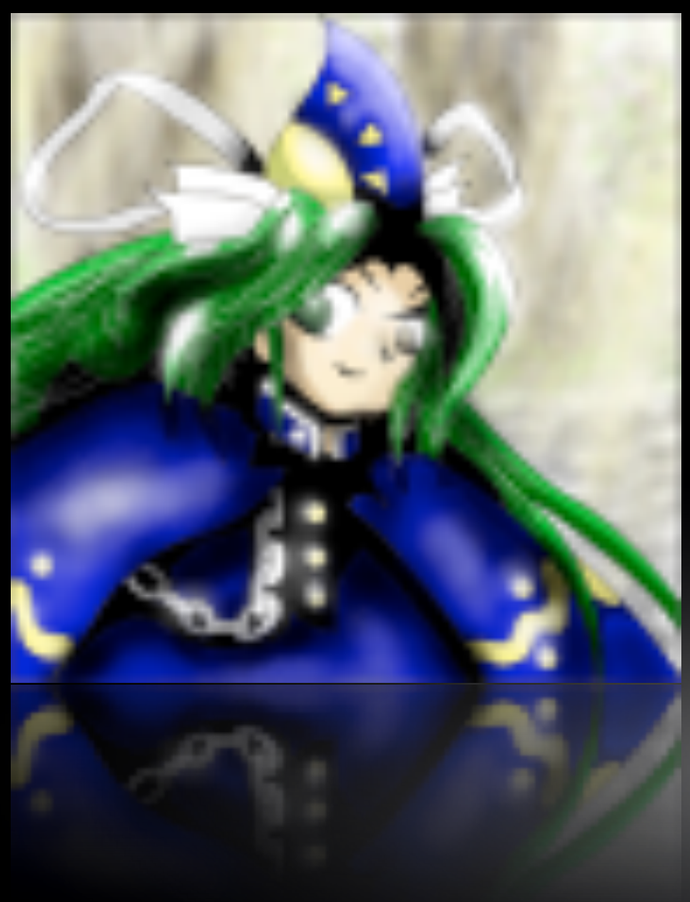
松林 尚理

自己紹介

2011年11月5日土曜日

はじめまして、おひさしぶりです。松林です。★Twitterでは@fadis_とかいう名前です。

松林 尚理



Twitter

@fadis_

自己紹介

2011年11月5日土曜日

はじめまして、おひさしぶりです。松林です。★Twitterでは@fadis_とかいう名前です。

松林 尚理
githubはじめました

https://github.com/Fadis/

自己紹介

- githubやってます、★今回紹介するコードもこちらで公開しています。

松林 尚理
githubはじめました
https://github.com/Fadis/
今回紹介するコードも
ここで公開中
自己紹介

githubやってます、★今回紹介するコードもこちらで公開しています。

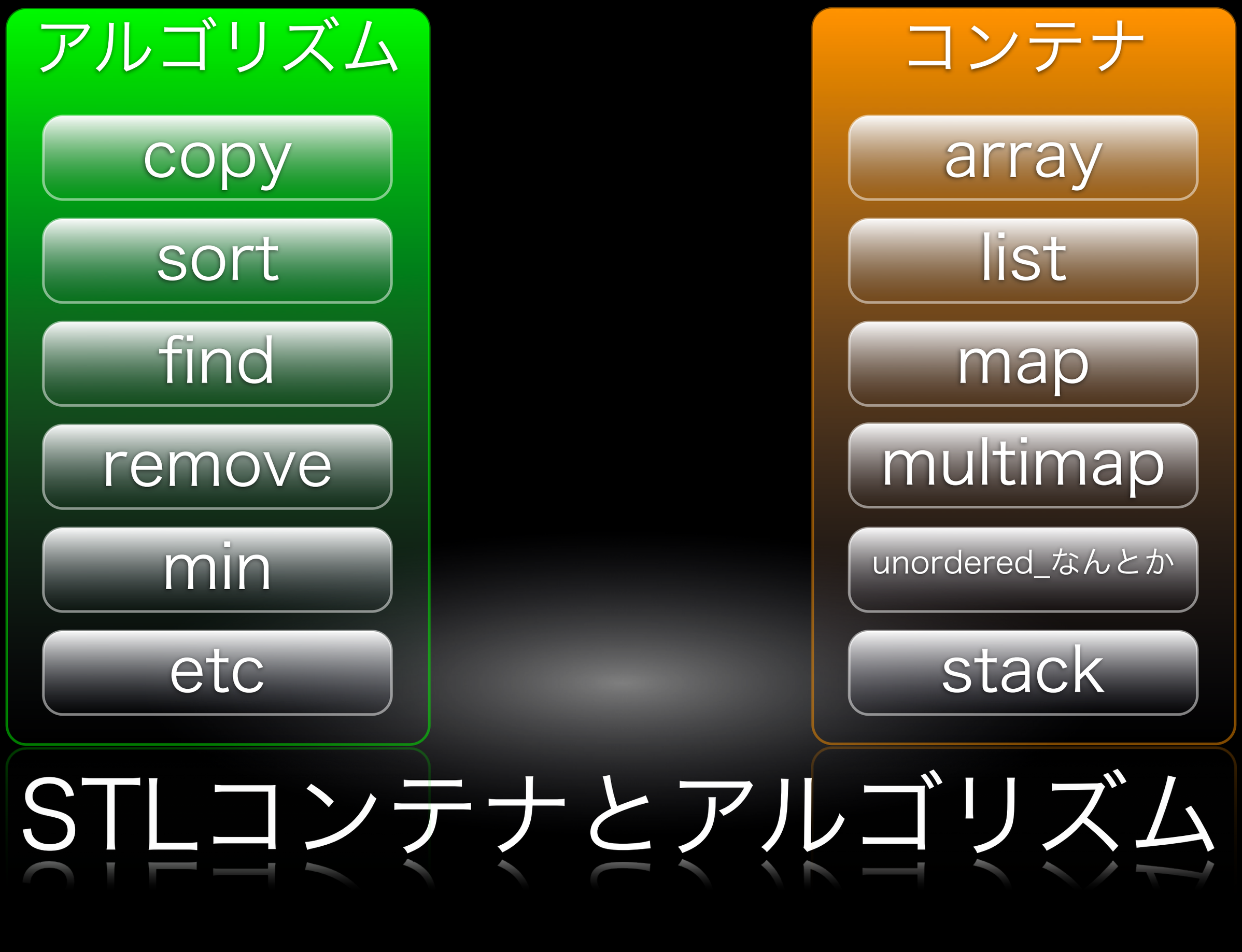
STLコンテナとアルゴリズム

2011年11月5日土曜日

今回はSTLコンテナネタなので、はじめにSTLのコンテナとアルゴリズムの関係をおさらいしておきましょう。★STLには沢山のデータを保持しておくためのフォーマットとして、様々なデータ構造を用いたコンテナと、★沢山のデータに対して操作を行う為のアルゴリズムが用意されています。★しかし、コンテナはそれぞれインターフェースが異なっているため、アルゴリズムの引数としてコンテナを直接渡すようにしてしまうと、同じアルゴリズムをコンテナの数だけ用意しなければならなくなってしまいます。

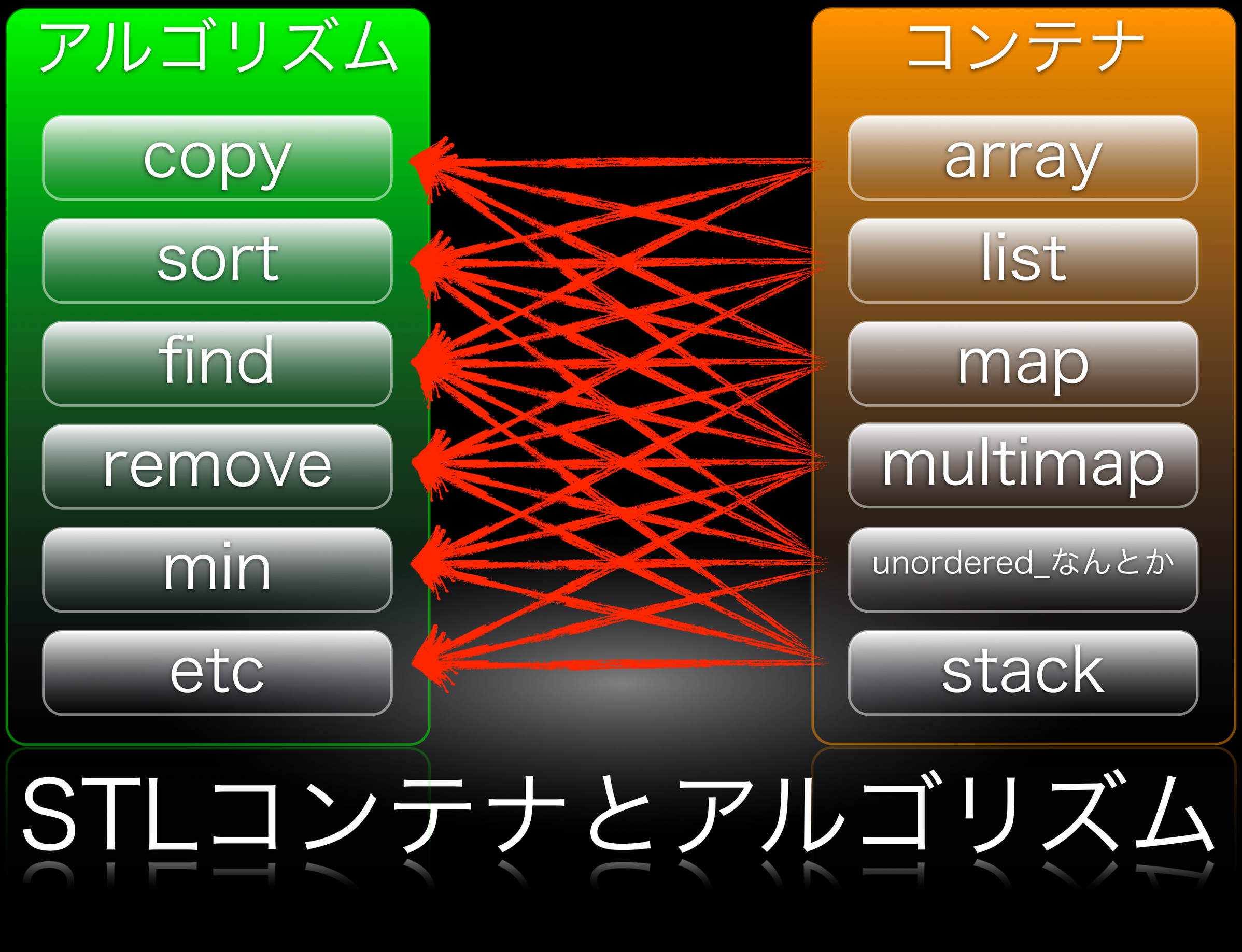
2011年11月5日土曜日

今回はSTLコンテナネタなので、はじめにSTLのコンテナとアルゴリズムの関係をおさらいしておきましょう。★STLには沢山のデータを保持しておくためのフォーマットとして、様々なデータ構造を用いたコンテナと、★沢山のデータに対して操作を行う為のアルゴリズムが用意されています。★しかし、コンテナはそれぞれインターフェースが異なっているため、アルゴリズムの引数としてコンテナを直接渡すようにしてしまうと、同じアルゴリズムをコンテナの数だけ用意しなければならなくなってしまいます。



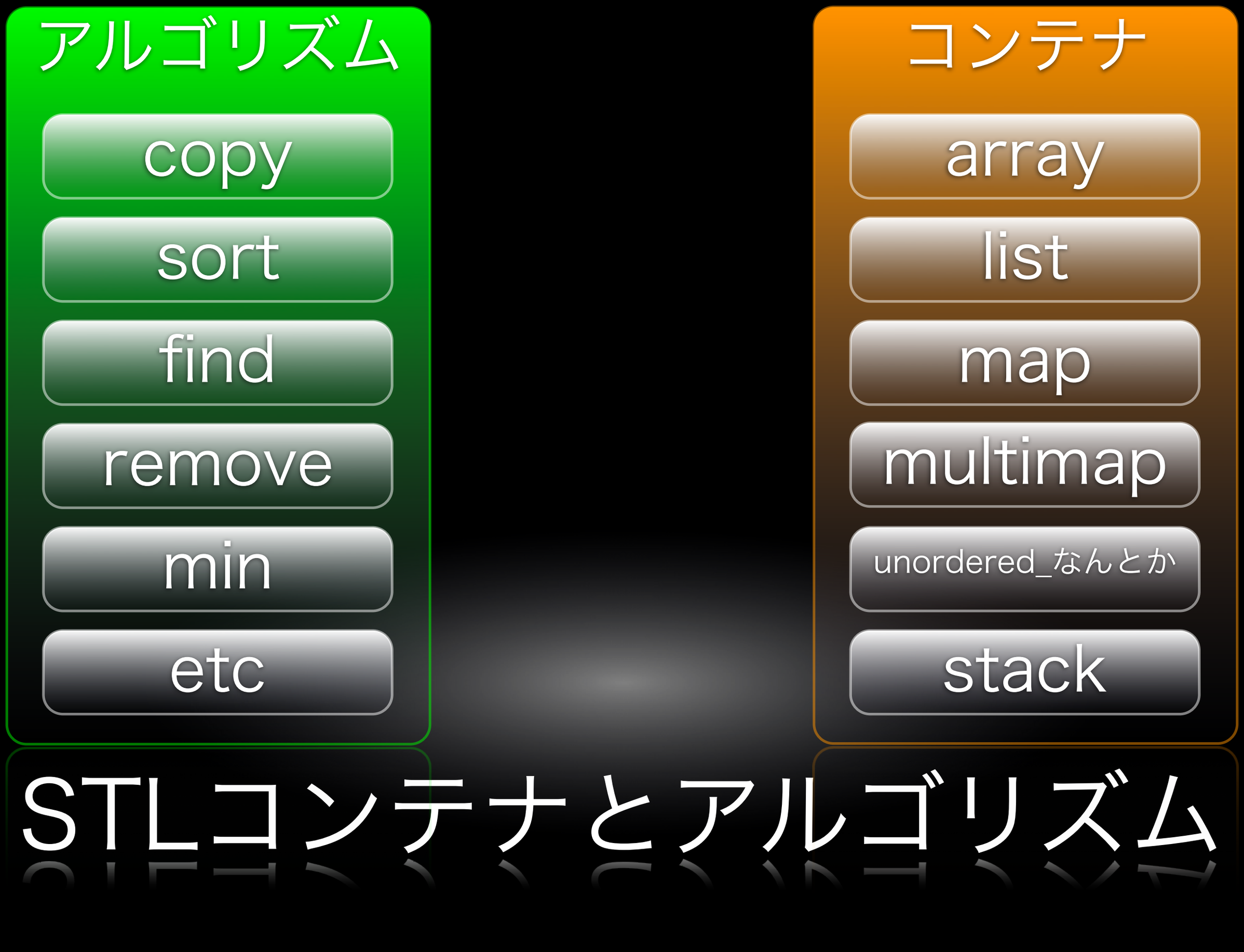
2011年11月5日土曜日

今回はSTLコンテナネタなので、はじめにSTLのコンテナとアルゴリズムの関係をおさらいしておきましょう。★STLには沢山のデータを保持しておくためのフォーマットとして、様々なデータ構造を用いたコンテナと、★沢山のデータに対して操作を行う為のアルゴリズムが用意されています。★しかし、コンテナはそれぞれインターフェースが異なっているため、アルゴリズムの引数としてコンテナを直接渡すようにしてしまうと、同じアルゴリズムをコンテナの数だけ用意しなければならなくなってしまいます。



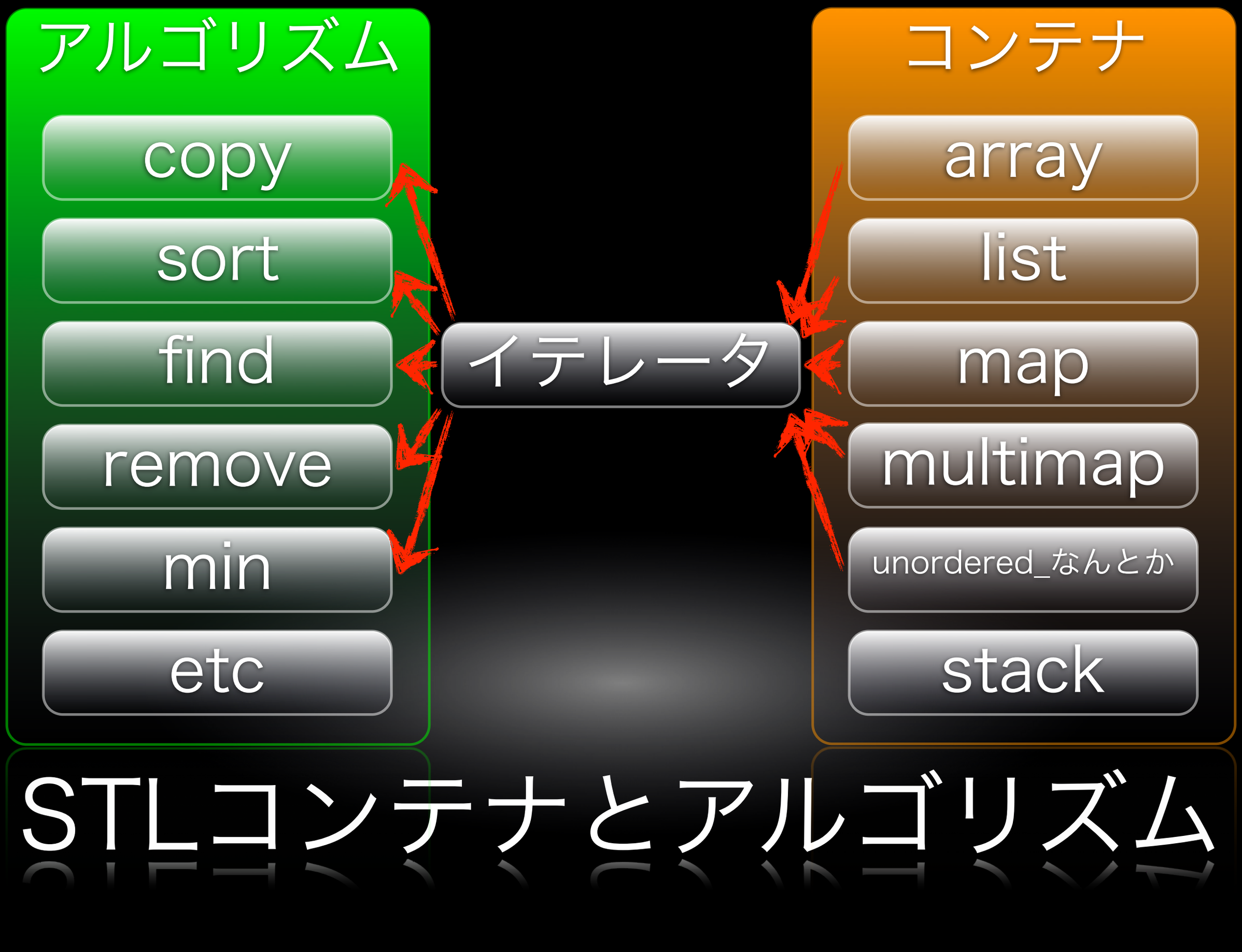
2011年11月5日土曜日

今回はSTLコンテナネタなので、はじめにSTLのコンテナとアルゴリズムの関係をおさらいしておきましょう。★STLには沢山のデータを保持しておくためのフォーマットとして、様々なデータ構造を用いたコンテナと、★沢山のデータに対して操作を行う為のアルゴリズムが用意されています。★しかし、コンテナはそれぞれインターフェースが異なっているため、アルゴリズムの引数としてコンテナを直接渡すようにしてしまうと、同じアルゴリズムをコンテナの数だけ用意しなければならなくなってしまいます。



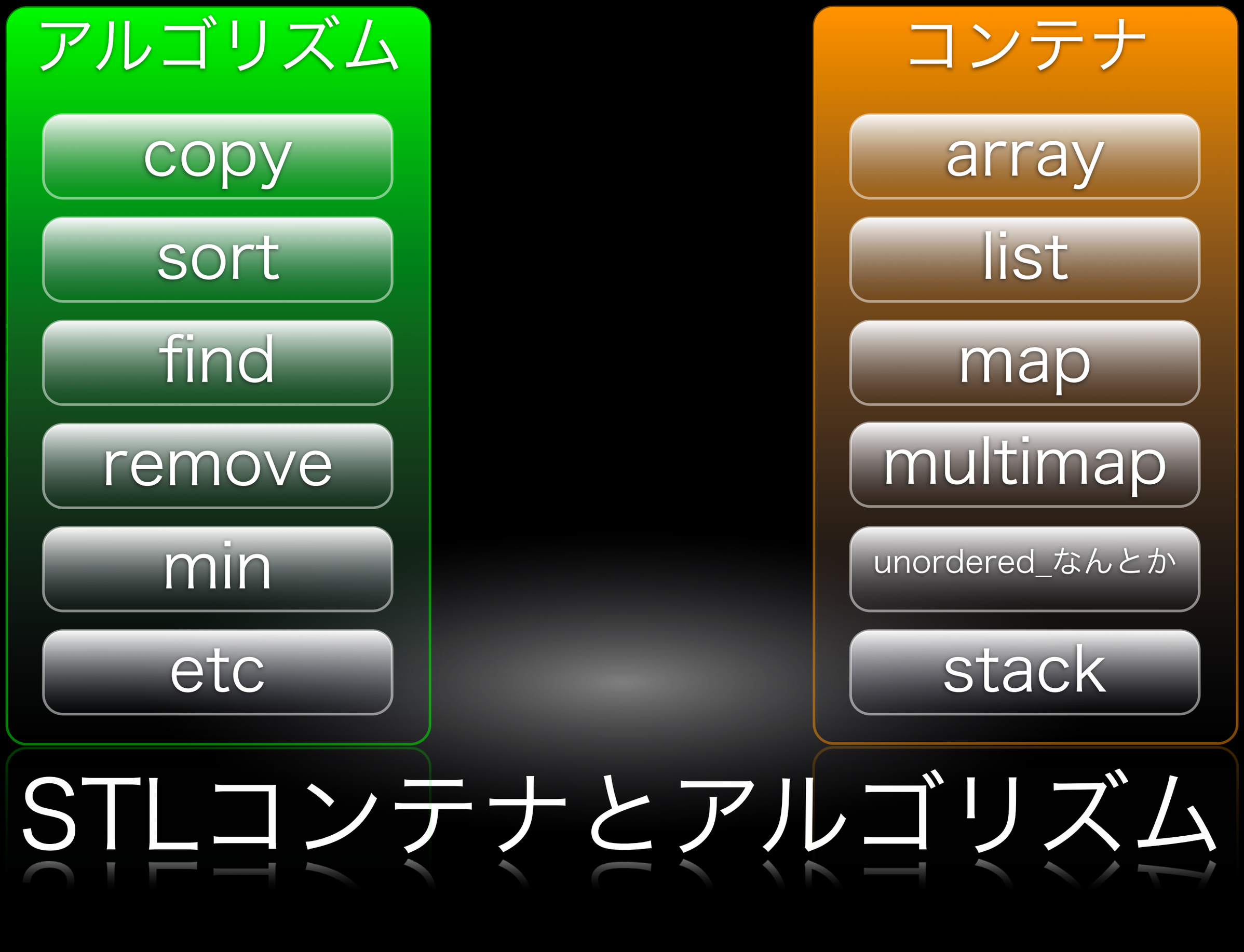
2011年11月5日土曜日

そこで、STLでは、アルゴリズムに直接コンテナを渡すのではなく、★間に共通したインターフェースを持つイテレータを挟む事で、コンテナ毎にアルゴリズムを用意しなければならない、という事態を回避しています。



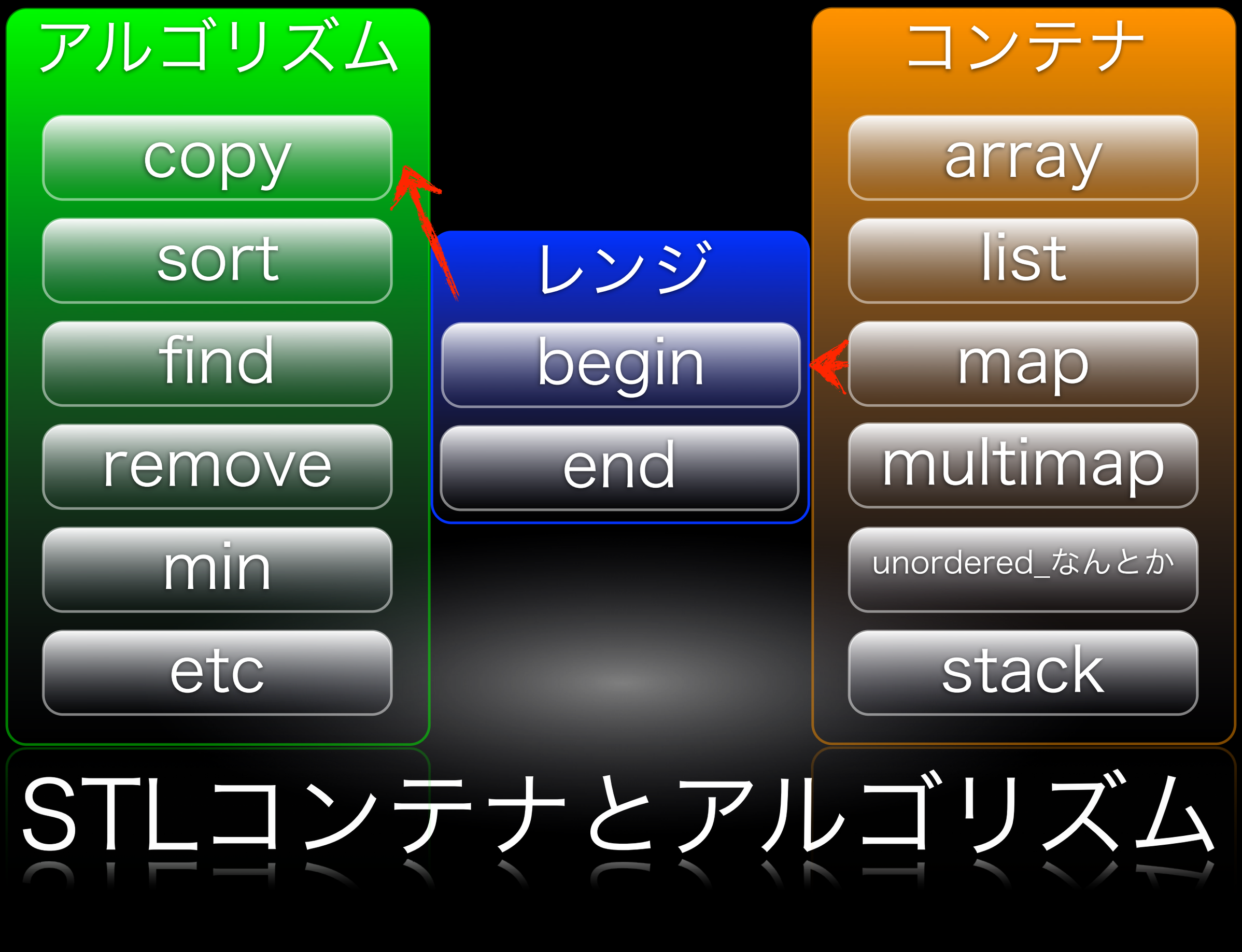
2011年11月5日土曜日

そこで、STLでは、アルゴリズムに直接コンテナを渡すのではなく、★間に共通したインターフェースを持つイテレータを挟む事で、コンテナ毎にアルゴリズムを用意しなければならない、という事態を回避しています。



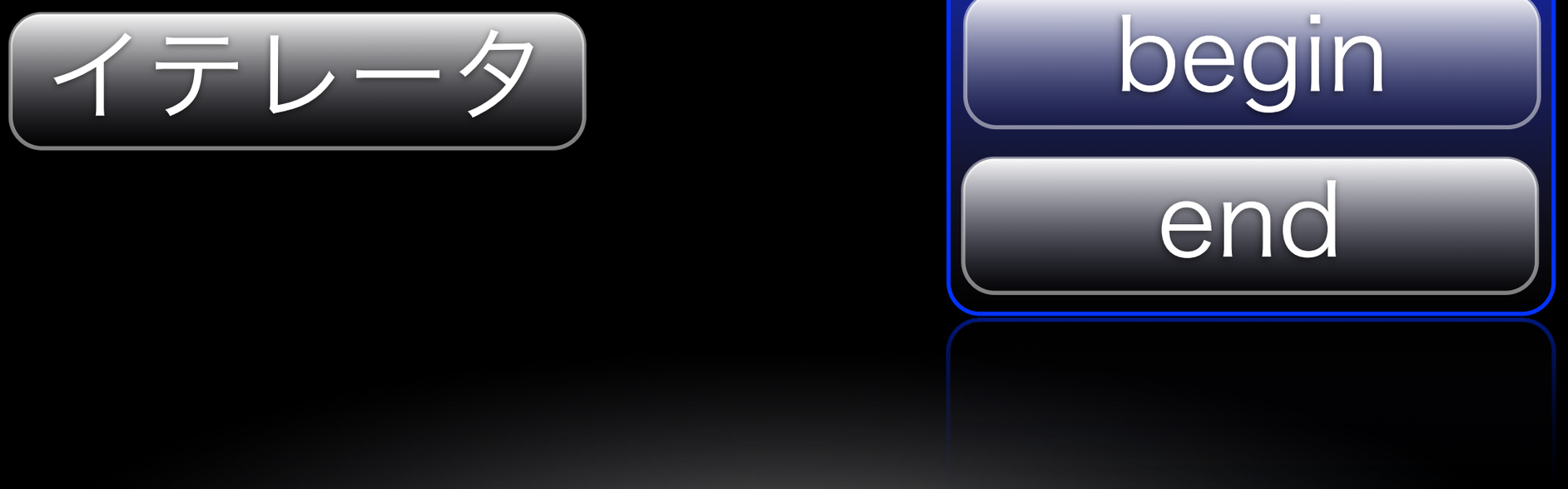
2011年11月5日土曜日

イテレータはポインタを抽象化したものなので、コンテナの中のある一点を指しています。多くのアルゴリズムは、アルゴリズムを適用する範囲の始点と終点という2点を要求するため、必然的にイテレータを2つ組で扱う状況が増えます。最近ではこの始点と終点のペアを抽象化した★レンジに対してアルゴリズムを用意しようという試みも活発に行われているようです。



2011年11月5日土曜日

イテレータはポインタを抽象化したものなので、コンテナの中のある一点を指しています。多くのアルゴリズムは、アルゴリズムを適用する範囲の始点と終点という2点を要求するため、必然的にイテレータを2つ組で扱う状況が増えます。最近ではこの始点と終点のペアを抽象化した★レンジに対してアルゴリズムを用意しようという試みも活発に行われているようです。



STLコンテナとアルゴリズム

2011年11月5日土曜日

イテレータやレンジによってコンテナに依存しないアルゴリズムが書けるようになりました。しかしイテレータはイテレータを通じてコンテナに既にある要素にアクセスすることが出来るだけなので、イテレータを受け取ったアルゴリズムはコンテナに対して要素の挿入削除を行うことが出来ません。この点レンジも同じで、イテレータペアをレンジと見なす以上、レンジに対してsizeを実装することは出来ても、resizeを実装することは出来ません。

イテレータを使わずに、
実装すべき関数の数が膨大にならないような
アルゴリズムを実現できないだろうか

そこで要素の挿入削除を伴うアルゴリズムのために、イテレータではなく直接コンテナを受け取り、
しかし実装すべき関数の数が膨大にならないようにする方法はないだろうかと考えました。

コンテナの種類

2011年11月5日土曜日

まず、すぐに思いつくのがコンテナのカテゴリ毎にアルゴリズムを実装するというものです。STLスタイルのコンテナは個々に無秩序なメンバ関数を備えているわけではなく、いくつかのカテゴリに分かれており、同じカテゴリに属すコンテナは似たようなメンバ関数を備えています。例えば★vectorとlistは同じような振る舞いをするメンバ関数を備えていますし、★mapとmultimapも似たようなメンバ関数を備えています。これらのカテゴリには標準できちんと名前が与えられており、vectorやlistのような要素が順番に並ぶコンテナを★シーケンスコンテナ、mapやmultimapのような要素がキーに応じてソートされて保持されるコンテナを★アソシエイティブコンテナ、C++11で標準に追加されたunordered_mapのような要素がキーに応じて保持されるけどソートはされないコンテナを★アンオーダードアソシエイティブコンテナ、そしてstackやqueueのように他のコンテナを土台として操作を制限するものを★コンテナアダプタと呼びます。

コンテナの種類

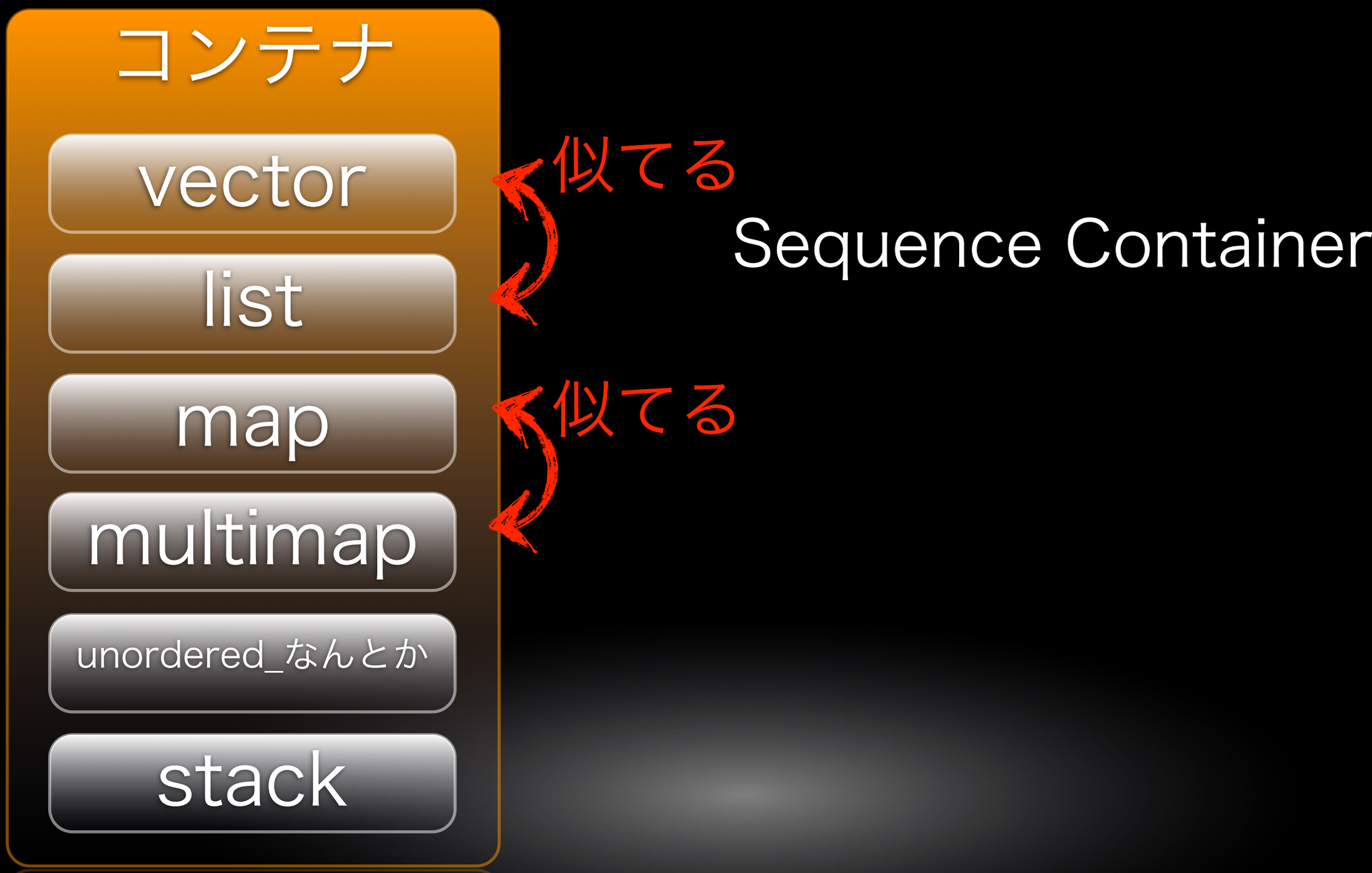
2011年11月5日土曜日

まず、すぐに思いつくのがコンテナのカテゴリ毎にアルゴリズムを実装するというものです。STLスタイルのコンテナは個々に無秩序なメンバ関数を備えているわけではなく、いくつかのカテゴリに分かれており、同じカテゴリに属すコンテナは似たようなメンバ関数を備えています。例えば★vectorとlistは同じような振る舞いをするメンバ関数を備えていますし、★mapとmultimapも似たようなメンバ関数を備えています。これらのカテゴリには標準できちんと名前が与えられており、vectorやlistのような要素が順番に並ぶコンテナを★シーケンスコンテナ、mapやmultimapのような要素がキーに応じてソートされて保持されるコンテナを★アソシエイティブコンテナ、C++11で標準に追加されたunordered_mapのような要素がキーに応じて保持されるけどソートはされないコンテナを★アンオーダードアソシエイティブコンテナ、そしてstackやqueueのように他のコンテナを土台として操作を制限するものを★コンテナアダプタと呼びます。

コンテナの種類

2011年11月5日土曜日

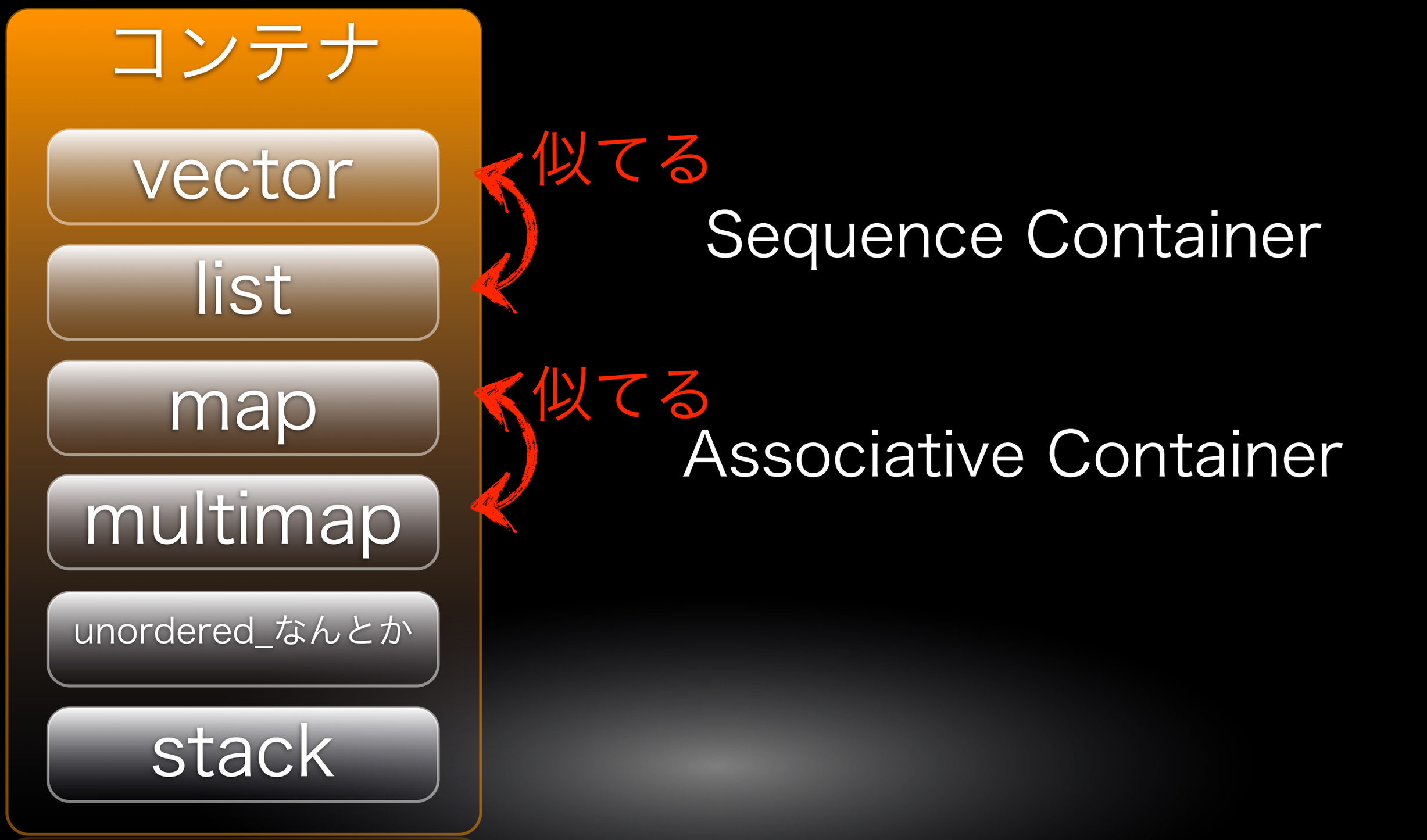
まず、すぐに思いつくのがコンテナのカテゴリ毎にアルゴリズムを実装するというものです。STLスタイルのコンテナは個々に無秩序なメンバ関数を備えているわけではなく、いくつかのカテゴリに分かれており、同じカテゴリに属すコンテナは似たようなメンバ関数を備えています。例えば★vectorとlistは同じような振る舞いをするメンバ関数を備えていますし、★mapとmultimapも似たようなメンバ関数を備えています。これらのカテゴリには標準できちんと名前が与えられており、vectorやlistのような要素が順番に並ぶコンテナを★シーケンスコンテナ、mapやmultimapのような要素がキーに応じてソートされて保持されるコンテナを★アソシエイティブコンテナ、C++11で標準に追加されたunordered_mapのような要素がキーに応じて保持されるけどソートはされないコンテナを★アンオーダードアソシエイティブコンテナ、そしてstackやqueueのように他のコンテナを土台として操作を制限するものを★コンテナアダプタと呼びます。



コンテナの種類

2011年11月5日土曜日

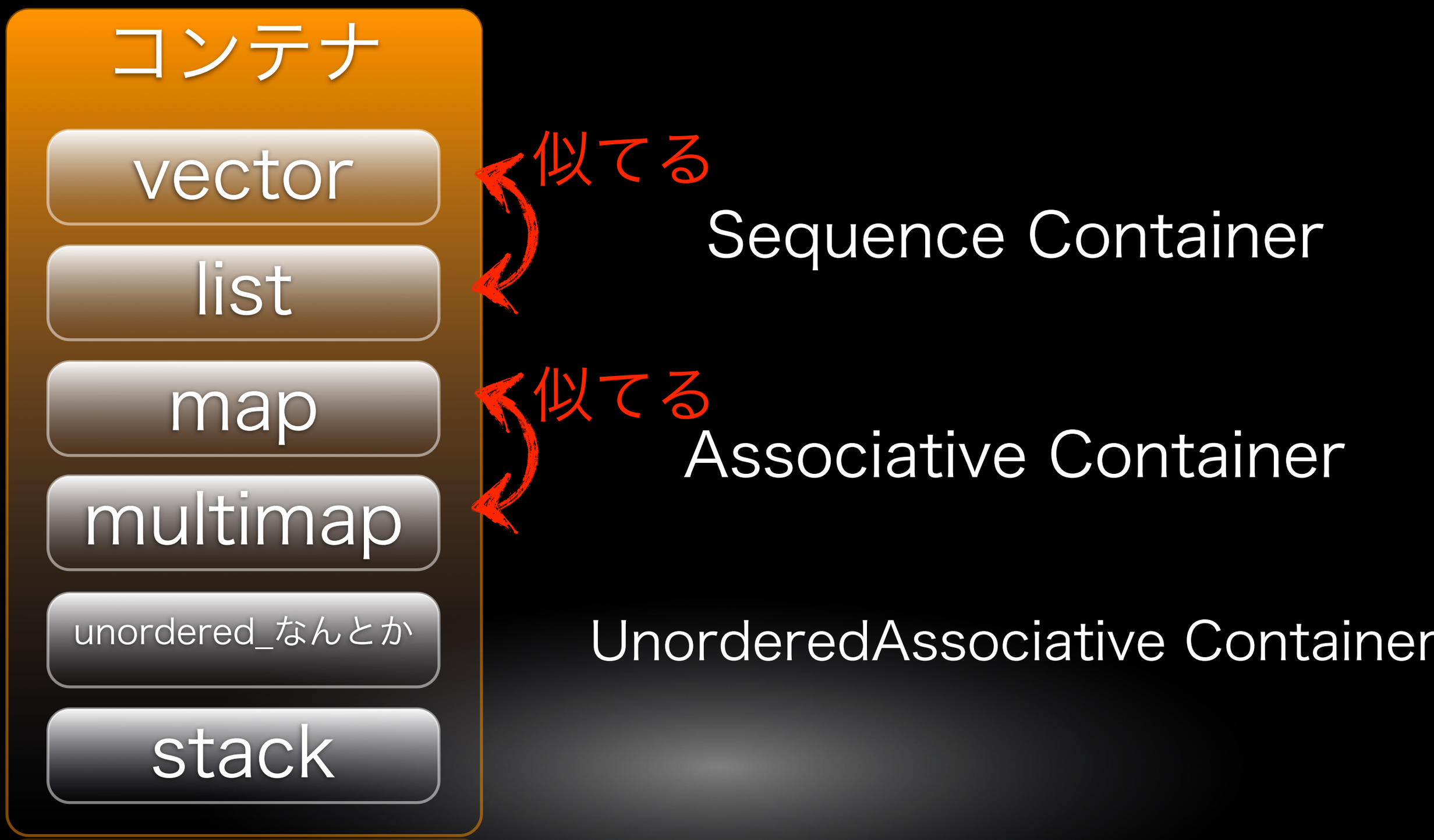
まず、すぐに思いつくのがコンテナのカテゴリ毎にアルゴリズムを実装するというものです。STLスタイルのコンテナは個々に無秩序なメンバ関数を備えているわけではなく、いくつかのカテゴリに分かれており、同じカテゴリに属すコンテナは似たようなメンバ関数を備えています。例えば★vectorとlistは同じような振る舞いをするメンバ関数を備えていますし、★mapとmultimapも似たようなメンバ関数を備えています。これらのカテゴリには標準できちんと名前が与えられており、vectorやlistのような要素が順番に並ぶコンテナを★シーケンスコンテナ、mapやmultimapのような要素がキーに応じてソートされて保持されるコンテナを★アソシエイティブコンテナ、C++11で標準に追加されたunordered_mapのような要素がキーに応じて保持されるけどソートはされないコンテナを★アンオーダードアソシエイティブコンテナ、そしてstackやqueueのように他のコンテナを土台として操作を制限するものを★コンテナアダプタと呼びます。



コンテナの種類

2011年11月5日土曜日

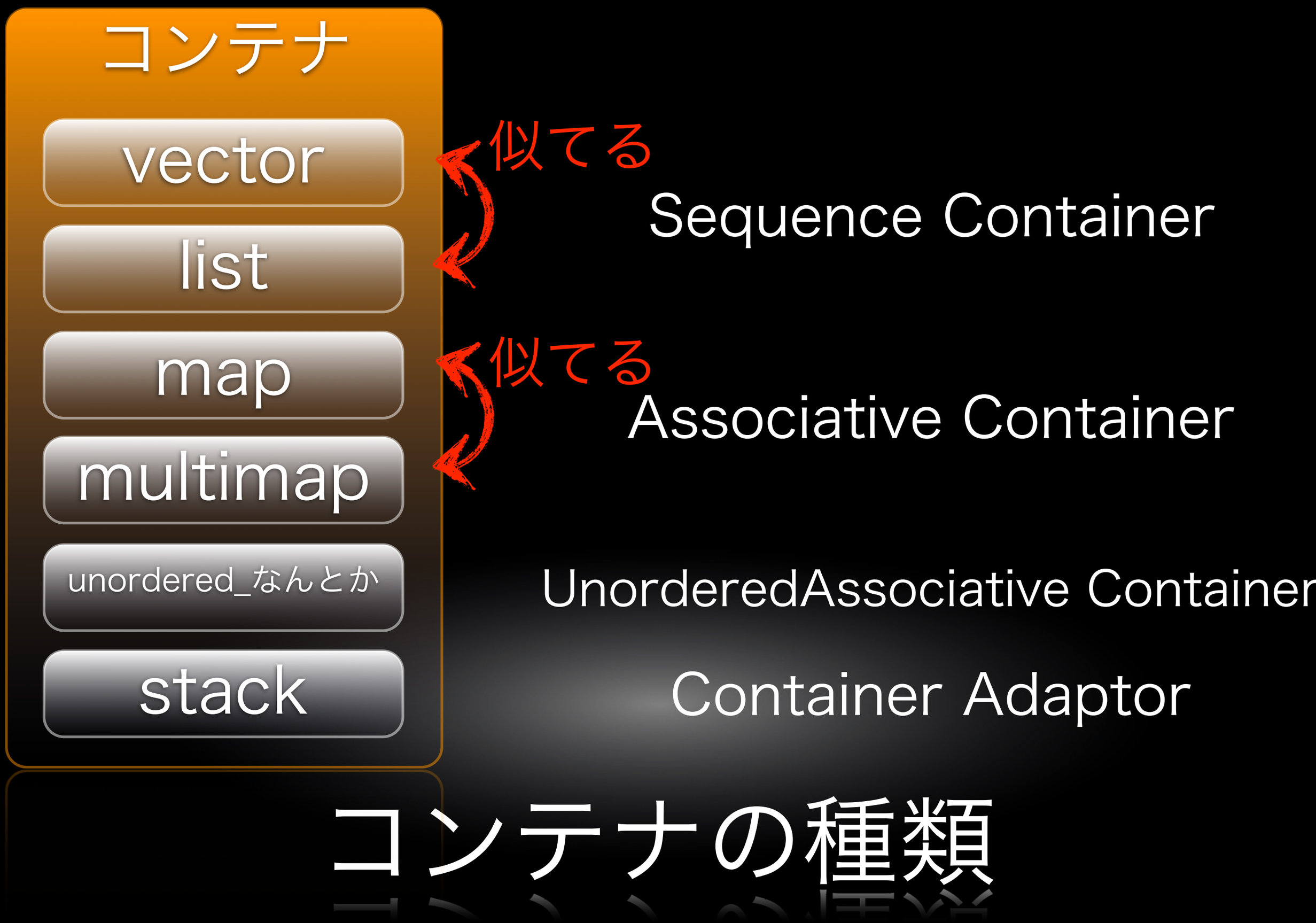
まず、すぐに思いつくのがコンテナのカテゴリ毎にアルゴリズムを実装するというものです。STLスタイルのコンテナは個々に無秩序なメンバ関数を備えているわけではなく、いくつかのカテゴリに分かれており、同じカテゴリに属すコンテナは似たようなメンバ関数を備えています。例えば★vectorとlistは同じような振る舞いをするメンバ関数を備えていますし、★mapとmultimapも似たようなメンバ関数を備えています。これらのカテゴリには標準できちんと名前が与えられており、vectorやlistのような要素が順番に並ぶコンテナを★シーケンスコンテナ、mapやmultimapのような要素がキーに応じてソートされて保持されるコンテナを★アソシエイティブコンテナ、C++11で標準に追加されたunordered_mapのような要素がキーに応じて保持されるけどソートはされないコンテナを★アンオーダードアソシエイティブコンテナ、そしてstackやqueueのように他のコンテナを土台として操作を制限するものを★コンテナアダプタと呼びます。



コンテナの種類

2011年11月5日土曜日

まず、すぐに思いつくのがコンテナのカテゴリ毎にアルゴリズムを実装するというものです。STLスタイルのコンテナは個々に無秩序なメンバ関数を備えているわけではなく、いくつかのカテゴリに分かれており、同じカテゴリに属すコンテナは似たようなメンバ関数を備えています。例えば★vectorとlistは同じような振る舞いをするメンバ関数を備えていますし、★mapとmultimapも似たようなメンバ関数を備えています。これらのカテゴリには標準できちんと名前が与えられており、vectorやlistのような要素が順番に並ぶコンテナを★シーケンスコンテナ、mapやmultimapのような要素がキーに応じてソートされて保持されるコンテナを★アソシエイティブコンテナ、C++11で標準に追加されたunordered_mapのような要素がキーに応じて保持されるけどソートはされないコンテナを★アンオーダードアソシエイティブコンテナ、そしてstackやqueueのように他のコンテナを土台として操作を制限するものを★コンテナアダプタと呼びます。

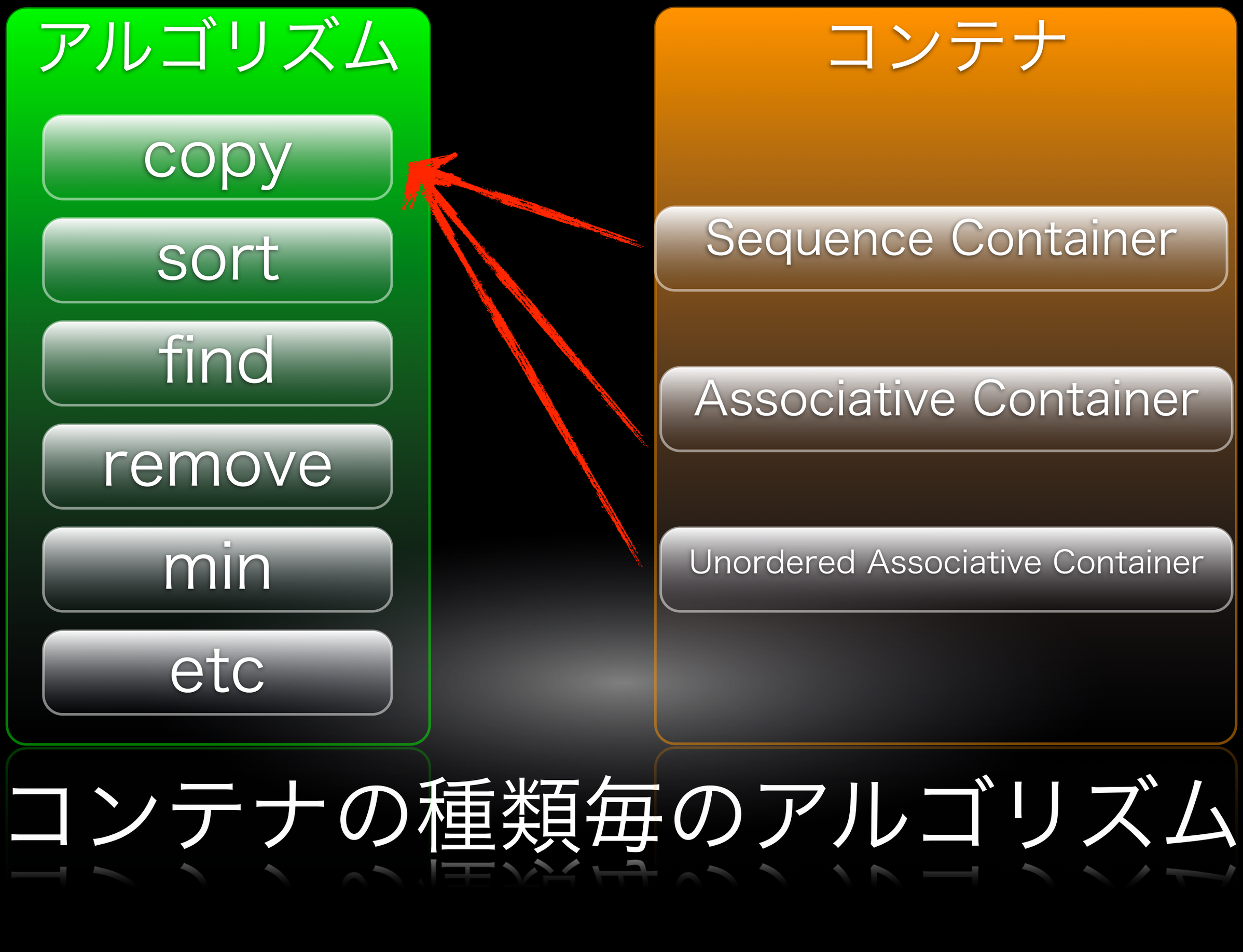


2011年11月5日土曜日

まず、すぐに思いつくのがコンテナのカテゴリ毎にアルゴリズムを実装するというものです。STLスタイルのコンテナは個々に無秩序なメンバ関数を備えているわけではなく、いくつかのカテゴリに分かれており、同じカテゴリに属すコンテナは似たようなメンバ関数を備えています。例えば★vectorとlistは同じような振る舞いをするメンバ関数を備えていますし、★mapとmultimapも似たようなメンバ関数を備えています。これらのカテゴリには標準できちんと名前が与えられており、vectorやlistのような要素が順番に並ぶコンテナを★シーケンスコンテナ、mapやmultimapのような要素がキーに応じてソートされて保持されるコンテナを★アソシエイティブコンテナ、C++11で標準に追加されたunordered_mapのような要素がキーに応じて保持されるけどソートはされないコンテナを★アンオーダードアソシエイティブコンテナ、そしてstackやqueueのように他のコンテナを土台として操作を制限するものを★コンテナアダプタと呼びます。

2011年11月5日土曜日

用途を制限するのが目的のコンテナアダプタは置いておくとして、その他のカテゴリのコンテナについて、共通しているメンバ関数をもとに3つの抽象的なコンテナを定義することができれば、★用意しなければならないアルゴリズムの種類はそれほど多くはなさそうです。しかしこの方法はうまくいきません。



2011年11月5日土曜日

用途を制限するのが目的のコンテナアダプタは置いておくとして、その他のカテゴリのコンテナについて、共通しているメンバ関数をもとに3つの抽象的なコンテナを定義することができれば、★用意しなければならないアルゴリズムの種類はそれほど多くはなさそうです。しかしこの方法はうまくいきません。

2011年11月5日土曜日

用途を制限するのが目的のコンテナアダプタは置いておくとして、その他のカテゴリのコンテナについて、共通しているメンバ関数をもとに3つの抽象的なコンテナを定義することができれば、★用意しなければならないアルゴリズムの種類はそれほど多くはなさそうです。しかしこの方法はうまくいきません。

同種のコンテナでも結構違う

2011年11月5日土曜日

なぜなら同一カテゴリの有ったり無かったりするメンバ関数が結構あるからです。例えば★シーケンスコンテナの場合、★arrayは一見vectorに似ていますが固定長なのでinsertやeraseは出来ません。★listに備わっているmergeや_といったメンバ関数は他のシーケンスコンテナには見られません。★arrayとvectorはメンバ関数dataから生の配列にアクセスできますが、それ以外のシーケンスコンテナはメモリ上に連続して要素が配置されているわけではないのでこれが使えません。★そしてstringはinsertやerase、findの返り値などで位置を表すのにイテレータではなく整数を使います。

同種のコンテナでも結構違う

2011年11月5日土曜日

なぜなら同一カテゴリの有ったり無かったりするメンバ関数が結構あるからです。例えば★シーケンスコンテナの場合、★arrayは一見vectorに似ていますが固定長なのでinsertやeraseは出来ません。★listに備わっているmergeや_といったメンバ関数は他のシーケンスコンテナには見られません。★arrayとvectorはメンバ関数dataから生の配列にアクセスできますが、それ以外のシーケンスコンテナはメモリ上に連続して要素が配置されているわけではないのでこれが使えません。★そしてstringはinsertやerase、findの返り値などで位置を表すのにイテレータではなく整数を使います。

arrayだけ
insert/eraseもできない

同種のコンテナでも結構違う

2011年11月5日土曜日

なぜなら同一カテゴリの有ったり無かったりするメンバ関数が結構あるからです。例えば★シーケンスコンテナの場合、★arrayは一見vectorに似ていますが固定長なのでinsertやeraseは出来ません。★listに備わっているmergeや_といったメンバ関数は他のシーケンスコンテナには見られません。★arrayとvectorはメンバ関数dataから生の配列にアクセスできますが、それ以外のシーケンスコンテナはメモリ上に連続して要素が配置されているわけではないのでこれが使えません。★そしてstringはinsertやerase、findの返り値などで位置を表すのにイテレータではなく整数を使います。

arrayだけ
insert/eraseもできない

listしかmergeできない

同種のコンテナでも結構違う

2011年11月5日土曜日

なぜなら同一カテゴリの有ったり無かったりするメンバ関数が結構あるからです。例えば★シーケンスコンテナの場合、★arrayは一見vectorに似ていますが固定長なのでinsertやeraseは出来ません。★listに備わっているmergeや_といったメンバ関数は他のシーケンスコンテナには見られません。★arrayとvectorはメンバ関数dataから生の配列にアクセスできますが、それ以外のシーケンスコンテナはメモリ上に連続して要素が配置されているわけではないのでこれが使えません。★そしてstringはinsertやerase、findの返り値などで位置を表すのにイテレータではなく整数を使います。

arrayだけ
insert/eraseもできない

listしかmergeできない

arrayとvectorしか
dataできない

同種のコンテナでも結構違う

2011年11月5日土曜日

なぜなら同一カテゴリの有ったり無かったりするメンバ関数が結構あるからです。例えば★シーケンスコンテナの場合、★arrayは一見vectorに似ていますが固定長なのでinsertやeraseは出来ません。★listに備わっているmergeや_といったメンバ関数は他のシーケンスコンテナには見られません。★arrayとvectorはメンバ関数dataから生の配列にアクセスできますが、それ以外のシーケンスコンテナはメモリ上に連続して要素が配置されているわけではないのでこれが使えません。★そしてstringはinsertやerase、findの返り値などで位置を表すのにイテレータではなく整数を使います。

arrayだけ
insert/eraseもできない

listしかmergeできない

arrayとvectorしか
dataできない

stringは
イテレータでinsert/eraseしない

同種のコンテナでも結構違う

2011年11月5日土曜日

なぜなら同一カテゴリの有ったり無かったりするメンバ関数が結構あるからです。例えば★シーケンスコンテナの場合、★arrayは一見vectorに似ていますが固定長なのでinsertやeraseは出来ません。★listに備わっているmergeや_といったメンバ関数は他のシーケンスコンテナには見られません。★arrayとvectorはメンバ関数dataから生の配列にアクセスできますが、それ以外のシーケンスコンテナはメモリ上に連続して要素が配置されているわけではないのでこれが使えません。★そしてstringはinsertやerase、findの返り値などで位置を表すのにイテレータではなく整数を使います。

コンテナの特徴による分類

2011年11月5日土曜日

そこでちょっと視点を変えてみましょう。★今、ソートされた2つのコンテナをソートされた1つのコンテナに統合するmergeというアルゴリズムを考えます。この関数は二つのコンテナに対して、一方に要素の追加、一方に要素の削除を伴うため、引数をイテレータとして実装することはできません。★おそらくこのような形でコンテナを直接受け取ることになるでしょう。この関数は実際には何をするでしょうか。★まずコンテナがlistのように効率の良いmergeをメンバ関数として持っていれば、それを使うべきでしょう。★それが無い場合でも、★コンテナがinsertとeraseをメンバ関数として持っていれば、効率はともかく同等の結果を得る処理は可能でしょう。★insertやeraseがない、固定長のコンテナの場合はどうにもならないので、コンパイルエラーになってもらいたいところです。実際にはinsertを使って処理を行う場合他にもbegin、endがコンテナに備わっている必要がありますが、★それでもこのアルゴリズムがコンテナに対して要求するコンテナの特徴はたった5つであることが分かります。

2つのソートされたコンテナを統合する

コンテナの特徴による分類

2011年11月5日土曜日

そこでちょっと視点を変えてみましょう。★今、ソートされた2つのコンテナをソートされた1つのコンテナに統合するmergeというアルゴリズムを考えます。この関数は二つのコンテナに対して、一方に要素の追加、一方に要素の削除を伴うため、引数をイテレータとして実装することはできません。★おそらくこのような形でコンテナを直接受け取ることになるでしょう。この関数は実際には何をするでしょうか。★まずコンテナがlistのように効率の良いmergeをメンバ関数として持っていれば、それを使うべきでしょう。★それが無い場合でも、★コンテナがinsertとeraseをメンバ関数として持っていれば、効率はともかく同等の結果を得る処理は可能でしょう。★insertやeraseがない、固定長のコンテナの場合はどうにもならないので、コンパイルエラーになってもらいたいところです。実際にはinsertを使って処理を行う場合他にもbegin、endがコンテナに備わっている必要がありますが、★それでもこのアルゴリズムがコンテナに対して要求するコンテナの特徴はたった5つであることが分かります。

2つのソートされたコンテナを統合する

```
template< typename Cont >
void merge( Cont &_cont1, Cont &_cont2 ) { ... }
```

コンテナの特徴による分類

2011年11月5日土曜日

そこでちょっと視点を変えてみましょう。★今、ソートされた2つのコンテナをソートされた1つのコンテナに統合するmergeというアルゴリズムを考えます。この関数は二つのコンテナに対して、一方に要素の追加、一方に要素の削除を伴うため、引数をイテレータとして実装することはできません。★おそらくこのような形でコンテナを直接受け取ることになるでしょう。この関数は実際には何をするでしょうか。★まずコンテナがlistのように効率の良いmergeをメンバ関数として持っていれば、それを使うべきでしょう。★それが無い場合でも、★コンテナがinsertとeraseをメンバ関数として持っていれば、効率はともかく同等の結果を得る処理は可能でしょう。★insertやeraseがない、固定長のコンテナの場合はどうにもならないので、コンパイルエラーになってもらいたいところです。実際にはinsertを使って処理を行う場合他にもbegin、endがコンテナに備わっている必要がありますが、★それでもこのアルゴリズムがコンテナに対して要求するコンテナの特徴はたった5つであることが分かります。

2つのソートされたコンテナを統合する

```
template< typename Cont >
void merge( Cont &_cont1, Cont &_cont2 ) { ... }
```

Cont::mergeがある → Cont::mergeを使う

コンテナの特徴による分類

2011年11月5日土曜日

そこでちょっと視点を変えてみましょう。★今、ソートされた2つのコンテナをソートされた1つのコンテナに統合するmergeというアルゴリズムを考えます。この関数は二つのコンテナに対して、一方に要素の追加、一方に要素の削除を伴うため、引数をイテレータとして実装することはできません。★おそらくこのような形でコンテナを直接受け取ることになるでしょう。この関数は実際には何をするでしょうか。★まずコンテナがlistのように効率の良いmergeをメンバ関数として持っていれば、それを使うべきでしょう。★それが無い場合でも、★コンテナがinsertとeraseをメンバ関数として持っていれば、効率はともかく同等の結果を得る処理は可能でしょう。★insertやeraseがない、固定長のコンテナの場合はどうにもならないので、コンパイルエラーになってもらいたいところです。実際にはinsertを使って処理を行う場合他にもbegin、endがコンテナに備わっている必要がありますが、★それでもこのアルゴリズムがコンテナに対して要求するコンテナの特徴はたった5つであることが分かります。

2つのソートされたコンテナを統合する

```
template< typename Cont >
void merge( Cont &_cont1, Cont &_cont2 ) { ... }
```

Cont::mergeがある → Cont::mergeを使う

Cont::mergeがない

コンテナの特徴による分類

2011年11月5日土曜日

そこでちょっと視点を変えてみましょう。★今、ソートされた2つのコンテナをソートされた1つのコンテナに統合するmergeというアルゴリズムを考えます。この関数は二つのコンテナに対して、一方に要素の追加、一方に要素の削除を伴うため、引数をイテレータとして実装することはできません。★おそらくこのような形でコンテナを直接受け取ることになるでしょう。この関数は実際には何をするでしょうか。★まずコンテナがlistのように効率の良いmergeをメンバ関数として持っていれば、それを使うべきでしょう。★それが無い場合でも、★コンテナがinsertとeraseをメンバ関数として持っていれば、効率はともかく同等の結果を得る処理は可能でしょう。★insertやeraseがない、固定長のコンテナの場合はどうにもならないので、コンパイルエラーになってもらいたいところです。実際にはinsertを使って処理を行う場合他にもbegin、endがコンテナに備わっている必要がありますが、★それでもこのアルゴリズムがコンテナに対して要求するコンテナの特徴はたった5つであることが分かります。

2つのソートされたコンテナを統合する

```
template< typename Cont >
void merge( Cont &_cont1, Cont &_cont2 ) { ... }
```

Cont::mergeがある → Cont::mergeを使う

Cont::mergeがない

Cont::insertはある → 気合いのコピー

コンテナの特徴による分類

2011年11月5日土曜日

そこでちょっと視点を変えてみましょう。★今、ソートされた2つのコンテナをソートされた1つのコンテナに統合するmergeというアルゴリズムを考えます。この関数は二つのコンテナに対して、一方に要素の追加、一方に要素の削除を伴うため、引数をイテレータとして実装することはできません。★おそらくこのような形でコンテナを直接受け取ることになるでしょう。この関数は実際には何をするでしょうか。★まずコンテナがlistのように効率の良いmergeをメンバ関数として持っていれば、それを使うべきでしょう。★それが無い場合でも、★コンテナがinsertとeraseをメンバ関数として持っていれば、効率はともかく同等の結果を得る処理は可能でしょう。★insertやeraseがない、固定長のコンテナの場合はどうにもならないので、コンパイルエラーになってもらいたいところです。実際にはinsertを使って処理を行う場合他にもbegin、endがコンテナに備わっている必要がありますが、★それでもこのアルゴリズムがコンテナに対して要求するコンテナの特徴はたった5つであることが分かります。

2つのソートされたコンテナを統合する

```
template< typename Cont >
void merge( Cont &_cont1, Cont &_cont2 ) { ... }
```

Cont::mergeがある → Cont::mergeを使う

Cont::mergeがない

Cont::insertはある → 気合いのコピー

Cont::insertもない → エラー

コンテナの特徴による分類

2011年11月5日土曜日

そこでちょっと視点を変えてみましょう。★今、ソートされた2つのコンテナをソートされた1つのコンテナに統合するmergeというアルゴリズムを考えます。この関数は二つのコンテナに対して、一方に要素の追加、一方に要素の削除を伴うため、引数をイテレータとして実装することはできません。★おそらくこのような形でコンテナを直接受け取ることになるでしょう。この関数は実際には何をするでしょうか。★まずコンテナがlistのように効率の良いmergeをメンバ関数として持っていれば、それを使うべきでしょう。★それが無い場合でも、★コンテナがinsertとeraseをメンバ関数として持っていれば、効率はともかく同等の結果を得る処理は可能でしょう。★insertやeraseがない、固定長のコンテナの場合はどうにもならないので、コンパイルエラーになってもらいたいところです。実際にはinsertを使って処理を行う場合他にもbegin、endがコンテナに備わっている必要がありますが、★それでもこのアルゴリズムがコンテナに対して要求するコンテナの特徴はたった5つであることが分かります。

2つのソートされたコンテナを統合する

```
template< typename Cont >
void merge( Cont &_cont1, Cont &_cont2 ) { ... }
```

Cont::mergeがある → Cont::mergeを使う

Cont::mergeがない

Cont::insertはある → 気合いのコピー

Cont::insertもない → エラー

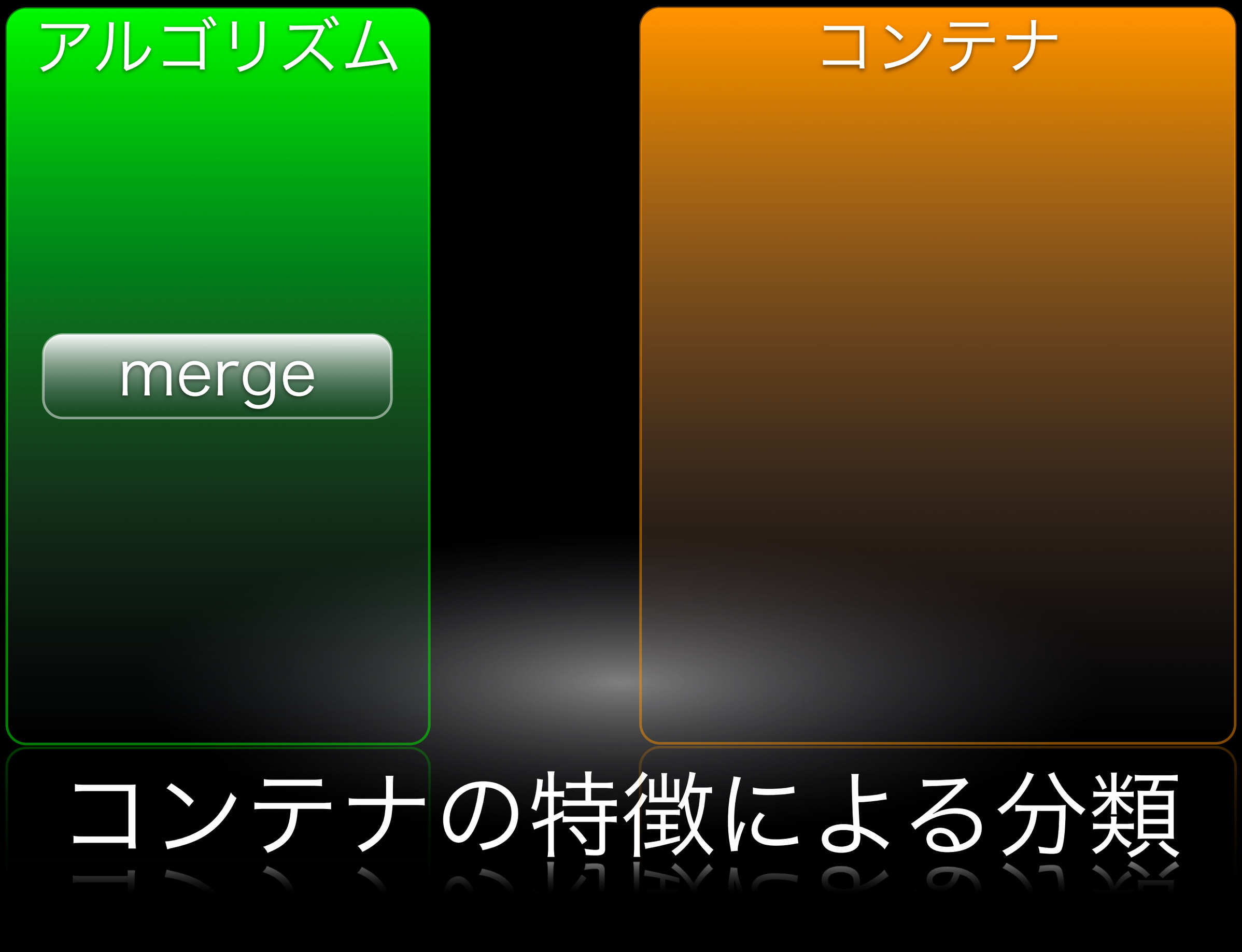
個々のアルゴリズムが要求する

コンテナの特徴は限られている

コンテナの特徴による分類

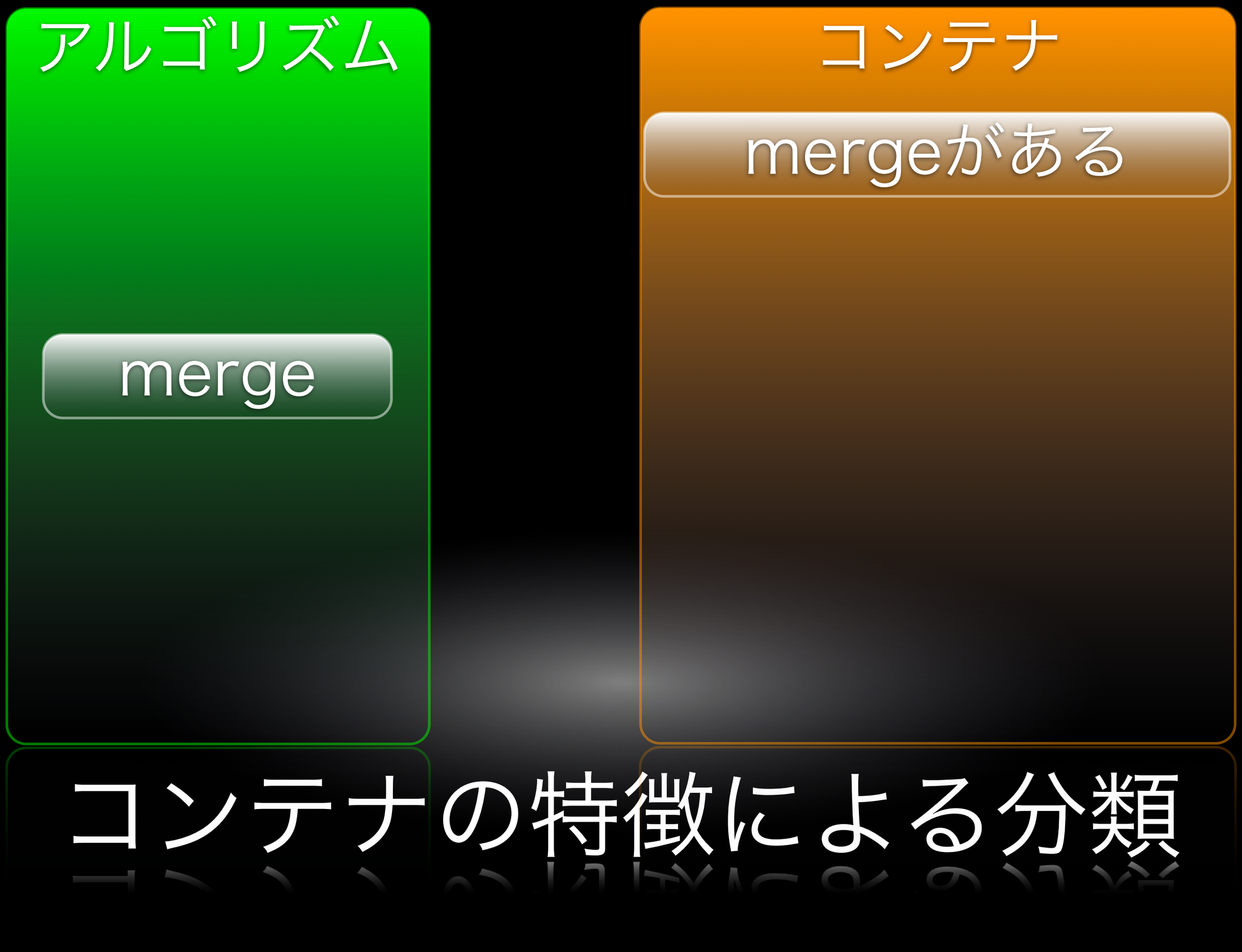
2011年11月5日土曜日

そこでちょっと視点を変えてみましょう。★今、ソートされた2つのコンテナをソートされた1つのコンテナに統合するmergeというアルゴリズムを考えます。この関数は二つのコンテナに対して、一方に要素の追加、一方に要素の削除を伴うため、引数をイテレータとして実装することはできません。★おそらくこのような形でコンテナを直接受け取ることになるでしょう。この関数は実際には何をするでしょうか。★まずコンテナがlistのように効率の良いmergeをメンバ関数として持っていれば、それを使うべきでしょう。★それが無い場合でも、★コンテナがinsertとeraseをメンバ関数として持っていれば、効率はともかく同等の結果を得る処理は可能でしょう。★insertやeraseがない、固定長のコンテナの場合はどうにもならないので、コンパイルエラーになってもらいたいところです。実際にはinsertを使って処理を行う場合他にもbegin、endがコンテナに備わっている必要がありますが、★それでもこのアルゴリズムがコンテナに対して要求するコンテナの特徴はたった5つであることが分かります。



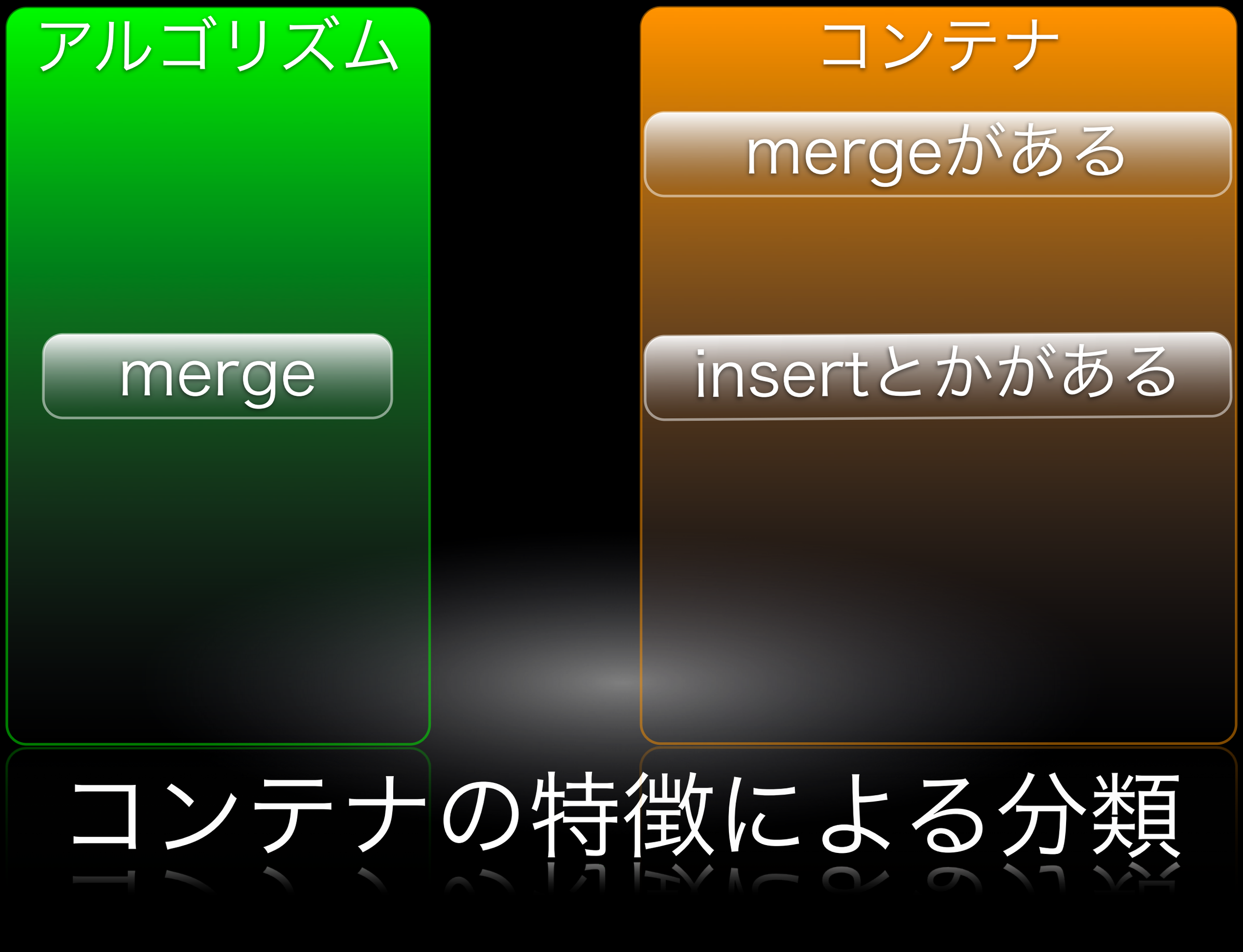
2011年11月5日土曜日

仮にbeginがあってendがない、insertはあるけどeraseがない、といったコンテナがあったとしても、そういったコンテナに対してmergeを実装することはできないと考えられるため、考慮しなければならないのは★メンバ関数mergeがある、★メンバ関数insert/erase/begin/endがある、★それ以外の3つのケースだけである事になります。どうやら、コンテナが特定のメンバ関数を持っているかどうかを調べることができれば、★それほど多くないパターン数で多くのコンテナに適用できるアルゴリズムを実装できそうです。



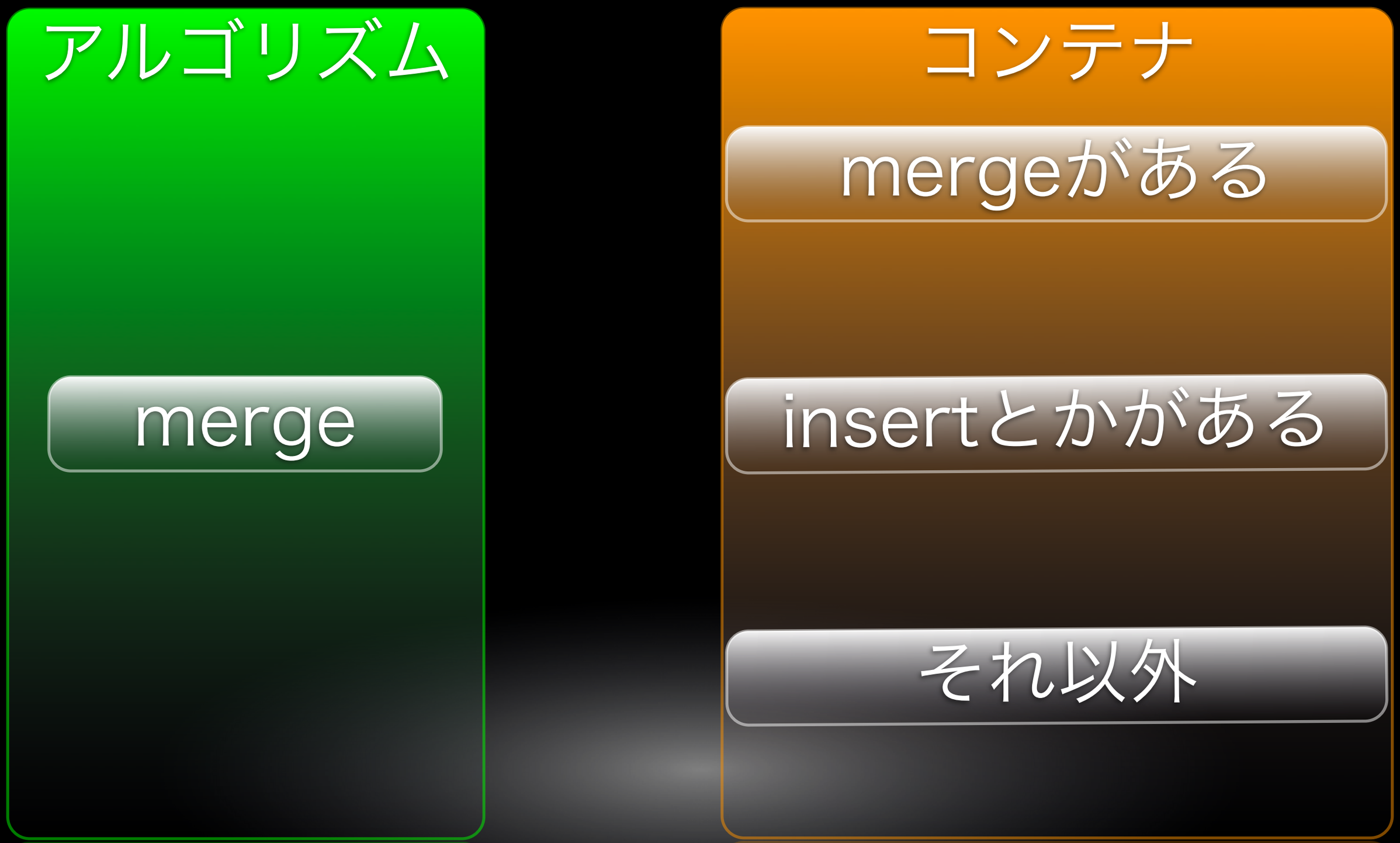
2011年11月5日土曜日

仮にbeginがあってendがない、insertはあるけどeraseがない、といったコンテナがあったとしても、そういったコンテナに対してmergeを実装することはできないと考えられるため、考慮しなければならないのは★メンバ関数mergeがある、★メンバ関数insert/erase/begin/endがある、★それ以外の3つのケースだけである事になります。どうやら、コンテナが特定のメンバ関数を持っているかどうかを調べることができれば、★それほど多くないパターン数で多くのコンテナに適用できるアルゴリズムを実装できそうです。



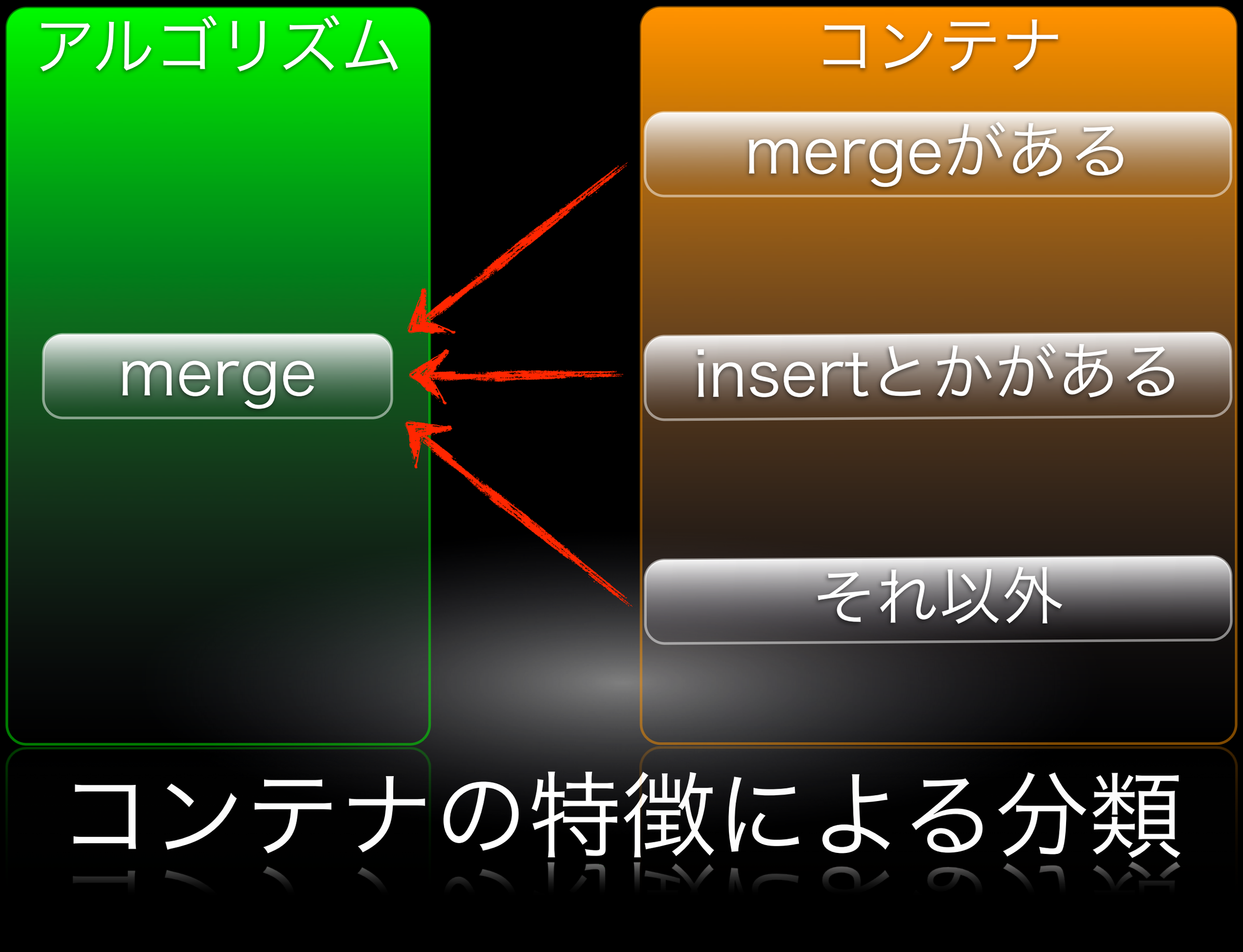
2011年11月5日土曜日

仮にbeginがあってendがない、insertはあるけどeraseがない、といったコンテナがあったとしても、そういったコンテナに対してmergeを実装することはできないと考えられるため、考慮しなければならないのは★メンバ関数mergeがある、★メンバ関数insert/erase/begin/endがある、★それ以外の3つのケースだけである事になります。どうやら、コンテナが特定のメンバ関数を持っているかどうかを調べることができれば、★それほど多くないパターン数で多くのコンテナに適用できるアルゴリズムを実装できそうです。



2011年11月5日土曜日

仮にbeginがあってendがない、insertはあるけどeraseがない、といったコンテナがあったとしても、そういったコンテナに対してmergeを実装することはできないと考えられるため、考慮しなければならないのは★メンバ関数mergeがある、★メンバ関数insert/erase/begin/endがある、★それ以外の3つのケースだけである事になります。どうやら、コンテナが特定のメンバ関数を持っているかどうかを調べることができれば、★それほど多くないパターン数で多くのコンテナに適用できるアルゴリズムを実装できそうです。



2011年11月5日土曜日

仮にbeginがあってendがない、insertはあるけどeraseがない、といったコンテナがあったとしても、そういったコンテナに対してmergeを実装することはできないと考えられるため、考慮しなければならないのは★メンバ関数mergeがある、★メンバ関数insert/erase/begin/endがある、★それ以外の3つのケースだけである事になります。どうやら、コンテナが特定のメンバ関数を持っているかどうかを調べることができれば、★それほど多くないパターン数で多くのコンテナに適用できるアルゴリズムを実装できそうです。

コンテナを利用可能なメンバによって
分類するための仕組み

ContainerTraits

2011年11月5日土曜日

というわけでまず、あるコンテナがどんなメンバ関数を備えているかを調べるための仕組み、ContainerTraitsを作りました。

has_ほげほげ

has_iterator< std::vector< int > >::type

has_iterator< std::stack< int > >::type

has_hasher< std::vector< int > >::type

has_key_type< std::map< int, int > >::type

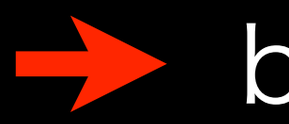
ContainerTraits

2011年11月5日土曜日

あるコンテナにどんな型が定義されているかは、has_ほげほげで調べることができます。例えば、★has_iterator< vector >の結果はtrueになりますが、★has_iterator< stack >の結果はfalseになります。クラスに特定の型が定義されているかどうかはBoost.MPLのhas_xxxを使って調べる事ができ、このhas_ほげほげもそれを使って実装していますが、もう少し小細工がなされています。それについては後ほどContainerFacadeのところで説明したいと思います。

has_ほげほげ

has_iterator< std::vector< int > >::type



boost::mpl::bool_< true >

has_iterator< std::stack< int > >::type

has_hasher< std::vector< int > >::type

has_key_type< std::map< int, int > >::type

ContainerTraits

2011年11月5日土曜日

あるコンテナにどんな型が定義されているかは、has_ほげほげで調べることができます。例えば、★has_iterator< vector >の結果はtrueになりますが、★has_iterator< stack >の結果はfalseになります。クラスに特定の型が定義されているかどうかはBoost.MPLのhas_xxxを使って調べる事ができ、このhas_ほげほげもそれを使って実装していますが、もう少し小細工がなされています。それについては後ほどContainerFacadeのところで説明したいと思います。

has_ほげほげ

has_iterator< std::vector< int > >::type
→ boost::mpl::bool_< true >

has_iterator< std::stack< int > >::type
→ boost::mpl::bool_< false >

has_hasher< std::vector< int > >::type
→ boost::mpl::bool_< false >

has_key_type< std::map< int, int > >::type
→ boost::mpl::bool_< true >

ContainerTraits

2011年11月5日土曜日

あるコンテナにどんな型が定義されているかは、has_ほげほげで調べることができます。例えば、★has_iterator< vector >の結果はtrueになりますが、★has_iterator< stack >の結果はfalseになります。クラスに特定の型が定義されているかどうかはBoost.MPLのhas_xxxを使って調べる事ができ、このhas_ほげほげもそれを使って実装していますが、もう少し小細工がなされています。それについては後ほどContainerFacadeのところで説明したいと思います。

get_ほげほげ

get_iterator< std::vector< int > >::type

get_iterator< std::stack< int > >::type

get_hasher< std::vector< int > >::type

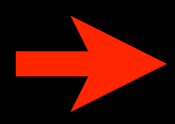
get_key_type< std::map< int, int > >::type

ContainerTraits

有るのは分かったからその型をよこせ、という時のために★get_ほげほげも用意しています。has_ほげほげの結果がfalseになる場合、get_ほげほげは★not_available_tという型を返します。

get_ほげほげ

get_iterator< std::vector< int > >::type



std::vector< int >::iterator

get_iterator< std::stack< int > >::type

get_hasher< std::vector< int > >::type

get_key_type< std::map< int, int > >::type

std::map< int, int >::key_type

ContainerTraits

2011年11月5日土曜日

有るのは分かったからその型をよこせ、という時のために★get_ほげほげも用意しています。has_ほげほげの結果がfalseになる場合、get_ほげほげは★not_available_tという型を返します。

get_ほげほげ

get_iterator< std::vector< int > >::type
→ std::vector< int >::iterator

get_iterator< std::stack< int > >::type
→ not_available_t

get_hasher< std::vector< int > >::type
→ not_available_t

get_key_type< std::map< int, int > >::type
→ std::map< int, int >::key_type

ContainerTraits

2011年11月5日土曜日

有るのは分かったからその型をよこせ、という時のために★get_ほげほげも用意しています。has_ほげほげの結果がfalseになる場合、get_ほげほげは★not_available_tという型を返します。

メンバ関数もhas_ほげほげ

has_at_sequence< std::vector< int > >::type

has_insert_associative< std::vector< int > >::type

has_const_begin< std::vector< int > >::type

has_pop_front< std::vector< int > >::type

ContainerTraits

2011年11月5日土曜日

さて、今知りたいのは型が定義されているかどうかではなく、メンバ関数が使用可能かどうかです。メンバ関数もhas_ほげほげすることができます。ただしメンバ関数はしばしばオーバーロードされており、同じ名前の関数が複数有るため、全てのメンバ関数にユニークな名前を付けました。★例えば、シーケンシャルコンテナのat、つまりsize_typeを受け取ってvalue_typeを返すatが存在するかどうかはhas_at_sequenceで調べることができます。★constなbeginが存在するかどうかはhas_const_beginで調べられます。これらのメタい関数はまず先ほどの型が定義されているかどうかを調べ、必要な型が定義されていた場合、要求された名前と引数と返り値を持つメンバ関数が存在するかどうかを調べます。クラスの中にメンバ関数が存在するかどうかを調べるようなhas_xxxはBoost.MPLの中には無いのですが、fcpptというBoostyなC++ライブラリに含まれていたので、

メンバ関数もhas_ほげほげ

has_at_sequence< std::vector< int > >::type
→ boost::mpl::bool_< true >

has_insert_associative< std::vector< int > >::type

has_const_begin< std::vector< int > >::type

has_pop_front< std::vector< int > >::type

ContainerTraits

2011年11月5日土曜日

さて、今知りたいのは型が定義されているかどうかではなく、メンバ関数が使用可能かどうかです。メンバ関数もhas_ほげほげすることができます。ただしメンバ関数はしばしばオーバーロードされており、同じ名前の関数が複数有るため、全てのメンバ関数にユニークな名前を付けました。★例えば、シーケンシャルコンテナのat、つまりsize_typeを受け取ってvalue_typeを返すatが存在するかどうかはhas_at_sequenceで調べることができます。★constなbeginが存在するかどうかはhas_const_beginで調べられます。これらのメタい関数はまず先ほどの型が定義されているかどうかを調べ、必要な型が定義されていた場合、要求された名前と引数と返り値を持つメンバ関数が存在するかどうかを調べます。クラスの中にメンバ関数が存在するかどうかを調べるようなhas_xxxはBoost.MPLの中には無いのですが、fcpptというBoostyなC++ライブラリに含まれていたので、

メンバ関数もhas_ほげほげ

has_at_sequence< std::vector< int > >::type
→ boost::mpl::bool_< true >

has_insert_associative< std::vector< int > >::type
→ boost::mpl::bool_< false >

has_const_begin< std::vector< int > >::type
→ boost::mpl::bool_< true >

has_pop_front< std::vector< int > >::type
→ boost::mpl::bool_< false >

ContainerTraits

2011年11月5日土曜日

さて、今知りたいのは型が定義されているかどうかではなく、メンバ関数が使用可能かどうかです。メンバ関数もhas_ほげほげすることができます。ただしメンバ関数はしばしばオーバーロードされており、同じ名前の関数が複数有るため、全てのメンバ関数にユニークな名前を付けました。★例えば、シーケンシャルコンテナのat、つまりsize_typeを受け取ってvalue_typeを返すatが存在するかどうかはhas_at_sequenceで調べることができます。★constなbeginが存在するかどうかはhas_const_beginで調べられます。これらのメタい関数はまず先ほどの型が定義されているかどうかを調べ、必要な型が定義されていた場合、要求された名前と引数と返り値を持つメンバ関数が存在するかどうかを調べます。クラスの中にメンバ関数が存在するかどうかを調べるようなhas_xxxはBoost.MPLの中には無いのですが、fcpptというBoostyなC++ライブラリに含まれていたので、

http://fcppt.net/

2011年11月5日土曜日

これを使う事にしました。★今回使ったヘッダはこれですが、脳内コンパイルを試みたところ構文解析に失敗したので、どういう仕組みになっているのかはよくわかりません。

http://fcppt.net/

fcppt/type_traits/generate_has_member_function.hpp

fcppt

2011年11月5日土曜日

これを使う事にしました。★今回使ったヘッダはこれですが、脳内コンパイルを試みたところ構文解析に失敗したので、どういう仕組みになっているのかはよくわかりません。

これらをSFINAEの種にすることで
最小限のアルゴリズムの実装で
様々なコンテナに対応する

```
template< typename Cont >
void merge( Cont &_cont1, Cont &_cont2,
  typename enable_if< has_merge< Cont > >::type* = 0
) { ... }
```

ContainerTraits

これでメンバ関数の有無をコンパイル時に調べることが出来るようになったので、これをenable_ifの条件にします。例えば、先ほどのmergeのうち、コンテナがメンバ関数mergeを持っている場合について書くにはこのようになります。

STLライクなコンテナには代用可能な関数が多くある

insert がなくても

begin と end があって

resize できるコンテナなら

要素の移動と組み合わせてどうにかなる

問題

2011年11月5日土曜日

ところで先ほどの例ではmergeが無かった場合にinsert/erase/begin/endで代用しましたが、insert/eraseはbegin/end/resizeがあればパフォーマンスはともかく同じ結果を得ることができます。ではinsertとeraseのどちらかあるいは両方が無いけど、begin/end/resizeがある場合のmergeも用意すべきでしょうか。こうした代用可能なメンバ関数はSTLスタイルのコンテナには沢山有るため、それら全てのケースを想定してアルゴリズムを用意していたのでは、結局アルゴリズムは膨大な関数になってしまいます。

STLコンテナには代用可能な関数が多くある

pop_back を使って pop_front が
無いのを補おうとする場合、
そのコードはアルゴリズムによらず

問題

2011年11月5日土曜日

どうしたものでしょうか。今pop_frontが無いコンテナで、それをpop_back/begin/endを使って補おうとしているとします。そのコードは何をするアルゴリズムかに関わらず★このようなものになるでしょう。このように、殆どの場合メンバ関数の代用の為のコードはアルゴリズムに依存しません。であれば、★アルゴリズムから分離したところでこうした代用を実装すれば、アルゴリズムは欲しているメンバ関数相当の処理が出来るのか出来ないのかをストレートに調べるだけでよくなります。

STLコンテナには代用可能な関数が多くある

pop_back を使って pop_front が
無いのを補おうとする場合、
そのコードはアルゴリズムによらず

```
move( cont.begin(), ++cont.begin(), cont.end() );
cont.pop_back();
```

問題

2011年11月5日土曜日

どうしたものでしょうか。今pop_frontが無いコンテナで、それをpop_back/begin/endを使って補おうとしているとします。そのコードは何をするアルゴリズムかに関わらず★このようなものになるでしょう。このように、殆どの場合メンバ関数の代用の為のコードはアルゴリズムに依存しません。であれば、★アルゴリズムから分離したところでこうした代用を実装すれば、アルゴリズムは欲しているメンバ関数相当の処理が出来るのか出来ないのかをストレートに調べるだけでよくなります。

STLコンテナには代用可能な関数が多くある

pop_back を使って pop_front が
無いのを補おうとする場合、
そのコードはアルゴリズムによらず

```
move( cont.begin(), ++cont.begin(), cont.end() );
cont.pop_back();
```

こうしたコードをアルゴリズム毎に
用意するのは非合理的

問題

2011年11月5日土曜日

どうしたものでしょうか。今pop_frontが無いコンテナで、それをpop_back/begin/endを使って補おうとしているとします。そのコードは何をするアルゴリズムかに関わらず★このようなものになるでしょう。このように、殆どの場合メンバ関数の代用の為のコードはアルゴリズムに依存しません。であれば、★アルゴリズムから分離したところでこうした代用を実装すれば、アルゴリズムは欲しているメンバ関数相当の処理が出来るのか出来ないのかをストレートに調べるだけでよくなります。

コンテナには実現可能な全てのメンバ関数が
備わっていて欲しい

ContainerFacade

2011年11月5日土曜日

一般にSTLスタイルのコンテナでは、実装可能だけど実装してもパフォーマンス上のメリットが全く無いメンバ関数は利用可能になっていません。しかし今まで見てきたように、コンテナを抽象化するという観点から言えば速かろうが遅かろうが実現可能なメンバ関数は全て利用可能になっている方がありがたいわけです。この差を埋めるのがContainerFacadeです。

ContainerFacade

2011年11月5日土曜日

ContainerFacadeの使い方は簡単です。★このように元のコンテナをテンプレート引数としてインスタンスを作ると、元のコンテナに存在しないけど、★存在するメンバ関数の組み合わせで実現可能なメンバ関数が補われます。この例で使われているvectorには本来push_frontは有りませんが、パフォーマンスを気にしなければpush_frontを実装することはいくらでも可能なため、ContainerFacadeを通すとpush_frontが使用可能になります。

```
ContainerFacade< std::vector< int > > cont;
cont.push_front( 1 );
```

ContainerFacade

2011年11月5日土曜日

ContainerFacadeの使い方は簡単です。★このように元のコンテナをテンプレート引数としてインスタンスを作ると、元のコンテナに存在しないけど、★存在するメンバ関数の組み合わせで実現可能なメンバ関数が補われます。この例で使われているvectorには本来push_frontは有りませんが、パフォーマンスを気にしなければpush_frontを実装することはいくらでも可能なため、ContainerFacadeを通すとpush_frontが使用可能になります。

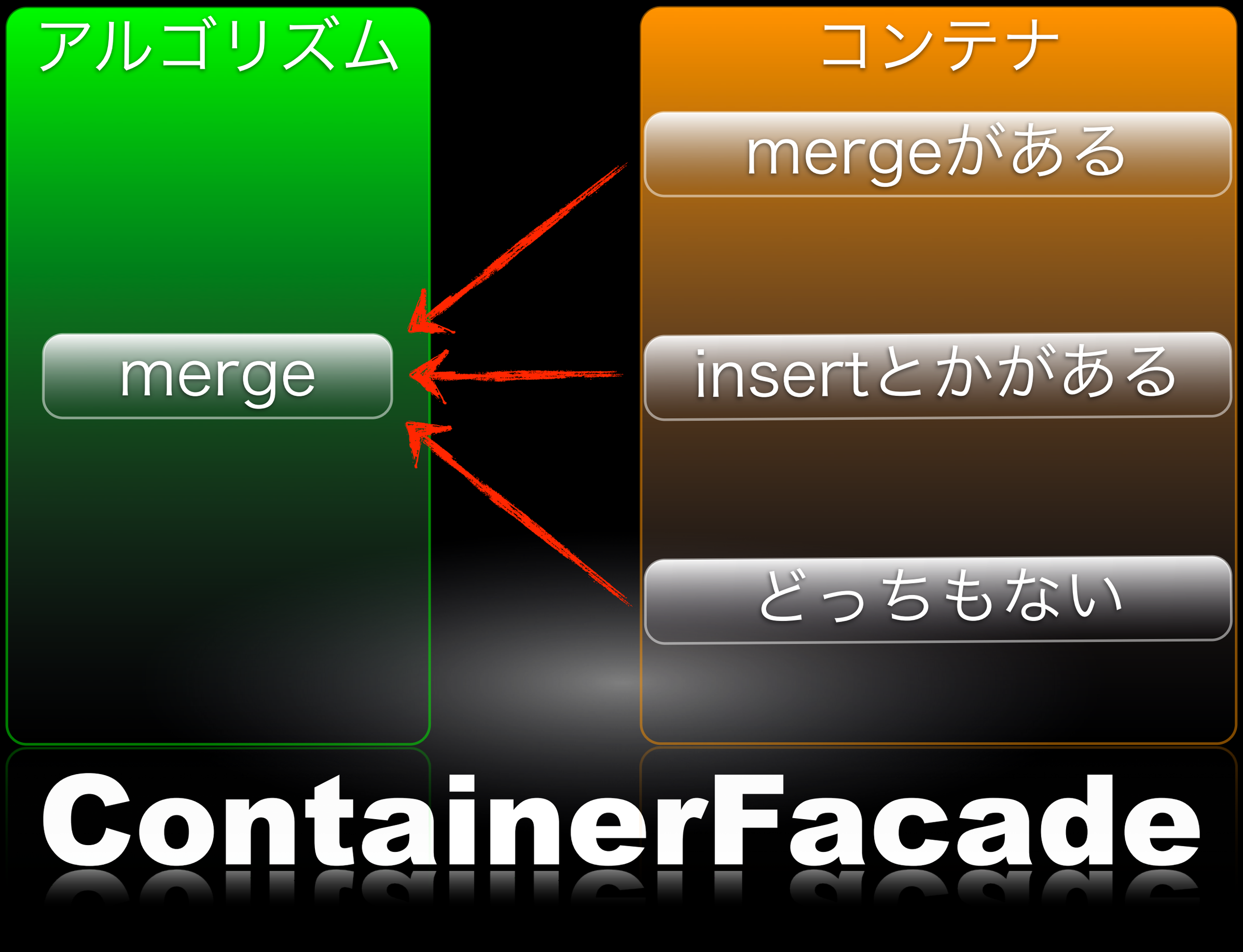
```
ContainerFacade< std::vector< int > > cont;
cont.push_front( 1 );
```

ContainerFacadeを通すと
元のコンテナには無くても実現可能なメンバ関数が
使用可能になる

ContainerFacade

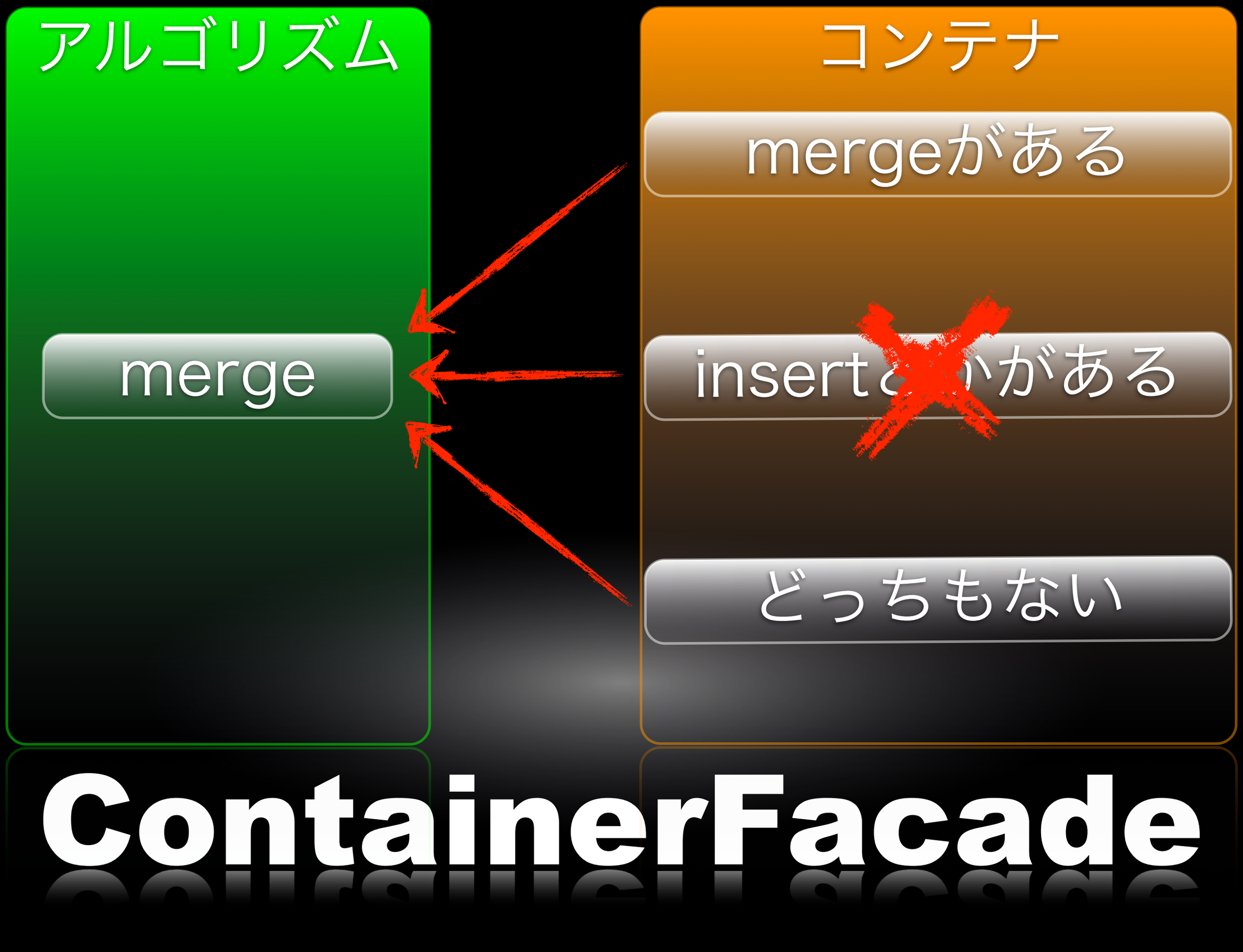
2011年11月5日土曜日

ContainerFacadeの使い方は簡単です。★このように元のコンテナをテンプレート引数としてインスタンスを作ると、元のコンテナに存在しないけど、★存在するメンバ関数の組み合わせで実現可能なメンバ関数が補われます。この例で使われているvectorには本来push_frontは有りませんが、パフォーマンスを気にしなければpush_frontを実装することはいくらでも可能なため、ContainerFacadeを通すとpush_frontが使用可能になります。



2011年11月5日土曜日

mergeの例をもう一度考えてみましょう。mergeと同等の処理が出来る条件が整っている場合、ContainerFacadeを通すとmergeが使用可能になるため、アルゴリズムはContainerFacadeを通した状態でmergeが使用可能かどうかを調べ、可能ならmergeを呼び出し、mergeが使用可能でないなら、即座に諦めれば良いことになります。先ほどもう一つの可能性として挙げられていたinsert/erase/begin/endが使用可能な場合については、★アルゴリズム側が考慮する必要は無くなります



2011年11月5日土曜日

mergeの例をもう一度考えてみましょう。mergeと同等の処理が出来る条件が整っている場合、ContainerFacadeを通すとmergeが使用可能になるため、アルゴリズムはContainerFacadeを通した状態でmergeが使用可能かどうかを調べ、可能ならmergeを呼び出し、mergeが使用可能でないなら、即座に諦めれば良いことになります。先ほどもう一つの可能性として挙げられていたinsert/erase/begin/endが使用可能な場合については、★アルゴリズム側が考慮する必要は無くなります

ContainerFacade

2011年11月5日土曜日

使用方法のシンプルさに反して、ContainerFacadeの中身はやや複雑になっています。
ContainerFacadeは先ほどのContainerTraitsを使ってSFINAEで関数を呼び分けています。なので、つい★このように実装したくなるのですが、これは正しく動きません。なぜなら、★ここで使われている型Baseは関数のテンプレート引数ではなく、★クラスのテンプレート引数だからです。SFINAEのつもりで書いたenable_ifはクラスのテンプレート引数Baseが確定した瞬間に評価され、★sizeが有れば2つめの関数がコンパイルエラーに、sizeが無ければ1つめの関数がコンパイルエラーになります。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  size_type size(
    typename enable_if< has_size< Base > >::type* = 0
  ) const { return base.size(); }
  size_type size(
    typename enable_if<
      !has_size< Base > && has_begin< Base > && has_end< Base > >::type* = 0
  ) const { return distance( base.begin(), base.end() ); }
  ...
};
```

ContainerFacade

2011年11月5日土曜日

使用方法のシンプルさに反して、ContainerFacadeの中身はやや複雑になっています。ContainerFacadeは先ほどのContainerTraitsを使ってSFINAEで関数を呼び分けています。なので、つい★このように実装したくなるのですが、これは正しく動きません。なぜなら、★ここで使われている型Baseは関数のテンプレート引数ではなく、★クラスのテンプレート引数だからです。SFINAEのつもりで書いたenable_ifはクラスのテンプレート引数Baseが確定した瞬間に評価され、★sizeが有れば2つめの関数がコンパイルエラーに、sizeが無ければ1つめの関数がコンパイルエラーになります。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  size_type size(
    typename enable_if< has_size< Base > >::type* = 0
  ) const { return base.size(); }
  size_type size(
    typename enable_if<
      !has_size< Base > && has_begin< Base > && has_end< Base > >::type* = 0
  ) const { return distance( base.begin(), base.end() ); }
  ...
};
```

ContainerFacade

2011年11月5日土曜日

使用方法のシンプルさに反して、ContainerFacadeの中身はやや複雑になっています。ContainerFacadeは先ほどのContainerTraitsを使ってSFINAEで関数を呼び分けています。なので、つい★このように実装したくなるのですが、これは正しく動きません。なぜなら、★ここで使われている型Baseは関数のテンプレート引数ではなく、★クラスのテンプレート引数だからです。SFINAEのつもりで書いたenable_ifはクラスのテンプレート引数Baseが確定した瞬間に評価され、★sizeが有れば2つめの関数がコンパイルエラーに、sizeが無ければ1つめの関数がコンパイルエラーになります。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  size_type size(
    typename enable_if< has_size< Base > >::type* = 0
  ) const { return base.size(); }
  size_type size(
    typename enable_if<
      !has_size< Base > && has_begin< Base > && has_end< Base > >::type* = 0
  ) const { return distance( base.begin(), base.end() ); }
  ...
};
```

ContainerFacade

2011年11月5日土曜日

使用方法のシンプルさに反して、ContainerFacadeの中身はやや複雑になっています。ContainerFacadeは先ほどのContainerTraitsを使ってSFINAEで関数を呼び分けています。なので、つい★このように実装したくなるのですが、これは正しく動きません。なぜなら、★ここで使われている型Baseは関数のテンプレート引数ではなく、★クラスのテンプレート引数だからです。SFINAEのつもりで書いたenable_ifはクラスのテンプレート引数Baseが確定した瞬間に評価され、★sizeが有れば2つめの関数がコンパイルエラーに、sizeが無ければ1つめの関数がコンパイルエラーになります。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  size_type size(
    typename enable_if< has_size< Base > >::type* = 0
  ) const { return base.size(); }
  size_type size(
    typename enable_if<
      !has_size< Base > && has_begin< Base > && has_end< Base > >::type* = 0
  ) const { return distance( base.begin(), base.end() ); }
  ...
};
```

コンパイルエラー

ContainerFacade

2011年11月5日土曜日

使用方法のシンプルさに反して、ContainerFacadeの中身はやや複雑になっています。ContainerFacadeは先ほどのContainerTraitsを使ってSFINAEで関数を呼び分けています。なので、つい★このように実装したくなるのですが、これは正しく動きません。なぜなら、★ここで使われている型Baseは関数のテンプレート引数ではなく、★クラスのテンプレート引数だからです。SFINAEのつもりで書いたenable_ifはクラスのテンプレート引数Baseが確定した瞬間に評価され、★sizeが有れば2つめの関数がコンパイルエラーに、sizeが無ければ1つめの関数がコンパイルエラーになります。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  template< typename _Base >
  size_type size(
    typename enable_if< has_size< _Base > >::type* = 0
  ) const { return base.size(); }
  template< typename _Base >
  size_type size(
    typename enable_if<
      !has_size< _Base > && has_begin< _Base > && has_end< _Base > >::type* = 0
  ) const { return distance( base.begin(), base.end() ); }
  ...
};
```

ContainerFacade

このように、メンバ関数にテンプレート引数を持たせれば、enable_ifは関数呼び出しの時点で評価され、SFINAEは正しく機能します。しかし、★このテンプレート引数アンダースコアBaseはメンバ関数の引数からは決定できないため、これらの関数を呼び出す度に★テンプレート引数を付けて呼ぶ必要が生じます。そしてクラスのテンプレート引数として渡された型とメンバ関数呼び出し時に渡された型が等しくないと正しく機能しないクラスなど、バグを作ってくださいと言っているようなものです。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  template< typename _Base >
  size_type size(
    typename enable_if< has_size< _Base > >::type* = 0
  ) const { return base.size(); }
  template< typename _Base >
  size_type size(
    typename enable_if<
      !has_size< _Base > && has_begin< _Base > && has_end< _Base > >::type* = 0
  ) const { return distance( base.begin(), base.end() ); }
  ...
};
```

ContainerFacade

このように、メンバ関数にテンプレート引数を持たせれば、enable_ifは関数呼び出しの時点で評価され、SFINAEは正しく機能します。しかし、★このテンプレート引数アンダースコアBaseはメンバ関数の引数からは決定できないため、これらの関数を呼び出す度に★テンプレート引数を付けて呼ぶ必要が生じます。そしてクラスのテンプレート引数として渡された型とメンバ関数呼び出し時に渡された型が等しくないと正しく機能しないクラスなど、バグを作ってくださいと言っているようなものです。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  template< typename _Base >
  size_type size(
    typename enable_if< has_size< _Base > >::type* = 0
  ) const { return base.size(); }
  template< typename _Base >
  size_type size(
    typename enable_if<
      !has_size< _Base > && has_begin< _Base > && has_end< _Base > >::type* = 0
  ) const { return distance( base.begin(), base.end() ); }
  ...
};
```

関数呼び出しの度に型を渡す必要が生じる

ContainerFacade

2011年11月5日土曜日

このように、メンバ関数にテンプレート引数を持たせれば、enable_ifは関数呼び出しの時点で評価され、SFINAEは正しく機能します。しかし、★このテンプレート引数アンダースコアBaseはメンバ関数の引数からは決定できないため、これらの関数を呼び出す度に★テンプレート引数を付けて呼ぶ必要が生じます。そしてクラスのテンプレート引数として渡された型とメンバ関数呼び出し時に渡された型が等しくないと正しく機能しないクラスなど、バグを作ってくださいと言っているようなものです。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  template< typename _Base = Base >
  size_type size(
    typename enable_if< has_size< _Base > >::type* = 0
  ) const { return base.size(); }
  template< typename _Base = Base >
  size_type size(
    typename enable_if<
      !has_size< _Base > && has_begin< _Base > && has_end< _Base > >::type* = 0
  ) const { return distance( base.begin(), base.end() ); }
  ...
};
```

ContainerFacade

2011年11月5日土曜日

C++11なら関数のテンプレート引数にデフォルト値を与えることが出来るため、この問題は丸くおさめることが出来ます。しかし、★たったこれだけの理由でC++03な環境を見捨てたくはありません。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  template< typename _Base = Base >
  size_type size(
    typename enable_if< has_size< _Base > >::type* = 0
  ) const { return base.size(); }
  template< typename _Base = Base >
  size_type size(
    typename enable_if<
      !has_size< _Base > && has_begin< _Base > && has_end< _Base > >::type* = 0
  ) const { return distance( base.begin(), base.end() ); }
  ...
};
```

only for C++11

ContainerFacade

2011年11月5日土曜日

C++11なら関数のテンプレート引数にデフォルト値を与えることが出来るため、この問題は丸くおさめることが出来ます。しかし、★たったこれだけの理由でC++03な環境を見捨てたくはありません。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  template< typename _Base >
  static size_type _size( const _Base &_base
    typename enable_if< has_size< _Base > >::type* = 0
  ) { return _base.size(); }
  template< typename _Base >
  static size_type _size( const _Base &_base
    typename enable_if<
      !has_size< _Base > && has_begin< _Base > && has_end< _Base > >::type* = 0
  ) { return distance( _base.begin(), _base.end() ); }
  size_type size() const { return _size( base ); }
  ...
};
```

ContainerFacade

2011年11月5日土曜日

そこでContainerFacadeはこのような実装になっています。まず★この行のメンバ関数sizeが呼び出されます。sizeは第一引数にBase型のコンテナを付けて★staticなメンバ関数アンダースコアsizeに★処理を丸投げします。アンダースコアsizeはテンプレート関数で、★第一引数の型によってアンダースコアBaseの型が決定します。このアンダースコアsizeはenable_ifで分岐されており、関数呼び出し時に第一引数から決定したコンテナの型によって適切な方が選択されます。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  template< typename _Base >
  static size_type _size( const _Base &_base
    typename enable_if< has_size< _Base > >::type* = 0
  ) { return _base.size(); }
  template< typename _Base >
  static size_type _size( const _Base &_base
    typename enable_if<
      !has_size< _Base > && has_begin< _Base > && has_end< _Base > >::type* = 0
  ) { return distance( _base.begin(), _base.end() ); }
  size_type size() const { return _size( base ); }
  ...
};
```


ContainerFacade

2011年11月5日土曜日

そこでContainerFacadeはこのような実装になっています。まず★この行のメンバ関数sizeが呼び出されます。sizeは第一引数にBase型のコンテナを付けて★staticなメンバ関数アンダースコアsizeに★処理を丸投げします。アンダースコアsizeはテンプレート関数で、★第一引数の型によってアンダースコアBaseの型が決定します。このアンダースコアsizeはenable_ifで分岐されており、関数呼び出し時に第一引数から決定したコンテナの型によって適切な方が選択されます。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  template< typename _Base >
  static size_type _size( const _Base &_base
    typename enable_if< has_size< _Base > >::type* = 0
  ) { return _base.size(); }
  template< typename _Base >
  static size_type _size( const _Base &_base
    typename enable_if<
      !has_size< _Base > && has_begin< _Base > && has_end< _Base > >::type* = 0
  ) { return distance( _base.begin(), _base.end() ); }
  size_type size() const { return _size( base ); }
  ...
};
```

ContainerFacade

2011年11月5日土曜日

そこでContainerFacadeはこのような実装になっています。まず★この行のメンバ関数sizeが呼び出されます。sizeは第一引数にBase型のコンテナを付けて★staticなメンバ関数アンダースコアsizeに★処理を丸投げします。アンダースコアsizeはテンプレート関数で、★第一引数の型によってアンダースコアBaseの型が決定します。このアンダースコアsizeはenable_ifで分岐されており、関数呼び出し時に第一引数から決定したコンテナの型によって適切な方が選択されます。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  template< typename _Base >
  static size_type _size( const _Base &_base
    typename enable_if< has_size< _Base > >::type* = 0
  ) { return _base.size(); }
  template< typename _Base >
  static size_type _size( const _Base &_base
    typename enable_if<
      !has_size< _Base > && has_begin< _Base > && has_end< _Base > >::type* = 0
  ) { return distance( _base.begin(), _base.end() ); }
  size_type size() const { return _size( base ); }
  ...
};
```

ContainerFacade

2011年11月5日土曜日

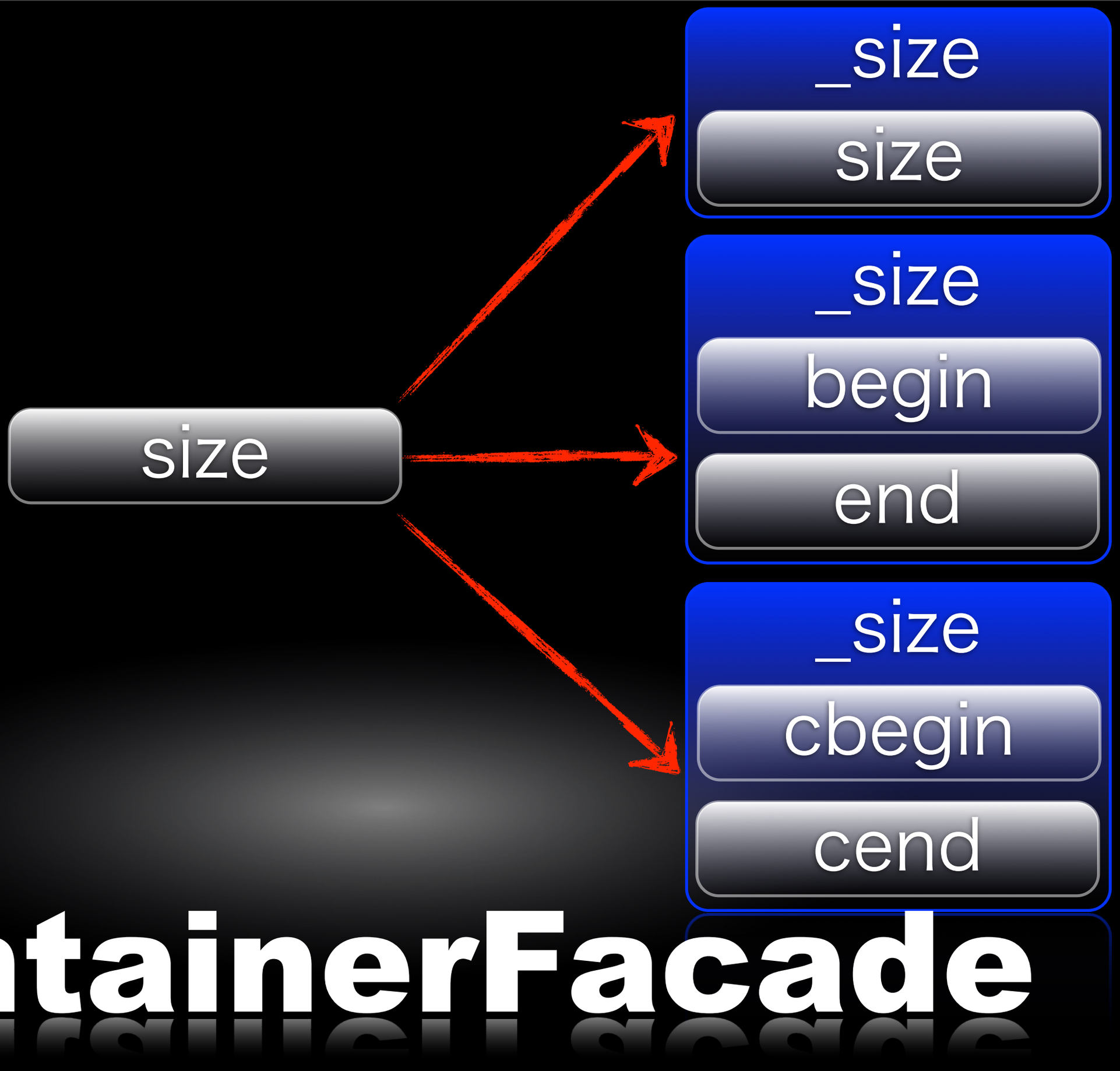
そこでContainerFacadeはこのような実装になっています。まず★この行のメンバ関数sizeが呼び出されます。sizeは第一引数にBase型のコンテナを付けて★staticなメンバ関数アンダースコアsizeに★処理を丸投げします。アンダースコアsizeはテンプレート関数で、★第一引数の型によってアンダースコアBaseの型が決定します。このアンダースコアsizeはenable_ifで分岐されており、関数呼び出し時に第一引数から決定したコンテナの型によって適切な方が選択されます。

```
template< typename Base >
class ContainerFacade {
public:
  ...
  template< typename _Base >
  static size_type _size( const _Base &_base
    typename enable_if< has_size< _Base > >::type* = 0
  ) { return _base.size(); }
  template< typename _Base >
  static size_type _size( const _Base &_base
    typename enable_if<
      !has_size< _Base > && has_begin< _Base > && has_end< _Base > >::type* = 0
  ) { return distance( _base.begin(), _base.end() ); }
  size_type size() const { return _size( base ); }
  ...
};
```

ContainerFacade

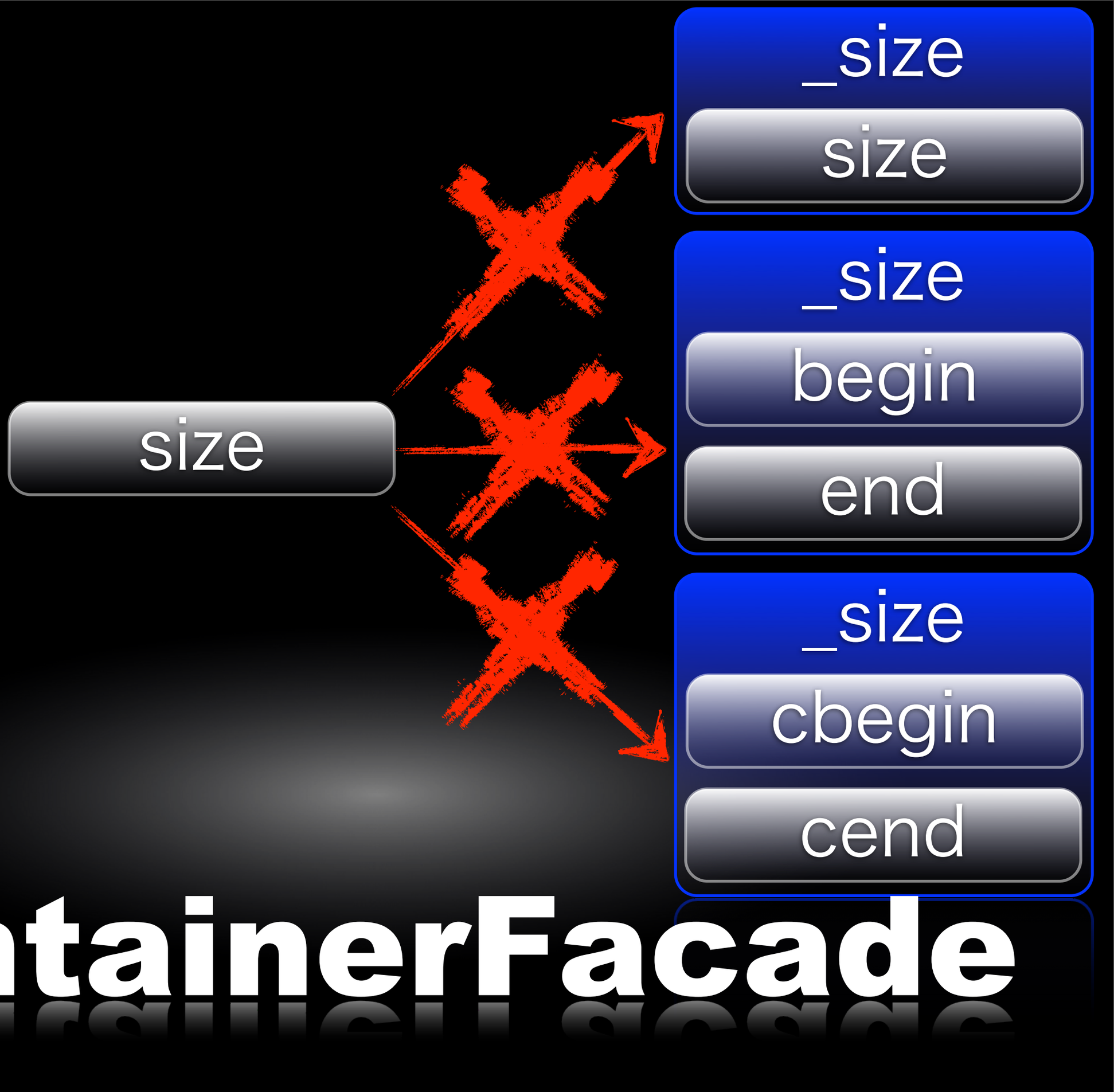
2011年11月5日土曜日

そこでContainerFacadeはこのような実装になっています。まず★この行のメンバ関数sizeが呼び出されます。sizeは第一引数にBase型のコンテナを付けて★staticなメンバ関数アンダースコアsizeに★処理を丸投げします。アンダースコアsizeはテンプレート関数で、★第一引数の型によってアンダースコアBaseの型が決定します。このアンダースコアsizeはenable_ifで分岐されており、関数呼び出し時に第一引数から決定したコンテナの型によって適切な方が選択されます。



2011年11月5日土曜日

このようなややこしい実装の結果別の問題が生じます。今、メンバ関数sizeを実現する方法がいくつか有ったとします。★コンテナがこれら全ての条件を満たしていない場合、全ての方法のenable_ifが成立しなくなります。しかし、★最初にそれぞれの方法に処理を丸投げするメンバ関数sizeは呼び出すとコンパイルエラーになるが、この関数自体は存在するという状態になる為、★ContainerFacadeに対してContainerTraitsを使うと、全ての関数が使用可能な状態という扱いになってしまいます。ContainerFacadeは内部でContainerTraitsを使っているため、これはContainerFacadeを何段も重ねることが出来ないことを意味します。



2011年11月5日土曜日

このようなややこしい実装の結果別の問題が生じます。今、メンバ関数sizeを実現する方法がいくつか有ったとします。★コンテナがこれら全ての条件を満たしていない場合、全ての方法のenable_ifが成立しなくなります。しかし、★最初にそれぞれの方法に処理を丸投げするメンバ関数sizeは呼び出すとコンパイルエラーになるが、この関数自体は存在するという状態になる為、★ContainerFacadeに対してContainerTraitsを使うと、全ての関数が使用可能な状態という扱いになってしまいます。ContainerFacadeは内部でContainerTraitsを使っているため、これはContainerFacadeを何段も重ねることが出来ないことを意味します。

2011年11月5日土曜日

このようなややこしい実装の結果別の問題が生じます。今、メンバ関数sizeを実現する方法がいくつか有ったとします。★コンテナがこれら全ての条件を満たしていない場合、全ての方法のenable_ifが成立しなくなります。しかし、★最初にそれぞれの方法に処理を丸投げするメンバ関数sizeは呼び出すとコンパイルエラーになるが、この関数自体は存在するという状態になる為、★ContainerFacadeに対してContainerTraitsを使うと、全ての関数が使用可能な状態という扱いになってしまいます。ContainerFacadeは内部でContainerTraitsを使っているため、これはContainerFacadeを何段も重ねることが出来ないことを意味します。

2011年11月5日土曜日

このようなややこしい実装の結果別の問題が生じます。今、メンバ関数sizeを実現する方法がいくつか有ったとします。★コンテナがこれら全ての条件を満たしていない場合、全ての方法のenable_ifが成立しなくなります。しかし、★最初にそれぞれの方法に処理を丸投げするメンバ関数sizeは呼び出すとコンパイルエラーになるが、この関数自体は存在するという状態になる為、★ContainerFacadeに対してContainerTraitsを使うと、全ての関数が使用可能な状態という扱いになってしまいます。ContainerFacadeは内部でContainerTraitsを使っているため、これはContainerFacadeを何段も重ねることが出来ないことを意味します。

チート

2011年11月5日土曜日

そうならないようにContainerTraits側に細工をします。調べるコンテナに★ContainerTraits型が定義されている場合、ContainerTraitsは実際にメンバ関数が存在するかどうかではなく、
★ContainerTraitsが提供する情報をもとにメンバ関数が使用可能かどうかを判断します。

```
template< typename Type >
class ContainerFacade {
public:
  typedef detail::AliasContainerTraits< Type > ContainerTraits;
};
```

チート

そうならないようにContainerTraits側に細工をします。調べるコンテナに★ContainerTraits型が定義されている場合、ContainerTraitsは実際にメンバ関数が存在するかどうかではなく、
★ContainerTraitsが提供する情報をもとにメンバ関数が使用可能かどうかを判断します。

```
template< typename Type >
class ContainerFacade {
public:
  typedef detail::AliasContainerTraits< Type > ContainerTraits;
};
```

ContainerTraits型がある場合
実際にメンバ関数が存在するかどうかではなく
ContainerTraitsが提供する情報をもとに
メンバ関数が使用可能かどうかを判断する

チート

2011年11月5日土曜日

そうならないようにContainerTraits側に細工をします。調べるコンテナに★ContainerTraits型が定義されている場合、ContainerTraitsは実際にメンバ関数が存在するかどうかではなく、
★ContainerTraitsが提供する情報をもとにメンバ関数が使用可能かどうかを判断します。

代用の代用

2011年11月5日土曜日

ContainerFacadeを何段も重ねる状況を想定しなければならないのには理由があります。★メンバ関数の代用は1回で済むとは限らないのです。例えば、今mergeが無いので★insert/erase/begin/endで代用しようとしたとします。しかしコンテナにはinsertも無く、★resize/size/begin/endで代用したくなるかもしれません。★resize/size/erase/begin/endがある場合について別で代用を用意すればそういったことも可能なのですが、こうした組み合わせは無数に有りそれら全てを人の手で用意するのは面倒です。

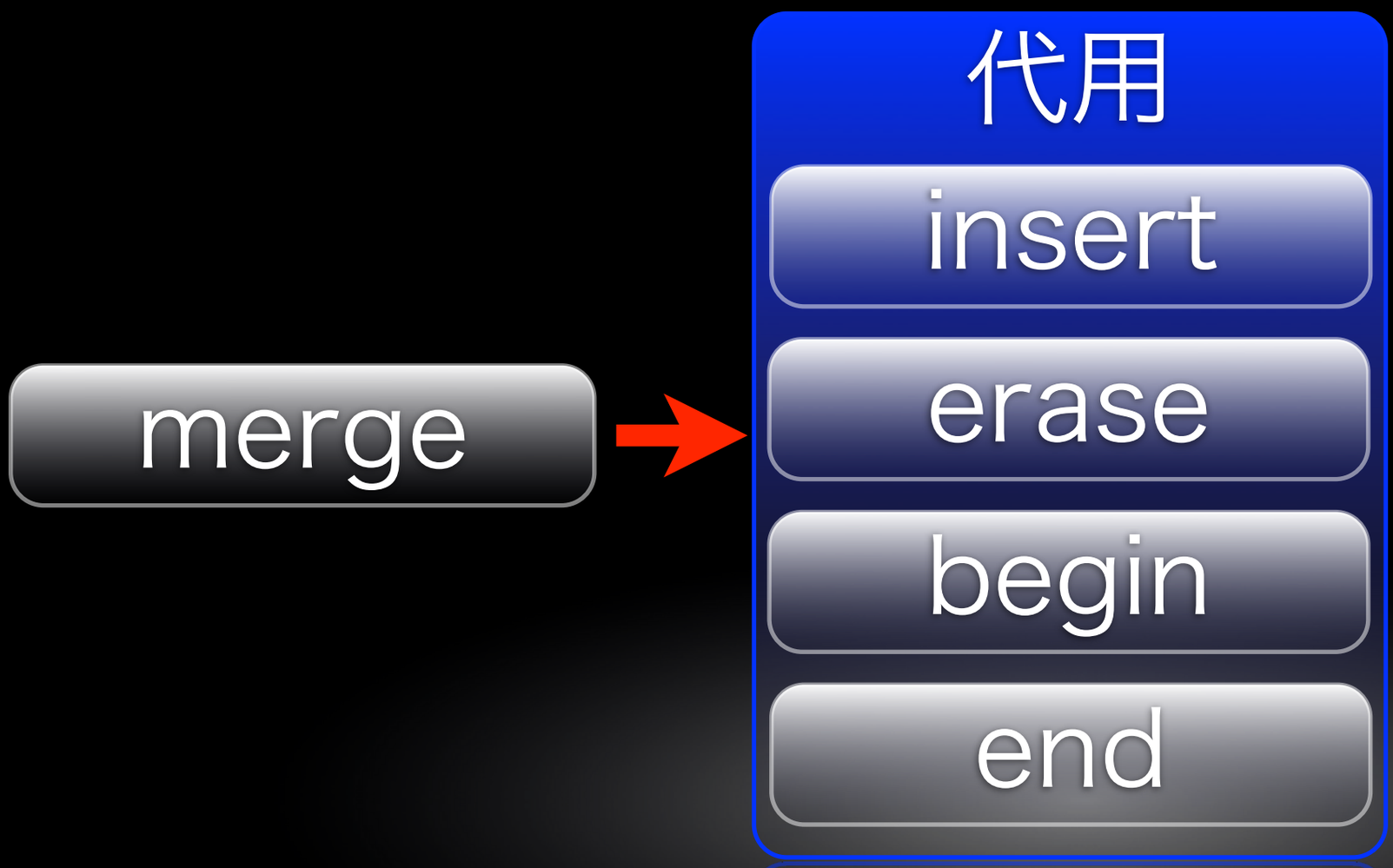
代用は1段階で済むとは限らない

代用の代用

2011年11月5日土曜日

ContainerFacadeを何段も重ねる状況を想定しなければならないのには理由があります。★メンバ関数の代用は1回で済むとは限らないのです。例えば、今mergeが無いので★insert/erase/begin/endで代用しようとしたとします。しかしコンテナにはinsertも無く、★resize/size/begin/endで代用したくなるかもしれません。★resize/size/erase/begin/endがある場合について別で代用を用意すればそういったことも可能なのですが、こうした組み合わせは無数に有りそれら全てを人の手で用意するのは面倒です。

代用は1段階で済むとは限らない

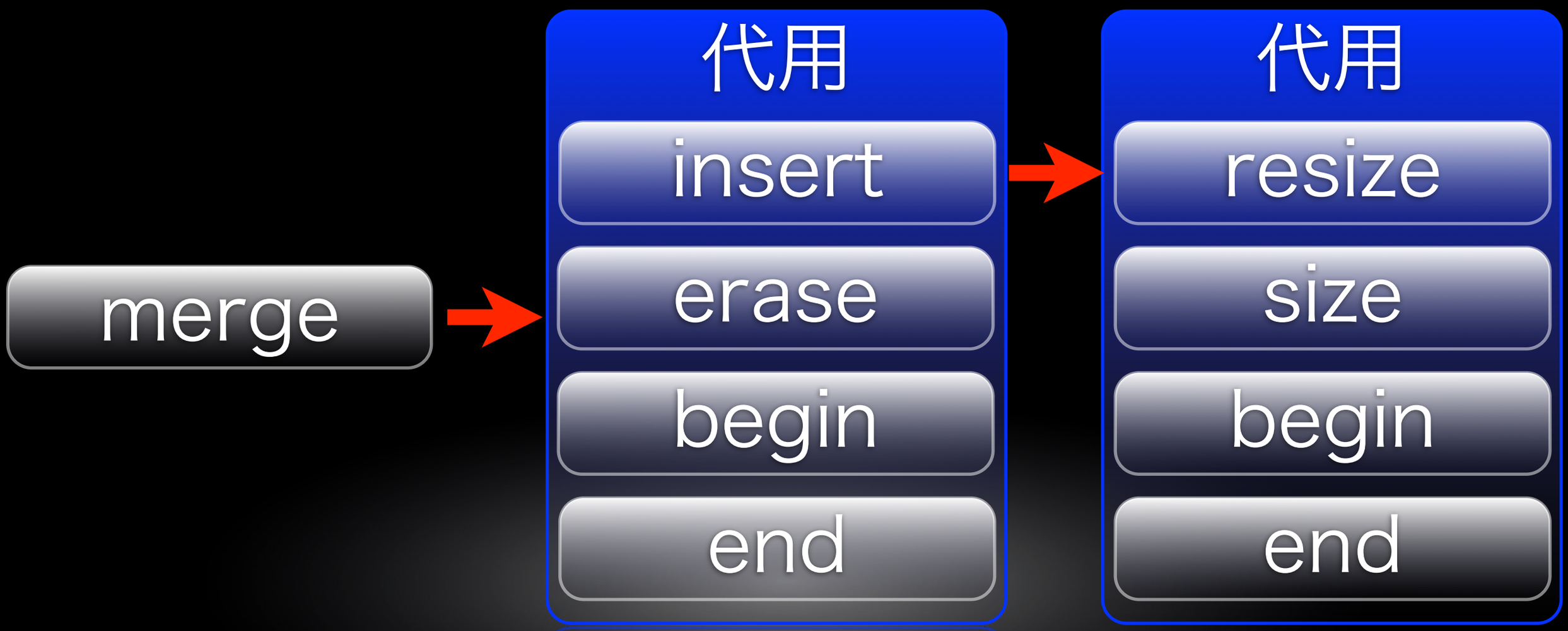


代用の代用

2011年11月5日土曜日

ContainerFacadeを何段も重ねる状況を想定しなければならないのには理由があります。★メンバ関数の代用は1回で済むとは限らないのです。例えば、今mergeが無いので★insert/erase/begin/endで代用しようとしたとします。しかしコンテナにはinsertも無く、★resize/size/begin/endで代用したくなるかもしれません。★resize/size/erase/begin/endがある場合について別で代用を用意すればそういったことも可能なのですが、こうした組み合わせは無数に有りそれら全てを人の手で用意するのは面倒です。

代用は1段階で済むとは限らない

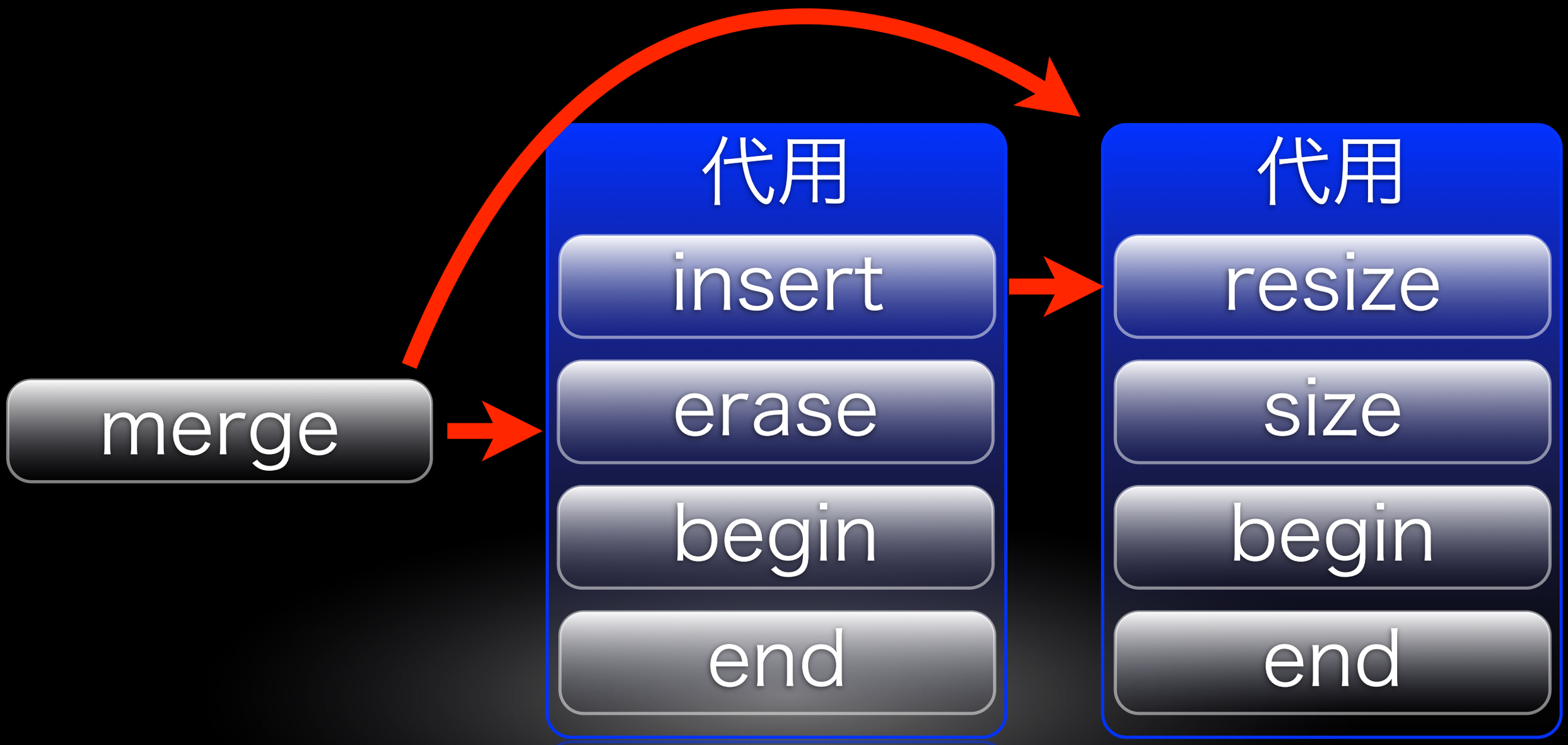


代用の代用

2011年11月5日土曜日

ContainerFacadeを何段も重ねる状況を想定しなければならないのには理由があります。★メンバ関数の代用は1回で済むとは限らないのです。例えば、今mergeが無いので★insert/erase/begin/endで代用しようとしたとします。しかしコンテナにはinsertも無く、★resize/size/begin/endで代用したくなるかもしれません。★resize/size/erase/begin/endがある場合について別で代用を用意すればそういったことも可能なのですが、こうした組み合わせは無数に有りそれら全てを人の手で用意するのは面倒です。

代用は1段階で済むとは限らない



代用の代用

2011年11月5日土曜日

ContainerFacadeを何段も重ねる状況を想定しなければならないのには理由があります。★メンバ関数の代用は1回で済むとは限らないのです。例えば、今mergeが無いので★insert/erase/begin/endで代用しようとしたとします。しかしコンテナにはinsertも無く、★resize/size/begin/endで代用したくなるかもしれません。★resize/size/erase/begin/endがある場合について別で代用を用意すればそういったことも可能なのですが、こうした組み合わせは無数に有りそれら全てを人の手で用意するのは面倒です。

```
template< typename Type >
class ContainerFacade :
  public detail::AliasLayer< detail::AliasLayer<
    detail::AliasLayer< detail::AliasLayer<
      detail::AliasLayer< detail::AliasLayer<
        Type > > > > > > {
public:
  typedef detail::AliasLayer< detail::AliasLayer<
    detail::AliasLayer< detail::AliasLayer<
      detail::AliasLayer< detail::AliasLayer<
        Type > > > > > > BaseType;
  MPP_CONTF_THROUGH_CONSTRUCTORS( 20, ContainerFacade, BaseType )
};
```

代用を実装するクラスを何段も重ねる

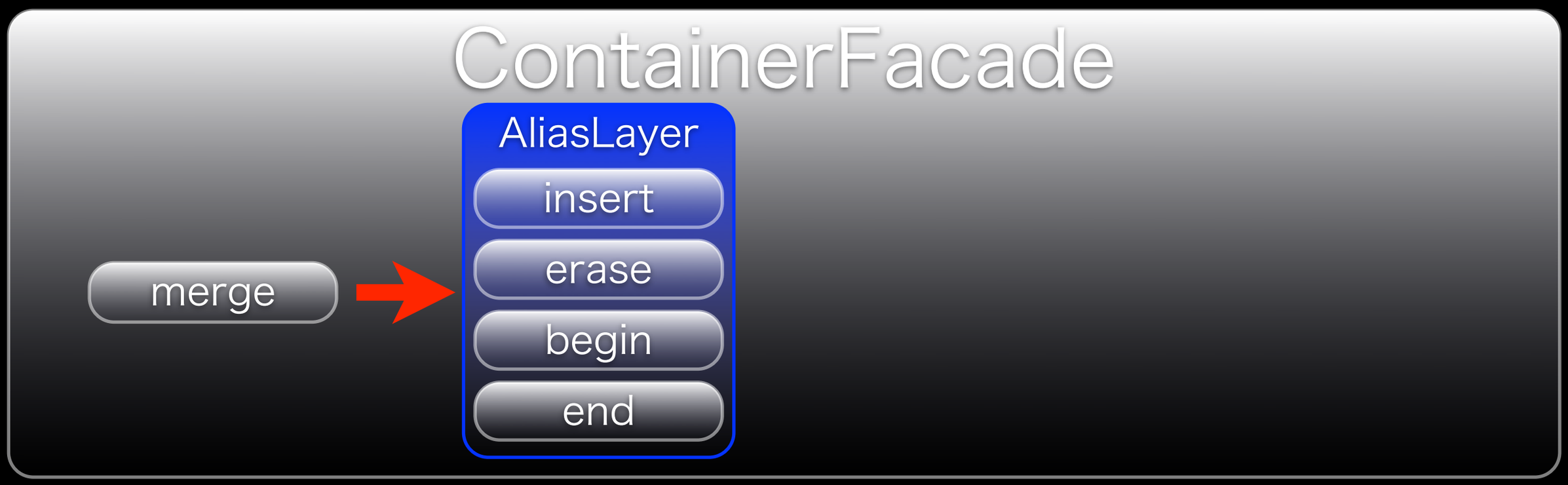
代用の代用

そこでContainerFacadeはこの代用を実装するクラスを何段も重ねたものを継承した形になっています。

代用の代用

2011年11月5日土曜日

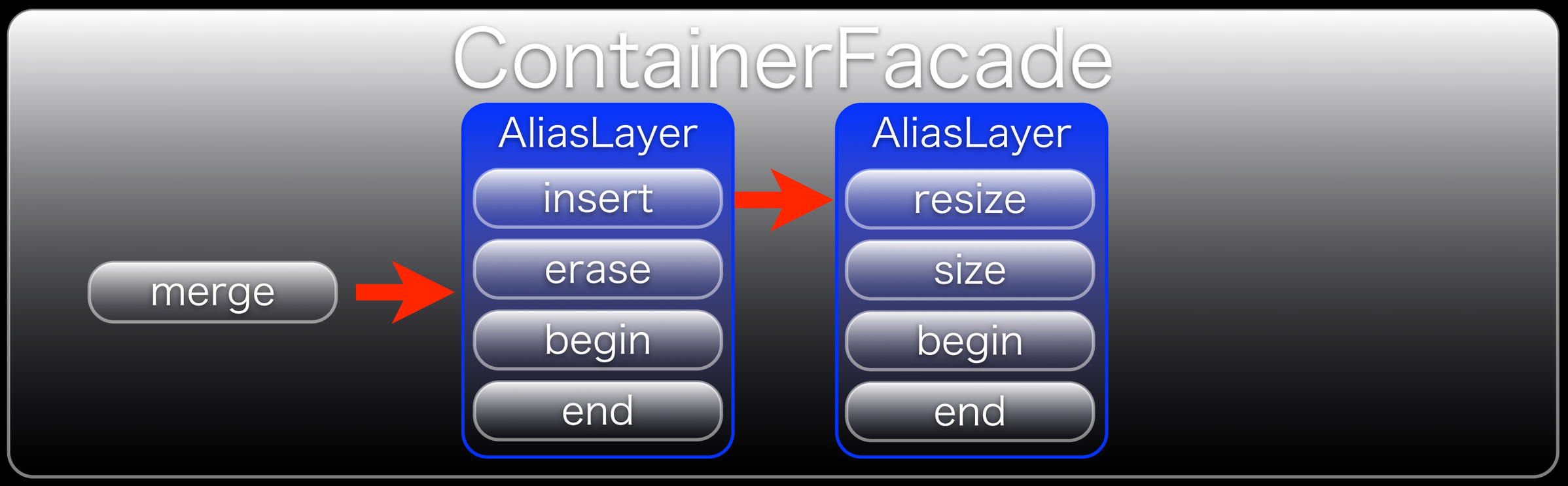
今メンバ関数mergeの無いコンテナに対して、ContainerFacadeを通してmergeが呼び出されました。★1段目のContainerFacadeはinsert/erase/begin/endを使ってmergeの代用をしようとしますが、困ったことにコンテナにはinsertも有りませんでした。★そこで2段目のContainerFacadeは1段目から要求されたinsertをresize/begin/endを使って代用しようとします。ところがコンテナにはresizeもありませんでした。★仕方ないので3段目のContainerFacadeは2段目から要求されたresizeをpush_back/pop_backで代用しようとします。結果としてmergeはpush_back/pop_back/erase/begin/endを使って代用されることになります。★この代用は最大で6段行われるようになっています。



代用の代用

2011年11月5日土曜日

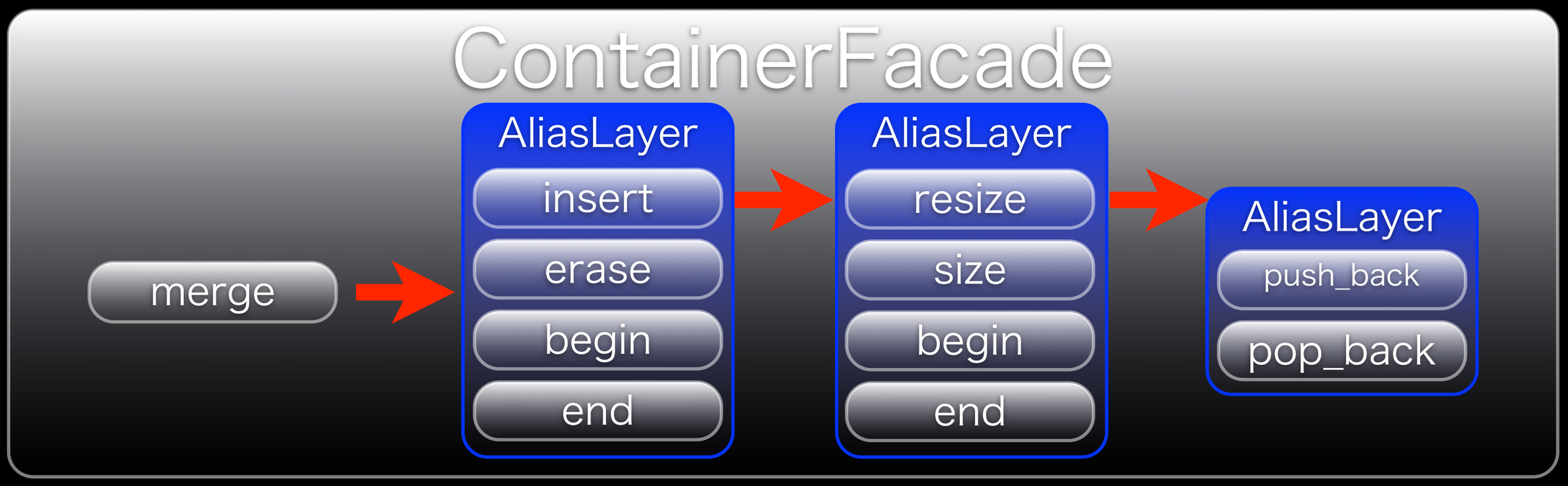
今メンバ関数mergeの無いコンテナに対して、ContainerFacadeを通してmergeが呼び出されました。★1段目のContainerFacadeはinsert/erase/begin/endを使ってmergeの代用をしようとしますが、困ったことにコンテナにはinsertも有りませんでした。★そこで2段目のContainerFacadeは1段目から要求されたinsertをresize/begin/endを使って代用しようとします。ところがコンテナにはresizeもありませんでした。★仕方ないので3段目のContainerFacadeは2段目から要求されたresizeをpush_back/pop_backで代用しようとします。結果としてmergeはpush_back/pop_back/erase/begin/endを使って代用されることになります。★この代用は最大で6段行われるようになっています。



代用の代用

2011年11月5日土曜日

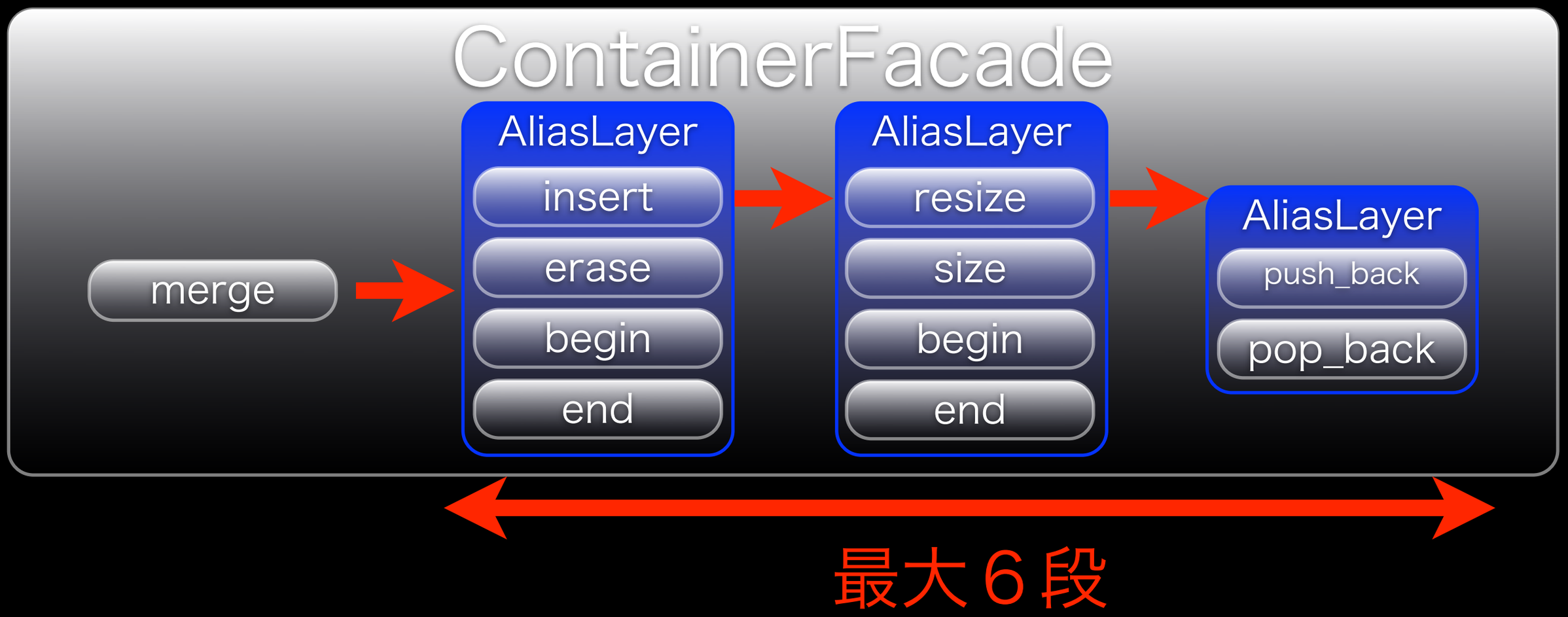
今メンバ関数mergeの無いコンテナに対して、ContainerFacadeを通してmergeが呼び出されました。★1段目のContainerFacadeはinsert/erase/begin/endを使ってmergeの代用をしようとしますが、困ったことにコンテナにはinsertも有りませんでした。★そこで2段目のContainerFacadeは1段目から要求されたinsertをresize/begin/endを使って代用しようとします。ところがコンテナにはresizeもありませんでした。★仕方ないので3段目のContainerFacadeは2段目から要求されたresizeをpush_back/pop_backで代用しようとします。結果としてmergeはpush_back/pop_back/erase/begin/endを使って代用されることになります。★この代用は最大で6段行われるようになっています。



代用の代用

2011年11月5日土曜日

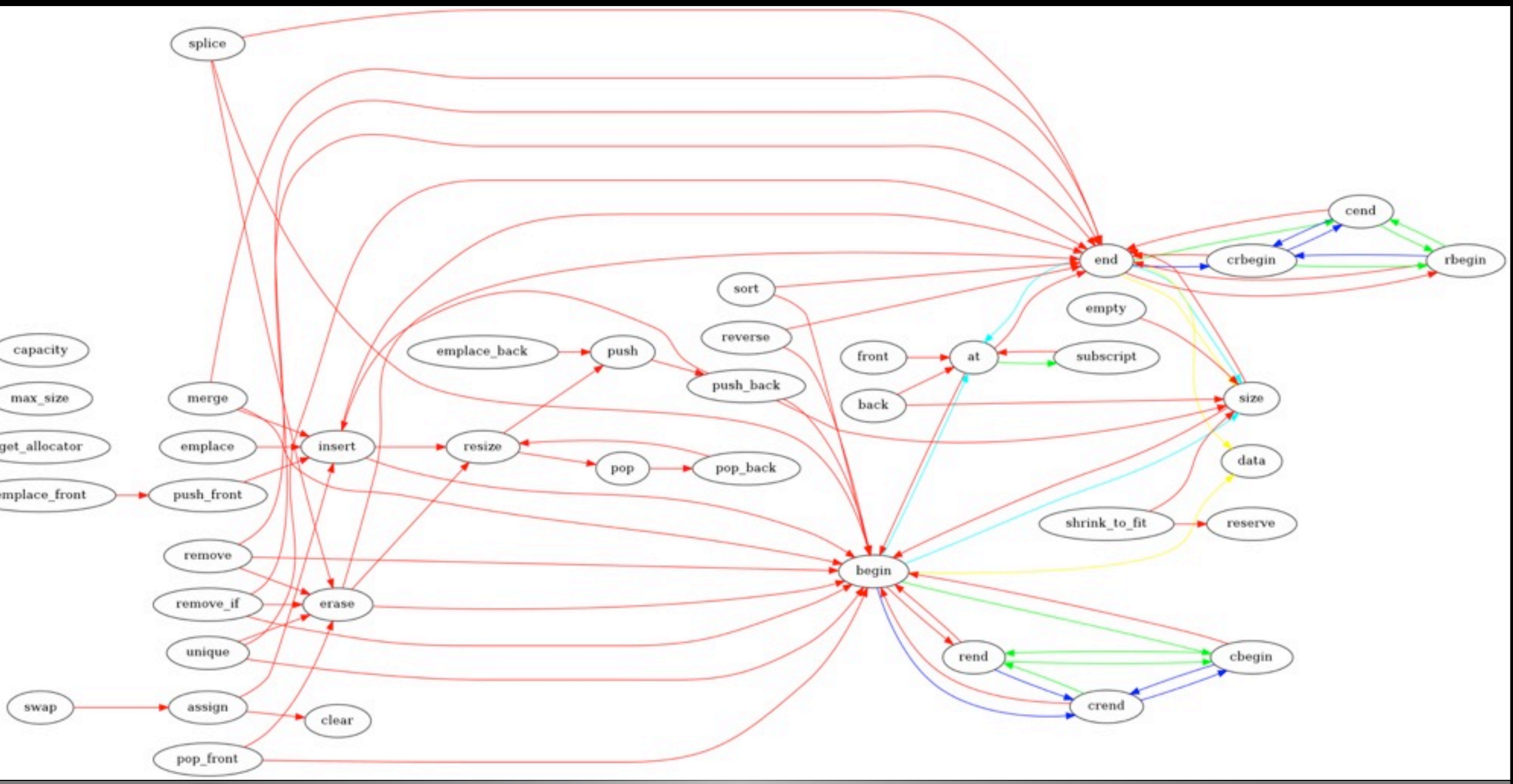
今メンバ関数mergeの無いコンテナに対して、ContainerFacadeを通してmergeが呼び出されました。★1段目のContainerFacadeはinsert/erase/begin/endを使ってmergeの代用をしようとしますが、困ったことにコンテナにはinsertも有りませんでした。★そこで2段目のContainerFacadeは1段目から要求されたinsertをresize/begin/endを使って代用しようとします。ところがコンテナにはresizeもありませんでした。★仕方ないので3段目のContainerFacadeは2段目から要求されたresizeをpush_back/pop_backで代用しようとします。結果としてmergeはpush_back/pop_back/erase/begin/endを使って代用されることになります。★この代用は最大で6段行われるようになっています。



代用の代用

2011年11月5日土曜日

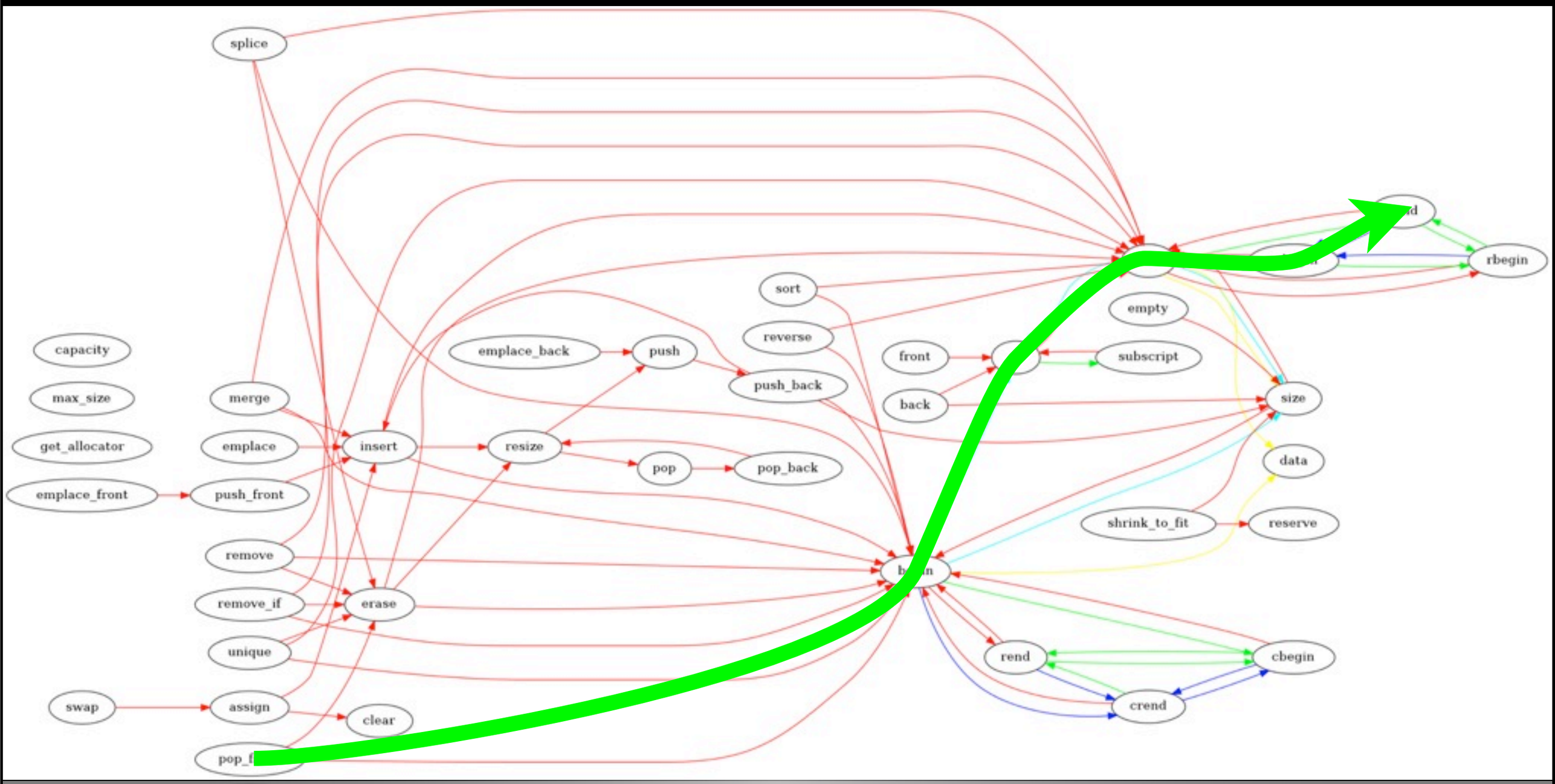
今メンバ関数mergeの無いコンテナに対して、ContainerFacadeを通してmergeが呼び出されました。★1段目のContainerFacadeはinsert/erase/begin/endを使ってmergeの代用をしようとしますが、困ったことにコンテナにはinsertも有りませんでした。★そこで2段目のContainerFacadeは1段目から要求されたinsertをresize/begin/endを使って代用しようとします。ところがコンテナにはresizeもありませんでした。★仕方ないので3段目のContainerFacadeは2段目から要求されたresizeをpush_back/pop_backで代用しようとします。結果としてmergeはpush_back/pop_back/erase/begin/endを使って代用されることになります。★この代用は最大で6段行われるようになっています。



再帰的な代用

2011年11月5日土曜日

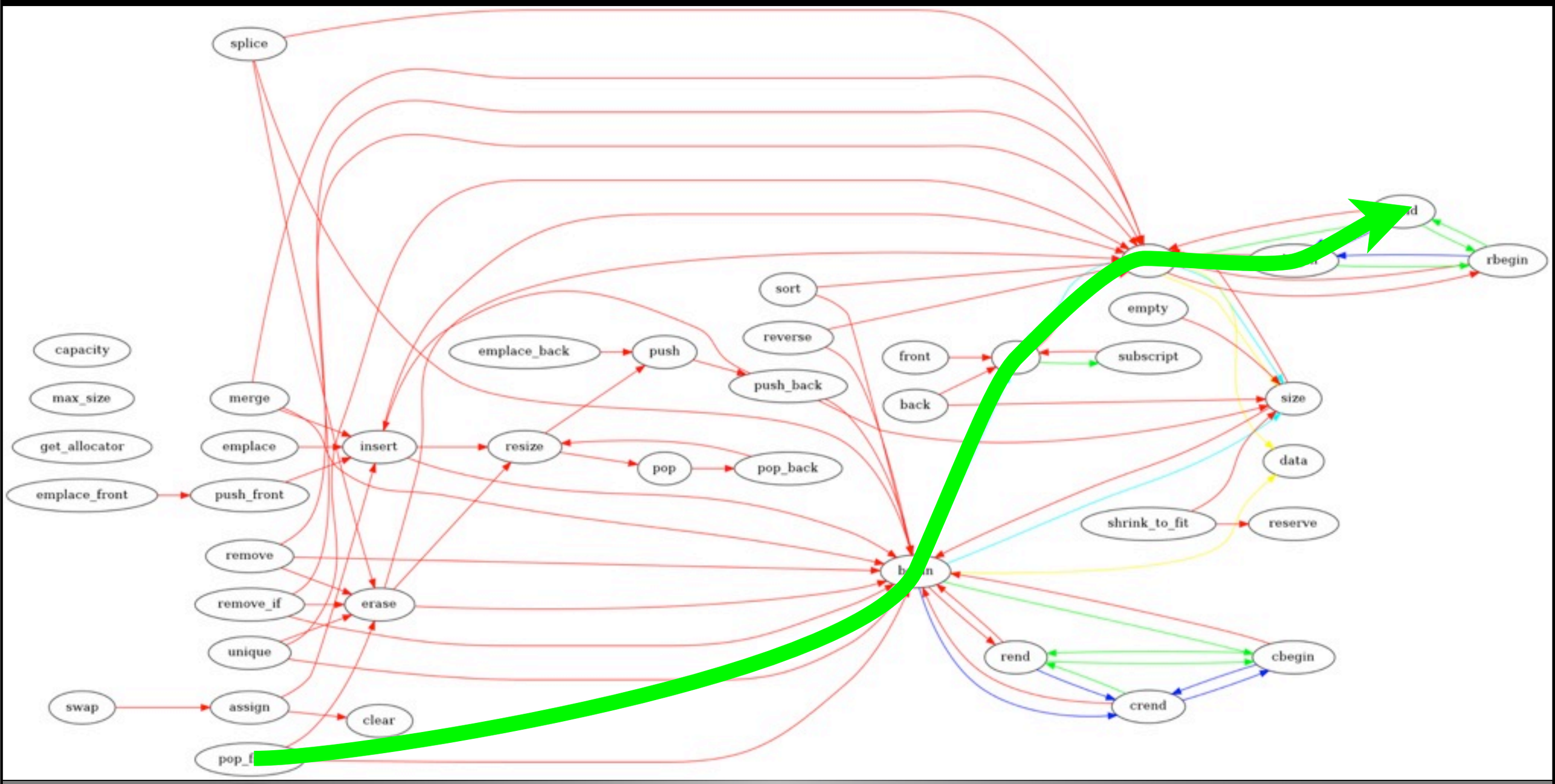
何故6段なのでしょうか。これはContainerFacadeを作るにあたって描いたSTLスタイルのコンテナが持つ主要な関数の「無かった場合に代用に使える関数」をまとめた有向グラフです。★このグラフから代用は最長でも緑の線で示した5段です。なので、ここで予想に漏れているものが有ったとしても★6段あれば足りるだろうと予想しました。ちなみにこのグラフには既にいくつかの間違いがあることが分かっており、本当に6段で十分か、あるいは6段も必要なのかについては再検討が必要かもしれません。



再帰的な代用

2011年11月5日土曜日

何故6段なのでしょうか。これはContainerFacadeを作るにあたって描いたSTLスタイルのコンテナが持つ主要な関数の「無かった場合に代用に使える関数」をまとめた有向グラフです。★このグラフから代用は最長でも緑の線で示した5段です。なので、ここで予想に漏れているものが有ったとしても★6段あれば足りるだろうと予想しました。ちなみにこのグラフには既にいくつかの間違いがあることが分かっており、本当に6段で十分か、あるいは6段も必要なのかについては再検討が必要かもしれません。



6段もあれば十分なんじゃないかな

再帰的な代用

2011年11月5日土曜日

何故6段なのでしょうか。これはContainerFacadeを作るにあたって描いたSTLスタイルのコンテナが持つ主要な関数の「無かった場合に代用に使える関数」をまとめた有向グラフです。★このグラフから代用は最長でも緑の線で示した5段です。なので、ここで予想に漏れているものが有ったとしても★6段あれば足りるだろうと予想しました。ちなみにこのグラフには既にいくつかの間違いがあることが分かっており、本当に6段で十分か、あるいは6段も必要なのかについては再検討が必要かもしれません。

実際の効果

2011年11月5日土曜日

それではContainerFacadeの効果を見てみましょう。★今、beginとendしかない残念なシーケンスコンテナが有ったとします。これをContainerFacadeに通すと、★添字演算子オーバーロード/at/front/back/top/rbegin/rend/empty/sizeが追加されます。ContainerFacadeはコンテナが提供するイテレータがbidirectional_traversalまたはrandom_access_traversalであることを期待してrbegin/rendをでっち上げるようになっているのですが、forward_traversalの場合はこれらの関数を追加しようとしないようにした方が良いかもしれません。

元のコンテナのメンバ関数

begin end

実際の効果

2011年11月5日土曜日

それではContainerFacadeの効果を見てみましょう。★今、beginとendしかない残念なシーケンスコンテナが有ったとします。これをContainerFacadeに通すと、★添字演算子オーバーロード/at/front/back/top/rbegin/rend/empty/sizeが追加されます。ContainerFacadeはコンテナが提供するイテレータがbidirectional_traversalまたはrandom_access_traversalであることを期待してrbegin/rendをでっち上げるようになっているのですが、forward_traversalの場合はこれらの関数を追加しようとしないようにした方が良いかもしれません。

実際の効果

2011年11月5日土曜日

それではContainerFacadeの効果を見てみましょう。★今、beginとendしかない残念なシーケンスコンテナが有ったとします。これをContainerFacadeに通すと、★添字演算子オーバーロード/at/front/back/top/rbegin/rend/empty/sizeが追加されます。ContainerFacadeはコンテナが提供するイテレータがbidirectional_traversalまたはrandom_access_traversalであることを期待してrbegin/rendをでっち上げるようになっているのですが、forward_traversalの場合はこれらの関数を追加しようとしないようにした方が良いかもしれません。

実際の効果

2011年11月5日土曜日

★resizeが使用可能な場合★要素数の増減を伴うメンバ関数も使えるようになります。

元のコンテナのメンバ関数

begin　end　resize

実際の効果

★resizeが使用可能な場合★要素数の増減を伴うメンバ関数も使えるようになります。

元のコンテナのメンバ関数

begin　end　resize

ContainerFacadeを通して使えるメンバ関数

operator[]　at sequence　front

back　top　begin

end　rbegin　rend

empty　size　clear

insert sequence　erase sequece　push_back

pop_back　push_front　pop_front

resize　merge

実際の効果

★resizeが使用可能な場合★要素数の増減を伴うメンバ関数も使えるようになります。

Sequence Containerの場合

size と operator[] があれば

コンテナの要素にランダムアクセスが可能だが

iteratorがあった方が何かと便利

iterator_emulator

2011年11月5日土曜日

ところで、シーケンスコンテナの場合、sizeと添字演算子オーバーロードさえあれば、コンテナの全ての要素にランダムアクセスが可能です。しかしこの2つのメンバ関数しかない場合イテレータが無いため、何かと不便です。そこで、★sizeと添字演算子オーバーロードがあるが、イテレータの無いコンテナに対して、ContainerFacade内でイテレータをでっち上げてしまいましょう。

Sequence Containerの場合

size と operator[] があれば

コンテナの要素にランダムアクセスが可能だが

iteratorがあった方が何かと便利

size と operator[] から

begin と end を作る

iterator_emulator

2011年11月5日土曜日

ところで、シーケンスコンテナの場合、sizeと添字演算子オーバーロードさえあれば、コンテナの全ての要素にランダムアクセスが可能です。しかしこの2つのメンバ関数しかない場合イテレータが無いため、何かと不便です。そこで、★sizeと添字演算子オーバーロードがあるが、イテレータの無いコンテナに対して、ContainerFacade内でイテレータをでっち上げてしまいましょう。

```
template< typename Type, typename value_type = typename Type::value_type >
class iterator_emulator : public boost::iterator_facade<
  iterator_emulator< Type, value_type >, value_type, boost::random_access_traversal_tag >{
public:
  iterator_emulator( Type *_container ) : container( _container ), position( 0 ) {}
  value_type &dereference() const { return container->at( position ); }
  void increment() { ++position; }
  void decrement() { --position; }
  bool equal( const iterator_emulator< Type, value_type > &_right ) const {
    return container == _right.container && position == _right.position;
  }
  void advance( int _count ) { position += _count; }
  int distance_to( const iterator_emulator< Type, value_type > &_right ) const {
    return _right.position - position;
  }
private:
  Type *container;
  int position;
};
```

iterator_emulator

もとになるコンテナにイテレータが定義されていない場合、ContainerFacadeはiterator_emulatorをコンテナのイテレータとして定義します。iterator_emulatorは今示しているようなシンプルなコードで、コンテナへのポインタと、今指している要素が何要素目かを保持しています。デリファレンス時はコンテナに対して添字演算を行います。

実際の効果

2011年11月5日土曜日

このiterator_emulatorの働きにより、元のコンテナに★このようにsizeと添字演算子オーバーロードしかなかったとしても、★ContainerFacadeを通すと赤で示したようなイテレータの取得が可能なコンテナになります。

元のコンテナのメンバ関数

operator[] | const operator[] | size

実際の効果

このiterator_emulatorの働きにより、元のコンテナに★このようにsizeと添字演算子オーバーロードしかなかったとしても、★ContainerFacadeを通すと赤で示したようなイテレータの取得が可能なコンテナになります。

実際の効果

2011年11月5日土曜日

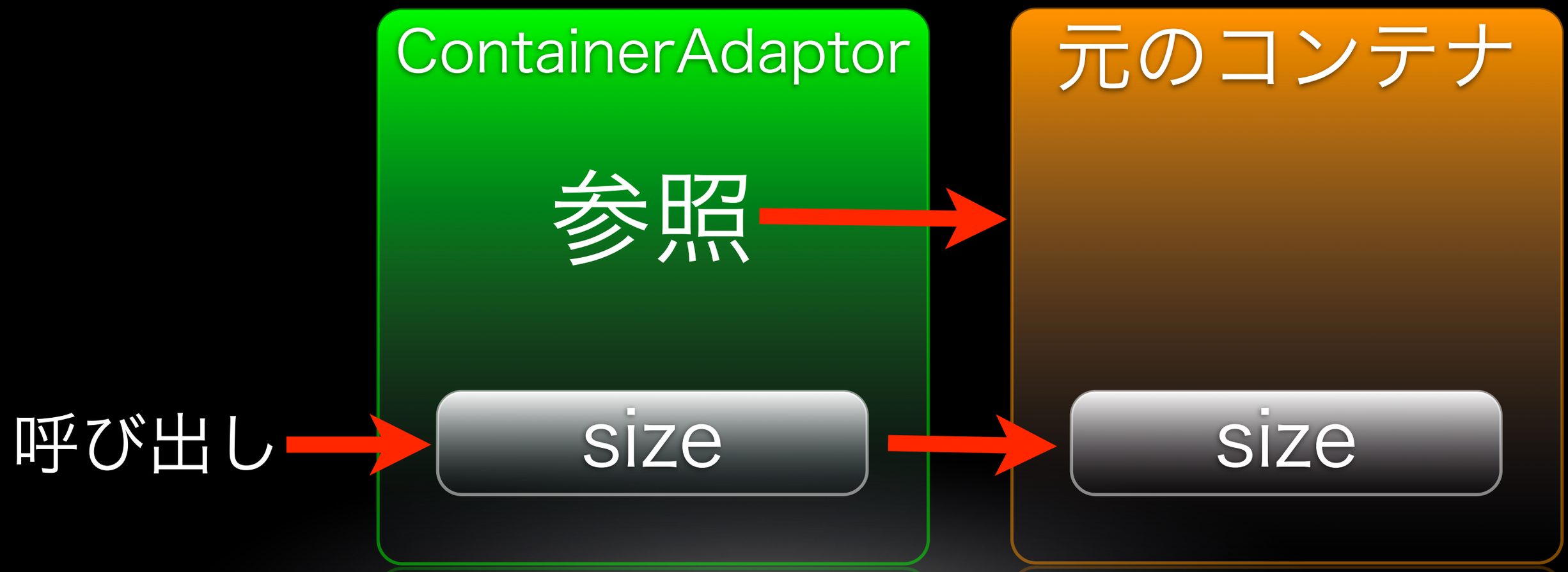
このiterator_emulatorの働きにより、元のコンテナに★このようにsizeと添字演算子オーバーロードしかなかったとしても、★ContainerFacadeを通すと赤で示したようなイテレータの取得が可能なコンテナになります。

既にあるコンテナに対して、
一時的にメンバ関数の代用を行いたい

ContainerAdaptor

2011年11月5日土曜日

ここまで見てきたContainerFacadeはそのインスタンスの中に土台となるコンテナのインスタンスを含んでいるため、元々土台となるコンテナが使われていた箇所を置き換える形で使うことになります。しかしどちらかと言うと、使用するコンテナは既に決まっていて置き換えることができず、アルゴリズムに通す時だけ一時的にContainerFacadeをかぶせたい、というケースの方が多いでしょう。その為の仕組みがContainerAdaptorです。ちなみにContainerAdaptorはiterator_facadeに対するiterator_adaptorと揃えて付けた名前ですが、コンテナの分類におけるContainerAdaptorと名前がかぶっていて紛らわしいので、そのうち名前を変えるかもしれません。



ContainerAdaptor

2011年11月5日土曜日

ContainerAdaptorはコンストラクタの引数として土台となるコンテナへの参照またはポインタのようにデリファレンスが可能な型をとります。ContainerAdaptorはSTLスタイルのコンテナのメンバ関数を備えており、呼び出すとそっくりそのまま参照先のコンテナに処理を丸投げします。なので、ContainerFacadeの土台となるコンテナをContainerAdaptorにすることで、既にインスタンス化されたコンテナに対して一時的にContainerFacadeのガワをかぶせることができます。

```
template< typename Cont >
void merge( Cont &_cont1, Cont &_cont2 ) {
  ContainerFacade< ContainerAdaptor< Cont > >
    cla( _cont1 );
  cla.merge( _cont2 );
}
```

mergeの実装

これを使って任意のコンテナに対してアルゴリズムmergeを実装すると今示しているようになります。

```
template< typename Bottom >
void merge( Bottom &_cont ) {
  _merge( getBase(), _cont );
}
```

mergeの実装

アルゴリズムから呼ばれたメンバ関数mergeは★staticなメンバ関数アンダースコアmergeを呼び出し、enable_ifによって適切な方法が選ばれます。

```
template< typename Bottom >
void merge( Bottom &_cont ) {
  _merge( getBase(), _cont );
}
```

```
template< typename Base, typename Bottom >
void _merge( Base &_base, typename Bottom &_cont,
  typename enable_if< has_merge< Base > >::type* = 0 ) {
  _base.merge( _cont2 );
}

template< typename Base, typename Bottom >
void _merge( Base &_base, typename Bottom &_cont,
  typename enable_if< mpl::bool_< !has_merge< Base >::value &&
    has_insert< Base >::value && has_erase< Base >::value &&
    has_begin< Base >::value && has_end< Base >::value > >::type* = 0 ) {
  (略)
}
```

mergeの実装

アルゴリズムから呼ばれたメンバ関数mergeは★staticなメンバ関数アンダースコアmergeを呼び出し、enable_ifによって適切な方法が選ばれます。

	Traits	Facade	Adaptor	Alternate	Algorithm
Common					
Sequence					
Associative					
Unordered Associative					
Property Tree					
Multi Index					
Concurrent					

実装状況

2011年11月5日土曜日

ContainerFacadeはまだ開発途中のライブラリです。メンバ関数の代用にはまだ未実装のものが多数あり、アルゴリズムに至っては全く実装されていません。今のところシーケンスコンテナとアソシエイティブコンテナに絞って実装を進めていますが、将来的にはアンオーダードアソシエイティブコンテナや、Boost.PropertyTree、Boost.MultiIndex、Intel TBB Concurrentコンテナ固有のメンバ関数にも対応したいと考えています。また、Boost.Preprocessorの使いすぎてコンパイル時間がとてもBoostするので、事前にプリプロセッサを通したヘッダを使う等の対策を考える必要がありそうです。

コンテナの型に依存せず、
サイズ変更を伴うアルゴリズムを実装することは
できないこともない

まとめ

まとめとしては、やろうと思えばイテレータを使わず、コンテナの型に依存しない、サイズ変更を伴うようなアルゴリズムを実装することは、出来なくもないということが言えようかと思います。

http://bit.ly/sdPNYC

ソースコード

2011年11月5日土曜日

ContainerFacadeを含む実験ライブラリのソースコードはここで公開しています。

ご清聴ありがとうございました

ご清聴ありがとうございました