C から入る C++

担当: 金丸隆志

第 3 回

1 C 言語復習 (主に関数について)

1.1 main 関数の形式

過去の講義で学んだように, C 言語ではプログラム本体を main 関数の中に記述する. main 関数の形式は以下の通りである.

```
int main(void){
    ...
    return 0;
}         /* 引数を取らない場合 */

int main(int argc, char *argv[]){
    ...
    return 0;
}         /* 引数を取る場合 */
```

図 1: main 関数の形式

実は main 関数には上に示した以外にも様々な形式が許されている (例えば, main() や void main(void) など) [1, 2].

しかし, 厳密な C の文法によると上記のように main 関数は int 型と定められている [3] ので, 本演習でもそれに従う.

なお, main 関数の最後に 「return 0;」と値が返されているが, これは OS に, 「プログラムが無事終了した」ことを知らせるための命令である. ただし, 本演習ではこの値を利用することは (恐らく) ないので決まり文句だと思って構わない.

また, Windows の GUI プログラミングでは main 関数の代わりに WinMain 関数を用いるが, それは本演習の範囲外である.

1.2 プログラムの大まかな形式

ラフに言えば, C 言語によるプログラムは以下のような形式をしているであろう. ただし, 簡単のため, 構造体やクラス, ソースファイルの分割等はこの例では考えていない.

```
#include <stdio.h>
    /* ライブラリのヘッダの読み込み */
...
    /* マクロ定義 (本演習では多分扱わない)*/
int x;
    /* グローバル変数定義 */
void func1(void);
int func2(double a, double b);
    /* 関数のプロトタイプ (宣言) */

int main(void){
    ... /* main 関数 */
}

void func1(void){
    ... /* func1 関数の定義 */
}
int func2(double a, double b){
    ... /* func2 関数の定義 */
}
```

図 2: C 言語によるプログラムの典型的な例

「能力確認テスト」の 3 番はちょうどこの様な形式になっていたことに注意しよう.

これを見るとわかるように, C 言語によるプログラミングにおいては,

- main 関数からいかに機能を切り出して関数化するか
- 切り出した関数の引数, 戻り値などをいかに決

定するか

などがポイント (の一つ) になることがわかるであろう.

しかし, これらは教科書を読めば身につく, というようなものではなく, ある程度の訓練が必要である. 自分でプログラムを書く時は「何か関数として切り分けるべき機能はないか」ということを常に意識して書くことにしよう. (さらに, C++ を学ぶ場合「何をクラスとして実現し, どう実装すべきか」という分析/設計も行わねばならない)

なお, main 関数から関数を切り出すべき理由として以下が考えられる.

- 一つの関数 (main 関数も含む) の長さを短くして, プログラムの可読性を高める. (何百行もある関数を読むのは誰でもつらい)
- 何度も呼び出される機能を関数化しておけば, プログラムの再利用性が高まる

1.3 変数のスコープ (グローバル変数とローカル変数)

図 3 のプログラムリストには, x という int 型の変数が 3 つ定義されている. 2 行目の $x = 5$ は**グローバル変数**であり, プログラムのすべての領域で使用できる. 一方, main 関数と scope_test1 関数でもそれぞれ x が定義されているが, これらは関数内でのみ使用できる**ローカル変数**である.

図には各 x の有効範囲 (スコープ) も示されている. 実行結果を見ると, main 関数と scope_test1 関数では各関数内で定義されたローカル変数 x が使われ, scope_test2 関数ではグローバル変数の x が使われているのがわかる.

このように, 同じ x という名前の変数であっても, 通用範囲を外れると全く異なる変数となることに注意しよう.

特に, 2 つの関数内 (ここでは main と scope_test1) でそれぞれ定義されたローカル変数が全く別物であることは注意が必要である.

例えば, 能力確認テストの 3 番において, int 型の変数 x, y がともに main 関数および larger 関数の両方で定義されていた (larger 関数では仮引数として定義されている) が, これも main 関数と larger 関数で実体は別物である.

```
#include <stdio.h>

int x=5;

void scope_test1(void);
void scope_test2(void);

int main(void){
  int x=0;

  printf("in main  x=%d\n",x);

  scope_test1();
  scope_test2();

  return 0;
}

void scope_test1(void){
  int x=1;

  printf("in scope_test1 x=%d\n",x);
}

void scope_test2(void){

  printf("in scope_test2 x=%d\n",x);

}
```

```
result:
    in main  x=0
    in scope_test1 x=1
    in scope_test2 x=5
```

図 3: グローバル変数とローカル変数

そのため, 関数を利用するときは, **引数**と**戻り値**をうまく用いてデータをやりとりしなければならない.

一方, グローバル変数を用いれば複数の関数でデータを共用できる. しかし, グローバル変数はどの関数からも参照できることから, データの管理が難しくなる, バグがの発見が難しくなるなどの傾向があり, 濫用は避けるべきであると一般に言われている [4]. 本演習でもグローバル変数の利用はなるべく避けることにする.

また, この例ではグローバル変数とローカル変数で同じ変数 x が定義されていたが, 現実のプログラ

ムでは混乱の元となるのでもちろんこれも避けるべきである.

1.4 関数の引数と戻り値

前節で触れたように, 関数を呼び出す時にデータを受け渡して共有するには関数の引数と戻り値を利用しなければならない. 関数の形式は以下のようになる.

```
[戻り値の型] [関数名]([引数 1], [引数 2], ···){
···
}
```

引数や戻り値の型は void/char/int/float/double などだけでなく, ポインタや構造体, クラスもとれる. なお, void は引数や戻り値を使用しない時に指定する.

戻り値がある場合, return 文を用いて値を返さねばならない. 以下は, int 型の数値の 2 乗を得る関数である [5].

```
int square(int x){
    int tmp;
    tmp = x*x;

    return(tmp);
}
```

この square 関数は y=square(x) のように利用できる. もちろん y には x^2 の値が格納される.

なお, 引数は複数の値をとれるのに対し, 戻り値には一つの型の値しか戻せないことに注意する.

[**レポート 課題 1**]
「能力確認テスト」の 1 番で,「1,2,··· n の整数の和を求める」ルーチンを示し, 前回のレポートでこのルーチンを main 関数に組み込み, printf 文で値を表示してもらった.

この「1,2,··· n の整数の和を求める」ルーチンを関数化し, それを main 関数から呼び出して値を printf 文で表示せよ.

なお, 作成する関数名は sum とし, main 関数から s = sum(n) という形式で呼び出されるように記述すること [6].

2 値渡し, ポインタ引数, 参照引数

本章では C 言語において関数へ引数を渡す方法を詳しく学ぶ.

基本的には, C 言語においては全て「**値渡し**」によって引数が渡される. しかし, ポインタ引数, 参照引数を用いることで, あたかも関数に変数の実体を渡しているかのように見せることができ, C/C++ では常套手段になっている [8].

本章全体で,「int 型の変数 x の値を一つ増加 (インクリメント) させる関数」を取り扱う. これは「ある関数のローカル変数の値を別の関数から変更する」ことがポイントになっており,「能力確認テスト」の 2 番 (scanf の問題) に関係する.

なお, 本章にはポインタの知識が必要である.

2.1 値渡し

「int 型の変数 x の値を一つ増加 (インクリメント) させる関数」を素直に考えると, 図 4 のリストのような関数が考えられる.

```
#include <stdio.h>
void increment(int x);

int main(void){
    int x=0;
    increment(x);
    printf("x=%d\n",x);

    return 0;
}

void increment(int x){
    x = x+1;
}
```

図 4: インクリメント関数 (失敗例)

しかし, このときコンソールに表示されるのは「x=0」であり, インクリメントはうまくいかない.

これは,「引数を通して関数へ渡されるのが変数の値のみだけであり, 変数そのものが渡されているわけではないからである (**値渡し**)」と説明される. これをもう少し詳しく見てみよう.

一般に, C 言語では関数が起動されたとき, ローカル変数のうちの自動変数 (他には静的変数, レジス

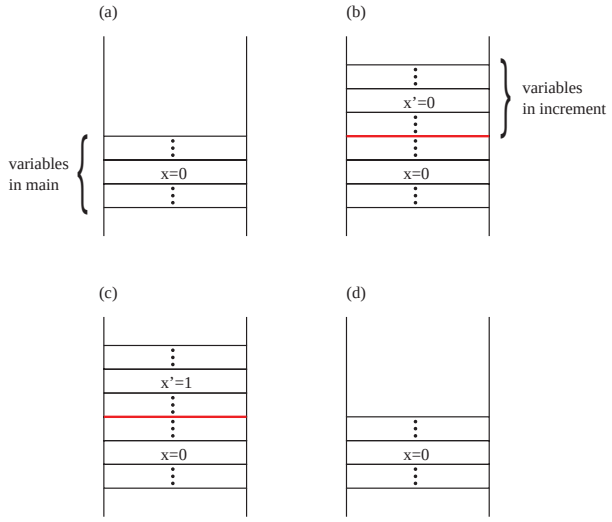

図 5: 値渡しの際のスタック領域の様子. (a) increment 呼び出し前. (b) increment 呼び出し直後. (c) increment 実行中. (c) increment 終了後.

タ変数, 外部変数があるが, いままで出て来たローカル変数は全て自動変数) はメモリ上の**スタック領域** に確保される [7]. そして関数での処理が終了したら, スタック領域は開放される.

これを上の increment 関数呼び出しに関して図示したのが図 5 である.

ポイントになるのは, 変数 x が increment 関数に渡される時, increment 関数で用いられるのは **x のコピー** (図では x' と書いた) である, という点である. そのため, increment 関数が終了後, main 関数の変数 x は変更を受けていない.

2.2 ポインタ引数

実際にインクリメント関数をうまく機能させるためには, 次のリストのようにポインタを引数として渡さなければならない.

なお, increment_p 関数では p はポインタとして定義されており, p の指す値は *p であることに注意しよう. 実行するとコンソールには「x=1」と表示され, この関数がうまく機能していることがわかる. この時のスタックの様子は図 7 の通り.

ところで, プログラミング 1 の後半において,「配列の先頭のアドレスを関数に渡し, 関数内部で配列操作を行う」ことを多用していた. これがなぜ可能だったのかを付録に示しておく.

```
#include <stdio.h>
void increment_p(int *p);

int main(void){
    int x=0;
    increment_p(&x);
    printf("x=%d\n",x);

    return 0;
}

void increment_p(int *p){
    *p = *p+1;
}
```

図 6: インクリメント関数 (ポインタ引数)

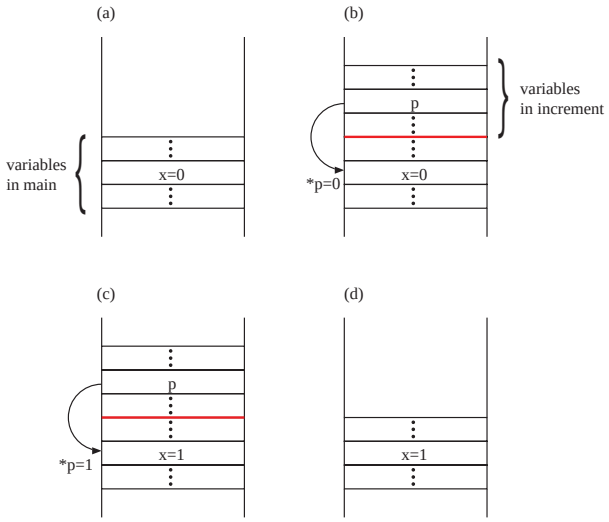

図 7: ポインタ引数を用いる際のスタック領域の様子. (a) increment_p 呼び出し前. (b) increment_p 呼び出し直後. (c) increment_p 実行中. (c) increment_p 終了後.

2.3 参照引数 (C++ 限定)

前節でみたように, ポインタを引数として与えれば, 関数内で別の関数の変数を書換えることができた. しかし, increment_p 内ではポインタが指している変数の値を *p と書かねばならず, それがやや煩わしいと感じる人もいるかもしれない.

そのような人のために, C++ では**参照**によって引数を渡すという手法が用意されている. 使用例は図 8 のリストの通り.

```
#include <stdio.h>
void increment_r(int &r);

int main(void){
    int x=0;
    increment_r(x);
    printf("x=%d\n",x);

    return 0;
}

void increment_r(int &r){
    r = r+1;
}
```

図 8: インクリメント関数 (参照引数)

関数の引数の定義に & がついているが, これが「関数の引数が参照である」ことを意味する. 関数内では値渡しの感覚で r=r+1 などと変数の操作ができるが, 効果はポインタ引数を用いた時と同じであり, 実行結果は「x=1」となる.

[**レポート 課題 2**]
int 型の変数 x, y がある時, 2 つの数を交換する関数 swap を作りたい. たとえば x=0, y=5 の時に swap 関数を実行すると, x=5, y=0 になる, というものである. 以下を実行せよ.

1. 値渡しによる swap 関数 「void swap_v(int x, int y)」を作成し, main 関数から呼び出して, それがうまく機能しないことを確認せよ. (ヒント： swap 関数内には, 値を一時的に保存するための変数が一つ必要である)

2. ポインタ引数による swap 関数 「void swap_p(int *x, int *y)」を作成し, main 関数から呼び出してみよ.

3. 参照引数による swap 関数 「void swap_r(int &x, int &y)」を作成し, main 関数から呼び出してみよ.

なお, 参照は C++ で導入された機構であるので, (BCC を用いる場合) ソースファイルの拡張子を cpp にしないと利用できないことに注意.

A 関数に配列へのポインタを渡す

図 9 のリストのような, 関数に配列の先頭へのポインタを渡して, 関数の中身で配列の値を変更するプログラムを考える.

これはプログラミング 1 の第 10 回目以降にしばしば登場している. これがうまく機能する理由を図

```
#include <stdio.h>
void func(int n, int *a);

int main(void){
    int a[3]={5, 3, 10};
    int n=3;
    func(n,a);

    return 0;
}

void func(int n, int *a){
   int i;
   for(i=0 ; i<n ; i++){
       a[i] = ···;
      /* a[i] に何らかの処理を施す */
   }
}
```

図 9: 関数に配列のポインタを渡して配列の値を変更するプログラム

10 のスタック領域の模式図で説明する.

まず, 図 10 (a) は func 関数呼び出し前のスタック領域の様子である. 変数 n と配列 a[3] がメモリ上に確保されていることがわかる.

一方, 図 10 (b) は func 関数呼び出し後のスタックの様子である. 変数 n のコピー n' と, 配列の先頭へのポインタ a のコピー a' が確保されていることがわかる. この func 関数から a'[i] を変更することは, 図に示したように main 関数上の配列を変更することを意味する.

func 関数の処理が終了すると, スタックから func 関数の自動変数が開放されて処理が main 関数に戻る.

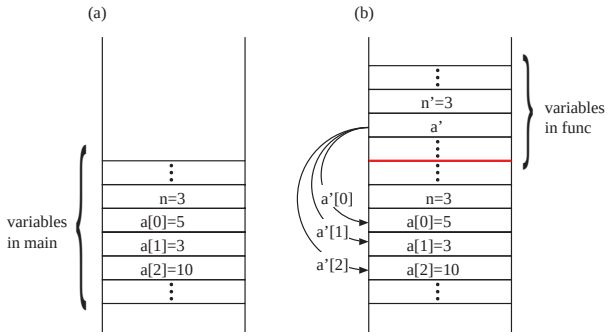

図 10: 関数に配列のポインタを渡す際のスタック領域の様子. (a) func 関数呼び出し前. (b) func 関数呼び出し中.

参考文献

[1] Steve Oualline, “C++ 実践プログラミング,” オライリー・ジャパン (1996).

[2] B.W. カーニハン/D.M リッチー, “プログラミング言語 C (第 2 版),” 共立出版株式会社 (1989).

[3] 例えば http://www.bohyoh.com/CandCPP/FAQ/FAQ00014.html など.

[4] 大抵の C 言語の入門書で, グローバル変数の濫用は控えるべきであるとされている. Web 上では例えば http://www.cmagazine.jp/src/kinjite/c/variable.html#index10

[5] 例では分かりやすくするために tmp という int 型の整数を用いたが, 実際には tmp を使わずに return(x∗x) でいきなり値を返しても構わない.

[6] $1 + 2 + \cdots + n = n(n+1)/2$ という公式があるので, この和を for 文で求めるのは実は現実的ではないが, プログラムの学習が目的なので, ここでも for 文を使ってプログラミングしてください.

[7] 自動変数がスタック領域に確保されることは C 言語の仕様で規定されているわけではないが, ほとんどの C 言語の実装ではスタック領域に確保されるようになっている.

また, スタック領域には自動変数以外にもいくつかの情報が格納されている. マイクロプロセッサの授業をとった人ならば, 関数呼び出し位置まで処理を戻すためのリターンアドレス $ra がスタックに格納されていたことについて覚えているかも知れない.

[8] (以下は, 混乱を招くかも知れないので, ある程度知識のある人向け)
関数に引数を渡す方法として代表的なものに, 「値渡し (call by value)」と「参照渡し (call by reference)」がある.

値渡しは資料で扱ったように変数の値のみを渡す手法であり, 参照渡しは変数の番地を渡す手法である. 値渡しを用いる代表的な言語は C 言語であり, 参照渡しを用いる代表的な言語に Fortran がある. (Pascal はどちらもサポートしている)

厳密には C 言語では全て値渡しで引数が渡される. 図 7 ではポインタの値 (すなわちアドレス) が渡されていると考えれば確かに値渡しである.

本資料で扱った「ポインタ引数」「参照引数」は値渡しの言語である C 言語で参照渡しを実現するための手法である, と理解すれば良い.
参考：
http://www.catnet.ne.jp/kouno/c_faq/c4.html の 4.11.

(似たような議論に「Java 言語は参照渡しか?」というものがある. 教科書などには「Java は参照渡しである」と書かれることもあるが,「Java は参照の値が渡されるので値渡しだ」というのが正解のようである.)