応用プログラム言語II / 演習II
#01
better C としての C++（その1）

情報システム学科
坂本政祐
sakapon@sit.ac.jp

教科書・補助サイトなど

- 教科書
  - 特にありません。
  - いろんな言語とライブラリを使うことになるので，それらのリファレンス・マニュアル（大抵Webにある）が教科書とも言えるでしょう。
- 補助サイト　http://comsys.sit.ac.jp/lecture.html
  - ここに，講義資料（パワポのハンドアウト等）を置いていく予定。
  - なくしたり，欠席したりした場合はここから自分で印刷してください。
- ソースコード
  - ソースコードは毎回提出。
  - 日本語ファイル名にはしないで下さい。採点できません（しません）。
  - 提出用フォルダはネットワークドライブに用意します。

この講義の前提条件

- （講義で教わった範囲で）Cで大抵のコードは書ける。
  - 書けなくても，自分で思い出せる手段を持っている。
- 自分で調べる意欲がある。
- プログラミングを楽しいと感じることができる。

最初の数回の予定

- better C としてのC++
  - Cには未成熟な部分が沢山あって，「アルゴリズムをコーディングしていきたい」という本質部分とは関係ない瑣末部分でしち面倒くさい部分がいろいろあります。文字列専用の型が無いとか，配列のサイズが静的に決まってないといけないとか。
  - C++ にはこれを解決する仕組みがあります。
  - つまり，「少し便利になったC（better C）」として使えます。
- オブジェクト指向言語としてのC++
  - オブジェクト指向という概念で，世の中のモノやコトを表現できるようになります。
  - C++ はオブジェクト指向言語のひとつです。C++ のオブジェクト指向の文法はどのようなものでしょうか。何が嬉しくなるのでしょうか。

本題に入る前に:
今後常に覚えていて欲しいこと

名前重要

- 今ここに2つの関数があります。機能は全く一緒です。何をする関数かわかりやすいのはどっち?

(a)

```
double func(double a, double b, double c)
{
  int z;
  z = (a + b) * c / 2.0;
  return(z);
}
```

(b)

```
double daikei_menseki(double joutei, double katei, double takasa)
{
  int menseki;
  menseki = (joutei + katei) * takasa / 2.0;
  return(menseki);
}
```


名前重要

- 当然(b)ですよね。なぜなら，各「名前」がわかりやすいからです。各名前とは:
  - 関数名:　daikei_menseki
  - 仮引数名:　joutei, katei, takasa
  - ローカル変数名:　menseki
- 例えばC言語では，変数名は少なくとも31文字までは認識されます(それ以上は処理系依存)。
- ので，長くても良いので，わかりやすい名前をつけましょう。
- よほど使い捨ての変数でもない限り，無意味に一文字な変数名は悪です。
- どうしても一文字変数名を使いたかったら，最低でも「その意味の頭文字」から取るようにしましょう。
  - 例: 無意味に a, b とかではなく，足し算結果なら s とか。（sumだから）

名前重要

- この講義でのサンプルプログラム＆解答例の命名方針:
  - 日本語をローマ字で表現。（本当は英語で表現した方がカッコイイでしょうが，そうすると初心者にはC言語の予約語と区別つかなくなるので）
  - 2語以上からなる場合は _ で接続。
    - 例: daikei_menseki
  - 1文字変数名はなるべく使わないが，以下は例外。
    - ループカウンタは例外。例: for(i=0; i<10; i++)
    - 名前のつけようがないような汎用的な仮引数。
  - 用途が具体的でない場合（汎用的な場合）は，ローマ字ではなく英語省略形を使うことあり（主にスペースの理由で）。
    - 例: 配列 ary　　array の省略形
    - 例: 文字列 str　　string の省略形
    - 例: ポインタ ptr　　pointer の省略形

名前重要

applied-progII #01

- ちなみに，一度「まずい名前付けちゃったな」と思っても大丈夫。現代のプログラミング環境には「リファクタリング」の機能があります。
  - ソースコード内の変数名を一括して賢く変換できる。
  - C# 2010ではデフォルト。
  - ただ VC 2010 には残念なことにその機能が無いみたいですが…。

9

インデント超重要

applied-progII #01

- インデントというのは，Tabキー（もしくはスペース複数個）による字下げのあれ。見やすいのはどっち?

インデントあり(タブ幅=4)

```
#include <stdio.h>

int main(void)
{
    int i;
    for(i=0; i<100; i++){
        if (i%2 == 0){
            printf("%dは偶数です¥n",i);
        } else {
            printf("%dは奇数です¥n",i);
        }
    }
    return 0;
}
```

インデントなし

```
#include <stdio.h>

int main(void)
{
int i;
for(i=0; i<100; i++){
if (i%2 == 0){
printf("%dは偶数です¥n",i);
} else {
printf("%dは奇数です¥n",i);
}
}
return 0;
}
```

10

- インデントの基本: 関数やブロックの中に入る度に1段インデント。＝「{」から「}」に挟まれた部分はインデント。
- ブロックの中にブロックが出てきたらさらに1段深く（より右側へと）インデント。
- ブロックの閉じカッコ「}」は，そのブロックがスタートした縦位置に置く。

```
#include <stdio.h>

int main(void)
{
    int i;
    for(i=1; i<100; i++){
        if (i%2 == 0){
            printf("%dは偶数です。¥n",i);
            if (i%4 == 0){
                printf("かつ4の倍数です。¥n");
            }
        }

    return(0);
}
```

（Tab, Tab, Tab, Tab）

インデント超重要

applied-progII #01

- インデントに限らず，ソースコードを規則正しいフォーマットで書くことは超重要。
  - プログラムの構造（≒アルゴリズムの構造）が一目瞭然。
    - 制御構造(if や forループ や whileループ)が，どこから始まってどこで終わっているのか。
    - ある行が，今どの深さのブロックに居るのか。
  - 間違えにくい。間違えてもそれを見つけやすい。
  - 修正しやすい。デバッグしやすい。
  - 保守しやすい ＝ 後でそのソースを見たときに読みやすい。
    - 中規模以上の大抵のプログラムは，一度作ってその場限りで動けば良いやというわけにはいかない。
  - 採点しやすい（私が）。

12

ここまでの小まとめ

applied-progII #01

- 名前重要
- インデント重要（きれいに書くこと重要）

13

C++の基本

14

コードの例

applied-progII #01

- 簡単な例

```
#include <stdio.h>

int main(void)
{
  printf("Hello, World.¥n");
  return 0;
}
```

15

C++の基本

applied-progII #01

- エントリポイント（プログラムが開始するところ）はmain()関数です。
- 基本的な文法もほぼCと同じです。
- 拡張子は:
  - .cpp（この講義ではこれに統一しましょう）
  - .c++
  - .cxx
  - .CC

16

開発環境

applied-progII #01

- 地味版の組み合わせ
  - 任意のエディタ　ソースコードを入力してファイルとして保存できる
  - Cygwin　コマンドを入力できる環境。UNIX OSに似せてる
  - g++　コンパイラ。ソースコードを実行ファイルに変換。

おすすめ

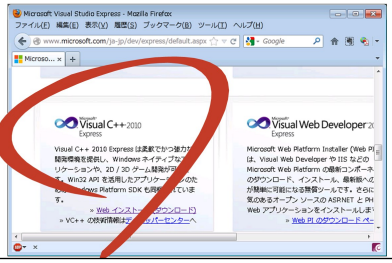
- Microsoft版
  - Visual Studio　ソースコードの入力，コンパイル，実行ファイルの実行がこれひとつでまとめてできる。「統合開発環境（IDE）」。
  - Visual C++　Visual Studio の中で実行されるコンパイラ。Visual Studio と一体化しているので通常は存在をあまり意識しない。通称VC。

17

開発環境ちなみに

applied-progII #01

- Visual Studioにはいくつかのグレードがあり，大学に入っているのは有料版のProffessional。
- 無料版のExpressもあり，機能も大して違わない。ので，自宅PCにインストールをおすすめ。

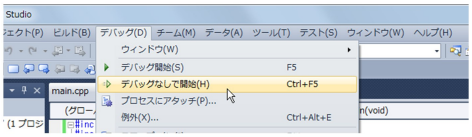


18

実行するときは

applied-progII #01

- デバッグ→デバッグなしで開始（Ctrl + F5）でやりましょう。プログラムが終了した時点で一旦停止してくれます。



19

文字列 std::string

20

Cでの文字列を思い出す

applied-progII #01

- Cでの文字列は，「char型の配列」として実現されていた。

mojiretsu.c

```c
#include <string.h>

int main(void)
{
  char str[20];

  strcpy(str, "saikoudai");

  return 0;
}
```

21

Cの文字列の頂けない点1

applied-progII #01

- 所詮配列なので，そのサイズ（文字列の長さ）は静的に決まっていないといけない。

mojiretsu.c

```c
#include <string.h>

int main(void)
{
  char str[20];

  strcpy(str, "saikoudai");

  return 0;
}
```

- 例えば，名簿を作りたかったらどうするか? ものすごく長い名前を想定して，その値に固定するしかない。
- それでもさらにそれより長い名前の人が出現したら?
  - 動的確保という手段も一応ありますが…。

22

Cの文字列の頂けない点2

applied-progII #01

- 代入が直感的でない。他の型と扱いが違いすぎる。

mojiretsu.c

```c
#include <string.h>

int main(void)
{
  char str[20];

  strcpy(str, "saikoudai");

  return 0;
}
```

- 例えば，一度宣言や初期化した後では，
  str = "saikoudai";
  みたいなことはできない。int型とかはできるのに！
- 仕方ないので，strcpy()という関数を使う。連結も同様にstrcat()を使う。

23

そこで std::string（スタンダードストリング）型

applied-progII #01

- そこでC++には std::string という型（厳密には「型」じゃないですが）があります。
- 前述の欠点はこれによって解消されています。

mojiretsu.cpp

```c
#include <string>

int main(void)
{
  std::string str;

  str = "saikoudai";

  return 0;
}
```

「.h」はつきません！
Point
最大長を予め決める必要は無い（可変長）
直感的に代入できる
Point

24

しかも連結も直感的にできる

applied-progII #01

- 文字列の連結も，strcat()などというダサい関数を使わなくても「+」演算子で直感的にできます。

mojiretsu.cpp
```
#include <string>

int main(void)
{
  std::string str;

  str = "saikoudai";

  str = str + " is nice.";

  return 0;
}
```
直感的に連結できる
Point

25

さらに比較も直感的にできる

applied-progII #01

- 文字列の比較も，strcmp()などというダサい関数を使わなくても「==」演算子で直感的にできま

mojiretsu.cpp
```
#include <string>

int main(void)
{
  std::string str1;
  std::string str2;

  str1 = "saikoudai";
  str2 = "saikoudai";

  if (str1 == str2){
    std::cout << "OK" << std::endl;
  }

  return 0;
}
```
Point
直感的に比較できる

- std::cout とかっていうのは画面出力（詳しくは後述）。

26

std::string 用の関数たち

applied-progII #01

- Cには文字列の長さを測るための strlen() とかありましたが，それは std::string には使えません。
- その代わり，std::string 用にもっと便利な関数が用意されています。

27

std::string 用の関数たち

applied-progII #01

- 例えば，長さを返す関数 size() は以下のように使います。

```
#include <string>
#include <iostream>

int main(void)
{
  std::string str = "sakamoto";
  int nagasa;

  nagasa = str.size();

  std::cout << nagasa << std::endl;

  return 0;
}
```
Point

28

オブジェクト.関数名()

applied-progII #01

- str.size()みたいな，オブジェクト名.関数名という書き方の詳しい意味については，オブジェクト指向のときにまたお話します。
- とりあえず，あるオブジェクトに対して何々をしたいというのを表現していると思って下さい。

29

std::string のためのその他の関数

applied-progII #01

- size()以外にも，std::string型のオブジェクトには以下の関数や演算子が用意されています（とりあえず使いそうなもののみ）。
- 以下では引数の仕様を書いていませんので，それは自分で調べてください。
  - 演算子[]　指定の位置の1文字(のリファレンス)を返す
  - substr()　指定の位置からの部分文字列を返す
  - insert()　指定の位置へ別の文字列を挿入する（破壊的）
  - replace()　指定の位置を別の文字列で置換する（破壊的）
  - find()　ある文字列が含まれているか検索する
  - compare()　別の文字列と比較する
  - clear()　すべての文字を削除する（破壊的）
  - erase()　文字列の一部を削除する（破壊的）
  - empty()　空かどうかチェックする

30

std::string 用の演算子，関数たちの使用例

```
#include <iostream>
#include <string>

int main(void)
{
  std::string str = "sakamoto";

  std::cout << "test1 [" << str[4]                          << "]" << std::endl;

  std::cout << "test2 [" << str.substr(3, 4)                << "]" << std::endl;

  str.insert(4, "_x_");
  std::cout << "test3 [" << str                             << "]" << std::endl;

  str.replace(0, 4, "hashi");
  std::cout << "test4 [" << str                             << "]" << std::endl;

  std::cout << "test5 [" << str.find("_x_")                 << "]" << std::endl;

  std::cout << "test6 [" << str.compare("hashi_x_moto")     << "]" << std::endl;

  str.erase(5, 3);
  std::cout << "test7 [" << str                             << "]" << std::endl;

  str.clear();
  std::cout << "test8 [" << str                             << "]" << std::endl;

  std::cout << "test9 [" << str.empty()                     << "]" << std::endl;

  return 0;
}
```

31

std::string 用の関数たちの使用例の実行例

applied-progII #01

実行例
```
test1 [m]
test2 [amot]
test3 [saka_x_moto]
test4 [hashi_x_moto]
test5 [5]
test6 [0]
test7 [hashimoto]
test8 []
test9 [1]
```

32

補足

applied-progII #01

- Cでは，1文字は 'z' のように書いて文字列は "z" のように書くという七面倒臭いクオート記号の区別がありました。
- 一方C++では，std::string とそれ用関数を使っている限りはすべて " " でうまくいきます。簡単ですね。

```
#include <iostream>
#include <string>

int main(void)
{
  std::string str = "sakamoto";

  if (str.substr(0, 1) == "s"){
    std::cout << "一致しました" << std::endl;
  }
  return 0;
}
```

33

ここまでの小まとめ

applied-progII #01

- C++には文字列専用の型として std::string がある。
- 代入や連結が直感的にできる。
- 関数もいろいろあり，文字列の操作がかなり簡単。

34

画面出力とキーボード入力

35

画面に表示するには

applied-progII #01

- Cのprintf()みたいなことをやりたいときは，「標準出力ストリーム」というのを使います。

```
#include <iostream>

int main(void)
{
  int num = 40;

  std::cout << num << std::endl;

  return 0;
}
```

Point

std::cout という画面を表すものに num を「流しこむ」イメージ。std::endl は改行。

実行例

```
40
```

36

標準出力ストリームの利点

applied-progII #01

- std::cout の利点
  - %d とか %f とかは必要ない。その辺は適当にうまくやってくれる。
  - いろいろ表示したいときは << でどんどん連結。

```
std::string name = "sakamoto";
int nenrei = 20;

std::cout << "私は" << name << nenrei << "歳です" << std::endl;
std::cout << "来年は" << nenrei + 1 << "歳です" << std::endl;
```

実行例

```
私はsakamoto20歳です
来年は21歳です
```

37

キーボードから入力するには

applied-progII #01

- Cのscanf()みたいなことをやりたいときは，「標準入力ストリーム」というのを使います。

```
#include <iostream>

int main(void)
{
  int num;

  std::cout << "数を入れてください: ";

  std::cin >> num;

  return 0;
}
```

Point

38

キーボードから入力するには

applied-progII #01

- 文字列も簡単です。

```
#include <iostream>
#include <string>

int main(void)
{
  std::string str;

  std::cout << "名前を入れて下さい: ";

  std::cin >> namae;

  std::cout << namae << std::endl;

  return 0;
}
```

39

std::string 画面表示の printf() 版

applied-progII #01

- 旧来からの printf() を使う方法もあります。

mojiretsu-print-printf.cpp

```
#include <string>
#include <stdio.h>

int main(void)
{
  std::string str;

  str = "saikoudai";

  printf("%s", str.c_str());

  return 0;
}
```

Point

str を c_str() で「C言語の文字列」化してから %s で表示。

40

ここまでの小まとめ

applied-progII #01

- 画面に出力するとき:
  - std::cout << 出力したいもの << std::endl;
  - std::cout << 出力したいもの;　（改行不要の時）
- キーボードから入力したいとき:
  - std::cin >> 入力対象の変数;
- どちらも，%d とか %f とかの七面倒臭いフォーマット指定子は不要。

41

配列よりも便利なコンテナ std::vector

42

Cの配列の欠点

applied-progII #01

- Cの配列の嫌なところは:
  - 嫌1: サイズが固定（固定長）。かつ静的に決めなきゃいけない。
  - 嫌2:サイズをプログラマが管理しなきゃいけない。

```
#define SIZE (10)

int main(void)
{
  int array[SIZE], i;

  for(i=0; i<SIZE; i++){
    array[i] = i;
  }
  return 0;
}
```

この10という数はコンパイル時に決まってないといけない。

ループが SIZE 回回る，というのをいちいち書かないといけない。（例えばまた別の配列があったら，それはまた別の回数回す必要があるし，それをプログラマが管理しなきゃいけない。）

Cの配列の欠点

applied-progII #01

- Cの配列の嫌なところは:
  - 嫌3:サイズをプログラマが管理しなきゃいけないので，関数に渡すときにサイズまで一緒に渡さなきゃいけない。

```
#define SIZE (10)

void print_each(int ary[], int size)
{
  int i;
  for(i=0; i<size; i++){
    printf("%d¥n", ary[i]);
  }
}

int main(void)
{
  int array[SIZE];
  ...

  print_each(array, SIZE);

  return 0;
}
```

Point

print_each()関数側では，ary[] の大きさがわからない。ので，サイズも一緒に渡してもらう必要がある。

Point

44

Cの配列の欠点

applied-progII #01

- 嫌1については動的確保で一応は何とかなる。
  - 動的。（実行時にサイズを決められる）
  - 可変長。（realloc()でサイズを伸ばせる）

```
#include <stdlib.h> // for malloc()

int main(void)
{
  int size;
  int *array;

  scanf("%d", &size);

  array = malloc(size * sizeof(int));

  array[0] = 10;
  array[1] = 20;
  ....
}
```

- が，嫌2，嫌3についてはどうにもならない。

45

そこでコンテナ

applied-progII #01

- C++には「コンテナ」というくくりのデータ構造が備わっている。
- いまあげつらった欠点を解消できる。
- コンテナはいくつかある。
  - std::vector　（スタンダードベクター；配列）
  - std::map　（スタンダードマップ；連想配列）
  - std::list　（スタンダードリスト；リスト構造）
  - std::set　（スタンダードセット；集合）
  - std::deque　（スタンダードデキュー；双方向キュー）
  - など。

46

コンテナ

applied-progII #01

- コンテナの利点
  - 動的に確保できる。
  - 可変長。（どんどん追加していける）
  - サイズはコンテナ自身が知っている（プログラマが管理する必要が無い）。なので，関数に渡すときもサイズを一緒に渡す必要がない。
  - コンテナにポインタを入れる場合は注意深く使う必要があるが，普通にオブジェクトを入れる分には便利な入れ物として使える。

47

std::vector

applied-progII #01

- std::vector
  - 普通の配列をより便利にしたもの。つまりこれも一次元に伸びていくデータ構造。
  - 格納する中身の型はCと同じく一種類のものしか格納できない。（0番目にはint型のものを，1番目にはdouble型のものを，とかってことはできない。※そういうことができる言語もあります。）

48

std::vector の例

applied-progII #01

■std::vector の例

```
#include <iostream>
#include <vector> // for std::vector

int main(void)
{
  // 宣言
  std::vector<int> vec;

  // データの追加
  vec.push_back(10);
  vec.push_back(20);

  // 中身へのアクセス
  std::cout << vec[0] << std::endl;
  std::cout << vec[1] << std::endl;

  return 0;
}
```

Point: vecという名前の，int型のstd::vectorを一つ宣言。この時点でサイズを決める必要は無い。サイズ0個の配列みたいなものを用意したということ。

Point: vecの末尾に「10」というデータを追加。サイズは1になる。

vecの末尾に「20」というデータを追加。サイズは2になる。

中身の読み取りは通常の配列と同じく [ ] 演算子。ゼロオリジン。

Point

実行例

```
10
20
```


std::vector のサイズ

applied-progII #01

■サイズは自分自身が知っている

- 先ほどのコードでは，push_back() によって2個の要素を追加したので，サイズは2になっているはず。
- Cの感覚では，こういうことを覚えておくためには例えば int counter という変数を作って，push_back() する度に counter++ などとする必要がある。
- std::vector ではそんなことする必要はなく，サイズは size() 関数でベクター自身に問い合わせれば良い。

size() の例

applied-progII #01

■size() の例

```
#include <iostream>
#include <vector> // for std::vector

int main(void)
{
  std::vector<int> vec;
  int vec_size;

  vec.push_back(10);
  vec.push_back(20);

  vec_size = vec.size();

  std::cout << vec_size << std::endl;

  return 0;
}
```

Point

右辺は，vec に対して，サイズはいくつ? と問い合わせている。その結果が左辺に代入される。

実行例

```
2
```


size() を使うとループがスマートに

applied-progII #01

■size() を使うと，まずループがスマートに書ける。

```
#include <iostream>
#include <vector> // for std::vector

int main(void)
{
  std::vector<int> vec;

  for(int i=0; i<100; i++){
    vec.push_back(i);
  }

  for(int i=0; i<vec.size(); i++){
    std::cout << vec[i] << std::endl;
  }

  return 0;
}
```

Point

vecのサイズがその時点で何個であろうが，すべて列挙したくば常にvec.size()回でよい。

別関数側でサイズがわかる（嫌3の解決）

applied-progII #01

■また，「嫌3」に関しては，以下のように，関数にサイズを一緒に渡す必要が無くなる。

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

void print_each(std::vector<int> ary)
{
  for(int i=0; i<ary.size(); i++){
    std::cout << ary[i] << std::endl;
  }
}

int main(void)
{
  std::vector<int> array;
  ...

  print_each(array);

  return 0;
}
```

ここにあった第2引数はもう要らない

ここでも size() でサイズがわかる

array だけを渡している

std::vector の例

applied-progII #01

■std::vector の例（サイズ指定版）

```
#include <iostream>
#include <vector> // for std::vector

int main(void)
{
  // 宣言
  std::vector<int> vec(100);

  // 代入
  vec[99] = 28;

  return 0;
}
```

Point

サイズ100個で宣言。

この場合すでに[0]～[99]までが使えるようになっているので，その間ならどれにでも [ ] 演算子で直接代入できる。

std::vector の例

applied-progII #01

■std::vector の例（サイズ指定＋可変版）

- 最初にサイズを指定し，後から追加ができる。

```
#include <iostream>
#include <vector> // for std::vector

int main(void)
{
  // 宣言
  std::vector<int> vec(100);

  // 末尾にデータを一つ追加
  vec.push_back(314);

  // サイズ確認
  std::cout << vec.size() << std::endl;

  return 0;
}
```

サイズ100個で宣言。[0]～[99]が使える。

一個追加。[0]～[100]が使えるようになる。

Point

101と表示される。

できないこともある

applied-progII #01

■これはできません

```
#include <iostream>
#include <vector> // for std::vector

int main(void)
{
  // 宣言
  std::vector<int> vec;

  // 代入
  vec[99] = 28;
  ....
}
```

push_back()で追加するのではなく，99番目に28を代入したいという気持ち。しかしこれは残念ながらできません。こう書いたら気を利かせて[0]～[99]が使えるようになると嬉しいのですが（実際，Rubyという言語とかではそういう振る舞いをします）。

std::vector のためのその他の関数

applied-progII #01

- push_back(), size()以外にも, std::vector型のオブジェクトには以下の関数が用意されています（とりあえず使いそうなもののみ）。
  - insert() 要素を途中に挿入する
  - front() 先頭要素を返す
  - at() ある位置の要素を返す
  - back() 最終要素を返す
  - clear() すべての要素を削除する
  - erase() ある要素（1個または複数）を削除する
  - pop_back() 最終要素を削除する
  - empty() 空かどうかチェックする

他の型も入れられます

applied-progII #01

- ここまでは std::vector<int>で説明してきましたが, もちろん他の型のベクターも作れます。
- <int>の部分を変えるだけです。

```
#include <iostream>
#include <vector> // for std::vector

int main(void)
{
  // 宣言                          Point
  std::vector<double> vec;

  // 代入                          Point
  vec.push_back(3.14);
  vec.push_back(6.28);

  // 中身へのアクセス
  std::cout << vec[0] << std::endl;
  std::cout << vec[1] << std::endl;

  return 0;
}
```


構造体も入れられます

applied-progII #01

- 自作構造体ももちろんOKです。

```
#include <iostream>
#include <string> // for std::string
#include <vector> // for std::vector

struct gakusei {
  std::string gakuseki;
  std::string namae;
};

int main(void)
{
  std::vector<struct gakusei> gakuseis;

  struct gakusei gaku1 = {"1003000", "鳰工花子"};
  struct gakusei gaku2 = {"1003999", "岡部太郎"};

  gakuseis.push_back(gaku1);
  gakuseis.push_back(gaku2);

  std::cout << gakuseis[0].gakuseki << std::endl;
  std::cout << gakuseis[0].namae    << std::endl;
  std::cout << gakuseis[1].gakuseki << std::endl;
  std::cout << gakuseis[1].namae    << std::endl;

  return 0;
}
```

push_back() はコピーをプッシュする

applied-progII #01

- ちなみに, push_back() はコピーを作ってからそれをベクターに追加するので, 一つの構造体でもpush_back()するたびに別ものとして入っていきます。

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

struct ningen {
  std::string namae;
};

int main(void)
{
  std::vector<struct ningen> ningens;
  struct ningen tmp;

  tmp.namae = "坂本";
  ningens.push_back(tmp);

  tmp.namae = "岡部";
  ningens.push_back(tmp);

  for(int i=0; i<ningens.size(); i++){
    std::cout << ningens[i].namae << std::endl;
  }
  return 0;
}
```

実行例

```
坂本
岡部
```


ここまでの小まとめ

applied-progII #01

- C++には, Cの配列よりもっと便利なコンテナがある。
- コンテナにはいくつか種類があるが, 連続した一次元データを扱うには std::vector が便利。
- std::vector は
  - 可変長。
  - サイズをそれ自身が覚えていてくれる。
  - std::vector<int>, std::vector<double>, std::vector<自作構造体> など, あらゆる型で作れる。
  - 加工したり値を取り出したりするための関数が用意されている。

今日の課題

applied-progII #01

- 課題1: キーボードから入力した文字列を, 1文字ずつ増やしながら表示するプログラムをC++で書いて下さい。例えば sakamoto と入力したら,

```
s
sa
sak
saka
sakam
sakamo
sakamot
sakamoto
```

のように出力されて欲しいということです。

今日の課題

applied-progII #01

- 課題2: キーボードから

```
1003000,sakamoto↵
```

のようにカンマ区切りで学籍番号と名前をひとつ入力すると, 以下のような構造体のそれぞれのメンバにこれらを代入するプログラムを書いてください。ひとつだけで良いです。

```
struct gakusei {
  std::string gakuseki;
  std::string namae;
};
```


今日の課題

applied-progII #01

- 課題3: キーボードから課題2のような行を

```
1003000,sakamoto↵
1003001,inoue↵
1003002,maeda↵
@↵
```

のように次々に入力していくと, それを std::vector に次々に追加していくというプログラムを書いて下さい。

- @＋エンターを入力したとき, 入力終了とします。
- うまくできたか確かめるために, 入力終了後に画面にすべて表示して下さい。

自分で調べてみるひとへ補足

applied-progII #01

- std::string ではなく string と書いてある資料もよくあると思いますが，全く同じものです。ただし，そのコードには必ず using namespace std; という一文があると思います。
- namespace は名前空間と言って，名前の衝突を避けるための仕組みです。意味やメリットや危険性をわからず闇雲に using namespace std; と書くのも良くないので，この資料では std:: を頭につける方に統一しています。

65

今日の落穂ひろい

applied-progII #01

- 「クラス」や「テンプレート」を説明しないうちに std::vector とかを説明するのはホントは邪道なのかも知れません。
  - が，Cのあまりの面倒くささに心が折れそうになっていると思いますので，現代はもっと楽に書けるんだよーというのを先に実感してもらいたいと思ってこうしました。

66