

```
char* argv[]
```

という形でも書かれる。これは `char**`と同じである。この2つが等しいのはパラメータリストとしてだけであることに注意。

入力-出力:

<code>cerr</code>	標準エラーストリーム。出力はバッファされない。
<code>clog</code>	ログのための標準エラー。出力はバッファされる。
<code>ifstream(cp)</code>	<code>char* cp</code> で示される名前のファイルに結び付けられた入力ストリーム。 <code>istream</code> と同様の操作ができる。
<code>ofstream(cp)</code>	<code>char* cp</code> で示される名前のファイルに結び付けられた出力ストリーム。 <code>ostream</code> と同様の操作ができる。入力・出力ファイルストリームは <code><fstream></code> ヘッダに定義されている。

メモリ管理:

<code>new T</code>	メモリを確保し、 <code>T</code> 型の新しいオブジェクトをデフォルト初期化で生成し、そのオブジェクトへのポインタを戻す。
<code>new T(args)</code>	メモリを確保し、 <code>T</code> 型の新しいオブジェクトを引数 <code>args</code> を使って初期化・生成し、そのオブジェクトへのポインタを戻す
<code>delete p</code>	<code>p</code> が指すオブジェクトを破棄し、 <code>*p</code> が使っていたメモリを解放する。このポインタは動的に生成されたオブジェクトを指すものでなければならない。
<code>new T[n]</code>	メモリを確保し、 <code>T</code> 型のデフォルト初期化された要素を <code>n</code> 個持つ配列を生成し、その最初の要素へのポインタを戻す。
<code>delete[] p</code>	<code>p</code> が指す配列のオブジェクトを破棄し、そのメモリを解放する。ポインタは動的に生成された配列の最初の要素を指すものでなければならない。

課題

10-0 この章のプログラムをコンパイルし、実行し、試してください。

10-1 § 9.6 の成績処理プログラムを ABC 評価を出力するものに書き換えてください。

10-2 § 8.1.1 の `median` 関数を `vector` に対しても組み込みの配列に対しても使えるものに書き換えてください。また、どのような数値型のコンテナでも扱えるようにしてください。

10-3 上の問の `median` 関数が正しく動作するかを確かめるテストプログラムを書いてください。特に、`median` 関数を使ってもコンテナ内の要素の順番は変わることを確かめてください。

10-4 `string` を保持するリスト構造のクラスを書いてください。

10-5 上の問のリストクラス `String_list`に対する双方向反復子を書いてください。

10-6 上の問の反復子を使って、`split` 関数を、出力が `String_list` になるように書き直し、反復子が正しく機能することを確かめてください。

抽象データ型を定義する

第9章では新しい型を定義するための言語的な仕組みを少し見ました。しかし、そこで定義した `Student_info` では、オブジェクトがコピーされるときや代入されるとき、また破棄されるときに何をするべきかは決めませんでした。この章で見るよう、クラスの制作者はこのようなときのオブジェクトの振る舞いをコントロールすることができるのです。少し驚くかもしれません、直感的で使いやすい型を作るにはこれらを正確に定義しなければならないのです。

ここまで何度も `vector` を使ってきていたので、同様のクラスを作り、クラスのデザインと実装がどのようなものか見てみることにしましょう。ただし、ここで紹介する実装は、考える操作を大いに単純化したもので、標準ライブラリの `vector` の機能を減らしていくたクラスなので、混乱しないように `Vec` という名前にします。主な注意点は、`Vec` クラスのオブジェクトのコピー、代入、破棄の方法ですが、より簡単なメンバ関数の実装から始めた方がわかりやすいでしょう。これらを完成させてから、クラスオブジェクトのコピー、代入、破棄のコントロールを再検討することにします。

11.1 Vec クラス

クラスをデザインするときは、提供するインターフェースの詳細から考え始めます。正しいインターフェースを決める1つの方法は、そのクラスを使ってユーザ（クラスを使うプログラマ）がどのようなプログラムを書けるかを考えることです。ここでは、標準の `vector` の有用な1部のみを実装したいので、これまで `vector` をどのように使ってきたかを思い出してみましょう。

```
// vectorオブジェクトの生成
vector<Student_info> vs; // 空のvector
vector<double> v(100); // 100要素のvector
// vectorの使う名前を得る
vector<Student_info>::const_iterator b, e;
vector<Student_info>::size_type i = 0;
// size関数とインデックス演算子を使いvectorの各要素を見る
for (i = 0; i != vs.size(); ++i)
    cout << vs[i].name();
// 最初の要素や最後の要素の1つ後を指す反復子を戻す
b = vs.begin(); e = vs.end();
```

もちろん、これらの操作は標準の `vector` が提供するもの一部にすぎません。しかし、この一部を実装することで、`vector` のインターフェースの大部分を構成するのに必要な言語の仕組みを理解できるのです。

11.2 Vec クラスの実装

インターフェースの仕様を決定したので、次に `Vec` を具体的にどのように表現するかを考えます。

いちばん簡単な決断は、テンプレートクラス (template class) を定義するということです。ユーザが `Vec` にいろいろな型のものを格納できるようにしたいのです。§ 8.1.1 では関数のテンプレートを紹介しましたが同様のことがクラスについてもできます。関数の場合、テンプレート関数を 1 つ書いて、それからいろいろな型の値に対する関数を作り出しました。同様に、テンプレートクラスを定義すると、テンプレートのパラメータリストの型だけ違うクラスをテンプレートから作り出せるのです。実際、`vector`、`list`、`map`などを使った例を見てみます。

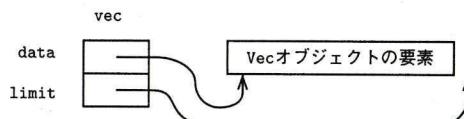
テンプレート関数のように、テンプレートクラスを定義するときは、これがテンプレートであるという印とクラスの定義の中で使う型のパラメータのリストを付ける必要があります。

```
template <class T> class Vec {
public:
    // インタフェース
private:
    // 実装
};
```

このコードは `Vec` がテンプレートクラスで、`T` という型パラメータを使うことを示しています。他のクラスの場合と同様に、`public` 部分と `private` 部分があり、それぞれがインターフェースと実装を表しています。

次に考えなければならないことは、保持するデータです。たぶん、`Vec` に要素を保持する場所が必要でしょう。また、`Vec` が保持している要素の数もわかるようにしておきたいのです。一番ストレートな方法は、動的に生成する配列で要素を保持するというものです。

配列に関してはどのような情報を持つべきでしょうか。`begin` と `end` と `size` 関数を付けたいでした。これは、最初の要素のアドレス、最後の要素の 1 つ後のアドレス、要素数を記憶しておく必要があるということです。しかし、これら 3 つのうち 1 つは他の 2 つから計算できるので、3 つともその値を保持しておく必要はありません。そのため、ここでは、最初の要素を指すポインタと最後の要素の 1 つ後を指すポインタをデータとして持つことにします。要素数は必要なときに計算するのです。これを図示すると次のようになります。



このように実装法を決めると、`Vec` クラスの中身は次のようにになります。

```
template <class T> class Vec {
public:
    // インタフェース
private:
    T* data; // Vecの最初の要素を指すポインタ
```

11.2 Vec クラスの実装

```
T* limit; // Vecの最後の要素の1つあとを指すポインタ
```

このクラス定義は、`Vec` がテンプレート型でパラメータが 1 つということを示しています。定義の中では、この型は `T` とよばれます。そしてこの `T` は、コンパイラによって、`Vec` の生成時にユーザの与えた名前に置き換えられます。つまり、たとえば、

```
Vec<int> v;
```

と書くと、コンパイラが、内部ですべての `T` を `int` に置き換えたバージョンの `Vec` を具体化 (§ 8.1.2) するのです。このコードはすべてのエクスプレッション中の `T` が最初から `int` であったように生成されます。`data` と `limit` の定義で `T` を使っていますが、この `T` は `Vec` オブジェクトの生成時に与えられる型 `int` になるのです。

`T` の具体的な型は `Vec` オブジェクトが生成されるまでわかりません。そして、一度 `Vec<int>`などと書けば、`data` と `limit` の型は、今の場合、`int*`になるわけです。同様に、`Vec<string>`と書けば、コンパイラは `T` を `string` にした別のバージョンの `Vec` を生成します。このとき、`data` と `limit` の型は `string*`になります。

11.2.1 メモリ確保

このクラスでは配列を動的 (§ 10.6.2) に生成します。そのため、`n` を整数として、`new T[n]` を使ってメモリを確保すると考えるかもしれません。しかし、`new T[n]` はメモリを確保するだけでなく、`T` のデフォルトコンストラクタを使って各要素の初期化をします。したがって、`new T[n]` を内部で使うと、`Vec<T>` が使えるのは `T` がデフォルトコンストラクタを持つときのみになってしまいます。標準の `vector` にはこのような制限はありません。今は `vector` を真似ているので、やはりこのような制限はなくしましょう。

実は、ライブラリはメモリ確保については、より細かいコントロールが可能なメモリ確保クラスを用意しています。これはまさに期待されるように動作するクラスで、`new` と `delete` の代わりに使うことができます。しかし、今そのクラスの詳細を見るより、最後にメモリ管理に便利な関数を書くので、そのとき見ることになります。今は、そのような関数があるとし、それによって `Vec` のコードが書けると仮定して、先に進みましょう。使い方を見ることで、何をしたいかがよりわかってきます。そして、実装のときには、どうすればよいかがわかつているはずです。

このようなユーティリティ関数は、このクラスの `private` 部分になります。これらは必要なメモリの確保と解放、`Vec` の要素の初期化と破棄を行います。また、これらはポインタ `data` と `limit` を処理してくれます。そしてメモリ管理関数だけが、これらの値を決められるのです。`Vec` の `public` メンバは、`data` と `limit` を読むことはできても、直接変更することはできないようにします。

新しい `Vec` オブジェクトの生成のように、`public` なメンバが `data` と `limit` の値を変える必要がでてきたときには、適当なメモリ管理のユーティリティ関数を使うことにします。このような方法に従うと、するべきことが 2 つに分かれることになります。1 つはユーザの使うインターフェースのためのメンバを作ることであり、もう 1 つは実装の詳細を考えるということです。

ユーティリティ関数の定義は § 11.5 で考えます。

11.2.2 コンストラクタ

コンストラクタは2つ必要です。というのは、次のような生成法が考えられるからです。

```
Vec<Student_info> vs; // デフォルトコンストラクタ
Vec<double> vs(100); // サイズを引数に取るコンストラクタ
```

標準のvectorには、同様の生成法と、すべての要素をそれで初期化する値とサイズを引数に取る第3のコンストラクタがあります。これはサイズを引数に取るコンストラクタと似ているので難しくはありません。Vecさて、どのコンストラクタもオブジェクトが正しく初期化されていることを保証しなければなりません。Vecのオブジェクトに対しては、dataとlimitの初期化を考えなければならないのです。そうするためにには、Vecの要素を保持するメモリを確保し、各要素を適当な値に初期化する必要があります。ただし、デフォルトコンストラクタの場合、空のVecを生成したいので、メモリを確保する必要はありません。サイズを引数に取るコンストラクタは、必要なメモリを確保しなければなりません。もし、ユーザが値とサイズを与えれば、その値を使ってメモリ上に生成したすべての要素を初期化しなければなりません。しかし、ユーザがサイズしか与えなかった場合、これらの要素を初期化するためにTのデフォルトコンストラクタを使います。今は、後で書くメモリ管理関数を使ってdataとlimitを初期化したり、要素の生成・初期化を行うことにします。

```
template <class T> class Vec {
public:
    Vec() { create(); }
    explicit Vec(size_type n, const T& val = T()) { create(n, val); }
    // 残りのインターフェース
private:
    T* data;
    T* limit;
};
```

デフォルトコンストラクタは引数を1つも取らず、空のVec(つまり要素が1つもないもの)を生成します。これには後で書くメンバ関数createを使っています。このようにcreateを使うと、dataとlimitは0に設定されると考えます。

2番目のコンストラクタは、始めて見るexplicitというキーワードを使っています。この意味はすぐに説明しますが、その前に、このコンストラクタの働きを理解する必要があります。第2引数をデフォルト引数(§7.3)にしたので、このコンストラクタは実質的には、2つのコンストラクタの定義になっています。1つはsize_t型の値のみを引数に取るもの、もう1つはsize_tとconst T&を引数に取るもので、どちらの場合も、サイズと値を引数にcreateを呼び出しています。この関数は§11.5で書きますが、n個のT型の要素を生成し、そして値をvalにする関数にします。ユーザはその初期値を指定するか、そうでなければTのデフォルトコンストラクタが§9.5で説明した値初期化のルールに従って生成する値を使います。

ここでexplicitを使う理由が出てきます。このキーワードは引数が1つだけのコンストラクタのみに適用できるものです。これを付けたコンストラクタは、ユーザがコンストラクタをきちんとその形で呼び出した場合にのみ使えるようになります。

```
Vec<int> vi(100); // OK. Vecを整数から明示的に生成している
```

11.2 Vecクラスの実装

```
Vec<int> vi = 100; // エラー。非明示的にVecを生成(§11.3.3)し、それをviにコピーしている
```

コンストラクタが使われるかどうかの場面で、explicitが重要な役割を果たします。これは§12.2で説明します。以下の例では必ずしも必要ないのですが、標準のvectorクラスと同様にするため、explicitを使いました。

11.2.3 型の定義

標準テンプレートクラスの方式に従って、ユーザが使える型を定義し、その実装の詳細を隠したいと考えます。特に、constとconstでないVecに対する反復子と、そのサイズを表す型のためにtypedefが必要です。

実は、標準ライブラリのコンテナにはvalue_typeというのも定義されています。これは、そのコンテナが保持する型を表すものです。少し先ではVecにpush_backのような関数を付け、ユーザがコンテナのサイズを動的に大きくできるようにしたいと思います。value_typeを定義しておけば、back_inserter(これはpush_backとvalue_typeに依存します)を定義し、それによってVecのサイズを大きくする出力反復子を生成することができるようになります。

このような型の定義で一番難しいのは、どの型を選ぶかということです。ここまで見てきたように、反復子はコンテナの内部を指し、その要素にアクセスするためのオブジェクトです。反復子自身、大抵はクラス型です。たとえば、連結リスト構造を実現するクラスを考えたとしましょう。そのようなクラスを実装する論理的な方法は、リストをノード(節)の集まりとし、それぞれのノードが値と次のノードへのポインタを持つというものでしょう。このようなクラスの反復子は、これらのノードへのポインタを持ち、++演算子でそのポインタをリストの次のノードを指すポインタへ進めるものです。このような反復子の実装はクラスとしてのみ可能です。

今考っているVecは、その要素を保持するのに配列を使います。そのため、Vecの反復子には普通のポインタを使えるのです。このポインタは、配列dataの中の要素を指すものになります。§10.1で見たように、ポインタはランダムアクセス反復子のすべての操作をサポートします。ポインタを反復子として使えば、それがそのままランダムアクセス反復子になり、標準のvectorと同様にできるのです。

それでは他の型はどうしましょう。value_typeは明らかにTです。サイズを表す型はどうしましょう。size_tは配列の要素数を表すのに十分な大きさの型です。Vecの内部には配列を使うので、size_tをVec::size_typeに使えます。これらを決めれば、次のようにVecを定義することができます。

```
template <class T> class Vec {
public:
    typedef T* iterator; // 追加した
    typedef const T* const_iterator; // 追加した
    typedef size_t size_type; // 追加した
    typedef T value_type; // 追加した
    Vec() { create(); }
    explicit Vec(size_type n, const T& val = T()) { create(n, val); }
    // インターフェースの残り
private:
    iterator data; // 変更した
    iterator limit; // 変更した
};
```

新しくtypedefをいくつか付け加え、また、ここで定義した新しい型を内部で使っています。typedefで定義

した名前をクラスの内部で使うことで、コードを読みやすくし、また、これらの型を変更するときに一部の変更のし忘れをしないで済むのです。

11.2.4 インデックスとサイズ

`size` を呼び出すことでユーザは `Vec` 内の要素数を知ることができるようにしたいのです。また、各要素にアクセスするために、`Vec` にはインデックスも付けます。たとえば、

```
for (i = 0; i != vs.size(); ++i)
    cout << vs[i].name();
```

のように使うために、`Vec` にメンバ関数 `size` とインデックスのための `[]` 演算子を付け加えます。`size` 関数は簡単です。これは引数を取らず、`Vec` の保持する要素の数を `Vec::size_type` 型で戻すだけです。インデックス操作に関しては、まず演算子のオーバーロードについて少し学ばなければなりません。

オーバーロードされた演算子は、他の関数の場合と同じように定義できます。演算子名を示し、引数と戻り値を特定して定義するのです。

オーバーロードする演算子の名前は、`operator` というキーワードに演算子の記号を付けることで示せます。今の場合には、`operator[]` ということになります。

演算子が単項演算子か二項演算子かは、対応する関数の持つ引数の数で決まります。演算子がメンバ関数でないときは、関数の引数の数が演算子のオペランドの数になります。前の引数が左オペランド、後の引数が右オペランドです。演算子がメンバ関数として定義されるときは、左オペランドは、その演算子が適用されるオブジェクトそのものになります。そのため、メンバ関数である演算子の引数の数は演算子が取るオペランドの数よりも1つ少ないことになります。

一般に、演算子関数はメンバかもしれないしメンバでないかもしれません。しかし、インデックス演算子は数少ない、メンバ関数でなければならない演算子なのです。ユーザが `vs[1]` と書いたら、これは `operator[]` というメンバ関数に1という引数を渡したことになるのです。

この関数の引数の型は、`Vec` が最大数の要素を持っても、その要素数を表せる大きな整数型でなければなりません。これは `Vec::size_type` です。決めなければならないことは、このインデックス演算子が戻すべき値です。

少し考えると、戻り値は `Vec` 内の対応する要素への参照であることがわかります。そうすることで、ユーザはこの要素を読んだり書き直したりできるわけです。前の例では、インデックスを使って要素を読むだけでしたが、ユーザが要素を書き換えることもあるでしょう。このように考えると、`Vec` クラスはさらに次のようになります。

```
template <class T> class Vec {
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef size_t size_type;
    typedef T value_type;
    Vec() { create(); }
    explicit Vec(size_type n, const T& val = T()) { create(n, val); }
// 新しい操作: sizeとインデックス
```

```
size_type size() const { return limit - data; }
T& operator[](size_type i) { return data[i]; }
const T& operator[](size_type i) const { return data[i]; }
private:
    iterator data;
    iterator limit;
};
```

`size` 関数は要素を保持する配列の最後を指すポインタから最初のポインタを引くことで、配列のサイズを計算する関数です。§ 10.1.4 で見たように、2つのポインタを引き算すると、その答えは `ptrdiff_t` 型で、2つのポインタ間にある要素数になるのです。`size` を使っても `Vec` は変わないので、これは `const` メンバ関数にします。こうすることで、`const` `Vec` オブジェクトに対しても `size` を使えるわけです。

インデックス演算子は内部の配列の対応する要素を見つけ、その参照を戻すわけです。参照を戻しているので、ユーザはインデックス演算子を使って `Vec` の要素を変更できます。要素を書き換えるために、実際に `const` の `Vec` オブジェクトと `const` でない `Vec` オブジェクトの2つの場合について考える必要が出てきます。`const` の場合は、`const` の参照を戻していることに注意してください。こうすることで、ユーザはインデックスを、書き込みなしの読み取り専用に使えるようになったのです。ここで、その場合でも、値ではなく参照を戻していることに注意してください。これにより標準の `vector` と同じになっているのです。このようにした理由は、コンテナ内のオブジェクトが大きな場合、コピーするより参照を渡すほうが効率がよいからです。

2つの演算子は、同じ名前で同じ型 `size_type` の値を1つだけ引数に取るので、驚くかもしれません。しかし、演算子も含めてすべてのメンバ関数は、非明示的にオブジェクトそのものを引数に取るのです。そのため、操作の対象になるオブジェクトが `const` かどうかで区別され、そのようなオーバーロードが許されるのです。

11.2.5 反復子を戻す関数

次に反復子を戻す関数を考える必要があります。先頭の位置を示す反復子を戻す `begin` と最後の1つ後を指す反復子を戻す `end` の実装です。これらは、次のようにできます。

```
template <class T> class Vec {
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef size_t size_type;
    typedef T value_type;
    Vec() { create(); }
    explicit Vec(size_type n, const T& val = T()) { create(n, val); }
    T& operator[](size_type i) { return data[i]; }
    const T& operator[](size_type i) const { return data[i]; }
    size_type size() const { return limit - data; }
    // 反復子を戻す新しい関数
    iterator begin() { return data; } // 追加した
    const_iterator begin() const { return data; } // 追加した
    iterator end() { return limit; } // 追加した
    const_iterator end() const { return limit; } // 追加した
private:
```

```
iterator data;
iterator limit;
};
```

`begin` と `end` 両方について、`Vec` が `const` かどうかによって異なる 2 つのバージョンを付け加えました。`const`

`Vec` クラスはまだかなり原始的ですが、本質的な準備は整いました。実際、`push_back` や `clear` などといった関数を少し付け加えるだけで、この本のすべてのサンプルについて、標準の `vector` の代わりに使えるクラスになるのです。ただ、非常に重要な点において、`Vec` クラスはまだ `vector` に足りないものがあります。これらを考えてみましょう。

11.3 コピー管理

この章の始めに、クラスの設計者は、オブジェクトの生成、コピー、代入、破棄を管理するという話をしました。ここまでで、オブジェクトの生成方法については説明してきましたが、コピー、代入、破棄はまだです。後で見ますが、これらの操作の定義をしないと、コンパイラがデフォルトのものを与えてくれます。これらは実際で望ましいものであるかもしれません。しかし、そうではない場合は、コンパイラの与えてくれるものは、予想外の動作をし、実行時エラーを引き起こすこともあります。

広く使われている言語としては、C++が、オブジェクトに対するこのレベルの操作が可能な唯一の言語です。当然、これらの操作を正しく定義することが、有用なデータ型を作るためにとても重要なのです。

11.3.1 コピー構造体

関数として値を渡すとき、また、関数が値を戻すとき、明示的ではありませんがオブジェクトがコピーされていました。たとえば、

```
vector<int> vi;
double d;
d = median(vi); // viをmedianのパラメータにコピー
string line;
vector<string> words = split(line); // splitからの戻り値をwordsにコピー
```

また、オブジェクトの初期化のために、もとのオブジェクトをコピーすることもあります。

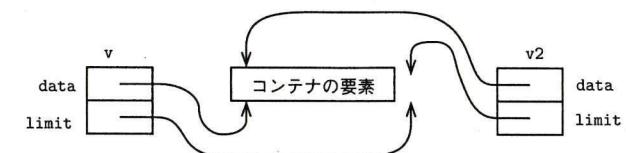
```
vector<Student_info> vs;
vector<Student_info> v2 = vs; // vsをv2にコピー
```

明示的な場合も非明示的な場合も、`copy-constructors` (copy constructor) とよばれる特別な構造体です。他の構造体のように、`copy-constructors` はクラスと同じ名前のメンバ関数ですが、これはすでに存在する同じ型のオブジェクトをコピーして新しいオブジェクトを初期化するものなので、自らのクラスの型の引数を 1 つだけとする構造体です。今、`copy` の意味を定義しているわけですが、それには関数の引数におけるコピーも含まれます。したがって、ここでパラメータを参照にすることは本質的に重要

です。また、`copy` を作ることが、もとのオブジェクトを変えることはないので、`copy` 元のオブジェクトの参照として、`const` なものを使うことになります。

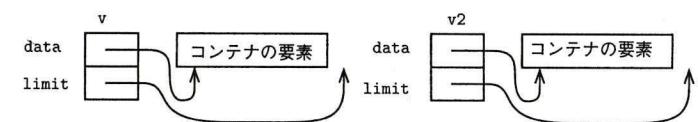
```
template <class T> class Vec {
public:
    Vec(const Vec& v); // コピー構造体
    // 前と同じ
};
```

コピー構造体の宣言をしたので、今度はその中身を考えなければなりません。一般に、コピー構造体はすでにあるオブジェクトのデータを新しいオブジェクトに「コピー」するものです。ここで「」を付いたのは、しばしば、単なるデータのコピー以上のことが必要になるからです。たとえば、`Vec` クラスには 2 つのデータ要素がありますが、それらは両方ともポインタです。もし、ポインタの値をコピーすると、もとのオブジェクトと新しいオブジェクトの両方のポインタが同じ配列を指すことになります。たとえば、`v` を `Vec` とそれを `v2` にコピーしたとします。ここでポインタをコピーすると下図のような構造を作ってしまいます。



明らかに、「コピー」のどちらかのオブジェクトで配列のデータを変更すると、もう一方のオブジェクトのデータも、同じものですから、変更されたことになります。つまり、たとえば `v[0]` にある値を代入すると、`v2[0]` もその値になってしまうのです。これは望ましい振る舞いでしょうか。

他の操作と同様に、この問題の答えを考えるのに標準の `vector` を見てみましょう。§ 4.1.1 では、`vector` のデータを `median` 関数に渡すため、それをコピーする必要がありました。そして、これがコピーであるため、関数内でコピーされたオブジェクトを変更してももとのオブジェクトは影響を受けません。`median` 関数の実例を見て考えてみると、標準の `vector` では、コピーとオリジナルが同じデータを内部で共有しているのではないことがわかります。`vector` のコピーはオリジナルから独立したもので、一方を変更しても他方に影響は与えないのです。



明らかに、`Vec` をコピーするときは、新しくメモリを確保しそこにオリジナルのデータをコピーする必要があります。ここで前に見たように、メモリ確保用にユーティリティ関数があるとし、`copy-constructors` の内部ではこの関数に仕事をさせることにします。

```
template <class T> class Vec {
```

```
public:
    Vec(const Vec& v) { create(v.begin(), v.end()); }
    // 前と同じ
};
```

このように、反復子を2つ引数に取る新しいバージョンの `create` が、メモリを確保し、2つの反復子でかこまれた要素をコピーして、新しい要素を初期化することになります。

11.3.2 代入

クラスの定義では、そのクラスのオブジェクトのコピーの仕方をコントロールするように、オブジェクトの代入演算子 (assignment operator) もコントロールします。オーバーロードによって、クラスはいくつもの引数の異なる代入演算子を定義することができますが、自分のクラスの `const` 参照を引数にとるものは特別です。これは、オブジェクトを他のオブジェクトに代入したときの振る舞いを規定するからです。この場合を特に「代入演算子」とよんで、`operator=` の他のバージョンと区別することもあります。インデックス演算子と同様に、代入演算子もクラスのメンバ関数でなければなりません。また、他の演算子と同様に、代入演算子には戻り値があるので、その型を決める必要があります。組み込み型の代入演算子と同様の動作をさせるには、左辺の参照を戻します。

```
template <class T> class Vec {
public:
    Vec& operator=(const Vec&);
    // 前と同じ
};
```

代入は、常にすでにある値（左辺のもの）を消し、新しい値（右辺のもの）で置き換えるので、コピーコンストラクタとは違います。コピーを作るときには、新しいオブジェクトを生成するので、消し去る古いデータはないのです。しかし、コピーコンストラクタと同様に、代入はデータの値の代入になります。そのため、ポインタがデータメンバであると、コピーのときと同様の問題が起こります。各オブジェクトが右辺にあるオブジェクトのデータのコピーを独立して持つようにしなければならないのです。

コードを書く前に考えておくべきことがあります。それは自己代入です。ユーザがなんらかの理由でオブジェクトを自分自身に代入することも考えられます。以下で見るように、このような自己代入をきちんと扱うことは非常に重要です。

```
template <class T>
Vec<T>& Vec<T>::operator=(const Vec& rhs)
{
    // 自己代入をチェック
    if (&rhs != this) {
        // 左辺の配列を破棄
        uncreate();
        // 右辺の要素を左辺にコピー
        create(rhs.begin(), rhs.end());
    }
    return *this;
}
```

11.3 コピー管理

この関数には新しい考え方が2、3使われていますので、説明しましょう。

まず、クラス定義の外でテンプレート関数を定義する形式です。他のテンプレートのときと同様に、コンパイラにこれがテンプレートであることを知らせ、型パラメータを書く必要があります。それから戻り値ですが、これは `Vec<T>&` になっています。これはクラス定義（ヘッダーファイル）内の宣言では `Vec&` になっています。そこでは戻り値にわざわざ型パラメータを付けなかったのです。実は、入力の手間を省くために、テンプレートの内部では型パラメータを書かなくてもよいようになっています。そのため、クラス定義の中は、テンプレートの中なので、`<T>` を繰り返す必要がないのです。しかし、クラス定義の外で戻り値を書くときには、テンプレートのスコープ外なので、（あれば）型パラメータを明示的に書かなければならないのです。同様に、関数名も単純に `Vec::operator=` ではなく、`Vec<T>::operator=` です。しかし、1度、これが `Vec<T>` のメンバであると特定した後なら、その後では `<T>` を繰り返す必要はなくなります。そのため、引数の型は `const Vec&` とできるのです。これは `const Vec<T>&` と書くこともできます。

この関数のもう1つの新しいところは、`this` というキーワードを使っていることです。`this` はメンバ関数の中だけ有効なキーワードで、メンバ関数が動作しているオブジェクトへのポインタになっているのです。たとえば、`Vec::operator=` の中では、メンバ関数 `operator=` が動作する `Vec` オブジェクトへのポインタなので、`this` は `Vec*` です。代入演算子では、`this` は左辺のオペランドを表します。一般に、`this` はオブジェクト自身を表したいときに使われるのです。今の例では、`if` の条件と `return` のところでそのように使われています。

代入の右辺と左辺のオブジェクトが同じものかどうか判定するのに `this` を使っています。もし、同じものなら同じアドレスを持つはずです。§ 10.1.1 で見たように、`&rhs` は `rhs` のアドレスになります。これと `this` をくらべて自己代入かどうかを直接チェックしているのです。右辺左辺が同じオブジェクトなら、代入演算子のすることは何もありません。そこですぐに `return` してしまうのです。一方、異なるオブジェクトであった場合、左辺のオブジェクトのメモリを解放し、そこに新しく配列を生成し、右辺のオブジェクトのデータを代入します。あきらかにもう1つのユーティリティ関数で `Vec` の要素を破棄しそのメモリを解放するものが必要になります。これを `uncreate` とします。この関数 `uncreate` を呼ぶと古い値が破棄されるので、次に左辺のメモリの確保と右辺から左辺へのデータのコピーに `create` 関数が使えるのです。

代入演算子が自己代入をきちんと処理することは非常に重要です。ここでは、左辺と右辺を直接比較して同じかどうかをチェックすることで処理しました。この重要性を見るために、このチェックを代入演算子のコードから取り除いたらどうなるかを考えみましょう。この場合、常に左辺のオブジェクトに対し `uncreate` を実行し、データを破棄しメモリを解放してしまいます。ここで、左辺と右辺のオブジェクトが同じものなら、右辺のオブジェクトも同じメモリ空間を指すポインタをデータとして持っているわけです。そして、左辺のオブジェクトのデータを生成するために `create` を使えば、ひどいことになるでしょう。左辺のメモリを解放すると右辺のメモリも解放してしまうからです。`create` が `rhs` のデータをコピーしようとしたとき、そのデータは破棄されメモリは解放されているからです。

ここで行ったような直接的なチェックは、自己代入を扱うためのよくある方法ですが、これ以外にも方法がありますし、常にベストの方法でもありません。重要なことは、自己代入を正しく扱うということです。それを実際にどうするかは方法の問題に過ぎません。

`return` ステートメントに関してはもう1つ面白いことがあります。これは `this` の指すオブジェクトを取り出し、それを戻しています。参照を戻す場合に重要なことは、その関数が終了したあとも参照しているオブジェクトは残るものでなければならないということです。ローカルな変数の参照を戻すとひどいことになります。それは関数が終了した後では、その参照するオブジェクトは破棄されてしまい、参照はゴミを指すことになるから

です。代入演算子の場合、左オペランドのオブジェクトの参照を戻しています。このオブジェクトは代入演算子のスコープの外でも破棄されず、したがって演算終了後も生き残ることが保証されているのです。

11.3.3 代入は初期化ではない

私たちの経験では、初心者には代入と初期化の区別があいまいなことが多いようです。特に C を代表とする多くのプログラミング言語ではその違いを明確にしないので、多くのプログラマがそれを意識しないということが起こります。`=`というシンボルは、初期化と代入の両方で使われるため、その違いがわかりにくくなっています。実は、`=`が変数の初期化に使われるときには、`operator=`を呼び出しています。クラスの設計者は、クラスを正規を代入のエクスプレッションに使うときは、`operator=`を呼び出しています。クラスの設計者は、クラスを正しく実装するため、その違いを理解しておかなければなりません。それは、代入 (`operator=`) は常に前の値を破棄するのに対し、重要な違いは以下の事実からわかれます。初期化は、新しいオブジェクトを生成し、同時に値を与える期化ではそのようなことをしないということです。初期化は、新しいオブジェクトを生成し、同時に値を与えるのです。初期化が行われるのは次のような場合です。

- 変数宣言で
- 関数が呼ばれたときのパラメータで
- 関数が終了するときの戻り値で
- コンストラクタの初期化子で

代入はエクスプレッションの中で`=`が使われるときにだけ行われます。

```
string url_ch = "~;/?:@=&$_-._+!*'();"; // 初期化
string spaces(url_ch.size(), ' ');
string y; // 初期化
y = url_ch; // 代入
```

最初の宣言は新しいオブジェクトを生成しています。そのため、これはオブジェクトの初期化であり、コンストラクタを呼び出しているのです。ここで

```
string url_ch = "~;/?:@=&$_-._+!*'();";
// という形式は、文字列リテラルの"~;/?:@=&$_-._+!*'();"である const char* から string を生成している
// ことを意味しています。この形式により、const char* 型の引数を取る string のコンストラクタが呼ばれる
// ことになります。このコンストラクタは、文字列リテラルから直接 string オブジェクト url_ch を生成するものか、ある
// です。このコンストラクタは、名前のないオブジェクトを文字列リテラルから一時的に作って、それからコピーコンストラクタを呼んで
// います。url_ch に無名オブジェクトをコピーするものです。
// 2番目の宣言は初期化の別の形式です。これはコンストラクタに1つ以上の引数を直接渡しています。このよ
// うな場合、コンパイラは、引数の数や型に適合するコンストラクタを見つけてそれを適用するのです。今の場合
// は、引数を2つ取っていますが、前の引数は string に何個の文字を入れるかを指示し、後の引数はその文字を
// 指示するものです。結果、spaces は url_ch の文字数と同じ数の空白を持った string オブジェクトになるわ
// けです。
```

11.3 コピー管理

3番目の宣言はもっと簡単です。これはデフォルトコンストラクタを呼び出し、空の string オブジェクトを生成しているのです。そして、最後のステートメントは宣言ではありません。これは=演算子をエクスプレッションの一部として使っているので、代入なのです。この代入は、string の代入演算子によって実行されます。関数の引数と戻り値は、もう少し複雑になります。たとえば、line が1行の入力を保持しているとします。そして次のように § 6.1.1 の split を使うとします。

```
vector<string> split(const string&); // 関数の宣言
vector<string> v; // 初期化
v = split(line); // split 関数の開始時にパラメータが line で初期化される
// 終了時に戻り値が初期化され、
// それが v に代入される
```

split の宣言では、戻り値の型をクラス型と定義しているので注意が必要です。関数からの戻り値を代入するには2ステップの処理が必要になるのです。まず、コピーコンストラクタが戻り値を一時的な作業用オブジェクトにコピーします。それから、代入演算子がその一時オブジェクトの値を左辺のオペランドに代入するのです。初期化と代入は、それぞれ異なる操作なので、区別することが重要なのです。

- コンストラクタは常に初期化をコントロールする
- メンバ関数の `operator=` は常に代入をコントロールする

11.3.4 デストラクタ

Vec にはもう1つ付け加えるべきものがあります。これは Vec オブジェクトが破棄されるときにどうするかを定義するものです。ローカルなスコープで生成されるオブジェクトは、そのスコープ外に出るときに破棄されます。動的に生成されたオブジェクトは、そのオブジェクトへのポインタに delete を使ったときに破棄されます。たとえば、§ 6.1.1 の split 関数を考えてみましょう。

```
vector<string> split(const string& str)
{
    vector<string> ret;
    // str を単語に分解し ret に格納する
    return ret;
}
```

split 関数が終了すると ret はスコープ外に出るので、変数 ret は破棄されます。

コピーと代入のときと同様に、オブジェクトが破棄されるときにすべきことを定義しておくことができます。オブジェクトが破棄されるときに何をするかを定義しておくデストラクタ (destructor) と呼ばれる関数があるからです。デストラクタは、オブジェクトの生成を定義するコンストラクタに対応する特別な関数です。デストラクタの名前はクラスの名前の前にチルダ (~) を付けたものです。これは引数を取らず、戻り値も持ちません。

デストラクタは、オブジェクトが消えるときに必要な後始末をする関数なのです。典型的な後始末は、コンストラクタが確保したメモリのようなリソースを解放することです。

```
template <class T> class Vec {
public:
    ~Vec() { uncreate(); }
```

```
// 前と同様
};

Vec ではメモリをコンストラクタで確保していたので、これをデストラクタで解放しなければなりません。代入演算子が左辺のメモリを破棄するのとほぼ同じ仕事です。これは意外なことでなく、デストラクタでも同じユーティリティ関数を呼び出し、要素を破棄しそのメモリ領域を解放しています。
```

11.3.5 デフォルトの動作

第4章と第9章で定義した `Student_info` のようなクラスには、`コピー``コンストラクタ`、`代入演算子`、`デストラクタ`の定義がありません。それなら、そのようなクラスが生成され、`コピー`され、`代入`され、`破棄`されると、`デストラクタ`の定義がありません。クラスの制作者がこれらの関数を定義しなかった場合、コンパイラがデフォルトのものを生成するのです。

デフォルトの動作は再帰的です。つまり、オブジェクトのデータメンバの`コピー`、`代入`、`破棄`を、データメンバのルールにしたがって行うのです。データメンバがクラス型のオブジェクトなら、その型の`コピー``コンストラクタ`、`代入演算子`、`デストラクタ`が使われ、もしデータメンバが組み込み型なら、オブジェクトの`コピー`・`代入`、`代入演算子`、`デストラクタ`が使われる。それがポインタであつては、その値の`コピー`・`代入`になります。ただし、組み込み型のデータメンバに対しては、それがポインタであつても、オブジェクトのデストラクタは何もしないことに注意してください。特に、デフォルトデストラクタでポインタを破棄すると、ポインタの指すメモリ領域は解放されずに確保されたまま残ります。

ここで `Student_info` の、デフォルトの動作がわかります。たとえば、デフォルトの`コピー``コンストラクタ`はデータメンバを`コピー`します。そこで `string` と `vector` の`コピー``コンストラクタ`が呼び出され、`name` と `homework` がそれぞれ`コピー`されます。また、2つの `double` である `midterm` と `final` の値は直接`コピー`されます。

§ 9.5 で見たように、デフォルトコンストラクタというものがあります。クラスにコンストラクタをまったく書かなかつた場合、コンパイラは引数を取らないデフォルトのコンストラクタを生成するのです。デフォルトコンストラクタは、そのオブジェクトを再帰的に初期化します。つまり、デフォルトの初期化が必要な場合は、コンストラクタはデータメンバをデフォルト初期化します。また、値初期化が必要な場合はデータメンバを値初期化します。

ここでクラスにコンストラクタを1つでも書いた場合、それが`コピー``コンストラクタ`であっても、コンパイラはデフォルトのコンストラクタを生成しなくなることに注意してください。実は、デフォルトコンストラクタが非常に重要なこともあります。たとえば、コンパイラがデフォルトコンストラクタを生成するときもそうです。コンストラクタが定義されていないクラスのデータメンバは、それ自身がデフォルトコンストラクタを持っているからです。それゆえ、一般的のクラスには、第9章で見たように明示的にか、または第4章で見たように非明示的に、デフォルトコンストラクタを持たせておくのがよいのです。

11.3.6 3のルール

メモリのようなリソースを管理するクラスでは、`コピー`のコントロールをよく考えて行う必要があります。一般的に、そのようなクラスではデフォルトの動作は不十分です。`コピー`における失敗は、そのクラスを使うユーザーを混乱させ、しばしば実行時エラーを引き起します。

`Vec` で`コピー``コンストラクタ`、`代入演算子`、`デストラクタ`を定義しなかつた場合を考えてみましょう。§ 11.3.1

11.4 動的な Vec

で見たように、ユーザを驚かせる程度でめめばよい方です。`Vec` クラスのユーザは、そのオブジェクトをコピーすると、独立したオブジェクトが2つになったと考えるでしょう。どちらか一方を変更しても、別の方は変更されないと思うはずです。

さらに悪いことに、デストラクタを定義しておかないと、コンパイラが生成するデストラクタが使われてしまいます。これはポインタそのものは破棄しますが、そのポインタが指すメモリ領域は解放しません。結果はメモリリーク（メモリの解放し忘れ、メモリの無駄使い）を起こします。`Vec` が使ったメモリは不要になつても解放されないので、メモリリークをなくすことができますが、`コピー``コンストラクタ`と`代入演算子`を付け加えること、おそらく実行時にクラッシュを起こしてしまうでしょう。これらを付け加えないと、§ 11.3.1 の最初の図にあるように、2つの `Vec` オブジェクトが同じメモリ領域を指すポインタを持つようになります。ここで1つのオブジェクトが破棄されると、デストラクタがそのメモリ領域を解放します。もう1つのオブジェクトが持つポインタを使ってその領域を参照するようことをすれば、エラーになつてしまうのです。

リソースをコンストラクタで確保するクラスでは、`コピー`がリソースを正しく扱うように定義する必要があるのです。もし、クラスがデストラクタを持つ必要があるなら、おそらく`コピー``コンストラクタ`と`代入演算子`も必要でしょう。クラスオブジェクトの`コピー`や`代入`では、オブジェクトを生成するときと同様なメモリ確保することになるでしょう。クラス `T` のオブジェクトの`コピー`をコントロールするには、次のものが必要です。

<code>T::T()</code>	コンストラクタ、引数を取るコンストラクタも必要かも
<code>T::~T()</code>	デストラクタ
<code>T::T(const T&)</code>	コピーコンストラクタ
<code>T::operator=(const T&)</code>	代入演算子

これらのメンバ関数を定義すると、コンパイラはオブジェクトの生成、`コピー`、`代入`、`破棄`の際に、これらを呼び出すことになります。オブジェクトの生成、`コピー`、`破棄`は、非明示的にも行われることを覚えておいてください。明示的であれ非明示的であれ、コンパイラは正しいメンバ関数を呼び出すのです。

コンストラクタ、デストラクタ、代入演算子は強く関係しているので、それらの関係は3のルールと呼ばれています。たとえば、デストラクタが必要なクラスなら、おそらく`コピー``コンストラクタ`と`代入演算子`も必要だろうということです。

11.4 動的な Vec

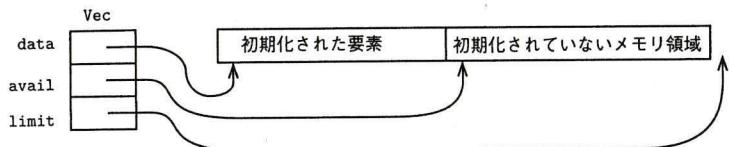
メモリ管理の関数の定義を書く前に、今の `Vec` は標準の `vector` より、重要な点で劣っていることを認める必要があります。`Vec` には `push_back` 関数は付いておらず、サイズは固定されているのです。`push_back` はその引数を `vector` の後に付け加える関数ですが、その処理で、`vector` の要素数を1つ増やしていることを思い出してください。

`Vec` オブジェクトの現在の要素に新しい要素を付け加える `push_back` 関数を付け加えることはできます。たとえば、要素1つ分大きくメモリを確保し、現在のすべての要素をその場所はじめの方から`コピー`し、次に `push_back` の引数を`コピー`して新しい要素を生成し、それを新しく要素を置いた場所の最後に付け加えるようにすることもできるでしょう。しかし、このような方法では、ユーザが頻繁に `push_back` を使うと、効率がかなり悪くなることがわかります。

このような問題には古典的な解決方法があります。それは、始めに、必要な分以上にメモリを確保しておくと

いうものです。そして、このメモリを使い切ったときに限り、新たにより多くメモリを確保するようにします。簡単のため、`push_back` が余分なメモリを必要とした場合、現在のメモリの倍のメモリを確保するようになります。たとえば、最初に `Vec` に要素を 100 個入れて生成したとします。すると `push_back` をはじめて呼び出すときに、200 要素分のメモリを確保するのです。そして最初の 100 個の要素を新しく確保したメモリの前半部にコピーし、新しい要素をその後に入れます。こうすると、その後の `push_back` の 99 回の呼び出しでは、メモリを新たに確保する必要はなくなるのです。

この方法では、内部のデータを保持する配列の扱いが変わってきます。もちろん、最初の要素を指すポインタは依然として必要です。しかし、「最後」を表すポインタは 2 つ必要になるのです。1 つは、末尾（最後の 1 つ後）の要素を指すポインタです。これは次に使える場所を指すポインタということもできます。もう 1 つのポインタは確保したメモリの末尾（最後の 1 つ後）を指すポインタです。`Vec` オブジェクトは次のような構造になります。



幸い、この新しいメンバを扱うのは、`push_back` 関数とまだ書いていないメモリ管理関数だけです。加えて、`push_back` そのものはとても簡単です。`grow`（成長という意味）と `unchecked_append`（チェックせずに追加という意味）という新しいメモリ関数を後で書くことにして、`push_back` の仕事の大変なところはそれらに任せてしまうのです。

```
template <class T> class Vec {
public:
    void push_back(const T& val) {
        if (avail == limit) // 必要ならメモリを確保
            grow();
        unchecked_append(val); // 新しい要素を付け加える
    }
private:
    iterator data; // 前と同様にVecの最初の要素へのポインタ
    iterator avail; // 保持されている末尾（最後の1つ後）を指すポインタ
    iterator limit; // 確保されているメモリの末尾（最後の1つ後）を指すポインタ
    // 残りのインターフェースと実装は前と同じ
};
```

11.5 柔軟なメモリ管理

`Vec` クラスを書き始めたときに、組み込みの `new` や `delete` でメモリ管理ができないことを見ました。これらを使うと、標準の `vector` に比べて `Vec` に柔軟性がなくなるからです。`new` は私たちの目的以上の仕事、つまりメモリ確保と初期化の両方をしてしまうのです。`new` を使って型 `T` の配列を作る場合、`T` にはデフォルトコンストラクタが必要です。このようなやり方では、ユーザに提供したい柔軟性を持たせられないのです。

11.5 柔軟なメモリ管理

また、`new` を使うことは、非常に代償が大きいともいえます。これは配列のすべての `T` オブジェクトを `T::T()` で初期化するからです。もし、`Vec` のすべての要素を自分で初期化したいと考えるなら、`new` で 1 回とユーザが値を入力するときの 1 回で、合計 2 回各要素を初期化することになるわけです。さらに悪いことがあります。`push_back` に関して説明したメモリ管理の方法を思い出してください。メモリが不足した場合は、使用メモリを現在の 2 倍にする方針です。このとき確保されるメモリを初期化する理由はないのです。これは `push_back` でのみ確保され、その後は新しい値を格納するだけだからです。もし `new` を内部の配列のメモリ確保に使えば、これらの要素は、使われる使われないにかかわらずすべて初期化されてしまうのです。

ここで、組み込みの `new` と `delete` 演算子の代わりに、柔軟なメモリ管理の方法をサポートしている標準ライブラリのツールを使うことができます。言語そのものにそのようなメモリ確保の方法はありません。それは、メモリの性質はいろいろ多様であるため、言語そのもので記述することができないからです。

たとえば、最近のコンピュータにはいろいろな種類のメモリがあります。同じマシン上でもメモリが異なれば、その速さは変わってきます。また、グラフィカルパッファや共有メモリのように特別な性質を持つメモリもあります。また停電でも内容を保持するメモリもあります。ユーザはこれらのうちで好きなもの（あるいは違うもの）を使えるので、メモリの確保と管理はライブラリに任せるのがよいのです。標準ライブラリも、これらのメモリのすべてをサポートするよう要求されているわけではありません。しかし、メモリ管理の仕組みを提供することで、メモリマネージャへの共通したインターフェースを持たせられるのです。これらのマネージャが固有の実装に結び付けられていたとしてもです。これは、入出力の仕組みを言語そのものではなくライブラリに任せた決定と同じです。メモリ管理をライブラリに任せたことにより、異なるメモリを使うことに柔軟性が出てくるのです。

`<memory>` ヘッダには `allocator<T>` というクラスがあります。これを使って、型 `T` のオブジェクトを格納するためにメモリを初期化せずに確保し、その先頭を指すポインタを得ることができます。このようなポインタを使うと、間違って望んでいない場所を指すことになるかもしれません。実際、異なるオブジェクトにも使えるので危険だということは覚えておいてください。ライブラリには、確保したメモリ内にオブジェクトを生成する方法と、メモリ自身は解放せずにオブジェクトを破棄する方法も与えられています。`allocator` を使う場合、メモリのどの部分にオブジェクトが格納され、どの部分は空いているかをきちんと把握しておくことは、プログラマの責任になるのです。

`allocator` クラスで、今の目的に使われる部分は、4 つのメンバ関数と付随した 2 つの非メンバ関数です。

```
template <class T> class allocator {
public:
    T* allocate(size_t);
    void deallocate(T*, size_t);
    void construct(T*, const T& );
    void destroy(T* );
    ...
};

template <class In, class Out> Out uninitialized_copy(In, In, Out);
template <class Out, class T> void uninitialized_fill(Out, Out, const T& );
```

`allocator` クラスのメンバ関数 `allocate` は指示された要素数分のメモリを確保しますが、その初期化は行い

ません。このメモリは T オブジェクトを保持するために使え、そのアドレスは T*型のポインタで表すことができます。また、初期化していないということは、そのメモリは領域を確保しただけで、その内にオブジェクトは生成していないということです。メンバ関数 `deallocate` は、この初期化されていないメモリを解放します。これは `allocate` によって確保されたメモリを指すポインタと、格納されている要素数を引数に取ります。メンバ関数の `construct` と `destroy` は、初期化されていないメモリ内でのオブジェクトの生成と破棄をする関数です。`construct` は、`allocate` で確保されたメモリを指すポインタとコピーしたい値を引数にして呼び出します。`destroy` は引数に渡されたポインタの指す要素について T のデストラクタを呼び出します。

テンプレートであるので、Tはコンパイル時に具体化されます。この具体化によって、特定の型に対する allocatorができるのです。今の場合、正しいallocatorを使うために、Vec<T>にallocator<T>をメンバとして付け加えておきます。これにより、T型オブジェクトのメモリが確保できるようになります。このメンバを付け加え、付随するライブラリ関数を使うことで、標準のvectorクラスと同様に効率的で柔軟なメモリ管理ができるのです。

11.5.1 Vec の最終版

メモリ管理関数の宣言まで含めた Vec の完成版は次のようにになります。ただし、メモリ管理関数の定義は後で書きます。

```
template <class T> class Vec {
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef size_t size_type;
    typedef T value_type;
    Vec() { create(); }
    explicit Vec(size_type n, const T& t = T()) { create(n, t); }
    Vec(const Vec& v) { create(v.begin(), v.end()); }
    Vec operator=(const Vec&); // §11.3.2で定義した
    ~Vec() { uncreate(); }
    T& operator[](size_type i) { return data[i]; }
    const T& operator[](size_type i) const { return data[i]; }
    void push_back(const T& t) {
        if (avail == limit)
            grow();
        unchecked_append(t);
    }
    size_type size() const { return avail - data; }
    iterator begin() { return data; }
```

```

const_iterator begin() const { return data; }
iterator end() { return avail; }
const_iterator end() const { return avail; }

private:
    iterator data;           // Vecの最初の要素を指す
    iterator avail;          // Vecの最後の要素（の1つ後）を指す
    iterator limit;          // 確保されたメモリの最後（のの1つ後）
    // メモリ確保の仕組み
    allocator<T> alloc;    // メモリ管理をするオブジェクト
    // 内部の配列のメモリ確保と初期化
    void create();
    void create(size_type, const T&);
    void create(const_iterator, const_iterator);
    // 配列内の要素の破棄とメモリの解放
    void uncreate();
    // push_backで使う関数
    void grow();
    void unchecked_append(const T&);

}:

```

最後に残ったのは、メモリを扱う `private` メンバ関数の実装です。これらのメンバを書いていきますが、有効な `Vec` オブジェクトに対しては次の 4 つのが成立することを覚えておくと、コードを理解しやすいでしょう。

1. `data` はデータがあるなら、その最初の要素を指し、そうでなければ 0 になっている。
 2. `data ≤ avail ≤ limit`
 3. データ要素は `[data, avail)` の範囲にある。
 4. `[avail, limit)` の範囲にはデータ要素はない。

このような条件をクラスの不变な表明 (class invariant) とよぶことにします。これは § 2.3.2 で紹介したループの不变な表明と同様で、クラスのオブジェクトが生成されるときこの不变な表明がすぐに成立します。そして、この不变な表明を保つようにメンバ関数を定義すれば、これは常に正しい表明になります。

`public` なメンバ関数はこの不变な表明を不成立にはできません。そのためには `data`、`avail`、`limit` の値を扱う必要がありますが、`public` なメンバ関数はこれらを直接操作することはないからです。

まず、いろいろなバージョンの `create` 関数を見てきましょう。これらはメモリの確保、そのメモリ内での要素の初期化、そしてポインタの設定を行う関数です。メモリの確保の仕方は違いますが、どの場合でも、実行後にポインタ `limit` と `avail` が等しくなることは同じです。つまり、生成された要素の最後のものが、確保されたメモリの最後に置かれるのです。以下の関数の実行終了後、クラスの不变な表明は成り立ったままでいることを確かめてみてください。

```
template <class T> void Vec<T>::create()
{
    data = avail = limit = 0;
}

template <class T> void Vec<T>::create(size_type n, const T& val)
{
    data = alloc.allocate(n);
    limit = avail = data + n;
}
```

```

        uninitialized_fill(data, limit, val);
    }

    template <class T>
    void Vec<T>::create(const_iterator i, const_iterator j)
    {
        data = alloc.allocate(j - i);
        limit = avail = uninitialized_copy(i, j, data);
    }
}

```

引数を取らない `create` は空の `Vec` を生成するので、その仕事はポインタをすべて 0 にすることだけです。

サイズと値を引数に取る `create` は、メモリを適当な分だけ確保します。`allocator<T>` のメンバ関数 `allocate` が、指定された数の `T` オブジェクトを格納できる分のメモリを確保します。つまり、`alloc.allocate(n)` で `n` 個のオブジェクト分のメモリが確保されるのです。この `allocate` 関数の戻り値は、メモリの最初を指すポインタになるので、これを `data` に格納します。`allocate` の確保したメモリは初期化されていないので、`uninitialized_fill` 関数を使って初期化します。これは、最初の 2 つの引数で示される範囲を 3 番目の引数の値を持つオブジェクトで埋める関数です。関数が終了すると、`allocate` で確保されたメモリ上に `val` という値を持ったオブジェクトが `n` 個入っていることになります。

最後の `create` も、その前の 2 つと似ていますが、ここでは `allocate` で確保したメモリを初期化するのに `uninitialized_copy` を使います。この関数は、最初の 2 つの引数で示される要素の列を、3 番目の引数が指定している場所に順にコピーする関数です。この関数の戻り値は、初期化された要素の末尾（最後の 1 つ後）を指すポインタになります。これはまさに `limit` と `avail` に入れるべきものです。

`uncreate` 関数は、`create` 関数のことをもとに戻し、`Vec` の使っていたメモリ領域を解放する関数です。

```

template <class T> void Vec<T>::uncreate()
{
    if (data) {
        // データ要素を（生成された逆順で）破棄する
        iterator it = avail;
        while (it != data)
            alloc.destroy(--it);

        // 確保されていたメモリを解放する
        alloc.deallocate(data, limit - data);
    }
    // ポインタをリセットし、Vecが再び空になったことを示す
    data = limit = avail = 0;
}

```

`data` が 0 の場合は何もしません。ここでもし `delete` を使っていれば、`data` を 0 と比べる必要もありません。というのは、0 であるポインタに `delete` を適用しても害はないからです。しかし、`delete` と違い、`alloc.deallocate` 関数は、仮に解放するメモリがない場合でも、0 でないポインタに適用しなければならないのです。そのため、最初に `data` が 0 かどうかをチェックしているのです。

`data` が 0 でなければ、反復子 `it` を使って `Vec` の要素 1 つひとつにアクセスし、`destroy` でそれらを破棄していきます。これは `delete[]` と同様に、最後の要素から前の要素に向かって破棄していきます。そしてすべての要素を破棄したら、`deallocate` を使ってそのメモリを解放します。この関数は解放するメモリの最初を指す

ポインタと、オブジェクト何個分のメモリを破棄するかを示す整数を引数に取ります。メモリをすべて解放したいので、`data` と `limit` の間のすべてのメモリを解放します。

最後は、`push_back` 関数内で使うメンバ関数です。

```

template <class T> void Vec<T>::grow()
{
    // メモリの確保領域を増やすときはこれまでの2倍確保する
    size_type new_size = max(2 * (limit - data), ptrdiff_t(1));
    // メモリを確保し、そこに今ある要素をコピーする
    iterator new_data = alloc.allocate(new_size);
    iterator new_avail = uninitialized_copy(data, avail, new_data);
    // これまで使っていたメモリ領域を解放する
    uncreate();
    // 新しいメモリ領域を指すようにポインタをセットする
    data = new_data;
    avail = new_avail;
    limit = data + new_size;
}

// availは確保された領域内の初期化されていないところを指しているとする
template <class T> void Vec<T>::unchecked_append(const T& val)
{
    alloc.construct(avail++, val);
}

```

`grow` のすることは、新しい要素のために十分なメモリ領域を確保することです。実際には、当面必要以上の領域を確保し、続く `push_back` がその領域を使えるようにしています。これはメモリ確保が頻繁になって効率が悪くなるのを防ぐためです。ここではメモリを確保するたびに、その前の領域の 2 倍の領域を確保する方針だと § 11.4 で書きました。ただ、`Vec` が空の場合もあります。そこで、「もとの領域の 2 倍」と 1 の大きい方を確保するメモリのサイズにするように、`max` を使っています。§ 8.1.3 で説明しましたが、`max` の 2 つの引数は厳密に同じ型でないとエラーになります。そこで、1 という数から `ptrdiff_t` 型オブジェクトを生成しています。これは § 10.1.4 で見たように `limit - data` の型なのです。

このように `grow` では、まず、確保するメモリ領域の大きさを `new_size` に格納します。それから `allocate` を使って、それだけメモリを確保し、`uninitialized_copy` を使って、現在の要素を新しいメモリ上にコピーします。そして、古くなったメモリ領域を解放しその要素を破棄するために、`uncreate` を呼びます。最後に、`data` が新しく生成された配列の先頭を指し、`limit` がその配列の最後の 1 つ後を指し、`avail` が `Vec` の最後の 1 つ後の要素（まだ初期化されていないもの）を指すようになっています。

`allocate` と `uninitialized_copy` の戻り値を、まず変数に格納していることは重要です。もし、これらの値を直接 `data` や `limit` に使うと、`uncreate` を呼んだときに、古くなった領域ではなく、新しく確保して初期化した配列を破棄してしまうことになるからです。

`unchecked_append` 関数は、初期化されている要素の直後に新しい要素を生成する関数です。この関数は、`avail` がまだ要素が生成されていない、しかしすでに確保されている領域を指していると仮定して動作します。`unchecked_append` 関数は、`grow` の直後で使うだけなので、これは安全なのです。

11.6 詳細

テンプレートクラス： § 8.1.1 で紹介したテンプレートの仕組みで作る。

```
template <class type-parameter [, class type-parameter] ... >
class class-name { ... };
```

これで、与えられた型のパラメータを持つ、*class-name* という名のテンプレートクラスの定義になる。型パラメータはテンプレートの中で型として使える。クラスのスコープ内では、クラス名に<>を付ける必要はない。クラスのスコープの外では必要である。例：

```
template <class T>
Vec<T>& Vec<T>::operator=(const Vec&){ ... }
```

テンプレート型のオブジェクトを生成するときに、ユーザが実際の型を決める。たとえば *Vec<int>* とすると、コンパイラはパラメータを *int* にしたバージョンの *Vec* を生成する。

コピー管理： 一般に、クラスはオブジェクトの生成、コピー、代入、破棄のしかたをコントロールする。コンストラクタは、オブジェクトの生成やコピーで呼び出される。代入演算子は代入のエクスプレッションの中で呼ばれる。デストラクタはオブジェクトが破棄されるときや、それがスコープの外に出るときに自動で呼び出される。コンストラクタでリソースを確保するクラスは、ほとんど確実にコピー・コンストラクタ、代入演算子、デストラクタが必要。代入演算子の定義を書くときは、自己代入をチェックすることが重要である。組み込み型の代入演算子に合わせるため、左辺のオペランドへの参照を戻すのがよい。

自動生成される操作： クラスの定義で、コンストラクタを書かないと、コンパイラはデフォルトコンストラクタを生成する。また、それはコピー・コンストラクタ、代入演算子、デストラクタも同様。自動生成される操作は、再帰的である。そのクラスのデータメンバに対し、適当な操作を再帰的に行う。

オーバーロードされた演算子： *op* という名前の演算子は *operator op* という名前の関数としてオーバーロードできる。パラメータの 1 つはクラス型でなければならない。演算子がクラスのメンバ関数であるとき、(2 項演算子なら) その左オペランド、または (1 項演算子なら) その 1 つだけのオペランドが、その演算子が定義されているクラスのオブジェクトになる。インデックス演算子と代入演算子はクラスのメンバでなければならない。

課題

11-0 この章のプログラムをコンパイルし、実行し、試してください。

11-1 第9章で定義した *Student_info* 構造体には、コピー・コンストラクタ、代入演算子、デストラクタを定義しません。なぜでしょうか。

11-2 しかし、*Student_info* 構造体にはデフォルトコンストラクタを定義しました。なぜでしょうか。

11-3 コンパイラが自動で付ける *Student_info* の代入演算子はどう動作するでしょうか。

11-4 コンパイラが自動で付ける *Student_info* のデストラクタが破棄するメンバはいくつでしょうか。

11.6 詳細

11-5 *Student_info* クラスに、そのオブジェクトがどれだけ生成され、コピーされ、代入され、破棄されるか回数を数える仕組みを作ってください。そして、そのクラスを第6章の成績処理プログラムに使ってみてください。それにより、ライブラリのアルゴリズムが何回コピーをするかがわかります。コピーの回数を調べることで、ライブラリのどのクラスを使うと効率がどう変わるかることができます。これを行い、解析をしてください。

11-6 要素を 1 つ破棄する操作を *Vec* に付け加えてください。また、*Vec* の全要素を破棄するものも付け加えてください。これらは *vector* の *erase* と *clear* に対応するものです。

11-7 *Vec* に *erase* と *clear* を付け加えたなら、この本の前方にあるほとんどのプログラムでは、*vector* の代わりに *Vec* を使えます。第9章の *Student_info* プログラムと第5章の文字で絵を描くプログラムを書き換えてください。

11-8 標準のリストクラスとそれに付随する反復子を簡略化したクラスを書いてください。

11-9 § 11.5.1 の *grow* 関数は、メモリが必要になるたびに、メモリの量を 2 倍にします。この方法の効率を評価してください。他の方法での効率を予測してから、そのように実装を変え、違いを測定してください。