# C から入る C++

担当: 金丸隆志

第 9 回

## 1　はじめに

今回は「**デストラクタ**」,「**コピーコンストラクタ**」という内容を扱う.

デストラクタは, コンストラクタとは逆にクラスが消滅するとき (記憶寿命を失うとき) に呼びだされる特殊なメンバ関数である. デストラクタが威力を発揮するのはクラスのデータメンバにポインタが存在する場合である. そのため, 今回は myComplex クラスから離れて, myString クラスを扱う [1]. myString クラスはポインタをデータメンバとして持つため, 今回の演習にはポインタの知識が必須である.

今回でクラスを用いたプログラミングを行う準備は終わり, 次回からはより大きなクラスを扱ったプログラミングを学ぶことになる.

## 2　C 言語での文字列の取り扱い

まず, 文字列を扱うクラス myString に触れる前に, C 言語での文字列の取り扱い方法について復習しておこう. ただし, ここで扱うは英数字 (ASCII 文字) のみであるとする.

```
#include <stdio.h>
int main(void){
    char s1[] = "Hello";
    printf("s1=%s\n", s1);

    return 0;
}
```

図 1: C 言語での文字列の取り扱い.

図 1 は, "Hello" という文字列を格納した配列 s1[] を定義し, それを画面に出力するプログラムのリストである. 「char s1[] = "Hello";」という命令で, s1[6] という配列がスタック領域に確保され, 図 2 のように文字列が配列に格納される. なお, '\0' は

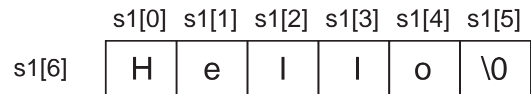


図 2: 配列 s1[6] の内部.

「ヌル (NULL) 文字」と呼ばれ, 文字列の最後に自動的につけられる.

さて, このようにして定義した文字列 s1 を, 別の変数 s2 にコピーすることを考えよう.

```
...
char s1[] = "Hello";
char *s2;
s2 = s1;
...
```

図 3: 文字列のコピー (失敗例)

ここで s2 を char 型のポインタとして定義し, 図 3 のように「s2 = s1;」を実行すると どうなるだろ

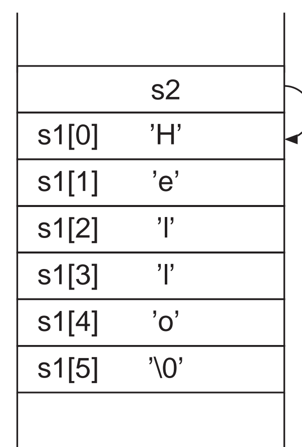


図 4: 「s2 = s1;」の後のメモリのスタック領域の模式図.

うか. この命令自体はエラーなく実行されるが, こ

れは意図した結果をもたらさない．それは図 4 を見ると明らかになる．図 4 は「s2 = s1;」を実行した後のメモリのスタック領域の模式図である．ポインタ s2 の指す先が配列 s1[] の先頭になっているが，”Hello” という文字列自体は一つしかなく，文字列のコピーが行われたわけではないことに注意しよう．

正しく文字列のコピーを行うには，例えば図 5 のリストのようにする必要がある．図 5 のリストのポイントは以下の通り．

```
#include <stdio.h>
#include <string.h> /* for strcpy, etc*/
int main(void){
    char s1[] = "Hello";
    char *s2;

    s2 = new char[ strlen(s1)+1 ];
        /* s2 の領域確保 */
    strcpy(s2,s1); /* 文字列のコピー */

    printf("s2=%s\n", s2);
    delete[] s2; /* メモリの開放 */
    return 0;
}
```

図 5: 文字列のコピー.

- strlen(s1) は文字列 ”Hello” の文字の長さ，すなわち 5 を返す関数である． よって，「new char[ strlen(s1)+1 ] 」によって，char 型の変数 6 個分の領域がヒープ領域に確保される．”+1” はヌル文字 ’\0’ の分である．

- strcpy(s2, s1) は，s1 の内容を s2 にコピーする．ただし，s2 は s1 をコピーできるだけの領域が確保されていなければならない．

- 確保したメモリはいつも通り「delete[] s2;」で開放している．

この時のメモリの模式図は図 6 のようになり，コピーが成功しているのがわかる．ただし，”Hello” が格納されている領域は簡略化して描いている．

以上，C 言語における文字列の取り扱いをみたがメモリの「確保/開放」が頻発することが，文字列の取り扱いをやっかいにしている．

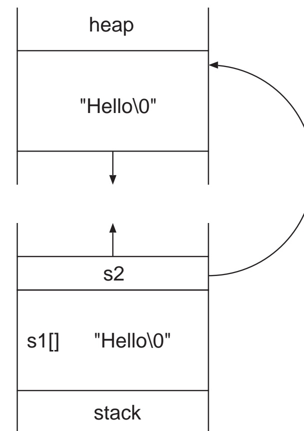


図 6: 図 5 のリスト実行後のメモリの状態の模式図.

以下では，メモリ操作をクラスの内部に隠蔽し，main 関数内ではメモリを意識させないようなクラスを作成することを目標としよう．

## 3 myString クラス〜デストラクタ

まず，myString クラスの第一版のリストを図 7 に載せた．このリストのポイントは以下の通り．

- myString クラスは文字列の長さを表す変数 length，および，文字列へのポインタである *s を持つ．

- 「myString s1;」と宣言された時に呼び出されるデフォルトコンストラクタ「myString::myString(void);」では length を 0 で，s を 0，すなわちヌルポインタ (どこも指さないと保証されているポインタ) で初期化する．なお，myComplex クラスと異なり，コンストラクタを cpp ファイルに記述している．その記述法の違いに注意しよう．

- 「myString s1(”Hello”);」と宣言された時に呼び出されるコンストラクタ「myString::myString(const char* str);」では，length の値設定と s の領域確保，そして s への str のコピーを行っている．

- 「~myString()」は，myString 型の変数が消滅する時に呼ばれる特別なメンバ関数で，「**デストラクタ**」と呼ばれる．典型的には，コンストラクタで確保されたメモリを開放するために用いられることが多い．

- データメンバ s をリターンするメンバ関数 getS(void) を定義している． これは「cout << s1.getS() << endl;」や「printf("%s\n", s1.getS())」などと用いる．

**(a) myString.h**

```
class myString{
    private:
        int length;
        char* s;
    public:
        myString(void);
        myString(const char* str);
        ~myString(){ delete[] s; }
        const char* getS(void){ return s; }
};
```

**(b) myString.cpp**

```
#include <string.h> /* for strlen, etc */
#include "myString.h"

myString::myString(void){
    length=0;
    s = 0;
} /* デフォルトコンストラクタ */
myString::myString(const char* str){
    length = strlen(str);
    s = new char[length+1];
    strcpy(s,str);
} /* コンストラクタ */
```

図 7: myString クラス (第一版)

さて, この myString クラスを利用例の図 8 に示す. この例では, s1 に対してコンストラクタを通して文字列を設定し, その内容を表示するだけである.

では,「myString s2;」と宣言された s2 に対して文字列を設定するにはどうしたらよいだろうか.

例えば, 文字列を設定済みの s1 を用いて「s2 = s1;」なる代入を実行するのはどうであろうか. しかし, これは図 4 と同じ状況に陥ってしまう. 具体得的には, 図 9 のような状況になり, s1 と s2 のデータメンバ s がどちらも同じ領域を指してしまう. すなわち, 文字列のコピーは行われていない.

この問題点を次章で修正し, myString クラスを拡張していこう.

**string_test.cpp**

```
#include <iostream>
#include "myString.h"
using namespace std;

int main(void){
    myString s1("Hello");
    cout << "s1=" << s1.getS() << endl;
    return 0;
}
```

図 8: myString クラス (第一版) の利用.

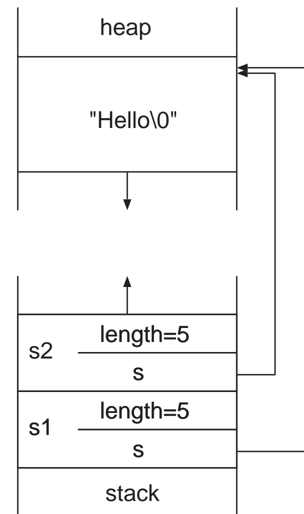


図 9: 「s2 = s1;」を実行した後のメモリの模式図 (コピーの失敗例).

## 4 myString クラスの拡張

### 4.1 代入演算子の多重定義

前章に引続き myString クラスを考えよう. 「myString s1("Hello");」, 「myString s2;」で宣言された s1, s2 に対し, 「s2 = s1;」を実行した時に文字列が正しくコピーされるようにしたい.

「=」演算が実行したときに行われる機能を追加 (変更) したいのであるから, 「= 演算子 (代入演算子)」の多重定義をすればよいと想像がつく.

具体的には図 10 の内容を図 7 のリストに追加すればよい.

ポイントは以下の通りである.

- 「s2 = s1;」が実行される時, 内部的には「s2.operator=(s1);」が実行される.

**(a) myString.h**

```
public:
    ...
    myString& operator=(const myString& x);
    ...
```

**(b) myString.cpp**

```
...
myString& myString::operator=(const myString&
x){ /* 実際は 一行 */
    delete[] s;
    length = x.length;
    s = new char[ length+1 ];
    strcpy(s, x.s);
    return(*this);
}
...
```

図 10: myString クラスにおける = (代入) 演算子の多重定義.

- 「s2 = s1;」が実行される時, s2 のデータメンバ s が指していた領域は一旦開放する必要があるため「delelte[] s;」命令を実行する.
- 文字列のコピーは第一章からおなじみのものである.
- 「return(*this);」は,「s3 = s2 = s1;」なる記法を許すためのものであるが, ここではとりあえず決まり文句と思って構わない.

以上により,「s2 = s1;」命令で正しく文字列がコピーされるようになる [3].

### 4.2 コピーコンストラクタ

さて, myString クラスにもう一つ機能を追加しよう. string_test.cpp で図 11 のように s2 を s1 で初期化する機能をつけたい.

```
...
myString s1(”Hello”);
myString s2 = s1; /* s1 による初期化 */
...
```

図 11: s1 による s2 の初期化.

= があるので, 代入演算子による代入が行われそうであるが, 実はそうではない. C++ では**代入**と**初期化**には厳密な区別があり, 図 11 では**代入**は行われず, **初期化**が行われるのである. そのため図 11 のリストをそのまま実行すると, 図 9 と同じ状況になり, やはり意図した動作ではない.

正しく初期化が行われるには,「myString 型のオブジェクトを受取って初期化を行うコンストラクタ」を定義しなければならない. これを**コピーコンストラクタ**と呼ぶ. 具体的には, 図 12 のように記述する.

**(a) myString.h**

```
public:
    ...
    myString(const myString& x);
    /* コピーコンストラクタ */
    ...
```

**(b) myString.cpp**

```
myString::myString(const myString& x){
    /* 演習として記述してもらいます */
}
```

図 12: コピーコンストラクタ

## 5 まとめ

myString クラスを記述し, さらに「代入演算子の多重定義」および「コピーコンストラクタ」を追加した.

ここまでで何が可能になったかを図 13 にまとめておこう. C 言語において図 13 のように「文字列の初期化・代入」などをしようと思うと, 図 5 のようにポインタ操作が必須である. しかし, C++ ではポインタ操作はクラスの内部に隠蔽されているため, 図 13 のように, ポインタを意識しない記述が可能になる.

つまり, myString クラスを作成しデバッグをしっかりした後は, myString クラスを利用することでポインタを操作する機会が減り, バグを減らすことができる, というわけである.

```
…
myString s1("Hello");
myString s2 = s1;
    /* コピーコンストラクタ呼び出し */
myString s3;
myString s4;

s3 = s1; /* 代入演算子呼び出し */
s4 = "Bonjour"; /* [3] を参照 */
…
```

図 13: myString 型の変数の様々な利用法.

[**課題**]

(1) 図 7 および図 8 のリストを記述し, コンパイルが通り, うまく機能することを確認せよ.

(2) (1) で記述した string_test.cpp に「myString s2;」なる宣言, および「s2 = s1;」という代入を追加し, 実行せよ. これはアプリケーションエラーとなるが, これは何故だろうか.

[**ヒント**]：代入後のメモリの状態は図 9 のようになっている. main 関数が終了するとき, s1 と s2 に対してデストラクタが呼ばれる. この時何が起こるか考えると良い.

(3) (1), (2) のソースを引続き用いる. 図 10 のリストを myString.h, myString.cpp に追加し, 代入演算子の多重定義を行え.

コンパイルおよび実行を行い,「s2 = s1;」という代入が正しく行われることを確認せよ.

(4) 前問までのソースをそのまま用いる. 図 12 のコピーコンストラクタを追加せよ. ただし, コピーコンストラクタ内部は自分で記述せよ.

[**ヒント**]：図 9 の代入演算子の多重定義とほとんど同じで, myString 型の変数 x の内容をデータメンバの length と s にコピーすればよい. ただし,「delete[] x;」と「return(*this);」は不要である.

(5) (**時間が余った人 or 興味のある人向け**)：付録 A の「+ 演算子の多重定義」を追加し,「s3 = s1+s2;」という命令が実行できることを確認せよ.

# A　+ 演算子の多重定義

今回の演習では, クラスのデータメンバにポインタがある例として myString クラスを作成し, 利用することを学んだ.

しかし, 今回作成した機能は「初期化, または代入して画面に表示するだけ」であり, ややもの足りなく感じるかもしれない.

そこで, 本章では myString クラスにおける + 演算子の多重定義を行う. これは,「s1="Hello "」,「s2="World"」であるときに「s3 = s1 + s2;」を実行すると,「s3 = "Hello World"」とすることができる演算である. 図 14 にリストを示した.

このように, myString クラスに機能を増やすことで, 実用的なクラスにすることができるようになる.

**(a) myString.h**

```
…
public:
    …
    myString operator+(const myString& x);
```

**(b) myString.cpp**

```
…
myString myString::operator+(const myString& x){
    int ll;
    char* ss;

    ll = length + x.length;
    ss = new char[ll+1];
    strcpy(ss, s);
    strcat(ss, x.s); /* ss の末尾に x.s を付加 */

    myString SS(ss);
    delete[] ss;

    return(SS);
}
```

図 14: myString 型における + 演算子の多重定義.

## 参考文献

[1] 「myComplex」,「myString」のようにクラス名の名前に ”my” を付けているが, これは特に必要なわけではない. ただ, C++ 標準ライブラリに「string クラス」や「complex クラス」は既に存在しており [2], 混同をさけるために ”myString”, ”myComplex” という名前をつけた, というわけである.

また, 既にライブラリにあるものを何故作るかというと, これらがクラスの学習に適していると考えているからである.

[2] Nicolai M. Josuttis, “C++ 標準ライブラリチュートリアル & リファレンス,” アスキー (2001).

[3] 代入演算子を多重定義したことで, 「s2 = ”Bonjour”;」という記述も可能になる. その理由を順を追って説明しよう. まず, ”Bonjour” を引数とし,「myString::myString(const char∗)」なるコンストラクタが呼ばれ**一時オブジェクト**が作られる (ここでのコンストラクタは char∗ を myString 型に変換する役割をするので, **変換コンストラクタ** とも呼ばれる). そして, 代入演算子 (operator=) が呼び出され, 一時オブジェクトが s2 にコピーされるのである.

なお, このような代入は, 関数が 2 回呼ばれるので実行効率は悪い. 実行効率をあげる方法は C++ の教本 (文献 [4] など) を参考にして欲しい.

[4] 柴田望洋, “C プログラマのための C++ 入門,” ソフトバンク (1992).