第11章「抽象データ型を定義する」

主なトピック

- コピーコンストラクタ
- デストラクタ

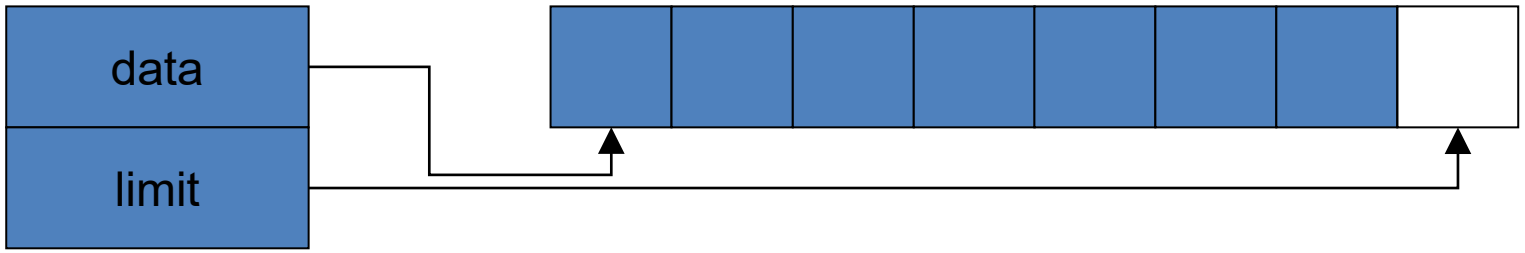
Vecクラス

- テンプレートクラスで実現
  - 色々な型のvectorを実現できるように

```
// この色の部分が追加
template <class T>
class Vec {
public:
        // インターフェースに該当する部分はこちらに
private:
        //   実装に該当する部分はこちらに
}
```

データの保持

- データの保持の仕方
  - 内部で配列を持つ
  - begin関数, end関数, size関数などを実現できるように
  - data: 配列の先頭の要素を指すポインタ
  - limit: 最後の要素の一つ後ろを指すポインタ



Vecクラス

```
template <class T>  class Vec {
public:
    //  インターフェース
private:
    T*  data;  //  最初の要素をさすポインタ
    T*  limit;  //  最後の要素の一つ後ろを指すポインタ
}
```

- Vec<int> v;
  - T は int型，T* は int*型を指す
- Vec<double> v;
  - T は double 型，T* は double *型を指す

メモリ確保

- 内部で確保する配列の要素数は不明
- そのため，必要になるたび配列を動的に確保する
  - 例えば，nを要素数として，
  - new T[n]; // 要素数 n のT型の配列の生成
  - しかし，上は T がデフォルトコンストラクタを持つときのみ実行可能
  - そうでないクラスでも動作するように，ユーティリティ関数を利用する→後で説明

コンストラクタ

- 2種類のコンストラクタ
  - Vec< Student_info > vs; // デフォルト
  - Vec< double > vs(100); // サイズを引数にとる
- Vecクラスのコンストラクタがする仕事
  - data と limit の値の初期化
  - 内部の配列のメモリ確保
  - 内部の配列の値の初期化
  - create という関数を作り，それにこれらの仕事をさせる（これは，最後の方で定義）

Vecクラス

```
template <class T>  class Vec {
public:
    //  この色の部分が追加
    Vec() { create(); }
    explicit Vec( size_type n, const T& val =T() )
            { create(n, val ); }
private:
    T*  data;
    T*  limit;
}
```

explicit

explicit Vec(size_type n, const T& val =T())

- explicit はコンストラクタが，引数が指定された形式で呼び出されたときのみ，適用可能
- C++では引数の自動型変化が行われるため，意図しない型変換を抑制するために使われる
  - Vec<int> v1(100, -1); // 要素数100，初期化の値は-1
  - Vec<int> v2(50); // 要素数50，初期化の値はデフォルト引数，この場合はint型のデフォルトの値で初期化
  - それ以外の方の引数で呼ばれたときはエラーを返す
  - Vecクラスではなくても動作する

型の定義

- クラスに関連した型を定義すると便利
  - iterator, const_iterator: イテレータ
  - size_type,: サイズを表す型
  - value_type: コンテナが保持する型
- これらをクラス内部で typedef する

Vecクラス

```
template <class T>  class Vec {
public:
    // この色の部分が追加
    typedef  T*  iterator;
    typedef  const T*  const_iterator;
    typedef  size_t  size_type;
    typedef  T  value_type;
private:
    iterator  data;
    iterator  limit;
}
```

インデックスとサイズ

- 添え字を使えるように，演算子（オペレータ）[ ]を定義
  - 添え字が与えられたら，その値を返す関数
  - 関数名は，operator[ ]
  - 引数は，size_type i
  - 戻り値は，T&
  - T& operator [] (size_type i) { return data[i]; }

インデックスとサイズ

- 要素数を知ることができるようにsize 関数を定義
  - 要素の数は limit - data で計算可能

- Vec<Student_info> vs;
  for(i = 0; i < vs.size(); ++i) {
  cout << vs[i].first_name();
  }

Vecクラス

```
template <class T>  class Vec {
public:
    //  この色の部分が追加
    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:
}
```

イテレータを戻す関数

- 先頭の要素のイテレータを返す begin 関数
- 最後の要素の一つ後のイテレータを返す end 関数

```
template <class T>  class Vec {
public:
    // この色の部分が追加
    iterator  begin()  { return data; }
    const_iterator  begin()  const  { return data; }
    iterator  end()  { return limit; }
    const_iterator  end()  const  { return limit; }
private:
}
```

コピー管理

- 明示的なコピー
  - vector<Student_info> vs;
  - vector<Student_info> v2(vs); // v2にvsをコピー
- 非明示的なコピー
  - vector<int> vi;
  - int d;
  - d = median(vi); // この瞬間 vi のコピーがmedian 関数に渡される
- どちらの場合も，コピーコンストラクタという特別なコンストラクタが使用される

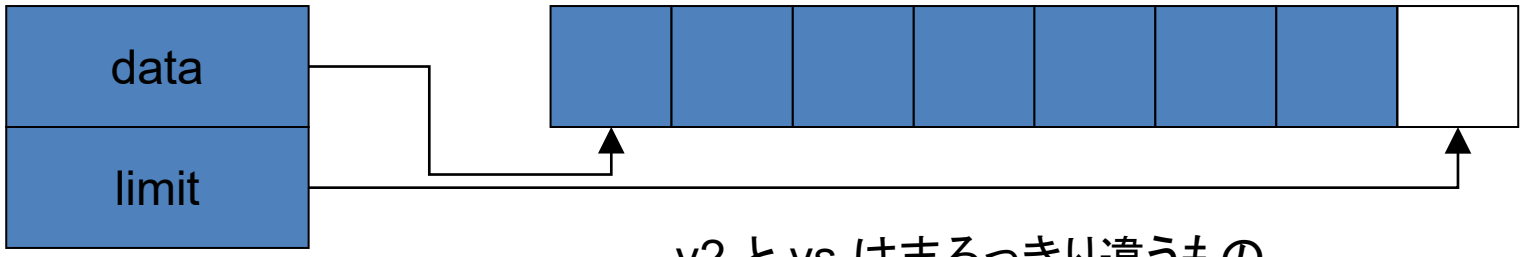
コピーコンストラクタ

- 引数を一つ取るコンストラクタ
- 引数の型は，自分のクラスの型の参照
- 元のオブジェクトを変更しないのでconst
- 新しくメモリを確保して，そこに元のデータを「コピーする」（deep copy）
  - shallow copyではない

Deep copy と Shallow copy

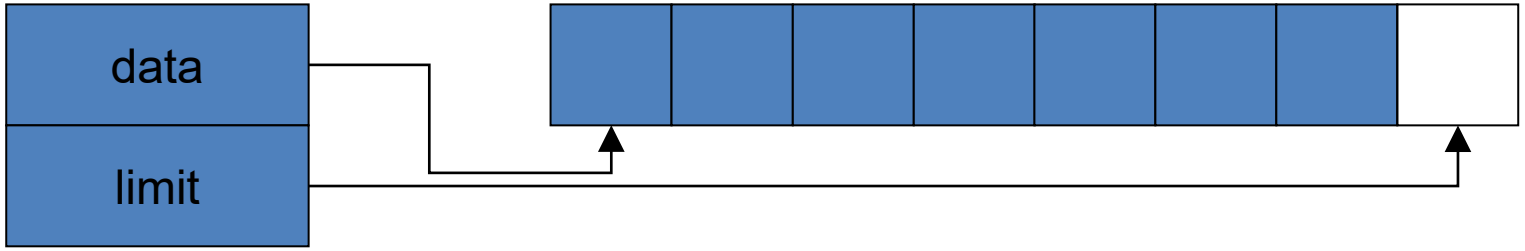
v2 = vs; Deep copyの場合

vs



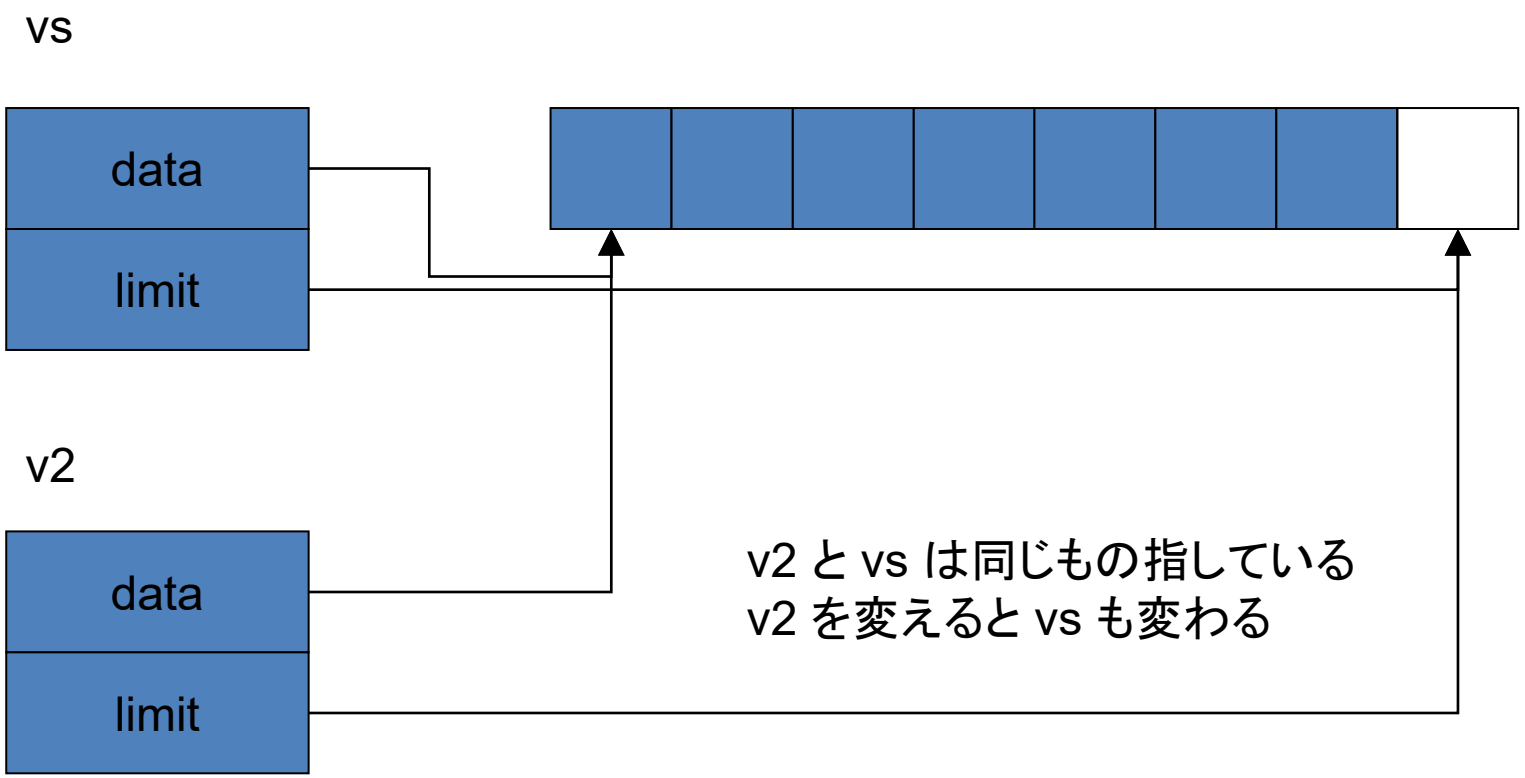
v2 と vs はまるっきり違うもの
v2 を変えても vs は変わらない

v2



Deep copy と Shallow copy

v2 = vs;　Shallow copyの場合



Vecクラス

```
template <class T>  class Vec {
public:
    // この色の部分が追加
    Vec( const Vec& v )  { create( v.begin(), v.end() ); }
private:
}
```

- create関数は，後ほど定義

代入

- 代入では，古い値を消し，新しい値で置き換える
  - v2 = vs;
  - uncreate という関数を作り，内部の配列を破棄
  - その後，create関数を利用して，配列の確保とデータのコピー
  - 自己代入のチェック
    - 自分自身を書き換えるのは，動作が不安定
    - 自分に自分自身を代入する必要はない（値が変化しないから）
    - そのため，自己代入をしないようなソースにする

Vecクラス

```
template <class T>  class Vec {
public:
    Vec& operator = ( const Vec& v );
}
template <class T>
Vec<T>&  Vec<T>::operator=(const Vec& rhs)
{
    if( &rhs != this){  //  自己代入でなければ
        uncreate();       //  古い内容破棄して,
        create( rhs.begin(), rhs.end() );  //  新しい内容に
    }
    return *this;
}
```

this

- メンバ関数の中だけで有効なキーワード
- 自分自身（メンバ関数が動作しているオブジェクト）へのポインタ
- 先程の例だと，thisは代入される方のオブジェクト
- *this はオブジェクトの実体

初期化と代入は違う！

- 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
- 初期化が行われるのは
  - 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
  - コンストラクタ初期化子で
- 一方，代入は，
  - 常に前の値を破棄し，その後値を与える

初期化と代入は違う！

- =という記号は初期化にも代入も用いられる
  - string a = “aaaaa”; // 初期化（コピーコンストラクタが呼び出される）
  - string b(10, ‘ ‘); // 初期化（引数が2のコンストラクタが呼び出される）
  - string y; // 初期化（デフォルトコンストラクタ）
  - y = a; // 代入
- メンバ関数のoperator=は代入を行う

デストラクタ

- オブジェクトが破棄されるときに行う動作
  - 内部の配列の破棄，メモリの開放
- デストラクタはクラスの名前に~（チルダ）を付けた関数名

```
template <class T>  class Vec {
public:
    // この色の部分が追加
    ~Vec()  { uncreate(); };
}
```

- uncreate 関数は，後ほど定義

デフォルトの動作

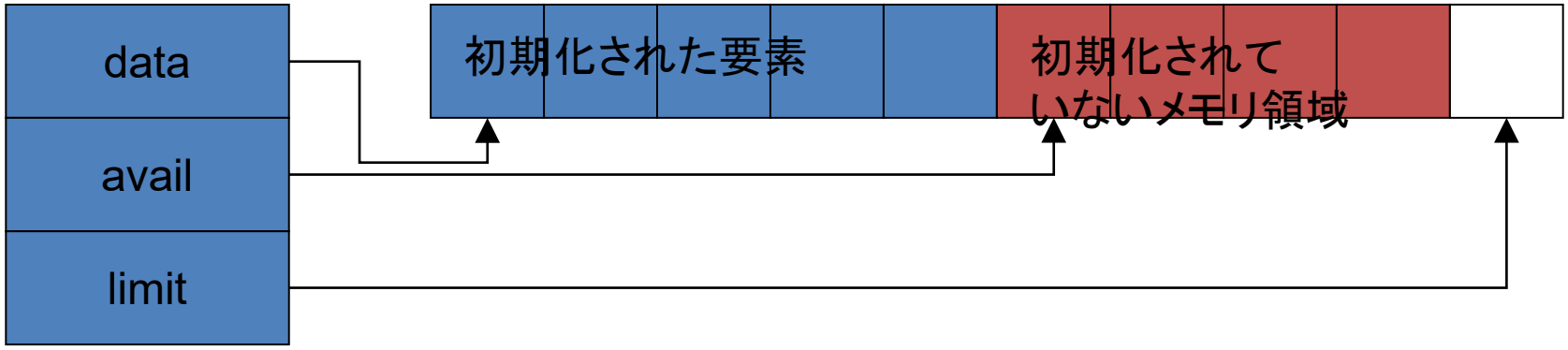
- 以前定義したStudent_infoクラスでは,
  - コピーコンストラクタ
  - 代入演算子
  - デストラクタの定義をしなかった
- このときは，各データメンバのルールに従う
  - FirstName, LastName, IDは string クラスの
  - Midterm, Finalは double 型の
  - Homeworkはvector<double>のルールに従う

3のルール

- 以下の3つは密接に関連しているので，どれか一つを定義する必要のあるクラスでは，残りの2つも定義せよ．
  - コピーコンストラクタ T::T(const T&)
  - 代入演算子 T::operator=(const T&)
  - デストラクタ T::~T()
    - コンストラクタが該当することも T::T()
- 今回のように内部でメモリ（配列）を利用する場合は，とくに気を付ける

動的なVec

- データの追加のための2つの方法
  - A: 必要となるたび，1つずつメモリを確保
    - メモリの無駄はないが，時間がかかる
  - B: 予めたくさん確保し，使い切るまで利用
    - メモリは無駄になるが，時間は効率的
- ここでは，Bの方法を採用



Vecクラス

```
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 );  //  新しい要素を付加
    }
private:
    iterator  data;  //  最初のデータへのポインタ
    iterator  avail;  //  最後のデータの1つ後へのポインタ
    iterator  limit;  //  確保されているメモリの最後の1つ後へのポインタ
}
```

柔軟なメモリ管理

- new と delete ではなく，より低レベルでのメモリの管理
- 手数はかかるが，細かな処理が可能
- allocator<T>というクラスを利用
  - #include <memory>
  - T型のオブジェクトを，初期化せずにメモリの確保が可能
  - 効率的だが，処理を間違うと危険
  - プログラマ次第

allocator クラス

- 今回使うallocatorクラスに関連した関数

```
template <class T>  class allocator  {
      T*  allocate(size_t);
      void  deallocate(T*, size_t);
      void  construct(T*, const T&);
      void  destory(T*);
};
template <class In, class Out> Out uninitilized_copy(In, In, Out);
template <class Out, class T> void uninitilized_fill (Out, Out, const T&);
```

Vecクラス

```
template <class T>  class Vec {
private:
    allocator<T>  alloc;  //  メモリ管理のためのオブジェクト
    //  内部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開放
    void  uncreate( );
    //  push_back関数で使用
    void  grow( );
    void  unchecked_append( const T& );
}
```

create 関数の実装

```
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 );
}
```

create 関数の実装

```
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 );
}
```

uncreate 関数の実装

```
template <class T>  void  Vec<T>::uncreate( )
{
    if( data )          {
            //  データを逆順に破棄
            iterator  it = avail;
            while( it != data )
                        alloc.destroy(--it);
            //  確保されていたメモリを開放
            alloc.deallocate( data, limit - data );
    }
    //  ポインタを0にリセットし，空になったことを示す
    data = limit = avail = 0;
}
```

grow 関数の実装

```
template <class T>  void Vec<T>::grow()       {
    //  今までの2倍の量のメモリを確保，その計算
    size_type new_size = max( 2 * (limit - data), ptrdiff_t(1) );
    //  メモリの確保と既存の内容のコピー
    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開放
    uncreate();
    //  新しいメモリ領域を指すようにポインタをリセット
    data = new_data;
    avail = new_avail;
    limit = data + new_size;
}
```

unchecked_append 関数の実装

```
template <class T>
void Vec<T>::unchecked_append(const T& val)
{
    //  確保済みのメモリに，val の値のオブジェクトを生成
    alloc.construct(avail ++, val);
}
```

- Vecクラスの詳細は，スケルトンファイルを参照

std::allocator

メモリ

- allocate　　メモリを確保する→このあとインスタンスを構築（初期化）が必要
- deallocate　　メモリを解放する←インスタンスの破棄をしてから呼び出す

インスタンス（オブジェクト＝変数）

- construct　　引数を元にインスタンスを構築する（初期化）
- destroy　　インスタンスを破棄する（終了処理）

std::uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x)

- 未初期化領域の範囲（first, last）を指定された値xで配置new（インスタンス初期化）する

std::uninitialized_copy(InputIterator first, InputIterator last, ForwardIterator result)

- 入力範囲[first, last)のコピーを未初期化出力範囲[result,)に書き込む。

```
int main()
{
  std::allocator<int> alloc;

  // メモリ確保。
  // この段階では、[p, p + size)の領域は未初期化
  const std::size_t size = 3;
  int* p = alloc.allocate(size);

  // 未初期化領域pを初期化しつつ、値2で埋める
  std::uninitialized_fill(p, p + size, 2);

  // 要素を破棄
  for (std::size_t i = 0; i < size; ++i) {
    alloc.destroy(p + i);
  }

  // メモリ解放
  alloc.deallocate(p, size);
}
```