C から入る C++

担当: 関根優年、金丸隆志

第 5 回

1 はじめに

過去の講義で学んだ C 言語を活かすために「C 言語的な発想でプログラムを作る」ことを学んで来た.

今回の「メモリの動的管理」で C 言語的な発想でのプログラミングに一区切りつけ, 次回からは C++ 的なオブジェクト指向プログラミングを学んでゆくことにする.

もちろん, 今回の内容も含めて C 言語の知識は必須となるので, 理解しておいて欲しい.

2 メモリの動的管理

2.1 メモリの動的管理 ～どんな時に必要か (1)

前回の「ソートプログラム」での配列の確保の仕方を思いだそう. 図 1 のリストのようにマクロ SIZE

```
#define SIZE 5

int main(void){
   int x[SIZE];
   ...
```

図 1: マクロ定義による, 配列 x[5] の定義

を定義し, 配列のサイズを変えたいときは SIZE の値 (1 行目) を変更してから再コンパイルしていた.

しかし, 配列のサイズを scanf などで取得したいときはどうすればよいだろうか. 素直に考えると, 図 2 のようにしたくなるかもしれないが, これはうまくいかない.

BCC で図 2 のリストをコンパイルしようとすると,「定数式が必要」というエラーが出てコンパイルに失敗する. つまり配列を宣言する際, 配列のサイズはコンパイル時に確定していなければならないのである.

```
int main(void){
   int N;
   scanf("%d",&N);       /* 配列サイズ取得 */
   int x[N];      /* サイズ N の配列の宣言 (?) */
```

図 2: scanf による配列サイズの取得 (?) (うまくいかない).

このように, プログラム中で「好きな時に」「好きなサイズの」配列 (変数) を取得することは今まで学んだ知識だけでは不可能であり, 今回学ぶ「メモリの動的管理」の知識が必要となる.

2.2 メモリの動的管理 ～どんな時に必要か (2)

「メモリの動的管理」の方法を学ぶ前に, もう一つの例を見ておこう.

卒業論文などで, 画像処理のプログラムを書かなければならなくなったとしよう. 画像のサイズは 1024 × 1024 であり, 256 階調の白黒濃淡画像を対象としているとする. 画像のピクセル値を格納する配列を確保するために図 3 のリストを書いたとする.

```
int main(void){
   int x[1024][1024];
    /* 1024 x 1024 の画像に対応する 2 次元配列 */
```

図 3: スタック領域への巨大な配列の確保.

このリストでは巨大な 2 次元配列 x[1024][1024] をスタック領域に確保しようとしている. このプログラムはコンパイルに成功するが, 実行するとプログラムが強制終了してしまう. これは, main 関数の実行時にメモリが確保できなかったからである. 一般にスタック領域に確保できる変数の大きさには制

限があり (BCC では 1MB [1]), スタック領域には巨大な配列は定義できない.

では, 上記の画像処理プログラムを書くにはどうしたら良いだろうか？一つの方法は [1] の URL に書かれているように, グローバル変数を使うことである (図 4 のリスト参照). 次節で学ぶ様にグローバル変数は, メモリ上の「静的領域」という領域に確保されるが, これは論理上使えるメモリの制限が 4 GB なので, 上記の配列を問題なく確保できる [2].

```
int x[1024][1024];
    /* 1024 x 1024 の画像に対応する 2 次元配列 */
int main(void){
    ...
```

図 4: グローバル変数を用いた静的領域への巨大な配列の確保.

しかし第二回にも触れたように, 今回の演習ではなるべくグローバル変数を用いないプログラミング技術を習得して欲しいと考えている. また, 図 4 のリストでも前節の問題, すなわち「配列のサイズがコンパイル時に確定していなければならない」という制限はあてはまる. すなわち, 画像サイズが異なる場合に柔軟に対応できない.

このように,「ローカル変数として巨大な配列を使用したい」際にも「メモリの動的管理」の知識が必須となる.

2.3 メモリ領域の模式図

前の二節で見たように,「メモリの動的管理」は「配列のサイズを柔軟に決定したい時」,「巨大なサイズの配列をグローバル変数を用いずに確保したい時」などに必要になる (もちろん他にもメリットはたくさん考えられるだろうが, ここでは代表的なものだけ挙げた). 実はこのような状況はプログラミングをしていると頻発するものであり,「メモリの動的管理」の知識は C/C++ によるプログラミングをする上で必須と言えるだろう [3].

本節では, C/C++ でプログラミングをする際に使用されるメモリ領域について概説する. C/C++ で用いられるメモリ領域は,「プログラム領域」,「静的領域」,「ヒープ領域」,「スタック領域」に分けられる. これらが実際のメモリにどのように割り当てられるかは処理系によるが, 典型的には図 5 の模式図のようになる.

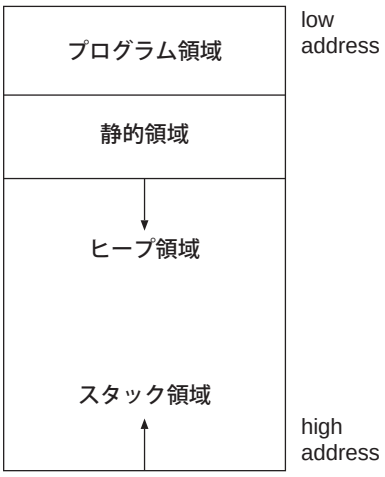

図 5: 4 領域の論理的なメモリ上の模式図

各領域の役割は以下の通り.

- **プログラム領域**: 機械語に翻訳されたプログラムが格納される. この機械語の命令が 1 行づつ実行されることでプログラムが動く.
- **静的領域**: グローバル変数などの静的変数が置かれる.
- **ヒープ領域**: 今回取り扱う「メモリの動的管理」で使用される領域. C++ では「フリーストア」と呼ばれることもある.
- **スタック領域**: 今までみたように, 自動変数 (多くのローカル変数) が置かれる.

2.4 メモリの動的管理 (C++ 版: new/delete)

それでは, 実際にメモリを動的に確保/開放する方法を見てみよう.

図 6 のリストは,「scanf で配列のサイズ N を取得し, サイズ N の int 型の配列 x[N] を確保し, 最後に開放する」というプログラムである.

まず, int 型のポインタ x を定義する (なお, C 言語では「int *x;」と定義するが, C++ では「int* x;」と定義することが多い. これは, x が「int*」という型, すなわち「int 型のポインタ」であることを明確にするためである. ただし, 実際にはどちらの記法も使用できる).

```
#include <stdio.h> /* for scanf */

int main(void){
    int N; /* 配列の要素数 */
    int* x; /* int 型のポインタ x の宣言 */

    scanf("%d", &N);

    x = new int[N]; /* 配列の確保 */

    for(int i=0 ; i<N ; i++){
        x[i] = ··· ; /* 配列の初期化 */
    }
    ··· /* 何らかの操作 */

     delete[] x; /* 配列の開放 */
     return 0;
}
```

図 6: new/delete による配列の確保/開放

そして, scanf で N の値を取得した後,「x = new int[N];」を実行すると, ヒープ領域にサイズ N の int 型の配列が確保され, x にはその先頭のアドレスが代入される. この様子を示したのが図 7 である.

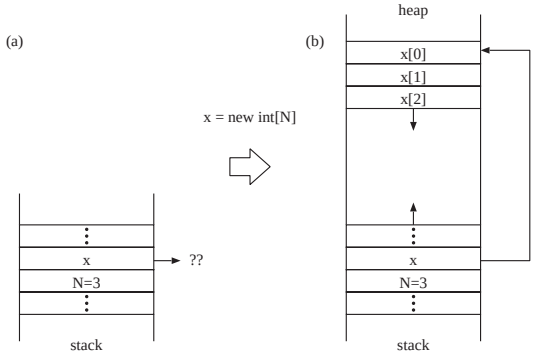

図 7: ヒープ領域に要素 3 の配列が確保される様子. (a) scanf 直後. N には 3 を指定したとする. (b) x=new int[N] 直後. ヒープ領域に x[3] が確保されていることがわかる.

このようにして確保されたメモリは, 今までの同様, 配列 x[i] として使用することができる. なお, この例では int 型だったが, double 型の配列を動的に確保したい場合は double 型のポインタを「double* x;」で定義し,「x = new double[N];」でメモリを確保すればよい. 他の型についても同様である.

また, このようにして確保したメモリは, 使用後に必ず開放しなければならない [4]. それが「delete[] x;」命令である. new と delete は対にして覚えよう.

さて, 今は x の指す先に配列を確保したが, 1 要素だけを確保することも可能である. メモリを開放する命令が配列の場合と異なっていることに注意しよう.

```
int main(void){
    int* x;
    x = new int; /* int を 1 要素分だけ確保 */
    *x = 0; /* x の指す先に 0 を格納 */
    ···
    delete x; /* x の指すメモリ領域を開放 */
    return 0;
}
```

図 8: new/delete による int 型の変数の確保/開放

なお, int 型を 1 要素だけ確保するのはあまり実用性がないかもしれないが, クラスの要素 (インスタンスという) をヒープ領域に 1 つ確保する場合は図 8 の様な記述を多く行うことになるだろう.

なお,「C 言語での動的メモリ管理」および「2 次元配列の動的管理」の方法を付録に記載した.

[**レポート 課題 1**]
前回の課題 1 で作成した「ソートプログラム」を, scanf で配列サイズを決定するように変更せよ. マクロ定義の行 (#define SIZE 5) を削除し, SIZE を int 型の変数として main 関数内で定義すると変更点が少なくて済む. (今まで触れなかったが, 変数は 1 文字である必要はなく, "SIZE" のような変数名でもなんら問題はない)

[**注意**]
前回「ソートプログラム」を完成させられなかった人は,「C 標準ライブラリ版」のクイックソートプログラムに対して, 上と同様のことを行うこと. なお,「C++ 標準ライブラリ版」の場合は「配列のサイズ」という概念がないので, この課題には適していない.

3 C++ 標準ライブラリによる入出力

今まで, C 言語の標準ライブラリに含まれる scanf/printf を用いた入出力のみを扱って来た. その理由は,「プログラミング I」で得た知識を活用して欲しかったからである.

実は C++ には C++ 固有の入出力方法があり, C++ の教科書にはその方法で記述されていることが多いので, ここで紹介することにする.

なお, 今まで見て来たように C++ では scanf/printf および, ここで扱う方法の両方を扱うことができる. どちらを用いるかはプログラマの好みによるが, せっかく C++ を学んでいるのだから, ここで新しい知識を得ておくのも悪くないであろう.

3.1 C++ の入出力 (新しい記法)

実は C++ の入出力 (正確には標準ライブラリ) には「古い記法」と「新しい記法」がある [5]. 教科書には古い記法が書かれていることが多いが, これから C++ を学ぶ人は新しい記法に馴染んだ方が良いと思われるので個人的にはこちらを推奨する.

新しい記法には簡略化した書き方とそうでない書き方があり, 図 9 のリストにその両方を挙げた. "using namespace std;" を書くことにより, "std::cin" 等を簡略化して書くことができることに注意しよう (この命令は C++ 標準ライブラリのクイックソートを用いたときも用いた).

プログラムの内容は, ユーザに N の値の入力を促し, その結果を表示するだけのプログラムである.

イメージとしては,「cin (入力) から変数 N へ値を流し込み, cout (出力) へ文字列や N の値を流し込む」と考えれば分かりやすいだろう. なお, cin, cout は C++ 的に言えばそれぞれ「標準入力」,「標準出力」をあらわすオブジェクトである.

なお "endl" を cout へ流し込むことにより改行が行われ, さらに cout のバッファがフラッシュされる. 通常は "endl" を改行の意味で理解してよいが, 大量のデータを出力する場合などは,「<< endl;」ではなく,「<< "\n";」で改行を行わせた方が, フラッシュが行われないためパフォーマンスが良いことに注意しておく [7].

(a) **簡略記法**

```
#include <iostream>
using namespace std;

int main(void){
    int N;

    cout << "N の値を入力してください ";
    cin >> N;
    cout << "N の値は" << N << "です" <<
endl; /* 実際は 1 行 */

    return 0;
}
```

(b) **非簡略記法**

```
#include <iostream>

int main(void){
    int N;

    std::cout << "N の値を入力してください ";
    std::cin >> N;
    std::cout << "N の値は" << N << "です"
<< std::endl; /* 実際は 1 行 */

    return 0;
}
```

図 9: C++ の入出力 (新しい記法).

3.2 C++ の入出力 (古い記法)

なお, 混乱を避けるため, 教科書に載っていることが多い古い記法についても触れておく.

図 9 の 1 行目を "#include <iostream.h>" のように拡張子をつけて記述し, 2 行目の "using namespace std;" 命令を削除すれば古い記法になる.

なお, 大抵の C++ コンパイラでは過去のバージョンと互換性が保たれているので, この記法でもコンパイルできる.

[**レポート 課題 2**]
レポート課題 1 のソートプログラムは scanf/printf を用いて作成されていると思うが, それを cin/cout を用いるように変更せよ.

この場合, <stdio.h> をインクルードする必要はもはやない. ただし. <stdlib.h> は乱数生成などに用いているので必要である.

A メモリの動的管理 (C 版: malloc/free)

new/delete は C++ にのみ存在する演算子であるので, C 言語では使えない. C 言語ではその代わりに malloc/free 関数を用いる. 具体例は図 10 のリストの通り.

```
#include <stdio.h> /* for scanf */
#include <stdlib.h> /* for malloc/free */

int main(void){
    int i; /* for 文用 */
    int N;
    int *x;

    scanf("%d", &N);

    x = (int *)malloc(sizeof(int)*N);

    for(i=0 ; i<N ; i++){
        x[i] = ···;
    }
    ···
    free(x);
    return 0;
}
```

図 10: malloc/free による配列の確保/開放

new/delete のときの様に, malloc と free を対にして使用することを心がけよう.

なお, int 型でなく double 型のときは, 「double *x;」で宣言し, 「x = (double *)malloc(sizeof(double)*N);」で確保する. 他の型も同様である.

B メモリの動的管理〜2 次元配列

第 2.2 節で画像処理用の 2 次元配列の確保の話をしたので, 2 次元配列の動的確保についても触れておこう.

なお, ポインタのポインタがでてくるので, 最初は理解が難しいかもしれない.

B.1 メモリの動的管理〜2 次元配列 (C++ 版: new/delete)

new/delete を用いた 2 次元配列の動的確保の方法は以下の通り. まず始めにヒープ領域に M 個のポインタを確保し, その各ポインタに対して N 個分の領域を確保している. それによって x[M][N] の 2 次元配列がヒープ領域に確保できる.

```
int main(void){
    int M, N; /* 配列 x[M][N] の確保用 */
    int** x; /* ポインタへのポインタ */

    ··· /* M, N を何らかの方法で決定 */

    x = new int*[M];
    for(int i=0 ; i<M ; i++){
        x[i] = new int[N];
    } /* 以上で配列 x[M][N] が使える */

    ··· /* x[M][N] を用いた何らかの処理 */

    for(int i=0 ; i<M ; i++){
        delete[] x[i];
    }
    delete[] x;
     /* 以上で配列 x[M][N] が開放される */
    return 0;
}
```

図 11: new/delete による 2 次元配列の確保/開放

興味のある人は, この時のメモリの使用状況の模式図を書いてみるとよいだろう.

B.2 メモリの動的管理〜2 次元配列 (C 版: malloc/free)

図 12 のリストに malloc/free を用いた 2 次元配列の動的確保の方法を示した.

```
#include <stdlib.h> /* for malloc/free */

int main(void){
    int i; /* for 文用 */
    int M, N; /* 配列 x[M][N] の確保用 */
    int **x; /* ポインタへのポインタ */

    ···/* M, N を何らかの方法で決定 */

    x = (int **)malloc(sizeof(int *)*M);
    for(i=0 ; i<M ; i++){
        x[i] = (int *)malloc(sizeof(int)*N);
    } /* 以上で配列 x[M][N] が使える */

    ···/* x[M][N] を用いた何らかの処理 */

    for(i=0 ; i<M ; i++){
        free(x[i]);
    }
    free(x);
    /* 以上で配列 x[M][N] が開放される */
    return 0;
}
```

図 12: malloc/free による 2 次元配列の確保/開放

B.3 2 次元配列は必要か?

以上, new/delete (C++) および malloc/free (C) を用いた 2 次元配列の動的確保をみたが, やや面倒であることがわかるであろう.

実は画像処理のプログラムを書く場合であっても, 必ずしも 2 次元配列 x[M][N] を用いる必要はない. 例えば 1 次元配列 x[M*N] を確保しておき, (i,j) の要素 ($0 \leq i < M$, $0 \leq j < N$) に対し x[N*i + j] にアクセスするようにすれば, 2 次元配列を用いた場合と同様に作業ができる.

どちらを用いるかは, その問題や, プログラマの好みによるだろう.

参考文献

[1] BCC におけるスタック領域の大きさは
http://www.borland.co.jp/qanda/lang/l0002734.html (最初の文字は「エル」)
にあるように 1 MB である. 一般にスタック領域の大きさはこのように小さいと考えてよい.

[2] ただし, OS や使用環境によっては静的領域にも制限がある場合も考えられる.

[3] メモリの動的確保はバグの温床だから使用すべきでないという, (挑発的な) 意見もある. 例えば
ポインタ不要論： http://www.tomozo.ne.jp/yamazaki/download/doc_without_pointer.htm
しかし, それを実践するには高度な知識が必要であるし, 現状では (良かれ悪しかれ) ポインタやメモリの動的管理の知識は必須であると思われる.

[4] new によって確保されたメモリ領域は, プログラム終了時に OS によって自動的に開放されることになっている. だから「new で確保したメモリを明示的に delete する必要はないのではないか」という議論も存在する. しかし, 実際に delete しない強気なプログラマはほとんどいないだろう.
また, 24 時間稼働し続けるようなプログラムを書く場合, メモリの開放し忘れが重大なバグを招くことがある. このようなバグは発見が難しいので, 「new したら delete する」ことを習慣づけるのが良いだろう.

[5] C++ の仕様の変更に伴い, C++ の標準ライブラリの使用法が変更されている. 新しい記述法は文献 [6] などで見ることができる.
雑誌では, CMagazine の 2002 年 4 月号から始まった「C++ 基礎講座」はこの新しい記法にのっとった連載になっている.
Web 上では STL や iostream に関する詳細な記述が例えば以下で見ることができる.
http://www.kab-studio.com/Programing/Codian/

[6] 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).

[7] endl の働きについては
http://www.bohyoh.com/CandCPP/FAQ/FAQ00009.html を参照.