PL04 プログラミングC++2023
授業概要

成瀬継太郎
naruse@u-aizu.ac.jp

授業の時間と形式

- 講義：毎週月・木曜日の13:20-15:00（M6），担当: 成瀬（CS, IT-SPR）
- 演習: 同じ日の15:10-16:00（演習室5，6），担当：成瀬（CS），イブラヒム（IT-SPR）

授業サイト

- 授業のWebページは大学共通のmoodleを使います
  - 講義：23講義 PL04 プログラミングC++
    - https://elms.u-aizu.ac.jp/course/view.php?id=6049
    - 講義のハンドアウト
    - クイズの採点
    - 期末試験情報など
  - 演習： 23演習 PL04 プログラミングC++ [演]
    - https://elms.u-aizu.ac.jp/course/view.php?id=6050
    - 演習課題
    - 演習の採点
- 授業のメーリングリスト
  - 講義: 18-1405-PL04-M6@course.u-aizu.ac.jp
  - 演習（成瀬クラス）：18-1407-PL04-std5@course.u-aizu.ac.jp
  - 演習（イブラヒムクラス）：18-1407-PL04-std6@course.u-aizu.ac.jp

評価

- 演習: 40%
  - 必修13
- 期末試験: 40%
- 出席とクイズ: 20%
  - クイズの提出回数が全体の2/3に満たない場合は無条件に不合格とする
    →10回の提出が必要
  - クイズは授業開始中に回答する
  - 単位を取得するには，演習，期末試験，出席とクイズのすべてを合格する必要がある

演習課題の提出

- 提出方法: 授業サイトからアップロード
- 提出期限：1週後まで
  - 授業が月曜として翌週火曜の日付のものは受け付けない
- 注意事項
  - 期限までに提出すること→遅れると採点しない
  - ただし，提出後の再提出は認める
    （学習効果を高めるためにも，採点後に積極的に再提出してください）

演習採点基準

- 各演習は10点満点
- 採点基準: 減点法(最低0点)
  - コンパイルが通らない: -10
  - コンパイルは通るが結果が間違い: -4から-8
  - コメントがない, 不適切: -2

（アンケート）演習の計算機環境について

- 皆さんが普段使っている計算機で演習を課題を行うことが多いと思いますので，質問です．
  皆さんのOSは何ですか？C++のコンパイラは何を使っていますか？
- Windows
  - Visual Studio 2022
  - gcc, clang on MinGW, Ubuntu / WSL2, Ubutu / VMWare Player
- MacOS
  - gcc, clang
- Ubuntu
  - gcc, clang
- CentOS
  - gcc, clang

授業予定前半

講義回	講義日	内容
1	10月5日 (木)	授業概要，第0章「さあ始めよう」第1章「stringを使う」
2	10月10日 (火) 月曜授業	第2章「ループとカウンタ」
3	10月12日 (木)	第3章「たくさんのデータを扱う」
4	10月16日 (月)	第4章「プログラムとデータの構成」
5	10月19日 (木)	第5章「シーケンシャルコンテナとstringの解析」
6	10月23日 (月)	第6章「ライブラリのアルゴリズムを使う」
7	10月26日 (木)	第7章「連想コンテナを使う」

授業予定後半

講義回	講義日	内容
8	10月30日 (月)	第8章「ジェネリック関数を書く」
9	11月1日(水) 月曜授業	第9章「新しい型を定義する」
	11月6日 (月)	学園祭翌日のため休講
10	11月9日 (木)	第10章「メモリ管理と低レベルのデータ構造」
11	11月13日 (月)	第11章「抽象データ型を定義する」
12	11月16日 (木)	第12章「値のように振舞うクラス」
13	11月20日 (月)	第13章「継承と動的結合を使う」
14	11月27日 (月)	第14章「メモリを（ほとんど）自動的に管理する」
期末試験	未定	期末試験期間中に実施する

この授業の到達点

- C++のコードを読める・書けるようになること
  - 将来的にc++を用いたライブラリを使えるように→画像処理（OpenCV），点群データ処理（Point Cloud Library），行列演算（Eigen）など
  - （蛇足）外部ライブラリを使うのなら python で良いのでは？
    - OpenCVもPCLもpythonライブラリがあるし，行列計算はNumPyがある
    - その通りだが，c++は実行速度の面で有利．用途に応じて使い分けるのが良いと思う

- プログラムの開発＝データ構造の設計＋アルゴリズムの適用ができるようになること

- プログラムのテストができるようになること
  →思い付きコード修正から系統的なテストへ

C++とは

- C言語と互換性あり
  - C言語のプログラムはC++でコンパイル可能
- C言語を「オブジェクト指向」化したもの
  - 自分で作った「クラス」が組み込みの変数や関数と同じように使えるようになる
  - 「＝」や「＋」が自分で定義できる
  - 使いやすい道具を作りやすい
- グループによる開発に向いている

C++を学ぶとは

- C++の組み込みのデータ形式を学ぶ
- 標準ライブラリ関数（STL）を学ぶ
- クラスの概念を用いたプログラム設計ができること
  - 何をクラス変数にするか
  - 何をメンバ関数にするか
  - どう設計すれば使いやすいクラスになるか
  - クラスのテストの仕方を学ぶ

本講義で学ぶC++のバージョン

通称	規格出版日	概要
C++98	1998-09-01	最初の標準化
C++11	2011-09-01	現在のC++の基本形．本講義ではこの規格のC++を学ぶ
C++14	2014-12-15	C++11のマイナーアップデートで，いくつかの機能が追加された
C++17	2017-12	新しい機能が追加され，一部の古い機能が廃止された 最近の標準になりつつある．しかし普段使うレベルでは C++11とC++17はほとんど違わないので， 本講義では部分的に便利なC++17の機能を紹介する

各バージョンの違いはC++日本語リファレンスを参照してください
https://cpprefjp.github.io/

C++の二つの面

- C++ as Better C
  - C++言語はライブラリが追加された使いやすいC言語としてつかうことができる
  - string, Standard Template Library (STL)
  - ポインタ, リファレンス（参照）, イテレータ
  - 講義の第1回から第8回まで
- C++ as Object-oriented C
  - 一方、C++言語はC言語をオブジェクト指向化したものとも見える
  - Class
  - 講義の第9回から第14回まで

第0章「さあ始めよう」

主なトピック

- コンパイルの方法
- コメントの書き方

最初のC++のソース

```
/*
    hello.cc
    概要：最初のC++プログラム
    Author: Keitaro Naruse
    Date: 2023-10-01
*/

#include <iostream>

int main()
{
    //   画面に文字列を出力し，改行
    std::cout << “Hello, World!” << std::endl;

    return 0;
}
```

ソースの編集とコンパイルの方法

- C言語とほぼ同じ
- エディタで編集
  - C++のソースファイルの拡張子は.cc .cppなど
- コマンドラインからコンパイル
  - > g++ hello.cc
  - 成功するとa.outという実行形式が生成
  - > ./a.out で実行
  - g++ -o hello hello.cc とすると，helloという実行形式ができる

ソースを見ると(1)

- １から6行目と12行目: コメント行
  - // で始まる行は行末までコメント
  - C言語で使われている /* */ も使用可能
- 8行目: #include < iostream >
  - プログラムの中で標準ライブラリを利用するときに使用
  - この場合はiostreamに含まれる標準ライブラリを使うとき

ソースを見ると(2)

- 10行目: int main()
  - メイン関数，必ず実行される
  - int型（整数型）の値を環境（OS）に返す
  - この場合は，外部から受け取るデータはない
- 11,16行目: { }
  - 中カッコで囲まれたのが，ブロック
  - ブロックの中に書かれるのが，ステートメント（文）

ソースを見ると(3)

- 13行目: std::cout << “Hello, world!” << std::endl;
  - 標準出力（画面）にHello,world!という文字列を出力する
  - std::cout 標準出力ストリーム，画面
  - std::endl 改行文字
  - std:: （標準の）名前空間
  - << 出力演算子
  - 文字列は””（ダブルクォーテーション）で囲う
  - 文末は;（セミコロン）

ソースを見ると(4)

- 15行目: return 0;
  - 関数の戻り値として0を返す（この場合は環境に）
- 自由形式
  - 文字列と文字列の間の改行や空白は自由に置いて良い
  - ソースが見やすくなるようにインデント

この科目の「美しい」ソースコード

```
/*
    hello.cc
    概要：最初のC++プログラム
    Author: Keitaro Naruse
    Date: 2023-10-01
*/

#include <iostream>

int main()
{
    //  画面に文字列を出力し，改行
    std::cout << “Hello, World!”
<< std::endl;

    return 0;
}
```

- 演習はこのスタイルに従ってください
  - ソースの先頭にプログラムの概要，作者，作成日を表すコメント
  - 見やすくなるように改行を多用
  - インデントの字下げは4文字
  - 処理の概要がわかるようなコメント

第1章「stringを使う」

主なトピック:

- ストリング型（文字列型）std::stringの使い方
- コンソール（std::cin, std::cout）の使い方

2番目のソース

```
/*
    greetings.cc
    概要：ユーザに名前を聞いて，あいさつをするプログラム
    Author: Keitaro Naruse
    Date: 2023-10-02
*/

#include <iostream>
#include <string>

int main()
{
    //  ユーザに名前を聞く
    std::cout << “Please input your name:”;
```

```
    //  名前を読み込む
    std::string name;           //  nameの定義
    std::cin >> name;  //  nameに読み込み

    //  あいさつを書く
    std::cout << “Hello, ” << name << “!” << std::endl;

    return 0;
}
```

このプログラムを実行すると

- 画面に
  - Please input your name:
- Vladimirと入力したとすると
  - Vladimir
- 画面には下が出力される
  - Hello, Vladimir!

string型(1/3)

- C言語では文字型 char
  - 文字列を扱うときは char型の配列として表現
  - char name[80];
  - 上の場合だと最大80文字
- C++言語では文字列型　string型
  - std::string name;
  - name = “Vladimir”;
  - 何文字であっても構わない

string型(2/3)

- #include <string>
  - string型を利用するためのインクルード文
- std::string　　name;
  - string型の変数nameを宣言
- std::cin >> name;
  - 標準入力(std::cin)から文字列を読み込み，nameに格納

string型(3/3)

- 文字列の長さを調べるにはsize()
  - string型のメンバ関数
  - nameの中身が”Keitaro Naruse”だとして
  - name.size()とすると14が返る
- stringの初期化
  - std::string name1(“Keitaro Naruse”);
  - std::string name2 = “Keitaro Naruse”;
  - std::string name3 = name1;
  - std::string name4(name1);
    - name1からname4の中身は”Keitaro Naruse”
  - std::string spaces(8, ‘*‘);
    - 8文字の＊（アスタリスク）からなる文字列

標準入力ストリームと標準出力ストリーム

- 標準入力ストリーム（キーボード） std::cin
  - std::string first, second;
  - std::cin >> first >> second;
  - C言語では
    - char first[80], second[80];
    - scanf(“%s”, first); scanf(“%s”, second);
- 標準出力ストリーム（画面） std::cout
  - std::cout << first << “ ”<<second << std::endl;
  - Std::string text=“abc¥n”;
  - Std::cout << text;
  - C言語では
    - printf (“%s %s¥n”, first, second);

型とオブジェクト

- 型：値と演算を定義する
  （例）int, double
- オブジェクト：特定の型の値を保持するメモリ
  （例）int a, double b;
- 値：型に従って解釈されるビット列
  （例）a = 0, b = 0.0;
- 変数：名前付きのオブジェクト
  （例）a, b

整数型

- int 型は整数の値と演算を定義する
- int a =7, b = 9, c;
  - この=は初期化演算子
  - 変数の値は指定された整数値
- c = a + b;
  - この=は代入演算子
  - + は加算の演算子
  - cの値は16

文字列型

- string 型は文字列の値と演算を定義する
- string a =“alpha”, b = “beta”, c;
  - この=は初期化演算子
  - 変数の値は指定された文字列
- c = a + b;
  - この=は代入演算子
  - + は文字列連結の演算子
  - cの値は”alphabeta”

サンプルコード

- std::string name;
- std::cin >> name; // ”Vladimir”が入力されたとして
- const std::string greeting = “Hello, ” + name + “!”;
  - greetingは”Hello, Vladimir!”になる
  - const宣言はプログラムの中で変数（この場合はgreeting）の内容を変更させない
  - const 宣言したときは，必ず値を与える
  - nameは変数宣言した後で値が変わる（代入される）からconst宣言できない

サンプルコード

- const std::string spaces(greeting.size(), ‘ ‘);
  - spaces というstring型の変数は,
  - greetingの文字列と同じ長さ（16文字）で，空白だけからなる文字列で初期化
- const std::string second = “* ” + spaces + “ *”;
  - second の中身は”*　　*”
  - 全体で20文字（2+16+2）
- const std::string first (second.size(), ‘*’);
  - first の中身は”**********” (20文字)

第2章「ループとカウンタ」

主なトピック

- ループと条件判断（C言語の復習）
- 名前空間(std::)

例えば，こんなプログラム (1/2)

```
//    1から100までの和を計算
sum = 0;
for(i = 1; i <= 100; i++)       {
      sum = sum + i;
}
```

- for()文による繰り返し（ループ）
- i という変数が繰り返し数を保存（カウンタ）

例えば，こんなプログラム (2/2)

```
//  3n + 1問題
n = 100;  //  n は任意の自然数
while(n != 1)  {
    if(n % 2 == 0)  {  //  2で割った時の余りが0, つまり偶数のとき
            n = n / 2;
    }
    else    {  //  奇数のとき
            n = 3 * n + 1;
    }
}
```

- while()文による繰り返し（ループ）
- n という変数とループの条件判定

ループとカウンタ

- ほとんどすべてのプログラムは
  - ループと
    - while(){}
    - for(){}
  - 変数（カウンタ）と
    - i ++;
    - n = 3*n + 1;
  - 条件分岐で
    - if(){} else if(){} else {}
- できている

while文 (1/5)

while(条件)
　　ステートメント;

- 条件が真の（成立している）間，ステートメントを繰り返し実行

while(条件) {
　　ステートメント1;
　　ステートメント2;
}

- 実行するステートメントが複数あるときはブロック{}の中に入れる

while文 (2/5)

- よくある失敗(1)
  - 実行されないかもしれないソース

```
// 確実な動作のためには i=0; がここに必要
while(i < 100){
    i ++;
}
```

  - iの値が初期化されていない（iの値は不定）
  - 条件が真かもしれない，偽かもしれない
  - while文の前に条件の初期値を確認すること

while文 (3/5)

- よくある失敗(2)
  - （ほぼ）無限ループ

```
i = 0;
while(i < 100){
    i --; // i++を間違えてしまう
}
```

  - 条件の更新を間違えると，終了しない
  - ちょっと複雑な条件や，複数の条件の組み合わせになると，よく間違えるので注意

while文(4/5)

- 何行あるか解らないデータを読み込むとき
- データファイル

  s0001 aaaa

  s0002 bbbb

  s0003 cccc
- データの最後はEOF(End of File; ファイルや入力の最後表す特別な記号)だとする
- キーボードからEOFを入力するには, UnixならCtrl+d, WindowsならCtrl+zをキー入力する

while文(5/5)

```
std::string id, name; //  IDと名前をstring型の変数に
int count; // 読んだデータの行数をint型の変数に
count = 0;
while(std::cin >> id >> name)     {
   count ++;
}
```

- 正常な入力データでは，whileの条件 std::cin >> id >> name は真(true)になる
- 入力データがEOFだったら，条件は偽(false)になる
- 1行読んだらcountが1増える，countにはデータの行数が入る

for文(1)

```
for(初期化ステートメント; 条件; 式){
    ステートメント;
}

int i, sum = 0;
for(i = 0; i < 100; i++){
    sum = sum + i;
}
```

- まず，初期化ステートメントを実行
  - 次に，条件が真の間，ステートメントを実行
  - 最後に式を実行
  - これを条件が偽になるまで繰り返す

for文(2)

- 同じ動作をする2つのループ式

```
for(初期化ステートメント; 条件; 式) {
    ステートメント;
}
```

```
初期化ステートメント;
while(条件)  {
    ステートメント;
    式;
}
```

- for文は初期化・条件・式を先頭で明示するため，一般的にプログラムミスが少ない
- while文は複雑なループを表現可能

条件分岐

```
if(条件1) {
        ステートメント1;
}
else if(条件2) {
        ステートメント2;
}
else {
        ステートメント3;
}
```

- 条件1が真ならばステートメント1を実行，条件2が真ならばステートメント2を実行，それ以外ならステートメント3を実行

条件の組み合わせ

- 論理演算子
  - AND, 論理積
    - 条件1 && 条件2
    - 条件1と条件2の両方が同時に真のときに，全体が真
    - if((a == 0) && (b == 0))
    - 演算子の優先順位があるので，if(a == 0 && b == 0) でも良い
  - OR, 論理和
    - 条件1 || 条件2；
    - 条件1と条件2のどちらかが（両者でも構わない）真のときに，全体が真
    - if((a ==0) || (b == 0))
    - 同様に if(a ==0 || b == 0)

C++言語に時々でてくる変数の型

- size_type型
  - 例えば，std::string::size_type c;
  - 標準の名前空間(std::)に含まれる文字列型(string)のサイズや長さを表すための型(size_type)の，変数c
  - 長さを表すから符号なし，0以上の数
  - しかし，最大値はシステム依存
    - 127？ 255？ 32,767？ 65,535？ 2,147,483,648? 4,294,967,295?
  - 汎用的に使えるように size_typeという型
  - 実際はunsinged int型であることが多い
  - string型だけでなく，他の型にもsize_type型
    - vector<double>::size_type など

小技(1)

```
int sum = 0;
for(int i = 0; i < 100; ++i){
    int a;
    sum = sum + i;
}
```

- C++ ではどこででも変数宣言できる
- for()の中で変数宣言(int i)可能
- ブロック{}の中でも変数宣言可能(int a)
  - その変数はブロックの中だけで有効
  - ブロック外に出ると，その変数は破棄される

小技(2)

- C++言語だと，++iと書くことが多い
  - i++は式を評価してからインクリメント（1増やす）
  - ++iはインクリメントしてから式を評価

```
a = 3; std::cout << a++;
// 出力は3
a = 3; std::cout << ++a;
// 出力は4
```

  - for文だとi++も++iも同じ動作
  - まぎらわしい動作をしないようなソースを書く

```
a = 3; //  このように式を分割する
++a; // これならa++, ++aとも同じ動作になる
std::cout << a;
```

小技(3)

- std::を省略する方法
  - std::cin, std::stringなどstdが多くて大変
  - using std::cin;とすると，それ以降cinでOK
  - 例えばプログラムの先頭で

```
#include <iostream>
#include <string>
using std::cin;
using std::cout;
using std::string;
int main()          {
}
```

名前空間

- std::は名前空間(namespace)と呼ばれる
  - cin, cout, string, などたくさんの型や変数がstdの中で定義されている
- 用途に応じて名前空間を定義可能
- 同じ変数名が衝突しないように
  - prv::が自分で作った名前空間として
  - std::string と prv::string は別物として扱われる
  - 巨大なプログラムを複数人で作るときに便利
- using namespace std;
  - stdという名前空間で定義されている全体の識別子（型や変数）を利用可能
  - using std::cin;
    - stdという名前空間の中の識別子を個別に指定

¥nとendlの違い

- OSによって改行コードが異なる
  - Unix: CR (0x0d)
  - MS-DOS -> Windows : CR(0x0d) + LF(0x0a)
- Unixの¥nは0x0dだが，Windowsでは0x0d+0x0a
- コード変換しないでUnixからWindowにテキストファイルを持ってくると改行が乱れることがある
- なのでC++では，改行コードの機種依存を防ぐために，生の制御コード(¥n)ではなく，endlという抽象的なオブジェクトで改行を表現することにした

演習課題2のポイント

- while文で，EOFが入力されるまでstd::cinからデータの読み込み
  - 読み込んだ行数を変数に保存
  - １行読み込むたびに，画面に出力
  - 5行ごとに仕切りのフレームを入れる
  - 各行には，きれいに見えるように，適切に空白を入れる

第3章「たくさんのデータを扱う」

主なトピック

- 要素数可変の配列　ベクトル型（std::vector）
- ライブラリのアルゴリズム（std::sort）の利用

個数がわからない大量のデータを扱う

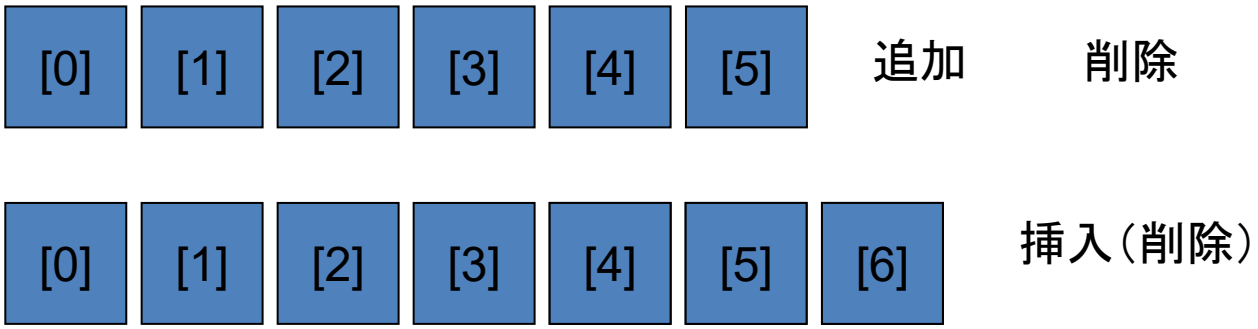
- 例えば，ファイルの読み込み
  - 何行あるか解らない
- 今回の課題だと，演習の提出回数
  - 個人により提出回数が異なる
- このようなデータをプログラムの中で保存したい
  - 配列だと最大個数をあらかじめ設定
  - 大きな配列をとったとしても，使わなかった無駄
  - 足りなかったら全然ダメ
  - どうする？

メモリを使う分だけ用意する

- メモリの動的確保
  - 使うときに使う分だけ，システムからメモリをもらう
  - 使い終わったらメモリをシステムに返す
  - C++なら，new と delete
  - C言語なら，malloc と free
  - でもC++には，簡単に使えるライブラリが用意
  - コンテナ・クラス
    - vector, list, map など多数

vector

- 配列に要素の追加と削除を可能
- データの順序を保持
- 様々な型のデータ（int, double, string など任意のクラス）を保管可能
  - テンプレート



vectorのサンプルソース

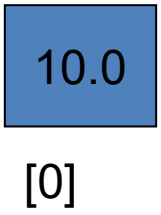
```
#include <vector>
#include <iostream>

int main()
{
	std::vector<double>  homework;
	homework.clear();
	homework.push_back(10.0);
	homework.push_back(5.3);
	homework.push_back(21.8);
	std::cout << homework[1] << std:: endl;
}
```

vectorのサンプルソースのイメージ

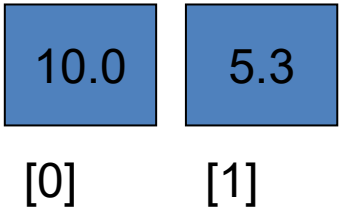
homework.clear();

データなし

homework.push_back(10.0);



homework.push_back(5.3);



homework.push_back(21.8);

サンプルソースの解説

- #include <vector>
  - vectorを使う時に必要なインクルードファイル
- std::vector<double> homework;
  - double型の（配列のような）vector の homework という名前の変数を宣言
- homework.clear();
  - homeworkという名前の変数に保存されているデータを消去する
- homework.push_back(10.0);
  - homeworkの最後に10.0というデータを追加
- homework[1];
  - homeworkの先頭から2番目のデータにアクセス

その他のvectorのメンバ関数(1/2)

- homework.pop_back();
  - 末尾の要素を削除
- homework.size();
  - homeworkに格納されている要素の数を返す
- std::vector<double>::size_type
  - vectorに格納されている要素の数を表現するための型
  - std::vector<double>::size_type size = homework.size();
- 他にもたくさんの関数がある
  - 例えば，insert(), erase()
  - これらはイテレータを学んでから、改めて学習

ソート

- データを大きさ順に並び替え
- C++では sort という関数が提供
- 対象はコンテナ，もちろんvectorも含まれる
- サンプルソース

```
#include <algorithm>
int main()
{
    std::vector<double> homework;
    //      すでにhomeworkにデータが格納されているとして
    std::sort( homework.begin(), homework.end() );
}
```

ソートのサンプルソースの解説

- #include <algorithm>
  - sort 関数が含まれるヘッダファイル
- std::sort(homework.begin(), homework.end());
  - sort(a, b);
    - aからbまでの範囲を昇順（非降順）に，つまりだんだん大きくなるようにソート
  - homework.begin()
    - homeworkの先頭の要素を表す
  - homework.end()
    - homeworkの末尾の要素の一つ後を表す

標準出力ストリームの書式

- 数字を出力するときの書式設定
  - 表示桁数，小数点以下の桁数
  - 固定小数点，指数表現
  - 右寄せ，左寄せ，など
- cout に書式の設定を出力，または書式設定の関数を適用
  - cout << setprecision(3) << a;
  - cout.precision(3); cout << a;
  - 上は両者とも小数点以下3桁で出力

オペレータと関数の違い

- オペレータ（操作子）
  - cout << setprecision(3) << a;
  - cout << b;
    - setprecision(3) はaの出力にのみ有効
    - b の出力はデフォルトの設定に戻る
- 関数
  - cout.precision(3);
  - cout << a;
  - cout << b;
    - この関数の適用以降ずっと（a も b も）有効

標準出力ストリームのサンプルソース

```
#include <iostream> //  標準ストリームを利用するときに必要
#include <iomanip>  //  書式設定を利用するときに必要

int main()
{
    //  129.65と出力される
    std::cout << std::setw(8) << std::setprecision(2) << 129.653 << endl;
    //  129.7と出力される
    std:: cout.precision(1);
    std::cout.width(8);
    std:: cout << 129.65395 << std::endl;
}
```

出力ストリームの書式設定

- std:: cout << std:: setprecision(n);
- std:: cout.precision(n);
  - 有効数字，小数点以下の桁数を指定
- std:: cout << std:: setw(n);
- std:: cout.width(n);
  - 出力するときの幅，文字数を指定
- std:: cout << std:: fixed;
- std:: cout.setf(std:: ios_base::fixed, std:: ios_base::floatfield);
  - 固定小数点での出力
- std::cout.setf(std:: ios::showpoint);
  - 小数点の強制表示
- 他にもたくさんの書式設定が可能

ストリームの評価

- std::cin >> x
  - 標準入力ストリームから x に代入
- (std:: cin >> x)
  - 上の代入文を「評価」
  - 正しく代入できたら，真(true)
  - 代入が失敗だったら（入力がない），偽(false)
  - while(std:: cin >> x){}
  - if(std:: cin >> x)　　{}

小技

- typedef
  - 変数の型の別名を定義
  - std:: vector<double>::size_type size = homework.size();
  - typedef std::vector<double>::size_type vec_sz;
  - 長い型名を vec_sz と短い別名
  - vec_sz size = homework.size();

std::vector< T >

std::vector< T >の特徴

- Tを格納される要素の型とする可変長の配列
- 各要素は線形に順序を保ったまま連続してメモリ格納される（配列と同じ）
- 配列と同じように，添え字やイテレータによるランダムアクセスが高速である
  - 各要素への添字アクセス（定数時間O(1)）
  - 全要素の両方向の走査（線形時間O(N)）
  - 末尾への要素の追加・削除（償却定数時間，たまに非常に長い時間がかかるが通常は短い時間で完了するために，平均的には定数時間と同等まで償却できるもの）
- 配列と違い，vectorでは自動的に領域の拡張が行われる

典型的なコード:データの追加・代入

```
//  Sample1.cpp
//  空のvectorにデータを追加
#include <vector>
#include <iostream>

int main() {
  const int n = 1000;
  //  int型で空のvectorを構築
  std::vector< int > v;
  //  0からn-1までの数値を末尾に追加
  for( int i = 0; i < n; i++ )
    v.push_back( i );
  //  0からn-1までの数値を画面に出力
  for( int i = 0; i < n; i ++ )
    std::cout << v[ i ] << " ";
  std::cout << std::endl;
  return 0;
}
```

```
//  Sample2.cpp
//  要素数が既知のvectorにデータを代入
#include <vector>
#include <iostream>

int main() {
  const int n = 1000;
  //  int型で要素数nのvectorを構築
  std::vector< int > v( n );
  //  第0要素から第n-1要素に数字を代入
  for( int i = 0; i < n; i++ )
    v[ i ] = n - i - 1;
  //  すべての要素を画面に出力
  for( int e : v )
    std::cout << e << " ";
  std::cout << std::endl;
  return 0;
}
```

典型的なコード:データの有無の判定
二部探索ならLog(N)で判定可能

```
//  Sample3.cpp
#include <vector>
#include <iostream>
#include <algorithm>
int main() {
  const int n = 1000;
  //  int型で要素数nのvectorを構築
  std::vector< int > v( n );
  //  第0要素から第n-1要素に数字を代入
  //  データは偶数で降順
  for( int i = 0; i < n; i++ )
    v.[ i ] = 2 * (n - i - 1) ;
  //  ソート
  std::sort( v.begin(), v.end() );
  //  昇順ののvectorに対して1000を二部探索
  bool is_found = std::binary_search( v.begin(), v.end(), 1000);
  if( is_found )
    std::cout << "1000 is in v" << std::endl;
  else
    std::cout << "1000 is not in v" << std::endl;
  return 0 ;
}
```

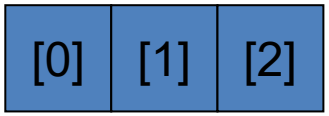
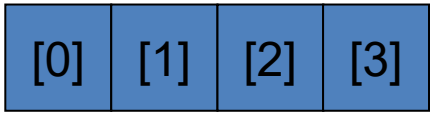
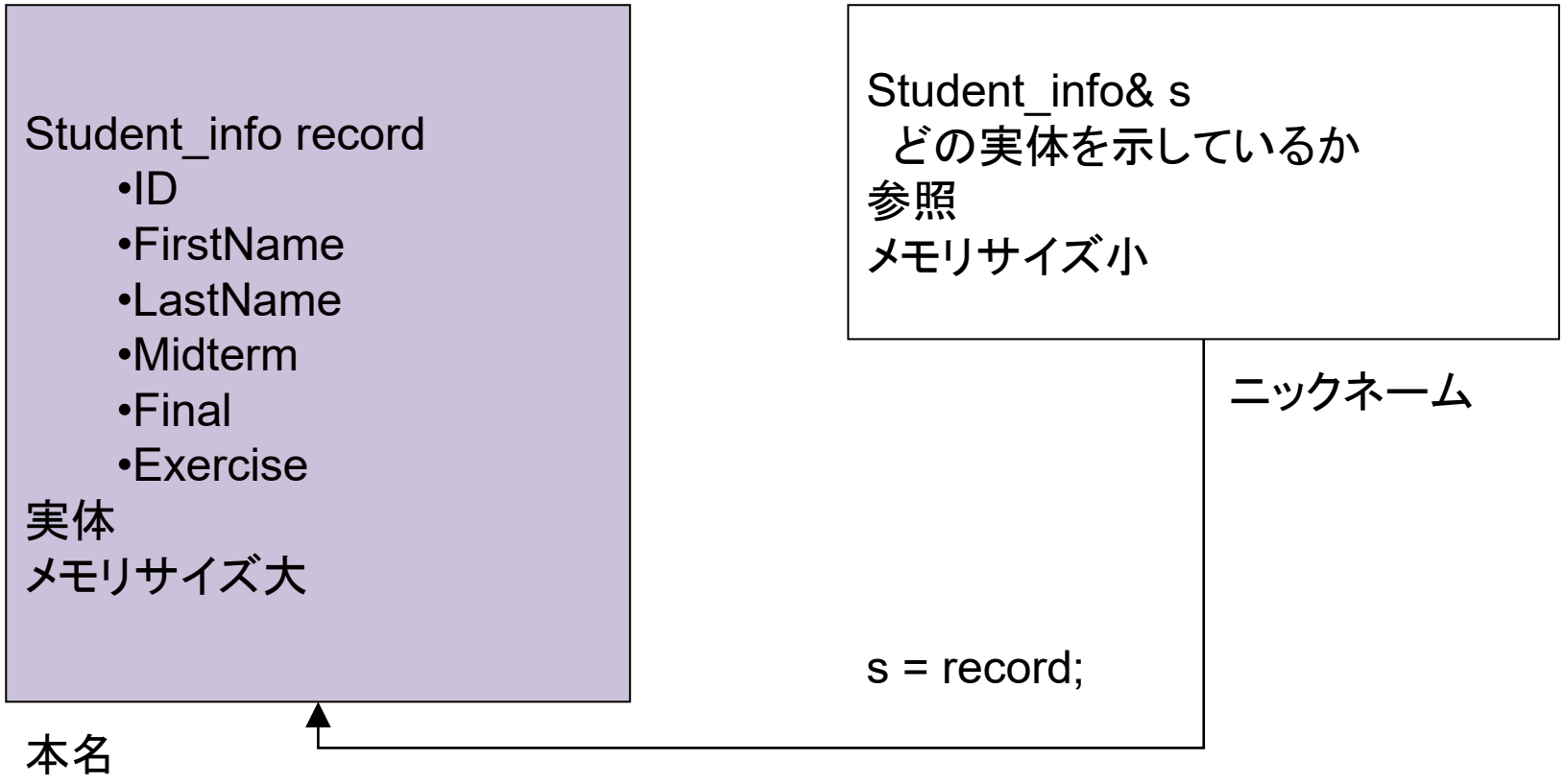
典型的なコード：与えられたデータ以上の位置を探索
二分探索ならLog (N)で判定可能

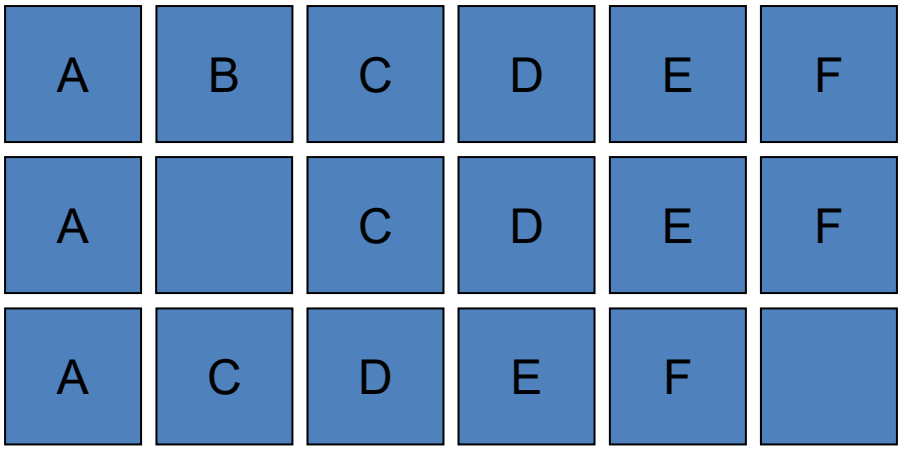
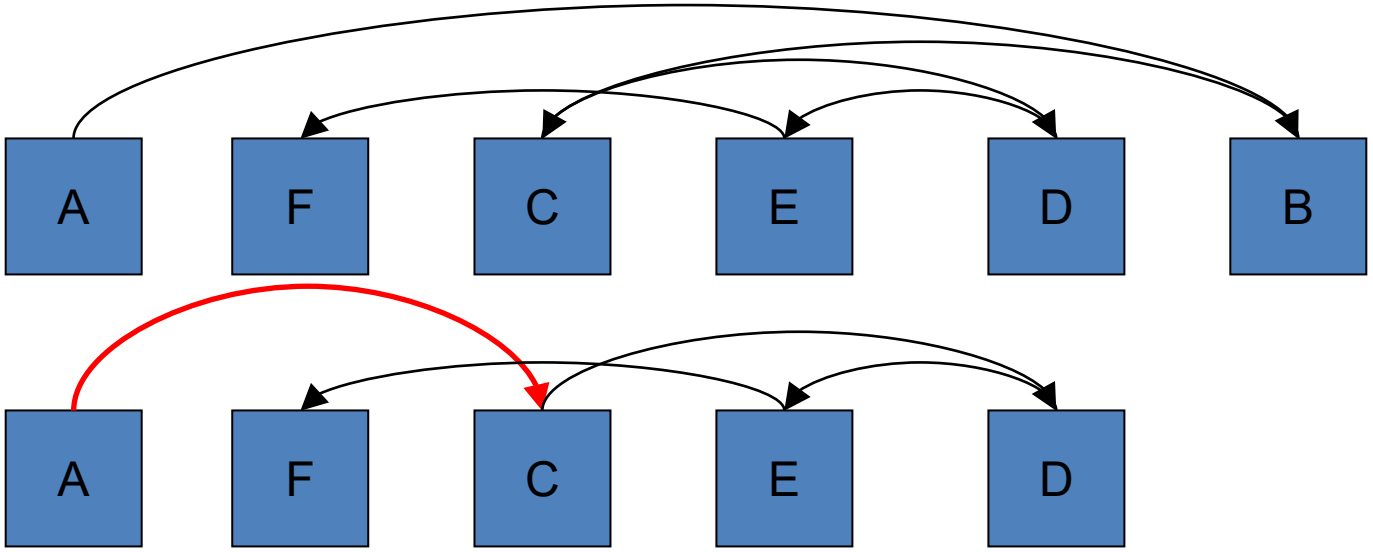
```
//  Sample3.cpp
#include <vector>
#include <iostream>
#include <algorithm>
int main() {
  const int n = 1000;
  //  int型で要素数nのvectorを構築
  std::vector< int > v( n );
  //  第0要素から第n-1要素に数字を代入
  //  データは偶数で昇順
  for( int i = 0; i < n; i++ )
    v.[ i ] = 2 * i;
  //  昇順のvectorに対してq以上のデータの位置を二部探索
  const int q = 999;
  std::vector<int>::iterator it = std::lower_bound( v.begin(), v.end(), q);
  if( it == v.end() )
    std::cout << q << "is larger than all of v" << std::endl;
  else
    std::cout << q << "is equal or smaller than " << *it << std::endl;
  return 0 ;
}
```

コンストラクタ（変数宣言と初期化）

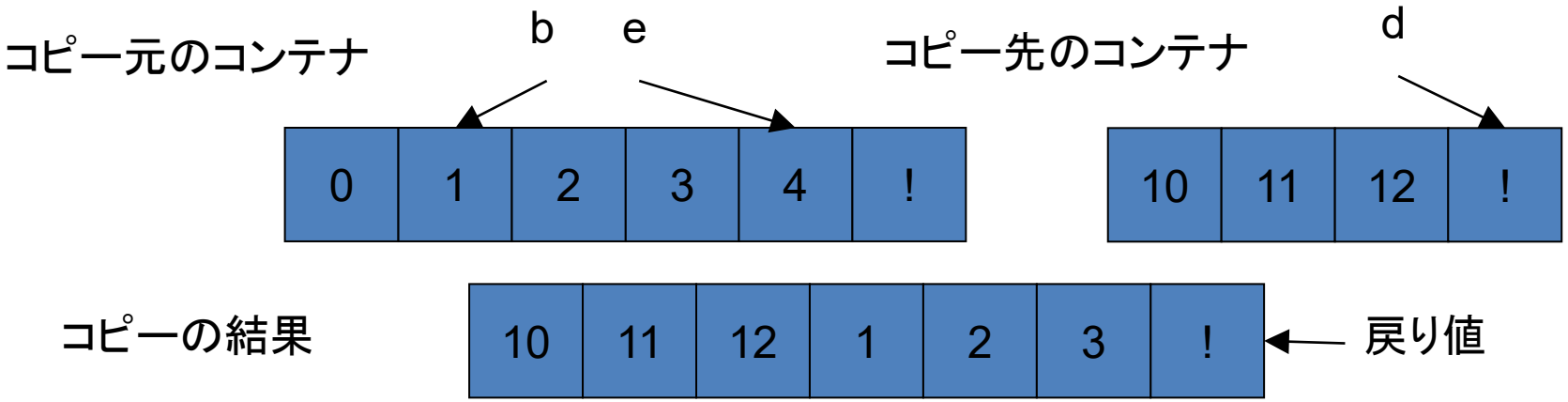
- `//  vector`を使う際に必要なインクルード
  ```
  #include <vector>
  ```
- `//  空(要素数0)の要素からなるint型のvector`を構築
  ```
  std::vector< int > v1;
  ```
- `//  4`個の要素からなる`vector`を構築，初期値は`int`のデフォルト値
  ```
  std::vector< int > v2( 4 );
  ```
- `// -1`で初期化された`8`個の要素からなる`vector`を構築
  ```
  std::vector< int > v3( 8, -1 );
  ```
- `//` イニシャライザ（初期化子）リストから`vector`を構築
  ```
  std::vector< int > v4 = {1, 2, 3};
  ```
- `//  イテレータ`で範囲を与えて`vector`を構築
  ```
  std::vector< int > v5( v4.begin(), v4.end() );
  ```
- `//  他の vector`と同じ値で`vector`を構築
  ```
  std::vector< int > v6( v5 );
  //  こちらに書き方でもOK
  std::vector< int > v6 = v5;
  ```

コンストラクタ（変数宣言と初期化）

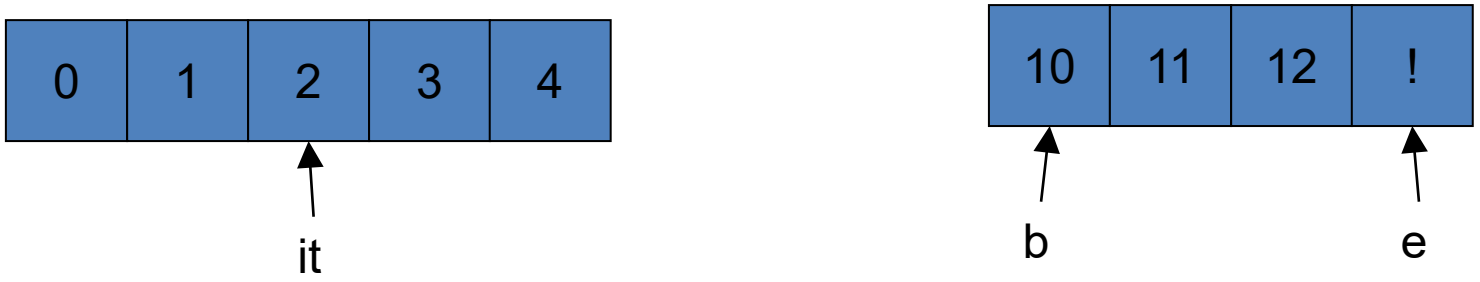
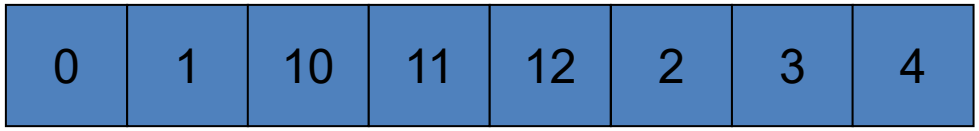
- ```
  //  int型の配列からvectorを構築
  int data[] = {4, 5, 6};
  std::vector< int > v7(data);
  //  こちらに書き方でもOK
  std::vector< int > v7 = data;
  ```
  ```

代入

- `//` コピー代入
```
std::vector< int > v8 = {1, 2, 3}, v9;
v9 = v8;
```
- `//` イニシャライザ(初期化子)リスト代入
```
std::vector< int > v10;
v10 = { 1, 2, 3 };
```

領域

- `std::vector< int > v = { 0, 1, 2 };`
- `//  vector`の要素数を取得する
  `int n = v.size();`
- `//  vector`の要素数を変更する
  `//  if n < v.size()` 現在の要素数よりも少ない`n`のときは，余剰な要素を削除する
  `n = 1;`
  `v.resize( n );`
  `//  if n > v.size()` 現在の要素数よりも多いのときは，足りない要素を追加する
  `n = 4;`
  `v.resize( n );`
- `//` 要素数が`0`かを判定する
- `if( v.empty() )`
  `  std::cout << "v is empty" << std::endl;`

コンテナの変更

- `std::vector< int > v;`
- `//  v`の末尾に要素`x`を追加する
  `v.push_back( x )`
- `//  v`の末尾に要素`x`を削除する
  `v.pop_back()`
- `//  イテレータit`で指定される位置に`x`を追加する. その後の要素は一つ後ろにずれる
  `v.insert( it, x )`
- `//  イテレータit`で指定される位置にイテレータ`b`から`e`の範囲の要素を追加する. その後の要素は一つ後ろにずれる
  `v.insert( it, b, e )`
- `//  イテレータit`で指定される位置の要素を削除する. その後の要素は一つ前にずれる
  `v.erase( it )`
- `//  イテレータb`から`e`の範囲の要素を削除する. その後の要素は一つ前にずれる
  `v.erase( b, e )`
- `//  `すべての要素を削除する
  `v.clear()`

イテレータ

- `std::vector< int > v;`
- `//` vの先頭を指すイテレータ
  `v.begin()`
  `//` 先頭からi番目の要素を指すイテレータ(vectorでは+-が使える)
  `v.begin() + i`
- `//` vの末尾を指すイテレータ. 要素は存在しない. 判定において要素が存在しないときにも使われる
  `v.end()`
- `//` vの先頭を指す読み込みイテレータ
  `v.cbegin()`
- `//` vの末尾を指す読み込みイテレータ
  `v.cend()`

イテレータ

- `std::vector< int > v;`
- `//  v`の末尾を指す逆イテレータ
  `v.rbegin()`
  `//  v`の先頭の前を指す逆イテレータ. 要素は存在しない
- `v.rend()`
- `//  v`の末尾を指す読み込み逆イテレータ
  `v.crbegin()`
  `//  v`の先頭の前を指す読み込み逆イテレータ
- `v.crend()`

要素アクセス

- `std::vector< int > v = { 0, 1, 2 };`
- `// 0`番目の要素を参照する. `a`を変更しても`v`の内容は変わらない
  ```
  int a = v[0];
  ```
  `// 1`番目の要素を参照する. `b`を変更したら`v`の内容も変わる
  ```
  int& b = v[2];
  ```
- `// n`番目の要素を参照する. 添え字範囲を超えらたら(`n >= size()`)例外を送出する
  ```
  int n = 0;
  int c = v.at( n );
  ```
- `//` 先頭の要素を参照する.
  ```
  int d = v.front();
  ```
- `//` 末尾の要素を参照する.
  ```
  int e = v.back();
  ```

std::vector<T>でよく利用するアルゴリズム

ソート

```
#include <algorithm>
std::vector<int> v = {8, 4, 2, 6};
//  std::sort( b, e, 判別関数 ), bからeを判別関数に従ってソートする
//  先頭[0]から末尾[3]までをデフォルトの判別関数（昇順）でソート
std::sort( v.begin(), v.end() );
//  昇順の判別関数を明示的に表現(<)
std::sort( v.begin(), v.end(), std::less<int>{} );
//  v[1]からv[2] までをデフォルトの判別関数（昇順）でソート
std::sort( v.begin()+1, v.end()-1 );
//  先頭[0]から末尾[3]までを降順でソート (>)
std::sort( v.begin(), v.end(), std::greater<int>{} );
```

```
#include <vector>
#include <algorithm>
const int x = 4;
std::vector<int> v = {2, 4, 6, 8};
if(std::binary_search( v.begin(), v.end(), x))
  std::cout << "found" << std::endl;
std::vector< int >::iterator = it;
it = std::lower_bound() v.begin(), v.end(), x));
if( it != v.end() )
  std::cout << x << “is greater than or equal to ” << *it
<< std::endl;
it = std::upper_bound() v.begin(), v.end(), x));
if( it != v.end() )
  std::cout << x << “is greater than ” << *it <<
std::endl;
```

演習課題4のポイント(1/2)

- データの読み取り方は2重ループ

```
//  外側のループは各行（学生一人ひとり）の読み取り
// 学生番号，姓，名，中間試験，期末試験などを読み込む
while()  {
      // 内側のループは演習の読み取り
      // -1が表れるまで読み続ける
      while()        {
            // vectorの末尾に追加;
      }
}
```

演習課題4のポイント(2/2)

- メジアン（中央値）
  - データを小さいものから順に並べたときの，中央の順位の値
  - 9件あるときは第5位
  - 10件あるときは，第5位と第6位の平均
  - vector, sort

a: ソートされているとして



メジアンはa[1]



メジアンは(a[1]+a[2])/2

第4章「プログラムとデータの構成」

主なトピック

- 構造体の定義の方法（クラスを定義するための準備段階）
- リファレンス（参照）型

プログラムの基本要素

- データ構造
  - 構造体
- 関数
  - 関数の定義，複数ファイルと分割コンパイル
- （クラス＝データ構造＋関数）

- 前回の演習は，データを処理しただけ
- 今回では，それを「構造化」していく
  - データ構造の定義と処理を関数として表現

データ構造

- 各学生のデータ
  - 学生番号: std::string ID;
  - 姓: std::string SurName;
  - 名: std::string GivenName;
  - 中間試験の成績: double Midterm;
  - 期末試験の成績: double Final;
  - 演習の成績: std::vector<double> Exercise;

構造体

- いくつかのデータをまとめて，新しい「型」をつくる
- 構造体の例

```
struct Student_info            {
      std::string  ID;            //  学生番号:
      std::string  SurName;  //  姓
      std::string  GivenName;           //  名
      double Midterm;          //  中間試験の成績:
      double Final;              //  期末試験の成績:
      std::vector<double> Exercise;   //  演習の成績:
};
```

構造体の定義

- struct 構造体の名前　　{
  - 変数の型　変数名;
  - これは，構造体のメンバ変数と呼ばれる
  - メンバ変数は，あるだけ続く
- };
- 構造体の定義の最後の「;」（セミコロン）を忘れずに
- 構造体の各メンバ変数へのアクセスの方法は
  - 構造体の変数名.メンバ変数名
  - これで普通の変数と同じように利用可能

構造体の使い方

- struct Student_info　　record;
  - Sutudent_infoという型（構造体）のrecordとい名前の変数
  - 学生一人分のデータ
- record.ID = “s0001”;
  - recordのIDというメンバ変数に”s0001”を代入
- std::cout << record.Final;
  - recordのFinalというメンバ変数の値を出力

学生全体のデータ

- std::vector< struct Student_info > students;
- 学生数が何人かは事前にはわからない
  - だからvectorを使う
- vectorの各要素が何を示すかというと
  - Student_info
- 使い方の例
  - students[0].ID
  - students.push_back(record);

関数の利用

- 処理のまとまりを関数（サブルーチン）にする
- 例えば，
  - read: データの読み込み
  - median: メジアンの探索
  - grade: 総合得点の計算
- 構造体と関数で，プログラムを見通しよくする（「構造化」）

read 関数の設計

- 概要
  - 学生1人分のデータを標準入力ストリーム(cin)から読み取り，Student_info 型の変数に保存する
- 関数の汎用性（他のプログラムでも使えるように）を考えて
  - データの入力はcin専用ではなく，他の入力ストリーム（例えばファイル入力）も使えるように，関数の引数にする

read 関数の設計

- std::istream& read(std::istream& is, struct Student_info& s)
- 引数
  - std::istream& is: 入力ストリーム(istream)の参照(&)型で，変数名がis
  - struct Student_info& record: struct Student_info型の参照(&)で，変数名がs
- 戻り値
  - std::istream&: 入力ストリーム，while文やif文で評価ができるように

参照（リファレンス）

- 参照
  - 別のところに実体がある変数を，他の名前で指し示す（「参照」する）
  - 使う時は，元の型に「&」をつける
- Student_info record;
  - 通常の変数，実体
- Student_info& s;
  - 参照，必ず実体を指し示す必要がある

参照（リファレンス）


Student_info record
•ID
•FirstName
•LastName
•Midterm
•Final
•Exercise
実体
メモリサイズ大
Student_info& s
どの実体を示しているか
参照
メモリサイズ小
ニックネーム
s = record;
本名

参照（リファレンス）

- 参照の使い方
- s = record;
  - これで s は record を指し示すことになる
  - s.ID = “s0002”; とすると record の内容が変更
  - record.ID = “s0003”; とすると s.ID も変化する
  - なぜなら s と record は同じものを表しているから

参照（リファレンス）

- 注意
  - 参照型の変数を宣言して，実体を示さずに使うことはできない
- 間違った例
  - Student_info& s; s.ID = “s0003”;
  - コンパイルエラー
  - これだと，sは実体を表していない
  - sは他の変数を参照しなければ使えない
  - 本名（実体）とニックネーム（参照）の関係

read 関数の使い方（ループの例）

```
int main()  {
   vector<Student_info> students;  //  学生全体のデータ
   Student_info record;  //  学生一人分のデータ
   //  標準入力ストリームからデータの読み込み
   //  読み込んだデータはrecordという変数に一時的に格納
   while( read( cin, record ) )   {
        //  recordをvectorに追加
        students.push_back( record );
   }
   //  読み込みデータがなくなるとループが終了
   //  read 関数の戻り値が偽： falseになるから
   //  データがないときにcinを評価すると false
}
```

read 関数の内容

```
std::istream& read( std::istream& is, Student_info& s )
{
    is >> s.ID >> s.GivenName >> s.SurName;
    is >> s.Midterm >> s.Final;
    //  演習の得点の読み込み，別の関数にする
    read_hw( is, s. Exercise );
    return is;
}
```

- 演習の得点の読み込みは，まとまった処理なので，別関数で行うことにする
- 引数のisを戻り値にすることで，関数を使いやすくしている

```
std::istream& read( std::istream& is, Student_info* s )
{
    is >> *s.ID >> *s.GivenName >> *s.SurName;
    is >> s->ID >> s-> GivenName >> s-> Surname;
}
```

read_hw 関数の設計

- 概要
  - 学生1人の演習の成績（回数不明）を入力ストリームから読み取り，vector<double>&型の変数に保存する
- 今回のデータの形式
  - s10000001 Taro Aizu 80 20 100 100 100
  - s10000002 Jiro Aizu 100 90 90
  - 演習の最後に -1がない

データの読み込み方法

- データの末尾に-1がないのに，どうやって各学生のデータの終わりを見つけるか？
  - 各学生のデータの先頭は学生番号，string型（文字列）
  - 各学生のデータの末尾は期末試験か演習の成績，double型（数値）
  - 演習の成績（double型）を読んでるつもりで，次の学生の学生番号（string型）を読み込むとエラー
- このエラーを利用

read_hw 関数の内容

```
std::istream&  read_hw(std::istream& is, std:: vector<double>& v)
{
   if( is )    {
      double x;  //  毎回の演習の成績
      v.clear();
      while( is >> x ){
          v.push_back( x );
      }
      is.clear();
    }
    return is;
}
```

read_hw 関数の解説

- std::istream& read_hw(std::istream& is, std::vector<double>& v)
  - 第1引数: 入力ストリームの参照
    - この関数の中で内容を更新するので参照型
  - 第2引数: 演習の成績保存するvectorの参照
    - この関数の中で内容を更新するので参照型
  - 戻り値: 入力ストリーム
    - 他の関数から使いやすくするように

read_hw 関数の解説

- if(is) {}
  - もし入力ストリームが空とかエラー状態だったら何もしない
- v.clear();
  - 初期化，前の学生の処理でvectorにデータが残っているかもしれない
- while(is >> x) { v.push_back(x); }
  - データをdouble型のxに読み込み，vectorに追加
  - 読み込みが失敗したときに，ループから脱出
  - ループが終わるのは，
    - 入力がなくなったときか
    - double型に数値以外のデータを代入しようとしたとき

read_hw 関数の解説

- is.clear();
  - 入力ストリームのエラーを解除
  - とくにdouble型に数値以外を代入のケース
- return is;
  - 使いやすいように関数の戻り値に入力ストリーム

median 関数の設計

- 概要
  - 演習の成績(std::vector<double>)を与えて，メジアンの値を返す
- double median(std::vector<double> v)
  - 引数は，参照型ではないことに注意
  - メジアンの計算では演習の成績をソートする必要
  - ここでは，元のデータの並びを変更しないようにして，元のデータのコピーを関数に渡す
  - 引数に「&」がないことに注意

関数の引数と参照

- 参照をよく使うのは，関数の引数
  - 参照型：関数の中で値を変更し，それを関数の外でも反映させたいときに使う
  - 通常：関数の中で値を変更しても，関数の外では元のままであって欲しいときに使う
  - 引数がコピーされ，コピーが関数に渡される

median 関数の中身

```
double median(std::vector<double> v)
{
   std::vector<double>::size_type size = v.size(), mid;
   if( size == 0 )
        throw domain_error("要素数が0のメジアン");
   std::sort( v.begin(), v.end() );
   mid = size / 2;
   if( size % 2 == 0 )
        return (v[ mid ] + v[ mid - 1 ]) / 2;
   else
        return v[ mid ];
}
```

例外処理

- 演習の回数が0だったとき
  - メジアンや平均点が計算できない
  - 暫定的にメジアンや平均点を0とできるが,
  - できれば例外として扱いたい
- C++における例外処理（詳しくは，また後で）
  - #include <stdexcept>が必要
  - throw: 例外が起こったときに,「例外を投げる」
  - try: 例外を見つける範囲を決める
  - catch: 投げられた例外を捕まえたときに，どう処理するか

grade 関数の設計

- 概要
  - 中間試験，期末試験，演習の成績を与えて，演習メジアン，演習平均，演習合計，総合得点を計算する
- double grade(double midterm, double final, const std::vector<double>& hw, double& ex_med, double& ex_avg, double& ex_sum)
  - constは，変数を定数として扱い，値を変更しない
  - const std::vector<double>& hw は，hwのデータを見るだけで，データは変更しない

grade 関数の設計

- 関数の内部で値を変更したくないときに，便利
- double& ex_med などは値を書き換え，関数の外側でもそれを使うので，参照型

grade 関数の中身

```
double grade(double midterm, double final, const
   std::vector<double>& hw, double& ex_med, double& ex_avg,
   double& ex_sum)
{
   if( hw.size() == 0)
        std::throw std::domain_error("演習回数が0");
   ex_med = median( hw );
   ex_sum = 0;
   for( std::vector<double>::size_type i = 0; i !=
   hw.size(); ++i){
        ex_sum += hw[i];
   }
   ex_avg = ex_sum / hw.size();
   return 0.2 * midterm + 0.4 * final + 0.4 * ex_med;
}
```

学生を名前順にソートして出力

- GivenName順にソート
  - もしGivenNameが同じだったら，次はSurNameでソート
- std::vector<Students_info> students　をソートする
  - 演習のソート，vector<double>のときは，大小関係は明らか
  - しかし，Student_info型の大小は決められていない
  - ソートのときにStudent_info用の大小関係を指定

vector<Students_info>のソート

- std::vector<Students_info> students
- std::sort(students.begin(), students.end(), compare);
  - students.begin() から students.end() までをcompareという関数に記述される大小関係に従ってソート
  - bool compare (const 型名& x, const 型名& y)
  - x が y より小さいときに真(true)を返す関数

compare 関数の内容

```
bool compare( const Student_info& x, const Student_info& y )

{
   //   xと y の GivenName が異なるときは GivenName を比較
   if( x. GivenName != y. GivenName )
        return( x. GivenName < y. GivenName );
   //  xと y の GivenName が同じときは SurName を比較
   else
        return( x. SurName < y. SurName );
}
```

main 関数

```
int main()
{
   std::vector<Student_info>  students;
   Student_info  record;
   //  まずデータの読み込み
   while( read( cin, record ) )       {
        students.push_back( record );
   }
   //  学生データを名前順にソート
   std::sort( students.begin(), students.end(), compare);
```

main 関数

```
//   学生のデータを出力
for( std::vector<Student_info>::size_type i = 0; i !=
    students.size(); ++i)  {
    //   総合得点，演習メジアン，演習平均，演習合計
    double total, ex_med, ex_avg, ex_sum;
    //   計算
    //   grade関数の中で例外が発生するかもしれないので
    try     {
        total = grade( students[ i ].Midterm,
        students[ i ].Final, students[ i ].Exercise,
        ex_med, ex_avg, ex_sum);
        //   画面にデータを出力（ここでは省略）
    }
    catch(std:: domain_error)  {
        //   ここに例外時の処理を書く
        //   今回はID, FirstName, LastNameの後に，
        //   演習回数が0だから総合得点が計算できない
        //   というエラーメッセージを出力
    }
}
```

第5章「シーケンシャルコンテナとstringの解析」

今日の内容

- list型
- イテレータ（iterator，反復子）

2種類のデータアクセス法

- ランダムアクセス
  - いろんな場所に一発でアクセスできる
  - 配列やvector型, data[i]
- シーケンシャルアクセス
  - 先頭から順番にしかアクセスできない
  - list型，ループとイテレータを使ってアクセス

vector型とlist型の特徴

- 両方ともコンテナ
  - 同じようにデータの追加が可能
- vector型
  - 要素へのアクセスが添え字を使える
- list型
  - 要素の追加や削除が高速

要素を削除するときのイメージ

vector: メモリ上に順に並んでいる



- データを詰める作業が必要
- データ数に依存した処理（データが多いときに前の方のデータを削除すると処理に時間がかかる）
- データの追加も同様

要素を削除するときのイメージ

list: 各データはメモリ上にバラバラに存在
各データが次のデータが何かを格納している



- データの接続だけを変更
- データ数に依存しない処理
- データの追加も同様に高速

list型

- 使い方は，vectorと共通なものもある
  - 変数の宣言
  - データの追加
- しかし，まったく異なる使い方もある
  - ループ
  - データへのアクセス
- ちょっとだけ違う使い方もある
  - ソート

list型のサンプルソース

```
#include <list>
//  前回の授業で定義した構造体 Student_info を使う
//  read 関数も同様
int main()
{
   list< Student_info > students;
   Student_info record;
   while( read( cin, record) )    {
        students.push_back( record );
   }
   //  前回の授業で定義した compare 関数
   students.sort( compare );
```

list型のサンプルソースの解説

- #include <list>
  - list型を使うときに必要なヘッダーファイル
- list< Student_info > students;
  - 変数の宣言
  - list<型名> 変数名;
- students.push_back(record);
  - データ(record)をlist型の変数 students の末尾に追加
- ここまではvector型と同じ操作法

list型のサンプルソースの解説

- students.sort(compare);
  - リスト型のデータのソート
  - list<double> のように型の大小関係が自明のときは compare 関数を省略することができる
  - vector型のときは，sort(students.begin(), students.end(), compare)
    - 上のように3つの引数をとる sort 関数を利用できるのはランダムアクセス型の変数のとき
    - vectorの他には，string型など

list型のサンプルソース

```
    //   さっきのソースの続き
    list< Student_info >::iterator it = students.begin();
    while( it != students.end() )  {
        //  画面にIDとFirstName, LastNameを出力
        cout << (*it).ID << " " << (*it).FirstName << " "
               << (*it).LastName << endl;
        ++ it;
    }
}
```

- vector型のときは for 文（インデックス）が使えた
- list型では 添え字は使えない，その代わりにイテレータと while 文を使う

list型のサンプルソースの解説

- list< Student_info >::iterator it = students.begin();
  - list< Student_info >::iterator という型の it という名前の変数を宣言
  - iter に students の先頭要素であるstudents.begin() で初期化
  - この it という変数がイテレータ
- while(it != students.end()) {}
  - it が students.end() と異なる間ループを実行

list型のサンプルソースの解説

- cout << (*it).ID
  - cout に イテレータ(it)が指しているデータ（Student_info型の変数）のIDを出力する
  - 通常の構造体の変数だと record.ID
  - イテレータの場合は (*it).ID
- ++ it;
  - イテレータをインクリメント，次の要素に進む

イテレータと添え字（インデックス）

- イテレータは添え字（インデックス）に似ているところもある
  - ++ it; // 整数を1増やすのに似ている
- しかし，違うところもある
  - it = students.begin();
    - 上の文は，studentsに蓄えられた先頭を指し示す，整数を代入しているわけではない
  - (*it).ID
    - (*it)とするとイテレータがデータを指すことになる
    - 配列やvectorだと，students[i]

イテレータとは

- コンテナに含まれている要素を指し示すもの
  - 要素そのものではない
    - cout << it.ID とはできない
  - 要素にアクセスするときは，イテレータの先頭に＊を付ける
    - cout << (*it).ID なら大丈夫
    - または，cout << it->ID と書いてもOK
  - 次の要素を指したいときは，イテレータをインクリメント
    - ++ it

正確に言うと

- これまでは，students.begin()は students の先頭要素を示すと言っていた
- 正確には，studentsの先頭の要素を指すイテレータ
- 同様に students.end() は最後の要素の一つ後を指すイテレータ
- 繰り返しになるが
  - イテレータは要素そのものではない
  - しかし，要素を直接扱ってるように手軽に扱える

イテレータの種類

- list< Student_info >::iterator it;
  - 通常のイテレータ,
  - 読み書き可能（イテレータが指す要素の内容を書き換えできる）
    - it-> ID = “s0001”;
- list< Student_info >::const_iterator it;
  - コンスト・イテレータ
  - 読み込みのみ可能（内容を書き換えは不可）
    - it-> ID = “s0001”; とするとエラーになる

std::list< T >

std::list< T >の特徴

- Tを格納する双方向リンクリスト（配列とは異なる）
- 任意の位置への挿入や削除を定数時間で行う事が出来るが、高速なランダムアクセスは出来ず、常にシーケンシャルアクセスを行う必要がある。
  - 各要素へのアクセスは先頭または末尾からのシーケンシャルアクセス（O(N)）
  - 任意の位置への挿入と削除は定数時間（O(1)）

典型的なコード:データの追加・代入

```
//  Sample1.cpp
//  空のlistにデータを追加
#include <list>
#include <iostream>

int main() {
  const int n = 1000;
  //  int型で空のlistを構築
  std::list< int > l;
  //  0からn-1までの数値を末尾に追加
  for( int i = 0; i < n; i++ )
    l.push_back( i );
  //  0からn-1までの数値を画面に出力
  for( std::list<int>::iterator it =
l.begin(); it != l.end(); it ++ )
    std::cout << *it << " ";
  std::cout << std::endl;
  return 0;
}
```

```
//  Sample2.cpp
//  要素数が既知のlistにデータを代入
#include <list>
#include <iostream>

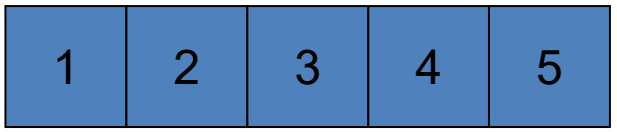
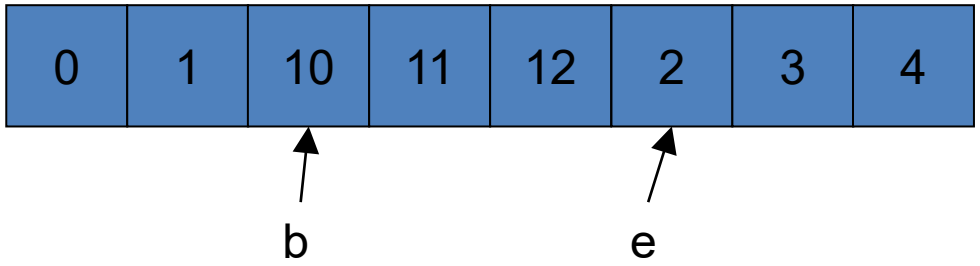
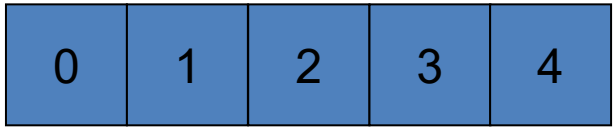
int main() {
  const int n = 1000;
  //  int型で要素数nのlistを構築
  std::list< int > l( n );
  //  リストの各要素に数字ををを代入
  int i = 0;
  for( std::list<int>::iterator it =
l.begin(); it != l.end(); it ++ )
    *it = i++;
  //  すべての要素を画面に出力
  for( int e : l )
    std::cout << e << " ";
  std::cout << std::endl;
  return 0;
}
```

典型的なコード：listのソート

```
//  Sample3.cpp
#include <list>
#include <iostream>
int main() {
  const int n = 1000;
  //  int型の空のlistを構築
  std::list< int > l;
  //  先頭にデータを追加
  for( int i = 0; i < n; i++ )
    l.push_front( i );
  //  ソート，リストのメンバ関数として実行
  l.sort();
  for( auto e : l )
    std::cout << e << " ";
  std::cout << std::endl;
  return 0 ;
}
```

典型的なコード：条件を満たす要素の削除

```
//  Sample4.cpp
#include <list>
#include <iostream>
int main() {
  const int n = 1000;
  std::list< int > l;
  for( int i = 0; i < n; i++ )
    l.push_front( i );
  for( auto it = l.begin(); it != l.end();) {
    // 要素が偶数なら削除
    if( *it % 2 == 0 ) {
      // 削除された要素の次を指すイテレータが返される。
      it = l.erase( it );
    }
    // 要素削除をしない場合に、イテレータを進める
    else {
      ++it;
    }
  }
  for( auto e : l )
    std::cout << e << " ";
  std::cout << std::endl;
  return 0;
}
```

コンストラクタ（変数宣言と初期化）

std::list< T >のコンストラクタの書式はstd::vector< T >のそれと同じ

- // listを使う際に必要なインクルード
  ```
  #include <list>
  ```
- // 空（要素数0）の要素からなるint型のlistを構築
  ```
  std::list< int > l1;
  ```
- // 4個の要素からなるlistを構築， 初期値はintのデフォルト値
  ```
  std::list< int > l2( 4 );
  ```
- // -1で初期化された8個の要素からなるlistを構築
  ```
  std::list< int > l3( 8, -1 );
  ```
- // イニシャライザ（初期化子）リストからlistを構築
  ```
  std::list< int > l4 = {1, 2, 3};
  ```
- // イテレータで範囲を与えてlistを構築
  ```
  std::list< int > l5( l4.begin(), l4.end() );
  ```
- // 他のlistと同じ要素でlistを構築
  ```
  std::list< int > l6( l5 );
  // こちらに書き方でもOK
  std::list< int > l6 = l5;
  ```

代入

- std::list< T >の代入の書式はstd::vector< T >のそれと同じ
- // コピー代入
```
std::list< int > l8 = {1, 2, 3}, l9;
l9 = l8;
```
- // イニシャライザ(初期化子)リストを代入
```
std::list< int > l10;
l10 = { 1, 2, 3 };
```

領域

- std::list< T >の領域関数はstd::vector< T >のそれと同じ
- std::list< int > l = { 0, 1, 2 };
- // listの要素数を取得する
  int n = v.size();
- // vectorの要素数を変更する
  // if n < l.size() 現在の要素数よりも少ないnのときは，余剰な要素を削除する
  n = 1;
  l.resize(n);
  // if n > l.size() 現在の要素数よりも多いのときは，足りない要素を追加する
  n = 4;
  l.resize(n);
- // 要素数が0かを判定する
- If(l.empty())
    std::cout << "l is empty" << std::endl;

コンテナの変更

- std::list< T >のコンテナ関数はstd::vector< T >のそれとほぼ同じ
- std::list< int > l;
- // lの末尾に要素xを追加する
  l.push_back(x)
- // lの末尾に要素xを削除する
  l.pop_back()
- // lの先頭に要素xを追加する
  l.push_front(x)
- // lの先頭の要素を削除する
  l.pop_front();
- // イテレータitで指定される位置にxを追加する
  l.insert(it, x);
- // イテレータitで指定される位置にxをn要素追加する
  l.insert(it, n, x);
- // イテレータitで指定される位置にイテレータbからeの範囲の要素を追加する
  l.insert(it, b, e);

コンテナの変更

- `//　イテレータitで指定される位置にイニシャライザリストの要素を追加する`
  `l.insert( it, {1, 2, 3} );`
- `//　イテレータitで指定される位置の要素を削除する. リターン値は削除された要素の次を指すイテレータ. そのような要素が存在しないときはl.end()を返す`
  `l.erase( it );`
- `//　イテレータbからeの範囲の要素を削除する. リターン値は削除された要素の次を指すイテレータ. そのような要素が存在しないときはl.end()を返すl.erase( b, e );`
- `//　すべての要素を削除する`
  `l.clear()`

リスト操作

- `std::list< int > l = {1, 2, 3};`
- `//` リスト`l`をソートする．デフォルトは昇順
  `l.sort();`
- `//` リスト`l`を降順でソートする．`std:greater`を使って降順を指定する
  `l.sort( std::greater<int>{} );`
  `//` ラムダ関数を使って，降順の判別関数を与える
  `l.sort( [](int a, int b) {return a > b} );`
- `//` リスト`l`を反転する
  `l.reverse();`
- `//` ソート済みのリストから重複した要素を削除する
  `l.unique();`

イテレータ

- std::list< T >のイテレータはstd::vector< T >のそれと同じ
- std::list< int > l;
- // lの先頭を指すイテレータ
  l.begin()
- // std::list< T >ではイテレータの足し算はできない
  // l.begin() + i
- // nの要素先を指したいならitをインクリメントする
  std::list< int >::iterator it = l.begin();
  for(int i = 0; i < n; i ++)
    it++;
- // lの末尾を指すイテレータ. 要素は存在しない. 判定において要素が存在しないときにも使われる
  l.end()
- // vの先頭を指す読み込みイテレータ
  l.cbegin()
- // vの末尾を指す読み込みイテレータ
  l.cend()

イテレータ

- `std::list< int > l;`
- `//  l`の末尾を指す逆イテレータ
  `ld.rbegin()`
  `//  l`の先頭の前を指す逆イテレータ. 要素は存在しない
- `l.rend()`
- `//  l`の末尾を指す読み込み逆イテレータ
  `l.crbegin()`
  `//  l`の先頭の前を指す読み込み逆イテレータ
- `l.crend()`

要素アクセス

- `std::list< int > l = { 0, 1, 2 };`
- `//` **std::list< T >では添え字[]もat()も使えない**
- `//` **先頭の要素を参照する.**
  `int d = l.front();`
- `//` **末尾の要素を参照する.**
  `int e = l.back();`
- `//` **要素アクセスはイテレータを使う**

```
for( std::list< int >::iterator it = l.begin(); it != l.end(); ++ it)
  std::cout << *it << " ";
std::cout << std::endl;
```

文字列処理

- string型はvector型と同じようにループを組むことができる

```
#include <cctype>
string s = “Sample text”;
for( string::size_type i = 0; i != s.size(); ++i ) {
    //  文字列sを各文字単位で小文字に変換して出力
    cout << tolower( s[ i ] );
}
```

- tolower(c)はcを小文字に変換する関数
  - #include <cctype>が必要

文字列処理

- s.substr(i, j);
  - i から始まり j 文字分の新しいstringを生成
  - 先頭を1文字削るには
    - string tmp = s.substr(1, s.size() -1);
  - 末尾を1文字削るには
    - string tmp = s.substr(0, s.size() - 1);

演習課題5のポイント

- 構造体を作る，例えば
```
struct WordCount {
    string Word;
    int Count;
};
```
- 標準入力ストリームから読み込んだ文字列をテキスト処理
- その後WordCount型に格納してから，list型に保存する
- そして，ソートと出力

演習課題5のポイント

```
string s;
list<WordCount> words;
while(cin >> s)        {
    //  sの文字列処理をここに入れる
    int isFound =0;
    list<WordCount>::iterator iter = words.begin();
    while( iter != words.end() )      {
        if( (*iter).Word == s) {
            isFound = 1;
            //   回数を1増やす処理を入れる
            break;
        }
        ++iter;
    }
    if(isFound == 0)   {
       //   データをwordsに追加する処理を入れる
    }
}
```

第6章「ライブラリのアルゴリズムを使う」

今日の内容

- ジェネリック・アルゴリズム（汎用アルゴリズム）
- イテレータの計算
- `copy`
- `insert`
- `erase`
- `search, find, find_if`
- `remove, remove_if , remove_copy, remove_copy_if`
- `partition, stable_partition`

ジェネリック・アルゴリズム

- 汎用アルゴリズム
  - 特定のコンテナに付随しているのではなく，引数の型からデータ形式を判断し，多くのコンテナに共通して作用できるアルゴリズム
  - `vector, list, string`型などを対象に同じ関数で利用可能
  - 通常，引数はイテレータで与えられる
  - `copy, search`など
    - 異なる例：`c.insert(), c.erase()`，クラスのメンバ関数
  - `#include <algorithm>`

copy

- `copy( b, e, d )`
  - 入力イテレータ `b` と `e` (eは含まない) の間の値を，出力イテレータ `d` で示される位置にコピーする
  - コピー先の最後の要素の一つを後を指すイテレータが返される
  - 同じ要素の型なら異なる種類のコンテナ間(`vector<T>`と`list<T>`など)でも適用可能



copyの例

```
vector<int> src, dst;
//  src[0] = 0; src[1] = 1; src[2] = 2; src[3] = 3; src[4] = 4;
//  dst[0] = 10; dst[1] = 11; dst[2] = 12;

//  srcの内容をすべてdstの最後にコピーする
copy( src.begin(), src.end(), back_inseter( dst ) );

//  srcのbからeの範囲をdstの最後にコピーする
vector<int>::iterator b, e;
//  bとeのイテレータを設定，例えば
b= ++(src.begin());  e = --( src.end() );
copy( b, e, back_inserter( dst ) );
```

イテレータアダプタ

- イテレータを返す関数
  - とくに以下のインサータがよく使われる
- `#include <iterator>`の中で定義
- `back_inserter( c );`
  - `c` というコンテナの最後に要素を追加するための出力イテレータを返す
  - `push_back` メンバ関数を持ってるコンテナにのみ利用可能(`vector`型, `list`型, `string`型など)

イテレータアダプタ

- `front_inserter( c );`
  - `c` というコンテナの先頭に要素を追加するための出力イテレータを返す
- `inserter( c, it );`
  - `c` というコンテナの `it` が指す要素の直前に値を挿入するための出力イテレータを返す

insert

- `c.insert( it, t )`
  - コンテナ `c` のイテレータ`it` が指す要素の直前に `t` を挿入する
  - 新しく挿入した `t` を指すイテレータを返す
- `c.insert( it, b, e )`
  - 同様にイテレータ`b` から `e` の間のシーケンスを挿入する，`e`そのものは挿入されない
  - `void` を戻す

insertのイメージ

- `c.insert( it, b, e )`

コンテナc



結果



insertの例

```
vector<int> src, dst;
//  src[0] = 0; src[1] = 1; src[2] = 2; src[3] = 3; src[4] = 4;
//  dst[0] = 10; dst[1] = 11; dst[2] = 12;

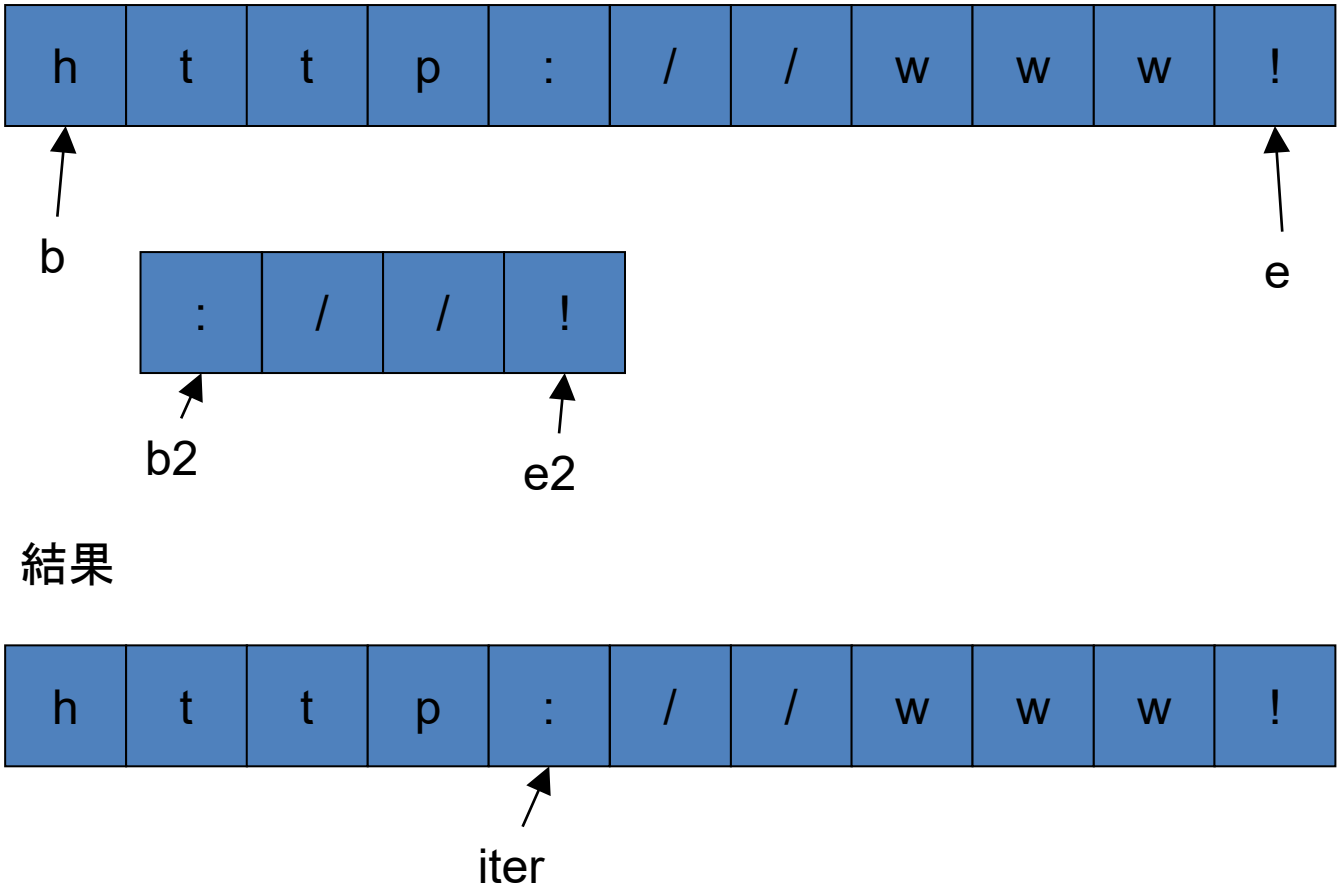
//  dst の先頭に src の全体を挿入
dst.insert( front_inserter( dst ), src.begin(), src.end() );

//  dst の2番目の要素の直前に src の全体を挿入
vector<int>::iterator it;
it = dst.begin(); ++it;
// dst.insert( it, src.begin(), src.end() );
dst.insert( inserter(dst, it), src.begin(), src.end() )
```

ループの中でのinsertの例

```
vector<int> src;
vector<int>::iterator it = src.begin();
int x;

while( it != src.end() ){
   if( 何らかの条件 ){
        //  もとのitの位置にxが挿入され,
        //  挿入された要素のイテレータがリターンされる
        it = src.insert(it, x);
        //  このインクリメントでもとのitの位置になる
        it ++;
   }
   //   次の要素に進める
   ++it;
}
```

イテレータの計算



- vectorとlistでは++と--が共通して使える
- vectorではさらに+と-が使える
  listでは使えない
- vector<int> src;
- vector<int>::iterator iter;
- iter = src.begin();
  - iter+1; // srcの2番目の要素を指すイテレータ
  - cout << *(iter+1); // 「2」が出力
- iter = src.end (); // 末尾の要素の一つ後を指す
  - iter-1; // srcの最後の要素を指すイテレータ
- iter + nやiter - nといった計算が可能

erase

- `c.erase( it )`
  - コンテナ `c` のイテレータ`it` が指す要素を削除する
  - 削除されたイテレータの直後を指すイテレータを返す
- `c.erase( b, e )`
  - 同様にイテレータ`b` から `e` の間のシーケンスを削除する
  - `e`そのものは削除されない
  - 削除された要素の直後を指すイテレータを返す

eraseのイメージ

- c.erase(b, e)

コンテナc



結果



ループの中でのeraseの例

```
vector<int> src;
vector<int>::iterator it = src.begin();

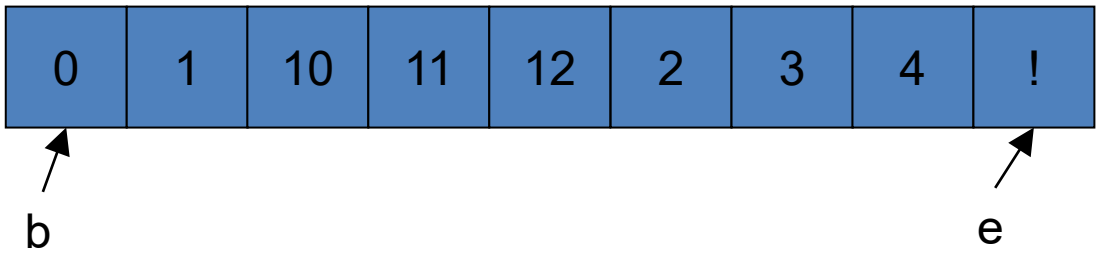
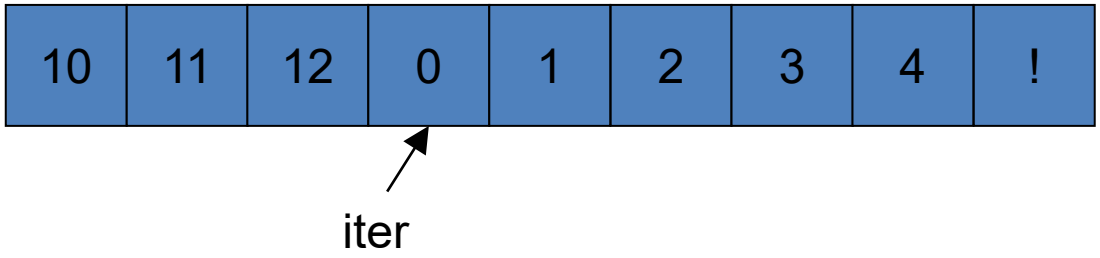
while( it != src.end() ){
   if( 何らかの条件 ){
        //   もとのitを削除し，その次の要素のイテレータをリターンする
        //   これで次の要素を指すことになる
        it = src.erase( it );
   }
   else {
        //   次の要素に進める
        ++it;
   }
}
```

eraseの例

```
vector<int> src;
vector<int>::iterator b, e;

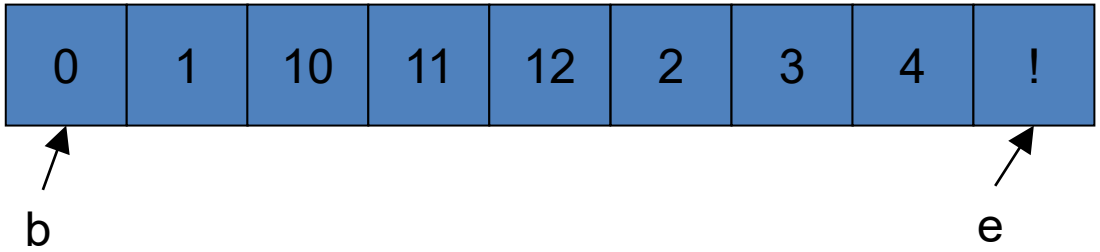
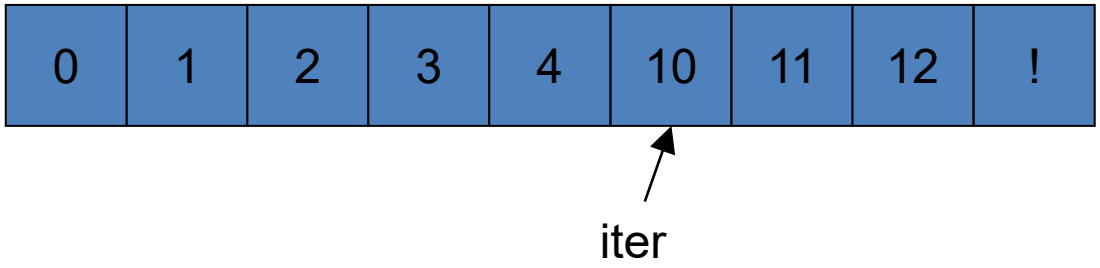
//  先頭から末尾の要素を削除
src.erase( src.begin(), src.end() );

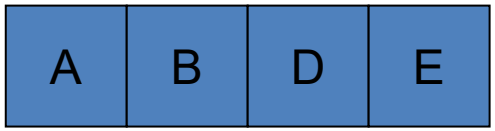
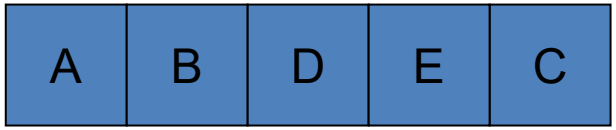
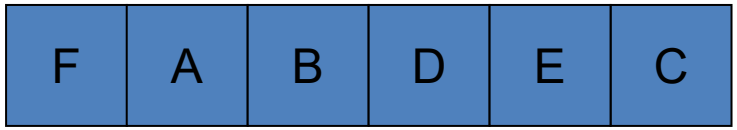
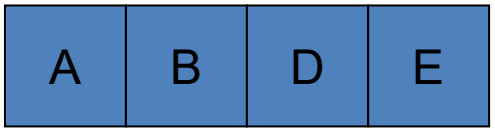
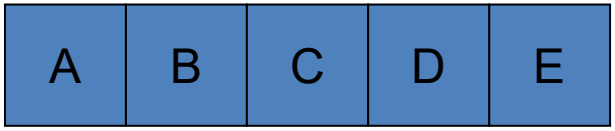
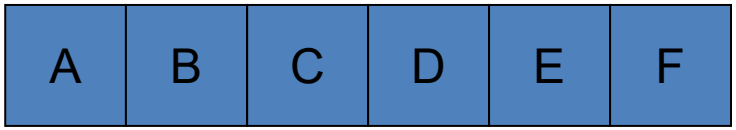
b = src.begin() + 1;
e = src.begin() + 5;
//  2番目の要素から5番目の要素を削除
src.erase( b, e );
```

search

- search(b, e, b2, e2)
- search(b, e, b2, e2, p)
  - イテレータ b と e で指定される範囲の中から，イテレータb2 から e2 のシーケンスが最初に現れる位置を示すイテレータを返す
  - マッチする文字列がなければ，e を返す
  - 上の例では==等しいかの判定をする
  - 下の例では判別関数pで等しいかを判定する

searchのイメージ



searchの例

```
string src = “http://www.u-aizu.ac.jp/”
string sep = “://”;
string::iterator it;

//  srcからsepの文字列に位置を探す
it = search( src.begin(), src.end(), sep.begin(),
sep.end() );
//  sepが含まれているので，iterは「:」の位置を指すことになる
```

find, find_if

- find(b, e, t)
  - イテレータ b と e のシーケンスから t と最初に一致するイテレータを返す
- find_if(b, e, p)
  - 同様に判別関数pと最初に一致するイテレータを返す
- findは要素，searchはシーケンスを探す

find_ifの例

```
string src = “http://www.u-aizu.ac.jp/”
string::iterator iter;

//  srcからアルファベット以外の文字列が最初に現れる位置を探す
iter = find_if( src.begin(), src.end(), compare );

bool compare(char c)
{
   return ( !isalpha(c) );
}
```

remove, remove_if

- remove(b, e, t)
- remove_if(b, e, p)
  - イテレータ b から e の間のシーケンスを並び替え，要素 t と一致しない要素，あるいは判別関数 p がfalseを返す要素を，シーケンスの前の方に集める
  - 戻り値は，集められた部分の一つ後ろを指すイテレータ（次のブロックの先頭）
  - removeという単語は，ここでは「削除」ではなく「除外」という意味

remove_ifのイメージ



p: 10未満がtrue

remove_ifの結果



remove_copy, remove_copy_if

- remove_copy(b, e, d, t)
- remove_copy_if(b, e, d, p)
  - イテレータ b から e の間のシーケンスの中から，要素 t と一致しない要素，あるいは判別関数 p がfalseを返す要素を，イテレータ d が示す位置にコピーする
  - b と e の間のシーケンスは変化しない
  - 戻り値は，コピー先の最後の要素の一つ後を指すイテレータ

remove_copy_ifの例

```
//  students には既にデータが入っていると仮定する
//  不合格の生徒を fail にコピーする
//  students のデータはそのまま
vector<Student_info> students, fail;

//  students の先頭から末尾を対象に，pgradeがfalseを返す（f不合格）
データをfailの末尾にコピーする
remove_copy_if( students.begin(), students.end(),
back_inserter(fail), pgrade);
```

remove_copy_ifの例

```
bool pgrade(const Student_info& s){
   if(sが合格なら)     {
        return true;
   }
   else        { // 不合格なら
        return false;
   }
}
```

partition, stable_partition

- partition(b, e, p)
- stable_partition(b, e, p)
  - イテレータ b から e の範囲を判別関数 p に基づいてグループ分けする.
  - p が true の要素が先頭にくる
  - 戻り値は p が false になる最初の要素
  - stable_partition は分類されたグループ内での順番が, もとからあった順番どおりになる

stable_partitionのイメージ



p: 10未満がtrue

stable_partitionの結果



partition関数だと0から4までの順序と,
10から12までの順序が保証されない

stable_partitionの例

```
//  students には既にデータが入っていると仮定する
//  students を合格と不合格に分類し，studentsを並び替える
vector<Student_info> students;
vector<Student_info>::iterator iter;

iter = stable_partition( students.begin(), students.end(),
   pgrade);
//  これで，studentsの前半には合格した生徒のデータが，後半には不合格の生徒のデータが集められる
//  iterには後半の先頭を指すイテレータが入る
```

第7章「連想コンテナを使う」

２種類のコンテナ

- シーケンシャルコンテナ
  - 列（シーケンス）になって格納
  - push_back関数などでデータを挿入する位置を指定可能
  - vector型，list型
- 連想コンテナ
  - データを追加するときに，データの値に応じて自動的に格納位置が決まる
  - map型

データの追加のイメージ

シーケンシャルコンテナ
vector型 c



c.push_back(“C”);



c.push_front (“F”);



連想コンテナ m



mに“C” を追加



mに “F”を追加



連想コンテナにおけるデータの表現

- 連想コンテナでは，データの格納位置が自動的に変化
- では，どうやってデータにアクセスする？
- データは，キー(Key)と値(Value)の組として表現
  - 例えば，前回までの課題だと
  - 単語(“C++”) と出現回数(10)
  - キー (“C++”)と値(10)
  - “C++”という文字列から10という整数を「連想」させる

連想コンテナにおけるデータの表現

- シーケンシャルコンテナでは,
  - 「添え字」（データの格納位置）を使ってデータにアクセス
  - イテレータを使って先頭から順番にアクセス
  - vector< WordCount > w;
- 連想コンテナでは,
  - キー (“C++”)を使って，値(10)にアクセス
  - イテレータを使って先頭から順番にアクセス
  - map<string, int> m;

map型

- 単語（string型）と出現回数（int型）の連想コンテナ
- map< string, int > counters;
  - map型の変数の宣言
  - map< キーの型，値の型 > コンテナの変数名;
- counters[“c++”]
  - map型の変数にアクセス
  - コンテナ変数名[キー]
  - ++ counters[“c++”]; 出現回数を1増やす
  - cout << counters[“c++”]; 出現回数を出力

map型のサンプルソース

```
//   標準入力から単語を読み込み，その出現回数を1増やす
map< string, int > counters;
string s;
while( cin >> s ) {
   ++ counters[ s ];
}
//  countersに格納された，すべての単語とその出現回数を出力
for( map< string, int >::iterator iter = counters.begin();
   iter != counters.end(); ++iter)  {
   cout << iter -> first << “¥t” << iter -> second << endl;
}
```

サンプルソースの解説

- counters[s]
  - mapでは，あるキーでアクセスした瞬間にその値が初期化される.
  - この場合は，int型のデフォルトの 0 で初期化
  - そのため，キーが出現済みかどうか確認する必要はない
- ++ counters[s]
  - counters[s] = counters[s] + 1と同じ

サンプルソースの解説

- map< string, int >::iterator iter;
  - map用のイテレータの宣言
- iter = counters.begin();
  - countersの先頭の要素
- iter != counters.end()
  - イテレータiterがcountersの最後の要素と違う
- ++ iter
  - イテレータを進める（次の要素に移る）

サンプルソースの解説

- iter -> first, iter -> second
  - この連想コンテナは，単語(string)と出現回数(int)の組を格納
  - 実際は string型と int型の pair (組)として格納
  - この場合は，pair< const string, int >
  - 一般に，pair< const K, V >
    - K: キーの型，V: 値の型
  - キーの型がconstなのは，一度登録したキーを途中で変更されないように保護するため

サンプルソースの解説

- mapのイテレータは，pairを指している
- iter -> first
  - pairの第1要素，キー（文字列型の単語）
  - (*iter).first　としても良い
- iter -> second
  - pairの第2要素，値（int型の出現回数）
  - 同様に(*iter).second

連想コンテナの原理

- どうして，添え字でなくて，キー（文字列）でデータにアクセスできる？
- mapの内部で，キー（文字列）の順序に応じてツリー（二分木）で表現

行単位での文字列の読み込み

- getline(is, s)
  - 入力ストリーム is から1行（改行まで）読み込み，改行文字を除いた文字列を s に格納する
  - s に以前に保存されていたデータは破棄される
  - 戻り値は，is への参照

行から単語の切り出し

- split 関数の作成
  - 引数: 1行の内容の文字列
    - 文字列の内容を変更しないようにコピーを渡すstring型か
    - 定数型の参照, const string&
  - 戻り値: 切り出された複数の単語を格納したvector<string>
  - vector<string> split(const string& str)

行から単語の切り出し

- str に対して，以下を繰り返す
- 先頭のスペースをとばす
  - スペース以外の文字が最初に現れる場所を探す
  - そこを単語の先頭とする
- 次に，そこから単語の区切り目を探す
  - スペースが最初に現れる場所を探す
  - そこを単語の末尾とする
- 単語の先頭と末尾が異なっていたら，vectorに登録

行から単語の切り出し

- string::iterator i, j;
- i = find_if(i, str.end(), not_space);
- j = find_if(i, str.end(), space);
- string(i, j) が単語となる
  - イテレータ i と j の間のシーケンスからなるコンテナ（文字列）を生成
- bool space(char c){ return(isspace(c)); }
- bool not_space(char c){ return(!isspace(c)); }

デフォルト引数

- 関数の引数が省略されたときに，自動的に使われる値
- 関数のプロトタイプ，または関数の宣言のどちらかでのみ，指定可能
  - void test1(double a = 1.0, double b = 10.0);
  - test1(); // aには1.0, bには10.0
  - test1(5.0); // aには5.0, bには10.0
  - test1(5.0, 20.0); // aには5.0, bには20.0
- デフォルト変数は最も右側から作用

左辺値

- コンパイルエラーで，lvalueとか現れたことありませんか？
- lvalue：左辺値，代入式の左辺に指定して良い値
  - 実体のあるオブジェクト
  - 一時的な値ではない
  - a=15/100としたとき，aは左辺値，15/100は左辺値ではない（計算が終わったら破棄される）
  - 参照を戻す関数の戻り値は左辺値として使えるが，変数の寿命に注意

変数の寿命に注意

動作は不安定：関数内の自動（スタック）変数だから

```
int main()  {
  func1()= 1;
}
int& func1()  {
  int a=0;
  return(a);
}
```

左の例と本質的に同一

```
int main()  {
  *func2()= 1;
}
int* func2()  {
  int a=0;
  return(&a);
}
```

動作は不安定：関数内の自動（スタック）変数だから

```
int main()  {
  func3()= 1;
}
int& func3()  {
  int a[1];
  a[0] = 0;
  return(a[0]);
}
```

動作は安定：vectorがヒープ変数だから

```
int main()  {
  func4()= 1;
}
int& func4()  {
  vector<int> a;
  a.push_back(0);
  return(a[0]);
}
```

演習7のポイント

- 入力
  - 行数Nの入力（改行コードの取り扱いのためgetline()を使うこと）
  - 行単位で文字列Siを読み込む（ getline() ）
  - 単語数Mの入力
  - 単語Wjの入力
- 前処理
  - map< string, vector<int> > counters;
  - for(int i = 1; i <= N; i++) { // i が行番号
  - 行を単語に分解 // split
  - 分解された各単語に対して
    counters[単語].push_back(行番号);
  - }

演習7のポイント

- クエリ
  - for(int j = 1; j <= M; j++) {
    counters[Wj]の行番号を出力
  - }

第8章「ジェネリック関数を書く」

ジェネリック関数

- 型に依存しない関数
- どんな型の変数が引数に与えられても使える関数
- 標準ライブラリのジェネリック関数の例
- search(b, e, b2, e2)
  - b から e で指定されるシーケンスの中から b2 から e2 のシーケンスを探す
  - b, e, b2, e2は，vector, list, stringなど，どんなコンテナでも利用可能

search関数の利用例

```
//  例1
string src = “http://www.u-aizu.ac.jp/”, sep = “://”;
search( src.begin(), src.end(), sep.begin(), sep.end() );

//  例2
vector<int> data, query;
//  data = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};, query = {4, 5,
   6};
search( data.begin(), data.end(), query.begin(),
   query.end() );
```

- 引数はすべてイテレータとして与えられている

ジェネリック関数とイテレータ

- イテレータの操作(++ など)は，元のデータの型に(char, int など)に依存しない
- コンテナそのものではなく，イテレータを引数にすることにより，データの型に依存しない処理が可能になる
- そのための手法がテンプレート関数

テンプレート関数の例

```
template < class T >
T median( vector<T> v)
{
   vector<T>::size_type size = v.size(), mid;
   if(size == 0)
       throw domain_error( “median of empty vector” );
   sort( v.begin(), v.end() );
   mid = size / 2;
   if( size % 2 == 0 )
       return ( v[ mid ] + v[ mid +1 ] ) / 2;
   else
       return v[ mid ];
}
```

テンプレート関数の例の解説

- 与えられた vector 型の変数のメディアンを計算する関数
- 引数は vector<T> 型，戻り値は T 型
  - 例えば，引数が vector<double> なら戻り値もdouble
  - 引数が vector<int> なら戻り値も int
- 実際の関数においては，Tが double だろうと int だろうと，処理に違いはない
- ただ，型が違うだけ

テンプレート関数の例の解説

- template < class T >
  - 以下の関数では，Tはクラスの型を表すと宣言
  - それ以降では，Tはクラスとして扱われ，コンパイルされる
  - T の実際の型は，median関数が呼び出されたときの引数で，自動的に判断

ジェネリック関数と型

```
//   2つの引数を取り，大きいほうの値を返す関数
//   この関数自体は，何の問題もない
template < class T >
// T max( const T& a, const T& b )
T max( T a, T b )
{
   if( a > b )
       return a;
   else
       return b;
}
```

ジェネリック関数と型

```
//   しかし，max関数の呼び出し方によっては，コンパイルエラーになる
int main()
{
   int a = 10;
   double b = 20.0;
   cout << max( a, b );
}
//   コンパイラは，テンプレートのTが int なのか double なのか判定できないため
```

データ構造非依存性

- 標準ライブラリの例
- find(c.begin(), c.end(), val);
  - なぜ，このように設計（引数がイテレータ）されているのか？
- c.find(val)でも，いいのでは？
  - 各コンテナ(vector, list, string)ごとにfindを定義しなければならない
- find(c, val)でも，いいのでは？
  - これだと，コンテナ全体からの探索のみ

アルゴリズムとイテレータ

- ジェネリック関数を書くためには，コンテナとイテレータの種類を理解する必要
  - 入力イテレータ
  - 出力イテレータ
  - 前方向イテレータ
  - 双方向イテレータ
  - ランダムアクセスイテレータ

入力イテレータ

- 以下の機能を満たすイテレータ
  - ++（前置と後置の両方）
  - ==（イテレータが等しいかの判定）
  - !=（イテレータが異なるかの判定）
  - *（イテレータが指す要素に読み込みアクセス）
  - ->（イテレータが指す要素のメンバに読み込みアクセス）

入力イテレータの例

- 順次読み込み専用アクセスの例, `find (1)`

```
template < class In, class X >
In find( In b, In e, const X& x )
{
   while( b != e && *b != x )      {
        ++ b;
   }
   return b;
}
```

- `b` と `e` が入力イテレータ
- `b` から `e` の間の最初に現れた要素`x`を指すイテレータを返す, なければ `e` を返すことになる

入力イテレータの例

- 順次読み込み専用アクセスの例, find (2)

```
template < class In, class X >
In find( In b, In e, const X& x )
{
   if (b == e || *b == x)   {
        return b;
   }
   b ++;
   return find( b, e, x );
}
```

- 再帰関数による表現

出力イテレータ

- 以下の機能を満たすイテレータ
  - 読み込みと書き込みの違いを除いて，入力イテレータと同じ

出力イテレータの例

- 順次書き込み専用アクセスの例, copy

```
template < class In, class Out >
In copy( In b, In e, Out d )
{
   while( b != e )   {
        *d = *b;  d++;   b++;
   }
   return d;
}
```

- b から e の間の要素を d が指す位置にコピー
- d が出力イテレータ, b と e は入力イテレータ

前方向イテレータ

- 以下の機能を満たすイテレータ
  - 入力イテレータと出力イテレータを合わせたもの
  - 読み書きのアクセス

前方向イテレータの例

- 順次読み書きアクセスの例, replace

```
template < class For, class X >
void replace ( For b, For e, const X& x, const X& y )
{
   while( b != e )   {
        if( *b == x )
                *b = y;
        ++ b;
   }
}
```

- b から e の間の要素の x を y に置換
- b と e が前方向イテレータ

双方向イテレータ

- 以下の機能を満たすイテレータ
  - 前方向イテレータに -- を加えたもの

双方向イテレータの例

- **可逆アクセスの例**, `reverse`

```
template < class Bi > void reverse ( Bi b, Bi e )
{
   while( b != e )   {
        -- e;
        if( *b != *e )
               swap(*b++, *e);
   }
}
```

- `b` から `e` の間の要素の順序を入れ替える
- `b` と `e` が双方向イテレータ

ランダムアクセスイテレータ

- 以下の機能を満たすイテレータ
  - 双方向イテレータにイテレータの計算を加えたもの
  - p と q をイテレータ，n を整数として
  - p + n, p - n, n + p
  - p – q
  - p[n], *(p + n) と同じ
  - p < q, p > q, p <= q, p >= q

ランダムアクセスイテレータの例

- ランダムアクセスの例, binary_search

```
template < class Ran, class X >
bool binary_search ( Ran b, Ran e, const X& x )
{
   while( b < e )    {
        Ran m = b + (e - b) / 2;
        if( x < *m )  e = m;
        else if( *m < x )    b = m + 1;
        else          return true;
   }
   return false;
}
```

- b と e がランダムアクセスイテレータ

イテレータとコンテナ

- すべての標準ライブラリのコンテナで適用可能
  - 入力イテレータ
  - 出力イテレータ
  - 前方向イテレータ
  - 双方向イテレータ
- vector, stringでのみ適用可能
  - ランダムアクセスイテレータ
  - sort関数が使えるのもランダムアクセスイテレータのみ

演習8のポイント

- template < class For1, class For2 >
  For1 my_search(For1 b, For1 e, For2 b2, For2 e2);
- template < class In, class Out >
  Out my_copy(In b, In e, Out d);

第9章「新しい型を定義する」

Student_info 構造体再掲

- 教科書で以前定義した構造体

```
struct Student_info {
   std::string FirstName, LastName, ID;
   double Midterm, Final;
   std::vector<double> Homework;
}
```

- 完全修飾型 `std::string` や `std::vector` を使う理由
  - 色々なプログラムやユーザに使われる
  - 構造体内で使われる `string` や `vector` は内部仕様で，この構造体を使うユーザに `using` を強要できないため

構造体のメンバ関数

- 構造体のデータ(メンバ変数)を操作する関数

```
struct Student_info {
   std::string FirstName, LastName, ID;
   double Midterm, Final, Ex;
   std::vector<double> Homework;

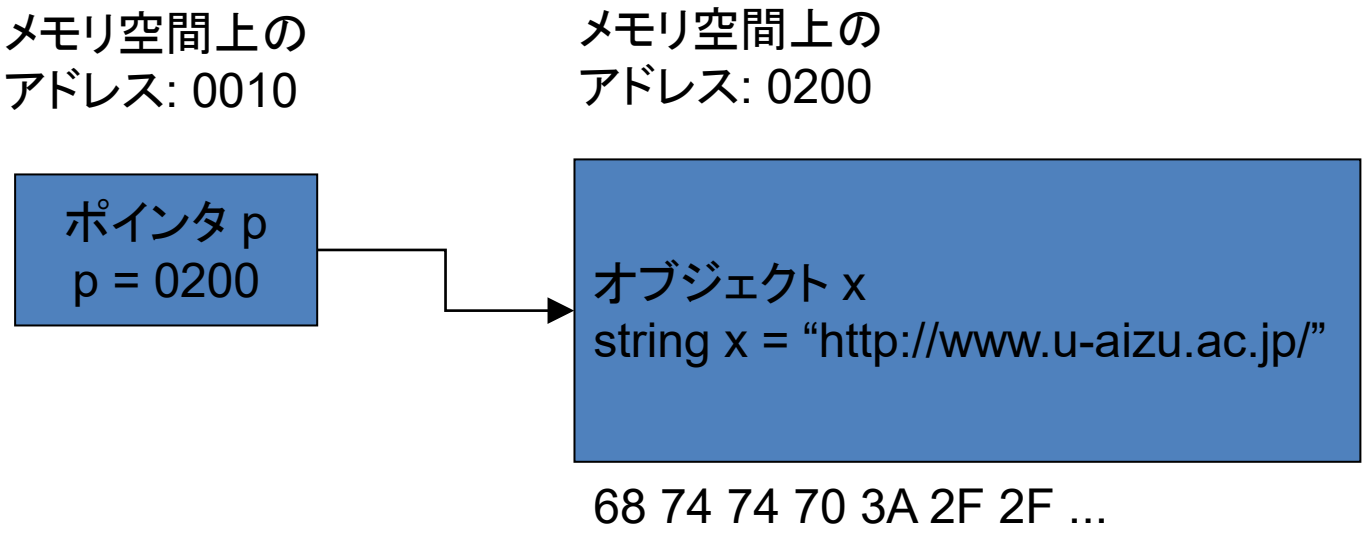
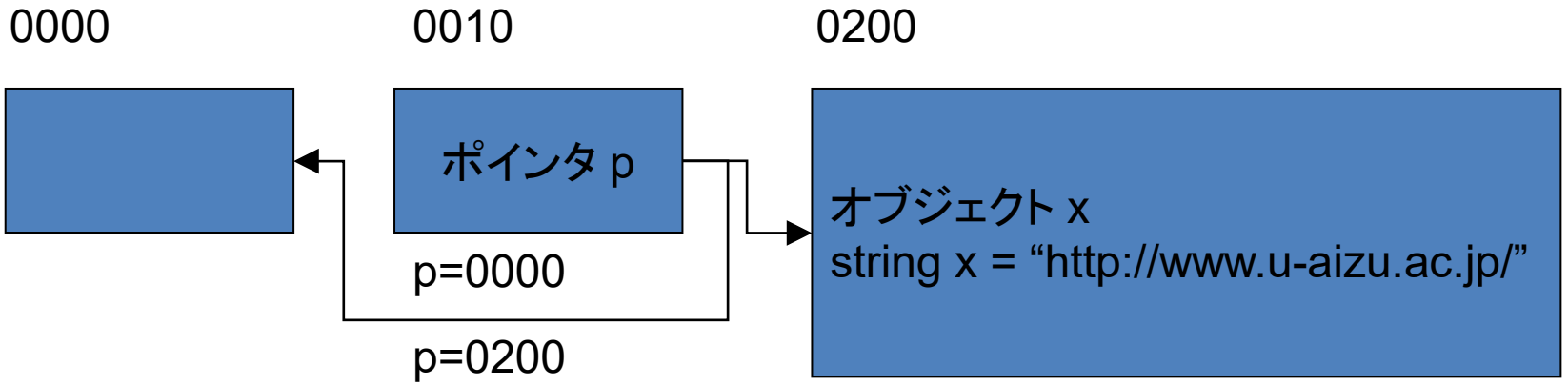
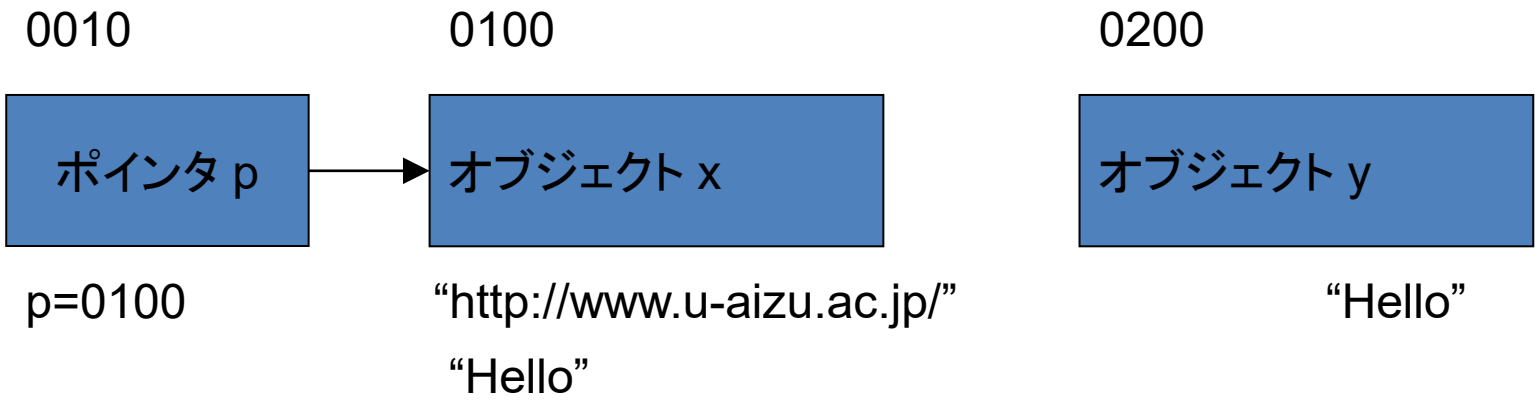
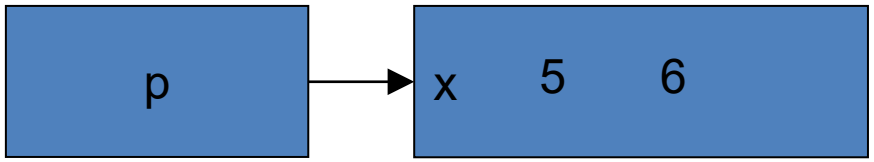
   //  入力ストリームからデータを読み込み,
   //  この構造体に格納する関数
   std::istream& read( std::istream& );
   //  最終成績を計算する関数
   double grade() const;
}
```

read関数の例

- メンバ関数のread関数

```
istream& Student_info::read( istream& in )      {
   in >> FirstName >> LastName >> ID;
   read_hw( in, Homework );
   return in;
}
```

- 従来の通常関数のread関数

```
istream& read( istream& in, Student_info& s )  {
   in >> s.FirstName >> s.LastName >> s.ID;
   read_hw( in, s.Homework );
   return in;
}
```

メンバ関数と通常の関数の違い

- 関数名
  - `read` から `Student_info::read`
  - クラス名::メンバ関数名
- 引数
  - `read( in, s )` から `read( in )`
  - 読み込み結果は，このオブジェクト(変数)に格納
- 関数の定義の内部
  - `s.FirstName` から `FirstName`
  - オブジェクトの要素に直接アクセス可能

メンバ関数のread関数の使用例

```
//   データを標準入力ストリームから読み込み
vector<Student_info> students;
Student_info record;

//   recordというオブジェクト（構造体変数）に
//   readというメンバ関数を適用
//   その結果はrecordに反映される

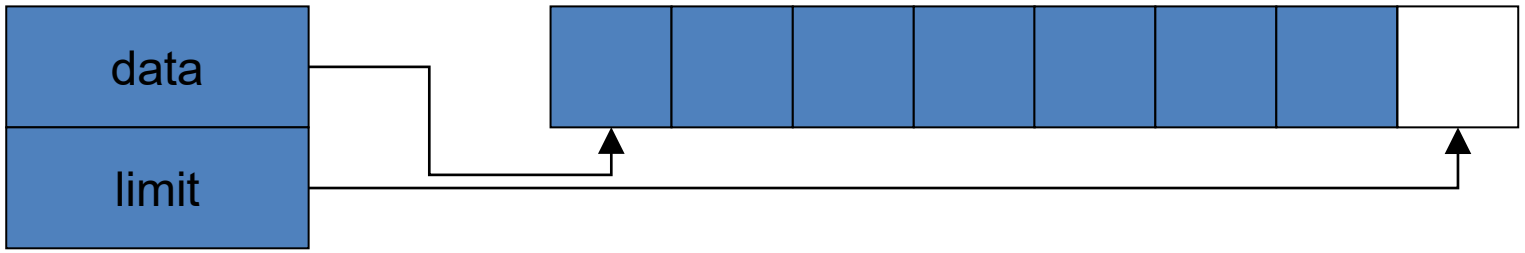
while( record.read( cin ) ) {
    students.push_back( record );
}
```

通常関数のread関数の使用例

```
//   readという通常関数を実行し，
//   その結果を参照型の引数で与えられた
//   recordというオブジェクトに格納

while( read( cin, record ) ) {
    students.push_back( record );
}

//   メンバ関数でも通常関数でも，結果は同じ
//   recordというオブジェクトに読み込みデータが格納される
//   プログラムの考え方（オブジェクト指向）の違い
```

grade関数の例

```
//  メンバ関数の grade 関数
double Student_info::grade() const {
   return ::grade(Midterm, Final, Homework);
}

//  通常関数の grade 関数
double grade( double midterm, double final, const
   vector<double>& hw )     {
   double ex;
   //  hw(演習)のメジアンを求め, ex に代入
   return 0.2 * midterm + 0.4 * final + 0.4 * ex;
}
```

grade関数の解説

- `double Student_info::grade() const`
  - このメンバ関数はオブジェクトの値を変更しないことを宣言
  - 引数に与える `const` と同じこと
  - `const` メンバ関数
- `::grade()`
  - メンバ関数の`grade`ではなく，通常関数の`grade`を呼び出す

構造体(struct)からクラス(class)へ

```
class  Student_info {
public:
   double grade() const;
   std::istream& read(istream&);
private:
   std::string  FirstName, LastName, ID;
   double Midterm, Final;
   std::vector<double>  Homework;
}
```

データ保護

- `public:`
  - ユーザが(クラスの外部から)使用可能
  - インターフェース(外部からの操作)
- `private:`
  - クラスのメンバ関数からのみ利用可能
  - 実装(データなど)
- `class` と `struct` の違いは，ほとんどない
  - ラベルがない時のデフォルトは，`class`は`private`，`struct`は`public`のアクセス制限

アクセス関数

- アクセス制限でデータメンバ(例えば，`FirstName`は`private`)を隠蔽
- では，データメンバにアクセスしたいときは，どうする？
- データメンバにアクセスするためのメンバ関数(アクセス関数)を作る
  - メンバ変数を直接操作するのではなく，アクセス関数を経由して操作
  - 不用意にデータ更新されないので，安全

アクセス関数の例

```
class  Student_info {
public:
   //  追加
   std::string  first_name() const { return FirstName; } ;
   std::string  last_name() const { return LastName; } ;
   std::string  id() const { return ID; } ;
   bool valid() const { return !Homework.empty(); };
   //  省略
private:
   std::string  FirstName, LastName, ID;
   double Midterm, Final;
   std::vector<double>  Homework;
}
```

アクセス関数の例の解説

- `std::string  first_name() const { return FirstName; } ;`
  - クラスの内部で，関数の定義を含めることが可能（インライン展開）
  - 関数が戻すのは `string`型なので，`FirstName`のコピーが返される
  - さらに，`const` が付いているのでメンバ変数は変更されない
  - これによりデータが保護（書き換えられない）

アクセス関数の利用例

```
bool compare( const Student_info& x, const Student_info&
   y)
{
   if( x.first_name() == y.first_name() )
       return x.last_name() < y.last_name();
   else
       return x.first_name() < y.first_name();
}
```

- compare 関数というクラス外部から，アクセス関数を利用してメンバ変数の値を利用可能

チェック用の関数

- `bool valid() const;`
  - 演習のメジアンの計算をするときに，演習の提出回数が0だと処理できない
    - 以前のプログラムでは例外にしていた
  - クラスの再利用性を考えると，演習が0回のときの処理は，ユーザやプログラムにより様々．
  - そこで，演習が0回かどうかを判定する関数を用意し，実際の処理はユーザにまかせる

コンストラクタ

- 必要となる初期化処理
  - オブジェクトを保持するための，メモリの確保（割付）
  - オブジェクト（データメンバの値）が初期化
- 初期化処理を行う特別なメンバ関数がコンストラクタ
  - コンストラクタの名前は，クラス名と同じ
  - 引数の型や個数に応じて，複数のコンストラクタを定義可能

Student_infoのコンストラクタ

- `Student_info s1;`
  - 変数を宣言するだけ，データは不定
- `Sudent_info s2(cin);`
  - 変数を宣言し，`cin`からデータを読み込み，その値で初期化

```
class  Student_info {
public:
    //  追加
   Student_info(); //  変数宣言だけ
   Student_info( std::istream& ); // データで初期化
    //  省略
}
```

デフォルトコンストラクタ

- 引数を取らないコンストラクタ
- 例えば,

```
Student_info::Student_info()      {
   Midterm = 0;
   Final = 0;
}
```

- MidtermとFinalを0で明示的に初期化
- FirstName, LastName, IDはstringクラスの
- Homeworkはvectorクラスのコンストラクタにより非明示的に初期化

デフォルトコンストラクタ

- 別の書き方，こちらがお勧め
- `Student_info::Student_info( ) : Midterm(0), Final(0) { }`
- コンストラクタ・イニシャライザ，「:」と「{」の間
  - データメンバの値をカッコの中のもので初期化
  - その後に，{}の内部が実行される
  - 同じ処理は{}内部でも行えるが，初期と代入で2度手間になる

引数を取るコンストラクタ

- `Student_info::Student_info( istream& is) { read(is); }`
- この例では，コンストラクタ・イニシャライザはない

コンストラクタによる値の初期化

- コンストラクタが定義されているクラスなら
  - コンストラクタに指定されている手続きに従って初期化
- 組み込み型なら
  - 値なら0，それ以外は不定値に初期化
- コンストラクタが定義されていないクラスなら
  - データメンバそれぞれのコンストラクタに従って初期化

今回の演習のポイント (1)

```
class Student_info{
public:
  //  インターフェース
  Student_info();
  Student_info( std::istream& );
  std::string first_name() const { return FirstName; };
  std::string last_name() const { return LastName; };
  std::string id() const { return ID; };
  double midterm() const { return Midterm; };
  double final() const { return Final; };
  double ex() const { return Ex; };
  double total() const { return Total; };
```

今回の演習のポイント (2)

```
  std::vector<double> homework() const { return
   Homework; };
  bool valid() const { return !Homework.empty(); };
  std::istream& read( std::istream& );
  double grade();
private:
  //  実装
  std::string FirstName, LastName, ID;
  double Midterm, Final, Ex, Total;
  std::vector<double> Homework;
};
bool compare(const Student_info&, const Student_info&);
```

今回の演習のポイント (3)

```
//   メンバ関数の定義
Student_info::Student_info( ) : Midterm(0), Final(0) { }
Student_info::Student_info( istream& is) { read(is); }
std::istream& Student_info::read( std::istream& is ){
   //   適切なコードを書く
}
double Student_info::grade(){
   //   適切なコードを書く
}
bool compare(const Student_info&, const Student_info&){
   //   適切なコードを書く
}
```

今回の演習のポイント (4)

```
int main()    {
   vector<Student_info> students;
   Student_info record;
   while( record.read( cin ) ) {
        students.push_back( record );
   }
   sort( students.begin(), students.end(), compare);
   for(vector<Student_info>::size_type i = 0; i !=
   students.size(); ++i)    {
        students[i].grade();
        //  結果の出力などvaild()
   }
}
```

第10章「メモリ管理と低レベルのデータ構造」

ポインタとは

- ポインタはオブジェクト（変数）のメモリ空間上のアドレスを表す
  - ポインタ p がオブジェクト x を指す
  - p は x を指すポインタ



アドレス演算子

- xがオブジェクトとすると，xのアドレスは&x
  - &はアドレス演算子
  - p = &x;
    - ポインタ変数 p に x のアドレスを代入
    - p がオブジェクト x を指すようにする



デリファレンス演算子

- pがアドレスだとすると，pが指すオブジェクトは*p
  - *はデリファレンス演算子
  - *p = y;
  - ポインタ変数 p が指すオブジェクトの内容に y の内容を代入する



ポインタ

- int　x;
  - x は int 型のオブジェクト（変数）
- int　*p;
  - *p が int 型を意味するので，pは int 型のオブジェクトを指すポインタ
- int*　p;
  - 上の同じ意味．pがポインタを意味することを強調するため，こう書かれることが多い

ポインタを用いたプログラムの例(1)

```
int  x = 5;
int* p = &x;
cout << “x=” << x << "*p=" << *p << endl;
*p = 6;
cout << "x=" << x << "*p=" << *p << endl;
```

- 出力は,
  - x=5 *p=5
  - x=6 *p=6



ポインタを用いたプログラムの例(2)

```
int  x = 5, y = 7;
int* p = &x;
cout << “x=” << x << "*p=" << *p << endl;
p = &y;
cout << “x=” << x << "*p=" <<  *p << endl;
```

- 出力は,
  - x=5 *p=5
  - x=5 *p=7

関数へのポインタの使用例

- 呼び出し側
  - 下の第3引数は，isEvenという関数へのポインタ

```
my_partition( c.begin(), c.end(), isEven );

bool isEven( int num )
{
  if( num % 2 == 0 ){
    return( true );
  }
  return( false );
}
```

関数へのポインタの使用例

- 呼び出された側
  - `Pr` は，`int` 型の引数を，`bool` 型の戻り値を持つ関数へのポインタ（の型）

```
template < class Bi, class Pr >
Bi my_partition( Bi b, Bi e, Pr p)
{
   //  ここまでも省略
     if( p( *b ) == false )  {
         while( b != e )  {
   //  以下も省略
}
```

関数のポインタ

- bool (*fp)(int)
  - int 型の引数を取り，bool 型の戻り値を返す関数への，fp という名前のポインタ
- fp = &isEven;
  - fp に isEven という関数のポインタを代入し，fp が isEven を指すようにする
- fp = isEven;
  - こちらの書き方も可能．上と同じ意味
  - 関数のポインタだけは，特別

- int data[80];
- int* p = data;
- Int* p = &(data[0]);

関数のポインタ

- bool a = (*fp)(i);
  - ポインタ変数を使った関数の呼び出し
  - 引数が i，戻り値が a に入る
- bool b = fp(i);
  - こちらも上と同じ意味.
  - 関数のポインタだけは，特別

配列

- 標準ライブラリではなく，言語そのものにあるコンテナの一種
- 1つ以上の同じ型のオブジェクト（型）を保持
- 配列の要素数は，コンパイル時に決められていなければならず，途中で増やしたら減らしたりできない
- size_t
  - 配列の要素数を表す型．実は unsigned 型
  - 参考：クラスの size_type
  - #include <cstddef>

配列

  - const size_t NDim = 3;
  - double coords [NDim];
- 配列の名前は，配列の最初の要素を表すポインタ
  - *coords = 1.5;
    - 配列の先頭の要素，coords[0] に 1.5 が代入される
  - coords + 1
    - 配列の2番目の要素を指すポインタ

ポインタの算術

- vector<double> v;
- copy(coords, coords + NDim, back_inserter(v));
  - coordsの先頭から最後の要素までを，v の末尾にコピーする
  - coords + NDim は，一番最後の要素の次を指している．有効な要素ではない．
  - 参考：イテレータの v.end()

ポインタの算術

- p と q を配列のポインタとして，
  - p - qは p と q の指す要素間の距離
- ptrdiff_t
  - ポインタの指す要素間の距離を表す型
  - #include <cstddef>
- ポインタは，イテレータの一種とみなすことができる
  - コンテナ用の標準ライブラリの関数の引数（イテレータ）に，ポインタを使うことができる

インデックスと配列の初期化

```
const int month_length[ ] = {
        31,28,31,30,31,30,
        31,31,30,31,30,31
};
```

- 明示的に要素を与えて初期化するときは，要素数を省略できる
- p[n] は *(p + n) と同じ

main関数の引数

- 例えば, g++ -o test10 test10.cc
  - コマンドライン引数の数は4
  - “g++” “-o” “test10” “test10.cc”
- コマンドライン引数は空白で区切られる

main関数の引数

- コマンドライン引数をプログラムで使うためには
- int main(int argc, char* argv[ ])
  - argc: コマンドライン引数の数
  - argv: コマンドライン引数の内容（文字列）
  - g++ の例だと,
    - argc = 4;
    - argv[0] = “g++” argv[1] = “-o” argv[2] = “test10” argv[3] = “test10.cc”
  - argv[i]が文字列（char *型）,それが配列になっているので char*[]という型

コマンドライン引数の例

```
//   3つの引数をとるプログラム
//   （コマンド名　入力ファイル名　出力ファイル名）
int main( int argc, char *argv[] )
{
   if( argc != 3)    {
        cerr << “Error” << endl;
        return -1;
   }
   ifstream infile( argv[1] );
   ofstream outfile( argv[2] );
}
```

- cerr は標準エラー出力：画面にエラーメッセージを出力するときに使われる．ファイルにリダイレクトされない
- File* fp = fopen(“test.txt”, “rt”);
- if(fp == NULL) …

ファイルの読み書き

- ifstream infile(argv[1]);
  - ifstream 型のinfileという名前の変数.
  - argv[1]という名前のファイルに読み込みでアクセスするときに使用
  - その後は，標準入力ストリーム cin と同じように使える
    - string FirstName, LastName, ID;
    - infile >> FirstName >> LastName >> ID;
  - #include <fstream>

ファイルの読み書き

- ofstream outfile(argv[2]);
  - ofstream 型のoutfileという名前の変数.
  - argv[2]という名前のファイルに書き込みでアクセスするときに使用
  - その後は，標準出力ストリーム cout と同じように使える
    - outfile << “|” << FirstName << “ | ” << LastName “ |” << ID << “|”<< endl;
  - #include <fstream>

メモリ管理

- 自動メモリ管理
  - ローカル変数に適用される
  - ブロックが終了すると自動的にメモリを開放し，その内容は無効になる

```
//    悪い例
int*  test_func1()
{
   int  x;
   return &x;
}
//   関数から出ると x は開放され，関数の戻り値は無意味
```

メモリ管理

- 静的メモリ管理
  - 関数が始めて実行されたときに初期化
  - プログラムが終了するまで，内容は保持

```
//    正しく動作する例
int*  test_func2()
{
   static int  x;
   return &x;
}
//   関数から出ても x は保持され，関数の戻り値は利用可能
```

オブジェクトの生成と破棄

- new
  - オブジェクト（メモリ）の動的な確保
- delete
  - 動的に確保したオブジェクトの破棄
- int *p = new int(42);
  - int型で，値が42のオブジェクト（メモリ）を確保
  - そのアドレスを p に代入
- delete p;
  - アドレスが p のオブジェクトを破棄（開放）

オブジェクトの生成と破棄

```
//   関数を利用した例
int main()
{
   int *p = test_func3();
   *p = 10;
   //   使い終わったら，開放すること
   delete p;
}

//  int型のオブジェクトを生成し，そのポインタを戻す関数
int test_func3()   {
   return new int(0);
}
```

配列の確保

- int *p = new int [64];
  - 要素数が64のint型のオブジェクト（配列）の確保
- delete[] p;
  - 配列全体のオブジェクトの開放（破棄）
  - 参考：delete p だとpが指す要素だけ

第11章「抽象データ型を定義する」

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
  - 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
  - 自動型変換（暗黙の型変換）のときに重要，Vecクラスではなくても動作

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`
  - `v2 = vs;`
- 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

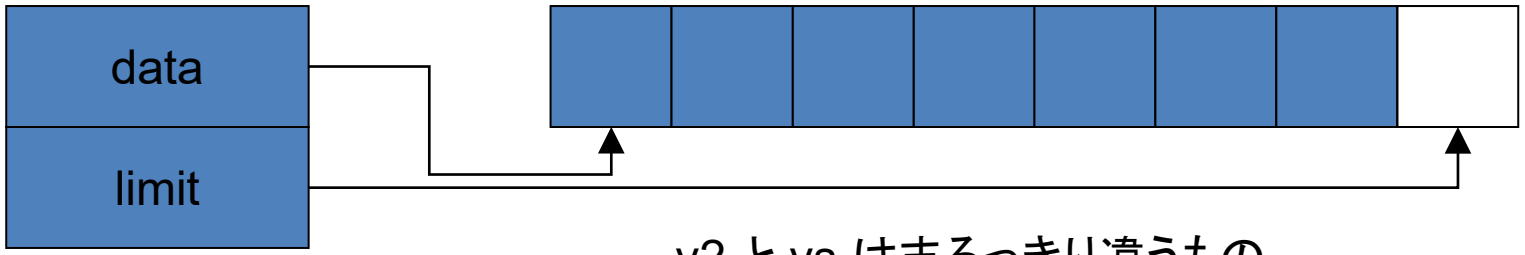
コピーコンストラクタ

- 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

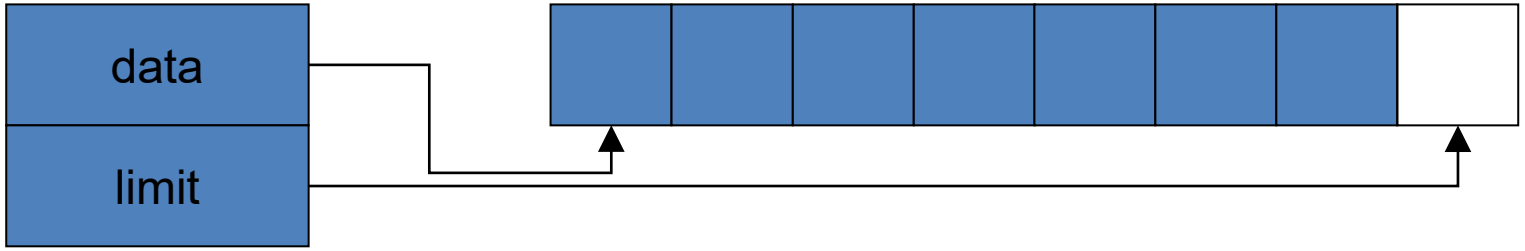
v2 = vs; Deep copyの場合

vs



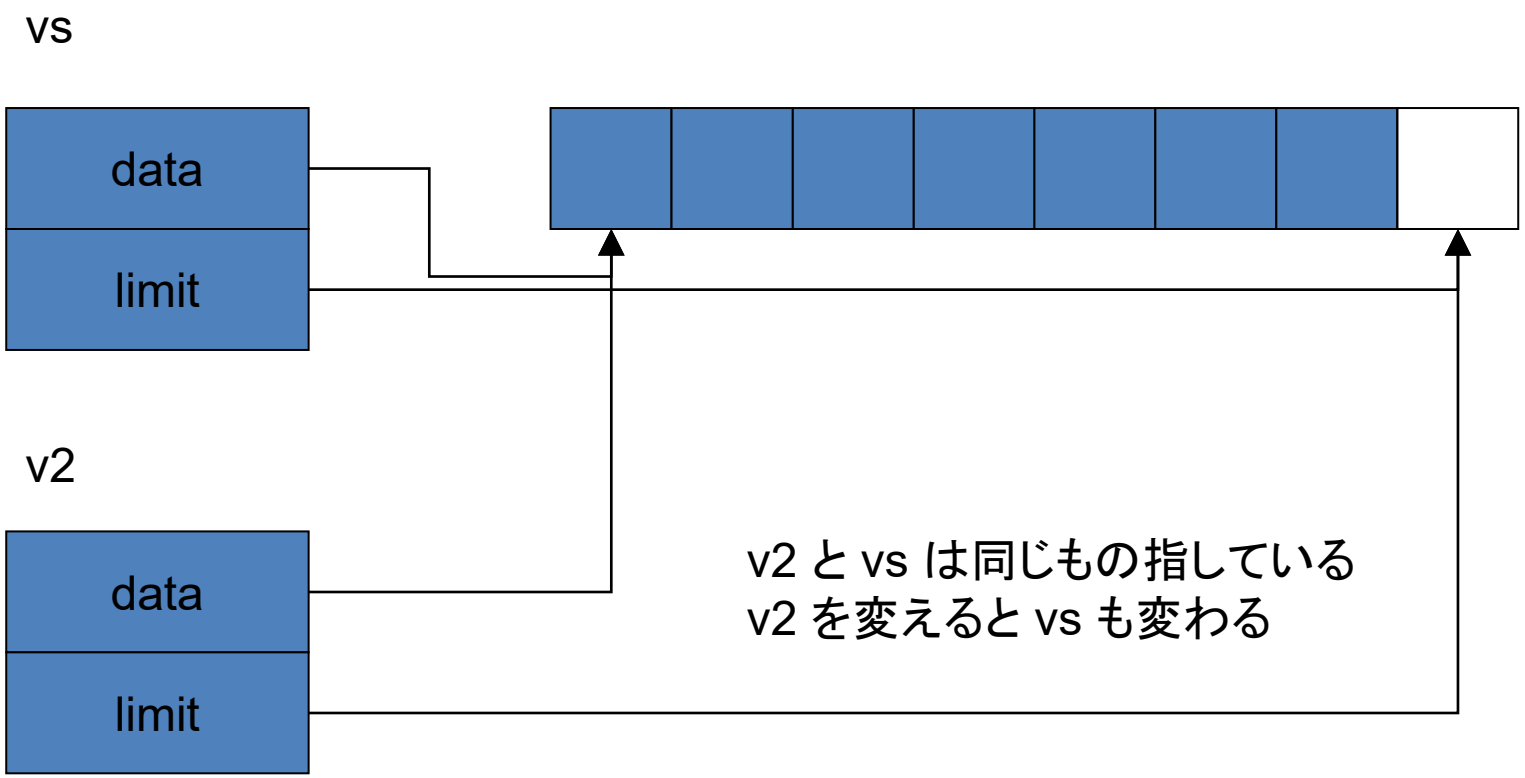
v2 と vs はまるっきり違うもの
v2 を変えても vs は変わらない

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;
}
Vec<int> v1, v2;
v1 = v1;
v1.operator=(v2);
```

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
  - コンストラクタイニシャライザ（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
- string a(“aaaaa”)
- string b=a;
- string b(a);

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

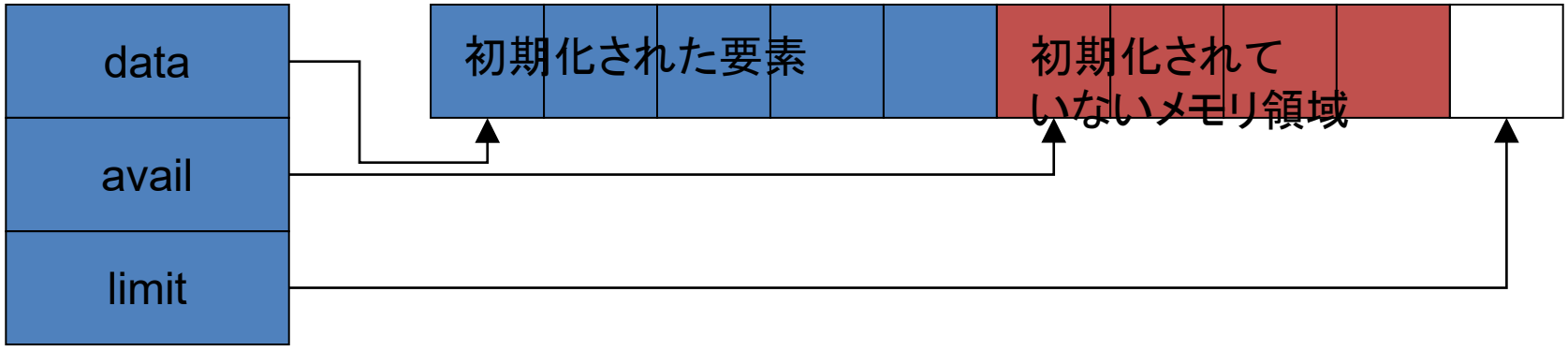
- 以前定義したStudent_infoクラスでは,
  - コピーコンストラクタ
  - 代入演算子
  - デストラクタ, の定義をしなかった
- このときは, 各データメンバのルールに従う
  - FirstName, LastName, IDは string クラスの
  - Midterm, Finalは double 型の
  - Homeworkはvector<double>のルールに従う

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

第12章「値のように振舞うクラス」

Strクラスのコンストラクタ

```
class Str  {
public:
     typedef  Vec<char>::size_type  size_type;
     //  4種類のコンストラクタ
     Str()  {}
     Str(size_type n, char c) : data(n, c)  {}
     Str(const char* cp)  {
        std::copy(cp, cp + std::strlen(cp),
         std::back_inserter(data));
     }
     template <class In> Str(In b, In e)      {
        std::copy(b, e, std::back_inserter(data) );
     }
private:
   Vec<char>  data;
}
```

自動の型変換

- Strクラスは，以下のような使い方ができると便利
  - Str s(“hello”); // オブジェクトの生成
  - Str t = “Hello”; // 初期化
  - s = “Hello!”; // 代入
- “hello”はconst char*型なので，型変換
  - これは，3番目のコンストラクタで対応済み

自動の型変換

- 1番目の「=」は，初期化なのでconst Str&を引数に取るコピーコンストラクタが使われる
  - そのようなコンストラクタは定義されていない！
- 2番目の「=」は，代入
  - const char*を引数にとる代入演算子は定義されていない！
- でも，これで大丈夫
  - ユーザ定義の型変換

ユーザ定義の型変換

- Str(const char*)というコンストラクタが定義されている
- Strが必要な場所でconst char*が使われると，自動的にこのコンストラクタが呼び出され，Str型への型変換が行われる

Strの演算子

```
class Str  {
public:
   //  以下を追加
   char& operator[ ]  (size_type i)  { return data[ i ]; }
   const char& operator[ ]  (size_type i) const { return
   data[ i ]; }
}
```

入出力演算子

- `Str  s;  cin >> s;` のように使いたい
- しかし，クラスの演算子として定義しようとすると，`cin.operator >> (s)` のように `cin`クラスの演算子になってしまう
- そこで，関数として実装

```
std::istream&  operator >> (std::istream&, Str&);
std::ostream&  operator << (std::ostream&, const Str&);

Str s(“abd”);
std::cout << s << std::endl;
```

入出力演算子

```
std::ostream&  operator << (ostream& os, const Str& s)
{
   for(Str::size_type i = 0; i != s.size(); ++i)
       os << s[ i ];
   return os;
}
```

Strクラス

```
class Str  {
public:
   //  以下を追加
   size_type  size()  const  { return data.size(); }
}
```

入出力演算子

```
std::istream&  operator >> (istream& is, Str& s)
{
   //   現在のデータを破棄
   //   Vecクラスにclear()を実装する必要
   s.data.clear();
   //   空白を読んで破棄
   char c;
   while( is.get(c) && isspace(c) )
        //   空白だったら何もしない
        ;
```

入出力演算子

```
    if(is)  {
        //  このままでは，ここでコンパイルエラー
        do s.data.push_back(c);
        while( is.get(c) && !isspace(c) );
        //  もし，空白を読み込んだら，ストリームに戻す
        if(is)
            is.unget();
    }
    return is;
}
```

- この演算子に`private`のメンバにアクセスさせる必要

フレンド

```
class Str  {
   //   以下を追加
   friend  std::istream& operator >> (std::istream&, Str&);
public:
}
```

- friendは，この演算子（関数）がStrクラスのprivateメンバにアクセスすることを許可する

他の2項演算子

- `s = s + s1;` や `s += s1;` のような演算子があると便利

```
class Str  {
public:
   //   以下を追加
   Str& operator += (const Str& s)        {
        std::copy(s.data.begin(), s.data.end(),
        std::back_inserter(data) );
        return *this;
   }
   Str& operator + (const Str& s, const Str& t) {
        Str r = s;
        r += t;
        return r;
   }
}
```

第13章「継承と動的結合を使う」

2種類の学生と3つのクラス

- 学部生
  - 名前，中間試験，期末試験，演習
  - Coreクラス
- 大学院生
  - 学部生のデータに付け加え，論文
  - Gradクラス
- ハンドルクラス
  - 上の2つを使いやすくするためのクラス
  - Student_infoクラス

Coreクラス

```
class Core{
public:
   Core( );
   Core( std::istream& );
   std::string name( ) const ;
   virtual double grade( ) const;
   virtual std::istream& read( std::istream& );
protected:
   std::istream& read_common( std::istream& );
   double midterm, final;
   std::vector<double> homework;
private:
   std::string n;
};
```

Coreクラスの解説

- public: すべてから利用可能
- protected: 派生クラスから利用可能
  - midterm, final, homework. read_common()はGradクラスでも利用可能
- private: 派生クラスからでも利用不可能
  - nはGradクラスからは直接は利用できない
  - ただし，name()という関数は利用可能

Gradクラス

```
class Grad : public Core {
public:
   Grad( );
   Grad( std::istream& );
   double grade( ) const;
   std::istream& read( std::istream& );
private:
   double thesis;
};
```

Gradクラスの解説

- class Grad : public Core
  - Gradクラスは，Coreクラスをpublic継承する
    - Coreクラスのpublicは，Gradクラスのpublicに
    - Coreクラスのprotectedは，Gradクラスのprotectedに
    - Coreクラスのprivateは，Gradクラスのprivateに
  - Gradクラスは，以下のCoreクラス
    - name(), read()というメンバ関数
    - n, midterm, final, homeworkというメンバ変数
  - を利用可能
  - さらに
    - thesisというメンバ変数を追加
    - grade(), read()というメンバ関数はGradクラスで再定義

関数の定義 Coreクラス

```
std::string Core::name( ) const { return n; }

double Core::grade( ) const {
   return ::grade( midterm, final, homework );
}

std::istream& Core::read_common( std::istream& in ) {
   in >> n >> midterm >> final;
   return in;
}
```

関数の定義 Coreクラス

```
std::istream& Core::read( std::istream& in ) {
   read_common( in );
   read_hw( in, homework );
   return in;
}
```

関数の定義 Gradクラス

```
std::istream& Grad::read( std::istream& in ) {
   read_common( in );
   in >> thesis;
   read_hw( in, homework );
   return in;
}

double Grad::grade() const {
   return min( Core::grade(), thesis );
}
```

継承とコンストラクタ

- 派生クラスのオブジェクトの生成
  - 全オブジェクト分のメモリを確保
  - 基底クラスのコンストラクタを実行，基底クラスの部分を初期化
  - 派生クラスのメンバを初期化（: 以降の部分）
  - 派生クラスのコンストラクタの中身（{}の中の部分）を実行

継承とコンストラクタ

```
Core::Core( ) : midterm(0), final(0) { }

Core::Core( std::istream& is ) { read(is); }

//  下の2つともCore()で基底クラス部分を初期化
Grad::Grad ( ) : theisis (0) { }

Grad::Grad( std::istream& is ) { read(is); }
```

多態性(ポリモルフィズム)と仮想関数

```
bool compare(const Core&c1, const Core& c2)
{
    return c1.name() < c2.name();
}
```

- Coreオブジェクトでも Gradオブジェクトでも実行可能
  - Core c1(cin), c2(cin); Grad g1(cin), g2(cin);
  - compare(c1, c2); compare(g1, g2);
  - compare(c1, g2);
  - Coreクラスも Gradクラスも Core::name()を利用するため

多態性(ポリモルフィズム)と仮想関数

```
bool compare_grades( const Core&c1, const Core& c2 )
{
    return c1.grade( ) < c2.grade( );
}
```

- 上の関数はCoreクラスのCore::grade()関数を実行
- しかし，Gradクラスではgrade()を実行すべき
- このままでは，正しく動作しない

virtual（仮想）関数

```
class Core  {
public:
    virtual double grade() const;
};
```

- `Core::grade()`に`virtual`を付けると，`compare_grades(c1, c2)`を実行時に`c1`と`c2`のオブジェクトの型を見て，該当するオブジェクトのクラスの`grade()`関数を実行
- `Grad::grade()`に`virtual`キーワードは付けなくても良い（継承されるため）

```
//  override 仮想関数を定義することを明示的に表現する
virtual double grade() const override;
//  final これ以降の仮想関数の定義を禁止する
virtual double grade() const final;
```

動的結合と静的結合

- 動的結合: プログラムの実行時にオブジェクトの型が決められる
  - 仮想関数で，参照かポインタを通して呼ばれる時
- 静的結合: プログラムのコンパイル時にオブジェクトの型が決められる
  - 仮想関数でも，オブジェクトを通して呼ばれる時は静的結合
  - 静的結合の例

```
bool compare_grades( Core c1, Core c2 )
{  return c1.grade( ) < c2.grade( );  }
```

動的結合と静的結合

- `Core c;`
- `Grad g;`
- `Core *p = &c;  p = &g;`
- `Core& r = g;  r = c;`
- `c.grade();  //  Core::grade()`に静的結合
- `g.grade(); //  Grad::grade()`に静的結合
- `p->grade(); //  `動的結合 `p`の指す型に依存
- `r.grade(); //  `動的結合 `p`の指す型に依存

ハンドルクラス

- Coreクラスは学部生，Gradクラスは大学院生
- 今のままでは，両者が混在した場合，それぞれのオブジェクト準備しなければならない
  - 学部生ならデータの先頭がu，大学院生ならg
  - 名前，中間試験，期末試験，大学院生なら論文（学部生はなし），演習の成績
  - u name 100 100 50 50 50
  - g name 100 100 80 50 50 50
- 両方を同時に扱うためのハンドルクラスを定義

virtualな型のコンテナ

- vector<Core> students;
- Core record;
  - 上の2つはCore型のオブジェクトのみ
  - Grad型のオブジェクトは扱えない
  - 静的結合だから

virtualな型のコンテナ

- vector<Core*> students;
- Core* record;
  - これだと動的結合になるので，Core型とGrad型のオブジェクトが利用可能
- 上はポインタだけで実体（メモリ）が確保されていないため実行不可能
- ハンドルクラスにはメモリ管理機能が必要

Student_infoクラス

```
class Student_info{
private:
   Core *cp;
public:
   Student_info( ) : cp( 0 ) { }
   Student_info( std::istream& is ) : cp( 0 ) { read(is); }
   Student_info( const Student_info& is );
   ~Student_info( ) { delete cp; }
   std::istream& read( std::istream& );
   std::string name( ) const {
        if(cp) return cp->name();
        else throw std::runtime_error("uninitialized");
   }
   static bool compare(const Student_info& s1, const
   Student_info& s2)  {
        return s1.name() < s2.name();
   }
};
```

staticなメンバ関数

- 特定のオブジェクトではなく，クラスに付随した関数
- クラスのスコープ内で定義
  - 常に`Student_info::compare`でアクセスされる
  - 通常の関数だと`s.compare`でアクセス
  - `sort`関数を呼び出す時に，他の`compare`関数と区別するために利用

ハンドルを読む

```
std::istream& Student_info::read( std::istream& is)
{
   delete cp;
   char ch;
   is >> ch;
   if(ch == 'u')     {
        cp = new Core( is );
   }
   else{
        cp = new Grad( is );
   }
   return is;
}
```

ハンドルオブジェクトのコピー

```
class Core{
   friend class Student_info;
protected:
   virtual Core* clone( ) const { return new Core(*this); }
//   以前と同じ
};

class Grad {
protected:
   virtual Core* clone( ) const { return new Grad
   (*this); }
//   以前と同じ
};
```

ハンドルオブジェクトのコピー

```
class Grad {
protected:
   virtual Core* clone( ) const { return new Grad
   (*this); }
//   以前と同じ
};
```

- GradではStudent_infoをfriendにする必要なし
  - Student_info はGrad::cloneを使わないから
  - friendは継承されない

ハンドルオブジェクトのコピー

```
Student_info::Student_info( const Student_info& s) :
   cp( 0 ) {
   if( s.cp ) cp = s.cp -> clone( );
}
```

ハンドルオブジェクトのコピー

```
Student_info& Student_info::operator=(const Studnet_info&
   s) {
   if(&s != this)    {
        delete cp;
        if(s.cp)
               cp = s.cp->clone();
        else
               cp = 0;
   }
   return *this;
}
```

ハンドルクラスを使う

```
int main()
{
   vector<Student_info> students;
   Student_info record;
   //   データの読み込み
   while( record.read( cin ) )     {
        students.push_back( record );
   }
   //   学生をアルファベット順に並び替える
   sort( students.begin(), students.end(),
   Student_info::compare );
```

ハンドルクラスを使う

```
for(vector<Student_info>::size_type i = 0; i !=
students.size(); ++i)    {
     cout << students[ i ].name() << " ";
     //  Student_infoクラスにvalid()という関数を追加
     if( students[ i ].valid() ) {
          cout << students[ i ].grade() << endl;
     }
     else {
          cout << "No homework" << endl;
     }
}
```

第14章「メモリを（ほとんど）自動的に管理する」

ポインタのようなハンドルクラス

- Handleはオブジェクトを指す
- Handleオブジェクトをコピーできる
- Handleオブジェクトが有効な内部データを指しているかテストできる
- Handleオブジェクトが派生クラスのオブジェクトを指す場合に多態性が表れる（正しい関数が自動的に選択され実行される）

Handleクラスの使い方

- vector< Handle<Core> > students;
- Handle<Core> record;
  - 前回のインプリメントでは vector<Core*> とするか，ハンドルクラスを利用した vector<Student_info>

Handle クラス

```
template <class T> class Handle {
public:
   Handle( ) : p(0) {};
   Handle( const Handle& s ) : p(0) { if(s.p) p=s.p-
   >clone(); }
   Handle& operator=( const Handle& );
   ~Handle( ) { delete p; }
   Handle( T* t ) : p(t) {}
   operator bool() const { return p; }
   T& operator*() const;
   T* operator->() const;
private:
   T* p;
};
```

Handle クラス

```
template <class T>
Handle<T>& Handle<T>::operator=(const Handle<T>& rhs)
{
  if( rhs != *this ){
    delete p;
    p = rhs.p? clone( rhs.p ): 0;
  }
  return *this;
}
```

Handle クラスの外部関数

```
template <class T>
T* clone(T* p)
{
  return new T(*p);
}
```

- すべての型で`clone()`関数が使えるようにテンプレート関数
- `T`型が `T( T& )` の形式のコンストラクタを持つこと

型変換の演算子

- `operator bool() const { return p; }`
  - `p`を`bool`型に変換，つまり有効(`p`が`0`以外)のとき`true`，無効(`p`が`0`)のとき`false`を返す
  - 型変換の演算子は，戻り値の型がない

型変換の演算子

- pの参照を返す演算子 *

```
template <class T>
T& Handle<T>::operator*() const  {
   if(p)  return *p;
   throw std::runtime_error("unbound Handle");
}
```

- Handle<Core> record;
  - *recordで，recordが保持しているポインタの参照を返す
- (*record).valid() のような使い方

型変換の演算子

- p(ポインタ)を返す演算子 ->

```
template <class T>
T* Handle<T>::operator->() const  {
   if(p)  return p;
   throw std::runtime_error("unbound Handle");
}
```

- record -> valid() のような使い方

Handleクラスは，ポインタの管理が可能

さらにもう一歩進めて

- Handle型はデータを持つオブジェクトのコピーと代入の際は、必要のない時でもコピーする
- 一方で、データを共有するなどコピーが必要ない時もある

- データを共有できるかをどうかを決められるハンドル: Ptr型を作成する

Ptr型

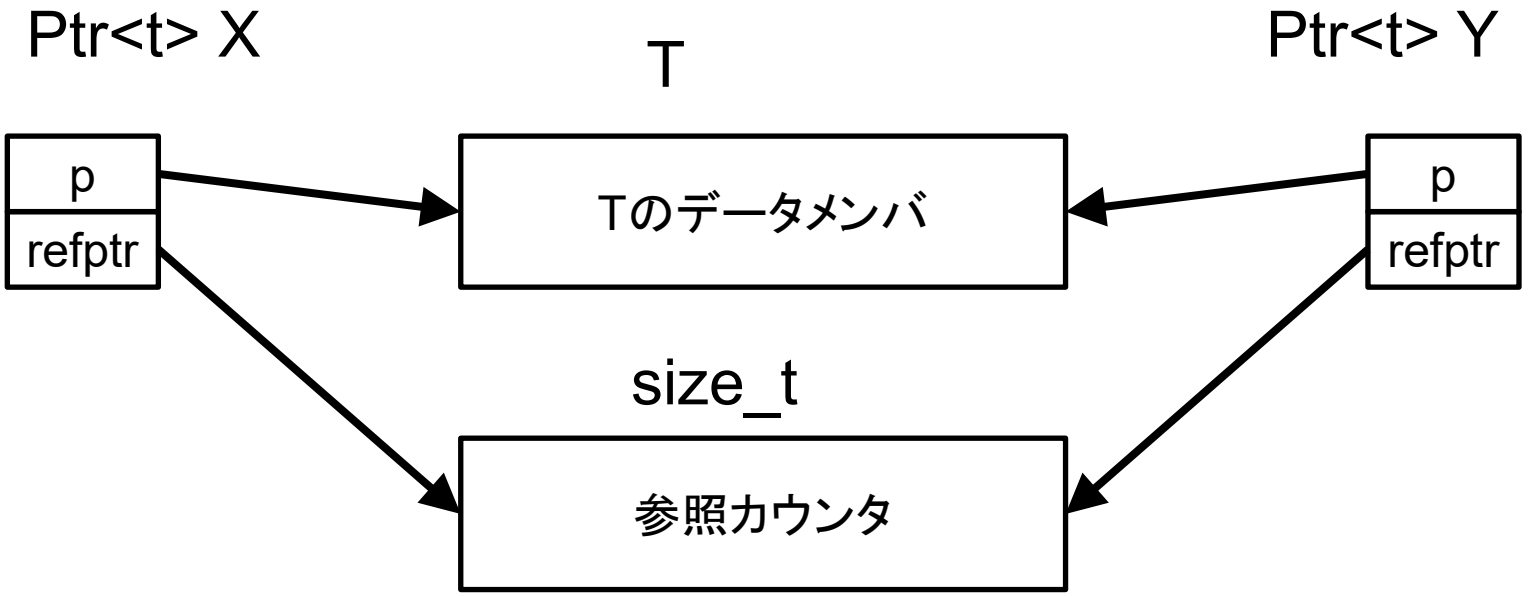
- 参照カウンタという変数を導入する
  - そのオブジェクトが他のどのオブジェクトと結びついているのかを保持
  - 他のオブジェクトにコピーするときに、データを複製せずに参照カウンタを1増やす
  - 必要のなくなったデータを削除する
- 必要なときには、他のオブジェクトのデータを複製して自分に持たせる

```
template<class T>
class Ptr  {
  private:
    T* p; //  データ保持するデータ
    size_t* refptr; //  参照カウンタ

  public:
    //  デフォルトコンストラクタ
    Ptr(): refptr( new size_t(1) ), p(0) {};
    Ptr(T* p): refptr(new size_t(1), p(t)) {};
    //  コピーコンストラクタ
    Ptr(const Ptr& h) : refptr(h.refptr), p(h.p)
{++*refptr};
```

```
    Ptr& operator=(const Ptr&);
    ~Ptr();
    operator bool() const {return p};
    T& operator*() const  {
      if(p) return *p;
      throw std::runtime_error("unbound Ptr handle");
    };
    T* operator->() const  {
      if(p) return p;
      throw std::runtime_error("unbound Ptr handle");
    };
    void make_unique(); //  複製に利用
}
```

データの複製

Prt<T> x, y; x = y; としたとすると



データの複製

- 前のスライドのように同じオブジェクト指しているときに、同じ内容の別オブジェクト作成するときに使う
- 元のオブジェクトの参照数を減らして、自分には同じデータを複製し、参照数を1とする

```
void Ptr::make_unique()  {
  if(*refptr != 1)  {
    --*refptr;
    refptr = new size_t(1);
    p = p? clone(p): 0;
  }
}
```

clone()関数

```
//  clone()というメンバー関数があるクラスには
template<class T> T*clone(T* tp)  {
  return tp->clone();
}

//  clone()というメンバー関数がないクラスには個別に定義する
Vec<char>*clone(const Vec<char>* vp)  {
  return new Vec<char>(*vp);
}
```

その他の関数

```
//   デストラクタ、データが必要がなくなったら削除
template <class T> Ptr<t>::~Ptr()  {
  if(--*refptr == 0)  {
    delete refptr;
    delete p;
  }
}
```

その他の関数

```
//   代入、データの複製はしない、必要がなくなったデータが発生したら削除
template <class T>
Ptr<t>& Ptr<t>::operator=(const Ptr& rhs)  {
  ++*rhs.refptr;
  //   データがどこから参照されず、必要なくなったとき
  if(--*refptr == 0)  {
    delete refptr;
    delete p;
  }
  //   右辺から値コピー
  refptr = rhs.refptr;
  p = rhs.p;
  return *this
}
```