




応用プログラム言語II / 演習II

#03

better C としての C++(その3)

情報システム学科
 坂本政祐
 sakapon@sit.ac.jp


1



前回までの復習


- better C としてのC++
 - 文字列専用型: `std::string`
 - 標準出力ストリーム: `std::cout`
 - 標準入力ストリーム: `std::cin`
 - コンテナの一つ: `std::vector`
 - コンテナの一つ: `std::map`
 - コンテナのトラバース: イテレータ

2



どんな型でもどんと来い: 関数のオーバーロード

3



Cでは同じ関数名で引数の並びが異なる 関数は作れない


- 例えば、「2つの数の大きい方を返却する関数」を作りたいとします。
- ところが、Cではこの場合、int型用のそれとdouble型用のそれは兼用できません。別々に作ることになります。
- 引数の個数が違ったらそれもまた別に作らなきゃいけません。

4


例

```

// C
#include <stdio.h>
int max2_for_int(int n1, int n2)
{
    return(n1>n2? n1:n2);
}
double max2_for_double(double d1, double d2)
{
    return(d1>d2? d1:d2);
}
int max3_for_int(int n1, int n2, int n3)
{
    int max = n1;
    max = max>n2? max:n2;
    max = max>n3? max:n3;
    return(max);
}
int main(void)
{
    printf("max of {d}, {d}: {d}\n", 32, 58,
        max2_for_int(32, 58));
    printf("max of {f}, {f}: {f}\n", 3.14, 6.28,
        max2_for_double(3.14, 6.28));
    printf("max of {d}, {d}, {d}: {d}\n", 32, 58, 24,
        max3_for_int(32, 58));
    }
                    
```



5



関数のオーバーロード


- 別々に作るのはまあ仕方ないとしても、呼び出し側の立場からしてみればどうせやりたいことは同じなのだから、「引数の型を見て良きに計らってくれよ」と思うわけです。
- そこでC++には、やりたいことが同じなら、**引数の型や個数が異なっても名前を兼用**することができる、という仕組みがあります。
- これが**関数のオーバーロード**です。

6


例

```

// C++
#include <iostream>
int max(int n1, int n2)
{
    return(n1>n2? n1:n2);
}
double max(double d1, double d2)
{
    return(d1>d2? d1:d2);
}
int max(int n1, int n2, int n3)
{
    int max = n1;
    max = max>n2? max:n2;
    max = max>n3? max:n3;
    return(max);
}
int main(void)
{
    std::cout << "max of {" << 32 << ", " << 58 << "}: ";
    std::cout << max(32, 58) << std::endl;
    std::cout << "max of {" << 3.14 << ", " << 6.28 << "}: ";
    std::cout << max(3.14, 6.28) << std::endl;
    std::cout << "max of {" << 32 << ", " << 58 << ", " << 24 << "}: ";
    std::cout << max(32, 58, 24) << std::endl;
    }
                    
```



7



関数のデフォルト引数

- また、C++では、本来引数を要求する関数に実引数をあえて渡さないと、デフォルトの値を使ってくれるという機能があります。
- これを**デフォルト引数**といいます。
- 仮引数の変数名の後ろに「= 値」とくっつけておくことで、呼び出し側でその引数を省略したときにその値を使ってくれます。

```

void func(int arg1, int arg2 = 100)
{
    ...
}
                    
```

8

デフォルト引数の例 SIT applied-progII #03

■ 馴染みの居酒屋で酒を注文する状況を関数化したものです。

- 何も言わないで注文(=実引数無し)しても、居酒屋の主人(=関数の側)は、いつもの奴ね(=デフォルト引数)ということだからしてくれるわけです。
- 逆に、たまに気分が変わったときは、あえてそれを言う(=実引数有り)と、主人はそれに合った動きをしてくれるわけです。

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

void chuumon(std::string shina = "ビール")
{
    if (shina == "ビール")
    {
        std::cout << "あいよ" << std::endl;
    }
    else if (shina == "日本酒")
    {
        std::cout << "お、珍しいね" << std::endl;
    }
    else if (shina == "カルーアミルク")
    {
        std::cout << "熱でもあるのかい?" << std::endl;
    }
}

int main(void)
{
    chuumon();
    chuumon();
    chuumon("日本酒");
}
```

デフォルト引数の使い途と制限 SIT applied-progII #03

■ デフォルト引数は、

- ふだん、多くの場合はこういうふるまいで良いのだけど、
- たまに例外的にこうふるまって欲しい場合がある
- というような場合に便利なくみです。

■ デフォルト引数は何個でも設定できます。

■ ただし仮引数リストの右側の方にあるものにはしか設定できません。

```
void func(int arg1 = 100);
void func(int arg1, int arg2 = 100);
void func(int arg1 = 100, int arg2);
```

ここまでの小まとめ SIT applied-progII #03

- C++では関数のオーバーロード(多重定義)ができます。
- 関数のオーバーロードを使うと、同じような機能で引数の型・個数が違う場合でも、一つの関数名に共通化できます。
 - ただし定義はそれぞれでなければいけません。
- また、関数のデフォルト引数も設定できます。「よくある場合」に楽をしたいときに便利です。

SIT

ファイル入出力

C++のファイル入出力 SIT applied-progII #03

- C++でももちろん、
 - ファイルからの読み込み(Cで言うところの fscanf())
 - ファイルへの書き出し(同 fprintf())ができます。
- しかも、std::cout と std::cin にそっくりに作ってあるので、
 - std::cout, std::cinの使い方を知っていれば、ほぼ同様に使えます。
 - 書式指定子(%何とか)もやはり必要ありません。良きにはかかってくれます。
 - といっても、さすがに int 型な変数に文字列を読み込むとかはできませんが。
- さらにクローズが自動なので、クローズし忘れの心配がありません。

ファイルへの書き出し SIT applied-progII #03

■ ファイルへの書き出しの手順

- まず #include <fstream>
- std::ofstream 型の変数をひとつ宣言。この変数が、画面で言うところの std::cout に相当する。
- ただし、画面はひとつしか無いのに対して、ファイルはファイル名によりいくつも作れるので、そのファイル名は変数の宣言時に、変数名の後ろに(" ")をつけて以下のように指定する。

```
std::ofstream 変数名("ファイル名");
```

例 `std::ofstream kaki_file("test.txt");`

ここで初めて登場した奇妙な変数宣言 SIT applied-progII #03

- ところで、今まで変数宣言は、ありとあらゆる場合で以下のような形式で宣言していた。
 - 型名 変数名;
 - 例1: int num;
 - 例2: double menseki;
 - 例3: char *ptr;
 - 例4: std::string str;
 - 例5: std::vector vec;
 - 例6: std::map<int, int> tokuten;
- だからこそ説明も理解もしやすかった。

ここで初めて登場した奇妙な変数宣言 SIT applied-progII #03

- ところが、ここで初めてその原則を破る異質な宣言の仕方が登場した。
 - 変数名の後ろに("ファイル名")という部分がくっつく、という形。これは今までなかった。
 - この形は今までの経験になぞらえると、関数呼び出しのように見えるかもしれない。
 - (今日はまだ理解する必要ないが)実際これはコンストラクタという関数を、変数を宣言すると同時にやっている。なので、変数宣言であり、かつ、関数呼び出しであるのだが、コンストラクタやオブジェクト指向の話は来週以降にちゃんとやるので、今はまだ「こう書くものなのだ」と思って下さい。

SIT applied-progII #03

ファイルへの書き出し

- ofstream 型の変数がひとつ宣言できたら、後はその変数に対して、ファイルに書きだしたいものを「<<」演算子で放り込んでいだけ。
 - std::cout と同じですよ。
 - 書き出したいものは、
 - 即値でも (例: 5 や 3.14)
 - 文字列リテラルでも (例: "sakamoto")
 - 変数でも
 - 改行 (std::endl) でも
 - OKです。

17

SIT applied-progII #03

ファイルへの書き出しの例

- test.txt というファイルに書き出す例

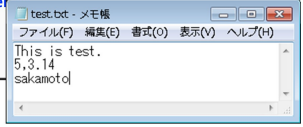
```
#include <fstream>
#include <string>

int main(void)
{
    std::ofstream kaki_file("test.txt");

    kaki_file << "This is test." << std::endl;
    kaki_file << 5 << ", " << 3.14 << std::endl;

    std::string namae = "sakamoto";
    kaki_file << namae << std::endl;

    return 0;
}
```



18

SIT applied-progII #03

ファイルからの読み込み

- ファイルからの読み込みの手順
 - まず #include <fstream>
 - std::ifstream 型の変数をひとつ宣言。この変数が、キーボードで言うところの std::cin に相当する。
 - ただし、キーボードはひとつしか無いのに対して、ファイルはファイル名によりいくつも指定できるので、そのファイル名は変数の宣言時に、変数名の後ろに (" ") をつけて以下のように指定する。


```
std::ifstream 変数名("ファイル名");
```

例 `std::ifstream yomi_file("test.txt");`

19

SIT applied-progII #03

ファイルからの読み込み

- ifstream 型の変数がひとつ宣言できたら、後はその変数から、ファイルから読み出した結果を代入したい変数へと、「>>」演算子で放り込んでいだけ。
 - std::cin と同じですよ。
 - 読み込みたいものは、
 - 即値なら int 型の変数や double 型の変数へ
 - 文字列なら std::string 型の変数へ
 - と読み込んでいく。

20

SIT applied-progII #03

ファイルからの読み込みの例

- test2.txt というファイルから読み込む例

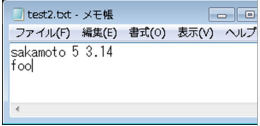
```
#include <fstream>
#include <iostream>
#include <string>

int main(void)
{
    std::ifstream yomi_file("test2.txt");
    std::string namae;
    int num;
    double pi;
    std::string str;

    yomi_file >> namae;
    yomi_file >> num;
    yomi_file >> pi;
    yomi_file >> str;

    std::cout << namae << std::endl;
    std::cout << num << std::endl;
    std::cout << pi << std::endl;
    std::cout << str << std::endl;

    return 0;
}
```



21

SIT applied-progII #03

ファイル読み込みの注意点

- この例では、文字列、整数、実数、文字列という順番で読んでいった。このように、あらかじめファイルのフォーマットが既知でないといけない。
 - もし既知でないなら、全部 std::string で読み込んでおいて、適宜数値に変換する。
- 読み込むものどうしの区切りは
 - 半角スペース1個
 - 半角スペース複数個連続
 - タブ
 - 改行
 - これらの組み合わせの連続並び

22

SIT applied-progII #03

ファイル読み込みまとめ

- 以上の原則さえ守れば、std::ifstream の使い方は非常に簡単。
- Cでファイルから読み込むときの fscanf() とか fgets() みたいに、「%0」は付くんだっけとか「%n」は付くんだっけとかは気にする必要がない。


23

SIT applied-progII #03


ここまでの小まとめ

- ファイルに書き込むには std::ofstream 型の変数に対して << で放り込むだけ。
- ファイルから読み込むには std::ifstream 型の変数から >> で変数に放り込むだけ。


24



ここまでの話をより実用的にするために




25




std::string とかは型です

- std::string や std::vector や std::map<何か, かんとか> は型と思って下さい。
 - int num; というのと std::string str; というのは見た目いっしょでしょ。「::」というたまたま見慣れないものが入っているというだけで。
- だから, Cで型が書けていたような場面で普通に書けます。
 - 関数の返却値の型
 - 関数の仮引数の型




26




std::string とかは型です

- しかも, Call by Value 的に渡せたり返せたりするわけです。
 - 次スライドのコードは, Rubyという言語にある rjust という「文字列を右寄せする」メソッドを, 今までに出てきた知識だけで書いてみた例です。



27



```

#include <string>
#include <iostream>


// 空白を補うことで文字列を右寄せする関数。
// str: 元の文字列
// haba: 全部で何文字の文字列とするか
// 返却値: strが右寄せされた文字列
std::string rjust(const std::string str, int haba)
{
    // habaがstrの長さ以下だったらstrを返却。
    if (str.size() >= haba){
        return(str);
    }

    // 補うべき空白の個数 space_kosuu を計算
    int space_kosuu = haba - str.size();


    // space_kosuu 回空白を連結する
    std::string space_str = "";
    for(int i=0; i<space_kosuu; i++){
        space_str += " ";
    }

    // 空白の並びと str とを連結して返却
    return(space_str + str);
}
    
```

続く



28



```

int main(void)
{
    std::string old_str = "foo";
    std::string new_str;

    new_str = rjust(old_str, 5);

    std::cout << "[" << new_str << "]" << std::endl;


    new_str = rjust(old_str, 7);

    std::cout << "[" << new_str << "]" << std::endl;
}
    
```


実行例

```

[  foo]
[   foo]
    
```



29



Cではできない

- Cではこんなことができなかつたですね。例えば, 以下のコードは正しく動きません(コンパイルは通っちゃいますが)。
 - str はローカル変数なので, そのアドレスを返しても, main() 関数に戻ってきた段階で str 自体が無くなってしまっていて, そのアドレスの中身は表示できないのです。

```


#include <stdio.h>

char *nake(void)
{
    char str[10] = "wan wan!!";


    return(str);
}

int main(void)
{
    printf("[%s]\n", naked());
}
    
```

実行例 [ワソレキ/鶴キ]




30




std::vector や std::map も

- std::string に限らず, std::vector や std::map もやはり関数の仮引数の型や関数の返却値の型になれるので,
 - 呼び出し側から vector などをごっそり渡したり,
 - 逆に関数側から vector などをごっそり返却したり,
 - しかもスコープ(変数の寿命)とかアドレスとかポインタとか気にしなくて良い。




31



富豪的プログラミング

- 「ごっそり」やり取りするのが効率の面で良くないという場面もあることはあります。が, 現在のコンピュータの資源(速さやメモリなど)は非常にリッチです。
- だとしたら効率なんて気にせずに「プログラマにとってのわかりやすさ最優先」で書く, という書き方も有りでしょう。こういうのを「富豪的プログラミング」と読んでいます。
- C++を含む現代的な言語では, こういうことが普通にできるわけです。セレブになりましょう。



32

SIT
applied-progII
#03

ここまでの小まとめ

- C++では文字列やコンテナが普通の型として扱える。そのおかげで、関数の呼び出し側と呼び出され側とのやり取りで細かいことに悩む必要がなくなる。
- 本質部分だけに集中してプログラミングできる。
- ちまちまと効率を気にすることなく、富豪になったつもりで大きい気持ちでプログラミングしよう。

33

SIT
applied-progII
#03

今日の課題

- 課題1: 関数のオーバーロードの応用例を1つ考えて、それを実装してください。
 - オリジナリティが必要です。

34

SIT
applied-progII
#03

今日の課題

- 課題2: 関数のデフォルト引数の応用例を1つ考えて、それを実装してください。
 - オリジナリティが必要です。

35

SIT
applied-progII
#03

今日の課題

- 課題3: 中～大規模なアプリケーションの場合、起動時に「設定ファイル」を読み込んで、その値により動作を変えるということを良くやります。そこで、以下のような設定ファイルを読み込み、設定項目と設定値を、それぞれキーと値として std::vector な変数に格納していくプログラムを書いてください。
 - 要所所が関数化されているとなお望ましいです。

設定項目

```
fullscreen : yes
bgm : yes
level : 3
server : server1.sit.ac.jp
```

設定値

設定項目と設定値との間は「:」があり、「:」の両側にはひとつ以上のスペースがあるとする。

36

SIT
applied-progII
#03

今週の落穂拾い

- 今回「良きに計らってくれる」という言葉が出てきました。つまり、プログラマの
 - 良く覚えていないけど、こんな感じだったかな?
 - std::vector では .size() があるから、std::map でも同じように .size() があるはずだ
 などという期待に答えてくれるわけです。
- Cは、この「良きに計らってくれる」というのが無さすぎてつらいわけですね。逆にC++ではいろんな場面でそうなっています。そもそも std::cout << 38 << "abc" << 3.14 << std::endl; だってそうですよね。(%d とか %f とか使ってませんし、文字列はポインタだから&か*をつけるんだっけつけないんだっけとか気にする必要はどこにもありません。)
- 良きに計らうという性質は、後でやるオブジェクト指向や、はたまたダックタイピングという考え方でも関わってきます。現代的な言語は、プログラマが楽ができるようになっているのです!

37