応用プログラミング

第4回　デバッグ、関数定義、変数の分類

メディア情報コース
平居　悠（ひらい　ゆたか）

到達目標

- C言語を通じてコンピュータの根幹部分を理解する。
- 目的に応じたプログラムを自由に設計できる。

前回

第１回　C言語の基本文法・言語処理系
第２回　数学的計算、配列、文字列
第３回　型変換、入出力処理
第４回　デバッグ、関数定義、変数の分離
第５回　プロトタイプ宣言、分割コンパイル
第６回　配列の受け渡し、ポインタ
第７回　まとめ、期末試験

前回の目標

C言語でデータ型変換、ファイル入出力処理ができるようになる。

前回学んだこと

1. 型変換
    1. データ型
    2. キャスト
2. 入出力処理
    1. ファイル入出力
    2. バイナリファイル

３つのデータ型

- 整数型
- 浮動小数点数型
- 文字型

整数型

- short型 (16ビット)
- int型 (32ビット)
- long型 (32ビット)

サイズ（コンパイラにより異なる）：

short ≤ int ≤ long

浮動小数点数型

- float型 (32ビット)
- double型 (64ビット)
- long double型 (64ビット)

サイズ（コンパイラにより異なる）：

float ≤ double ≤ long double

文字型

char型

(コンパイラによらず**1**バイト)

int型同士の割り算

- int型同士の割り算はint型 (整数)になる。
- 小数点以下は切り捨てられる。

キャスト演算子

式の型を変更する。

(型名X)式

ファイル入出力

- ファイルは**fopen**関数で開き、**fclose**関数で閉じる。
- ファイルへの出力は**fprintf**関数を用いる。

putc関数

putc(c, stdout)

ファイルに文字を出力する。標準出力（画面）に出力する場合は**stdout**を指定する。

fscanf関数

fscanf(fp, “%s”, text)

最初の引数はファイルポインタを指定。ファイルの終端まで来たらEOFを返す。

fgetc関数

fgetc(fp)

ファイルから1文字ずつ読み込み、int型で返す。ファイルの終端まで来たらEOFを返す。

バイナリファイルとは

コンピュータ内部で使用されるデータ形式をそのまま書き込んであるファイル。データサイズを小さくでき、処理速度も速い。

バイナリファイル入出力

1. fopen関数でファイルを開く。読み込みモードは“**rb**”、書き込みモードは“**wb**”。
2. 書き込みには**fwrite**関数、読み込みには**fread**関数を使う。
3. 読み書きが終了したらfclose関数でファイルを閉じる。

前回の問い

- 宣言したデータ型と異なるデータ型で計算したい場合はどうすれば良いか？
  - キャスト演算子でデータ型を変更する。
- C言語でファイル入出力はどう処理するのか？
  - ファイルを開くにはfopen関数、ファイルを閉じるにはfclose関数を用いる。

今回

第１回　C言語の基本文法・言語処理系
第２回　数学的計算、配列、文字列
第３回　型変換、入出力処理
第４回　デバッグ、関数定義、変数の分離
第５回　プロトタイプ宣言、分割コンパイル
第６回　配列の受け渡し、ポインタ
第７回　まとめ、期末試験

今回の目標

- プログラムをデバッグして正しく動くプログラムに修正できる。
- 関数を定義して自由にプログラムを設計できるようになる。

今回の問い

- プログラムをデバッグはどのように行うのか？
- C言語で関数はどのように定義すれば良いのか？

今回学ぶこと

1. デバッグ
2. 関数定義
3. 変数の分類

今回学ぶこと

1. デバッグ
2. 関数定義
3. 変数の分類

バグ

プログラムに含まれる論理的誤り

問題

右は平均点を求めるプログラムであるが、正しく動かない。どこにバグがあるか？

```
#include <stdio.h>

int main()
{
   int score[3] = {90, 60, 50};
   int i;
   int sum;

   for(i = 0; i < 4; i++){
      sum += score[i];
   }

   printf("平均点は%d点です。\n", sum/3);
   return 0;
}
```

デバッグ

プログラムのバグを取り除く作業

デバッグの方法

printfデバッグ

printfを要所要所で挟んでその時の変数状態を表示してデバッグする方法。

printfデバッグ

```
#include <stdio.h>

int main()
{
   int score[3] = {90, 60, 50};
   int i;
   int sum;

   for(i = 0; i < 4; i++){
      sum += score[i];
      printf(“sum = %d\n”, sum);
   }

   printf(“平均点は%d点です。\n”, sum/3);
   return 0;
}
```

実際の例

ASURA+BRIDGEコード

https://zenodo.org/records/11180637

```
    double MassInMsun = Ps->InitialMass*Pall.UnitMass/MSUN_CGS;
    double prob = SNIINumber*MassInMsun;
    if(prob >= 1.0){
        Ps->TypeIIProb = true;
    } else {
        double p = gsl_rng_uniform(RandomGenerator);
        fprintf(stderr,"%g %g\n",p,prob);
        if(prob > p){
            Ps->TypeIIProb = true;
        }else{
            Ps->TypeIIProb = false;
        }
    }
```

ソースレベルデバッガ

プログラムのソースとの対応を取りながら、プログラムを少しずつ動かしたり止めたりしつつデバッグを支援するツール。

GDB (GNU Debugger)

GNUプロジェクトで開発されたデバッガ。多くのUnix系システムで利用できる。

gdbによるデバッグ手順

1. コンパイル時に-gオプションを付ける。

```
gcc average01.c –g –o average01
```

gdbによるデバッグ手順

2. gdbの起動。gdbコマンドの後に実行プログラムを指定する。

```
gdb average01
```

gdbによるデバッグ手順

3. プログラムリストを見て行番号を調べる。
listコマンドを打つとプログラムが10行ずつ表示される。l(エル)のみでも可。

```
(gdb) list
```

list 3のように打つと、表示し始める行番号も指定できる。

gdbによるデバッグ手順

4. ブレークポイントを設定する。breakコマンドでプログラムの動作を止めたい行番号を指定する。例えば7行目で止めたい場合は以下のようにする。

(gdb) **break 7**
(または)
(gdb) **b 7**

gdbによるデバッグ手順

ブレークポイントには、関数名を指定しても良い。例えばmain関数の実行を最初から止めたい場合は以下のようにする。

(gdb) **break main**

gdbによるデバッグ手順

5. プログラムを走らせる。runコマンドでデバッグ実行する。

(gdb) **run**
(または)
(gdb) **r**

gdbによるデバッグ手順

プログラムの起動に引数を与えたい場合は
runコマンドに引数を与える。例えば、

```
./foo < data.txt
```

とするプログラムをgdbで起動するには

```
(gdb) run < data.txt
```

のようにする。

gdbによるデバッグ手順

6. 設定したブレークポイントの行でプログラムが止まるのでそのあと「ステップ実行」、「変数の値表示」などで調査する。

ステップ実行

(gdb) **s** (または **step**)

変数の値表示

(gdb) **p 変数** (または **print 変数**)

gdbによるデバッグ手順

7. gdbを終了する。

(gdb) **q** (または **quit**)
The program is running. Exit anyway? (y or n) **y**

countdown.cという1から0.1刻みでカントダウンし、0になったら終了する右のプログラムを作ってみよう。

```
#include <stdio.h>

int main()
{
   float count = 1.0;

   while (count != 0) {
      printf(“%fです。\n”, count);
      count -= 0.1;
   }

   printf(“おしまい\n”);
   return 0;
}
```

コンパイルと実行

```
gcc countdown.c –g –o countdown
```

このプログラムは実行すると暴走するので、**Ctrl+c**をタイプして止める。

gdbでデバッグ

gdb countdown

```
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5         float count = 1.0;
6
7         while (count != 0) {
8           printf("%fです。\n", count);
9           count -= 0.1;
10        }
```

この行で
止めたい

ブレークポイントの設定

(gdb) **b 9**

実行

(gdb) **run**

ステップ実行

(gdb) **s**

再度“count -= 0.1;”が表示されるまでsをタイプしよう。

count変数の値を確認

(gdb) **p count**

継続実行(continue; 省略形: c)

(gdb) **c**

count が0になるはずの10回目まで
繰り返そう

count変数の値を確認

(gdb) **p count**

$3 = -7.30156913e-08と表示される。

なぜ0にならないのか？

コンピュータ内部では浮動小数点数は2進数で処理される。10進数で0.1を2進数で表すと、0.0001100110011...と循環小数になり、正確に表せない。

gdbの終了

```
(gdb) q
A debugging session is active.

        Inferior 1 [process 4551] will be killed.

Quit anyway? (y or n) y
```

プログラムの修正

右のようにwhileの終了条件に不等号を利用する。

```
#include <stdio.h>

int main()
{
   float count = 1.0;

   while (count >= 0) {
      printf(“%fです。\n”, count);
      count -= 0.1;
   }

   printf(“おしまい\n”);
   return 0;
}
```

今回の問い

- プログラムをデバッグはどのように行うのか？
  - printfやgdbを利用する。
- C言語で関数はどのように定義すれば良いのか？

今回学ぶこと

1. デバッグ
2. 関数定義
3. 変数の分類

関数定義

```
戻り値の型　関数名(引数)
{
　やりたい処理;

　return 戻り値;
}
```

2つの整数の積を計算する関数

```
int func(int a, int b)
{
    int c;
    c = a * b;

    return c;
}
```

function01.cというファイルを作成し、右のプログラムを書いて実行してみよう。

```
#include <stdio.h>

int func(int a, int b)
{
	int c;
	c = a * b;
	return c;
}

int main()
{
   int a, b, c;
   a = 10;
   b = 20;
   c = func(a, b);
   printf(“%d x %d = %d\n”, a, b, c);

   return 0;
}
```

```
#include <stdio.h>

int func(int x, int y)
{
      int kotae;
      kotae = x * y;
      return kotae;
}

int main()
{
   int a, b, c;
   a = 10;
   b = 20;
   c = func(a, b);
   printf("%d x %d = %d\n", a, b, c);

   return 0;
}
```

引数

仮引数

実引数

引数を持たない関数

```
戻り値の型　関数名(void)
{
   やりたい処理;

   return 戻り値;
}
```

括弧内のvoidは省略できる。

関数の再帰呼び出し

関数内で自分自身を呼び出すこと。

階乗を求めるプログラム

factorial.cというファイルを作成し、右のプログラムを書いて実行してみよう。

```
#include <stdio.h>

int factorial(int n)
{
   if (n == 0)
      return 1;
   else
      return n * factorial(n-1);
}

int main()
{
   int i;
   for (i = 0; i < 11; i++)
      printf(“%d! = %d\n”, i, factorial(i));

   return 0;
}
```

factorial(3)とした場合の動作

1. 「4 * factorial(3)」がreturnされる。
2. factorial関数が呼ばれる。
3. 「3 * factorial(2)」がreturnされる。
4. factorial関数が呼ばれる。
5. 「2 * factorial(1)」がreturnされる。
6. factorial関数が呼ばれる。
7. 「1 * factorial(0)」がreturnされる。
8. ここで呼ばれたfactorial関数は0を返す。

今回の問い

- プログラムをデバッグはどのように行うのか？
  - printfやgdbを利用する。
- **C言語で関数はどのように定義すれば良いのか？**
  - **「戻り値の型　関数名(引数)」と定義する。**

今回学ぶこと

1. デバッグ
2. 関数定義
3. 変数の分類

変数の寿命

関数内で変数を宣言すると、その変数のためのメモリが確保される。return すると、確保されたメモリは破棄される。確保されてから破棄されるまでをその変数の**寿命**という。

寿命とスコープ

```
int func(int x, int y)
{
      int kotae;
      kotae = x * y;
      return kotae;
}

int main()
{
   int c;
   c = func(10, 20);
   printf("10 x 20 = %d\n", c);

   return 0;
}
```

kotaeの寿命

変数kotaeはこの範囲（スコープ）内でのみ可視。main関数からは見えない。

cの寿命

変数cはこの範囲（スコープ）内でのみ可視。func関数からは見えない。

変数の分類

- **ローカル変数**：関数内やブロック内のみで有効な変数
- **グローバル変数**：静的領域というメモリ上に記憶され、プログラムが終了するまで値を保持する変数

ローカル変数とグローバル変数

local.cというファイルを作成し、右のプログラムを書いて実行してみよう。

```
#include <stdio.h>

int i = 10;

int function()
{
   printf(“function関数: i = %d=\n”, i);
   return 0;
}

int main()
{
   int i = 5;
   {
      int i = 3;
      printf(“main関数の中のブロック内: i = %d\n”, i);
   }
   printf(“main関数内: i = %d\n”, i);
   function();

   return 0;
}
```

記憶クラス指定子

変数の前に以下の指定子を付けると変数の種類を指定して宣言できる。

- **static:** 関数がreturnしても値を保持し続ける。
- **auto:** その変数が関数内もしくはブロック内で宣言された変数（**自動変数**）であることを示す。
- **extern:** どこか別の箇所（他のソースファイルでも可）で定義されたグローバル変数を参照する変数を宣言する。

記憶クラス指定子

次ページの２つのプログラム、span01.c、span02.cを作成し、出力を比べてみよう。

```c
/* span01.c */

#include <stdio.h>

int add(int x)
{
   int gokei = 0;
   gokei += x;
   return gokei;
}

int main()
{
   int sum;

   sum = add(10);
   printf("sum = %d\n", sum);

   sum = add(20);
   printf("sum = %d\n", sum);

   return 0;
}
```

```c
/* span02.c */

#include <stdio.h>

int add(int x)
{
   static int gokei = 0;
   gokei += x;
   return gokei;
}

int main()
{
   int sum;

   sum = add(10);
   printf("sum = %d\n", sum);

   sum = add(20);
   printf("sum = %d\n", sum);

   return 0;
}
```

変数の分類まとめ

- 変数には**寿命**と**スコープ**がある。
- 関数の外で宣言した変数は**グローバル変数**となる。

今回学んだこと

1. デバッグ
2. 関数定義
3. 変数の分類

課題

円の半径を入力すると戻り値として円の面積を返す関数を作成せよ。その関数を使って円の面積を表示するプログラムarea.cを作成せよ。ただし、円周率は3.14とする。生成AIでのコード生成は不可。

提出先：yutaka.hirai@koeki-u.ac.jp

件名：応用プログラミング第4回課題 [学籍番号]

提出物：area.c

締め切り：10月29日 (木)

次回

第１回　C言語の基本文法・言語処理系
第２回　数学的計算、配列、文字列
第３回　型変換、入出力処理
第４回　デバッグ、関数定義、変数の分離
第５回　プロトタイプ宣言、分割コンパイル
第６回　配列の受け渡し、ポインタ
第７回　まとめ、期末試験