基礎プログラミング演習 II 教材　(#20)

■ 記憶クラス

★教科書 p.236, 13.2.1「変数の有効範囲とスコープ」を参照（第二パラグラフまで）

押さえて欲しいポイント：
・恒久的な記憶クラスと、一時的な記憶クラスがある
・グローバル変数は恒久的である
・ローカル変数は一時的である
・しかしローカルでありながら恒久的に値を維持したい場合もある

今までは「ローカル変数は一時的」「グローバル変数は恒久的」なものとして学んできました。しかし C 言語ではそれ以外にも「ローカル変数だが恒久的な変数」があり、そのために記憶クラスを明示的に指定する方法、記憶クラス指定子が用意されています。
記憶クラス指定子は次のように変数の定義の先頭に置きます。auto が一時的確保、static が恒久的確保に相当します。

```
auto int a;
static double x;
```

これは「a という名前で、整数型変数を一時的にメモリ上に確保しなさい」 「x という名前で、実数型変数を恒久的にメモリ上に確保しなさい」という指示をコンパイラに与えるものです。

ローカル変数に記憶クラス指定子を省略した場合は auto 変数（自動変数とも呼ぶ）とみなされます。グローバル変数は自動変数に指定することはできません。

結果的に、明示的に記憶クラス指定子を使う場面は static 記憶クラス指定子を局所変数につける場合（これを static 局所変数と呼ぶことにしましょう）だけと考えられます。次に static 局所変数を用いた例を示します。（他にも static を用いる場面、それら以外の記憶クラス指定子などがありますが、ここでは説明しません。）

□ static 局所変数

先に説明したように自動変数は関数やブロックに出入りするごとにメモリ領域の確保と解放を繰り返すため、関数を超えて変数の値を保持することが出来ません。つまり「同じ関数を実行するときに、前回実行した時の値を（覚えておいて）再び使う」ことができません。

これに対して static 局所変数はプログラムの実行開始時から変数の領域が確保されており、プログラムの終了時に初めて解放されます。すなわち局所変数であるにもかかわらず、プログラムの最初から最後までずっと変数の値を維持し続けることが出来ます。

右はそのようにして作られた「過去に渡された値を全て繰り込み続ける関数」の例です。

```
int carryover(int value)
{
  static int total = 0;

  total += value;
  printf("total=%d\n", total);
}

int main(void)
{
  carryover(1);  // total=1 と出力
  carryover(2);  // total=3 と出力
  carryover(3);  // total=6 と出力
  return 0;
}
```

□ 変数の初期化

既に説明した変数の初期化は右のようにstatic局所変数にも適用できます。

```
int x=100;
static int i=0;
```

初期設定が行われるタイミング、及び初期設定が行われていないばあいの初期値は大域変数と同じです。つまり、

・大域変数と static 局所変数の初期化はプログラムの実行開始時に一度だけ行われます。
・大域変数と static 局所変数では暗黙に 0（すべてのビットが 0 の値） に初期設定されます。

従って上のサンプルプログラムでは、実際には static 局所変数 total の初期設定をしなくても正しく動作します。

□ 課題 1.

以下の二つのプログラムは共に関数の呼び出し回数を（関数自身が）数えようとするものです。しかし左側はそれがうまくできず、右側はできています。なぜそうなるか、なぜ両者の結果が異なるのかを説明してください。また実際に実行してその予想と合っているか確認してください。

```
int f(void)
{
  int x=0;

  x++;
  return x;
}

int main(void)
{
  printf("answer=%d\n", f());
  printf("answer=%d\n", f());
  printf("answer=%d\n", f());
  return 0;
}
```

```
int f(void)
{
  static int x=0;

  x++;
  return x;
}

int main(void)
{
  printf("answer=%d\n", f());
  printf("answer=%d\n", f());
  printf("answer=%d\n", f());
  return 0;
}
```

（予想せず実行するのは避けましょう。どんな結果になり、なぜ両者で異なるかが重要です。）

□ 課題 2. （宿題）

三つの数を入力すると、二番目に大きい数を出力するプログラムを作ってください。
（三数を引数に渡すと二番目の数を戻り値で返す関数を作るのですよ）

少し C 言語の細かな機能について続いたので、static 局所変数や関数プロトタイプとは違う、幾らか楽しめる課題を用意しました。方法はいろいろ思いつくでしょう。複数試してください。

・正直に if 文を並べる
・配列を使って処理する
などなど。

講師が最初に思いついた方法は「三数のうち最大の値と最小の値を調べる関数を別に用意して、それを使って二番目の数を求める」でした。パッとしませんね、、、

■ 関数プロトタイプ

右のプログラムを入力してコンパイル、実行して下さい。
このプログラムは double 型の実数を二つ、加算した結果を出力するものです。

```
#include <stdio.h>

double add(double a, double b)
{
  return a+b;
}

int main(void)
{
  printf("answer=%f\n", add(3.0, 4.0) );
  return 0;
}
```

問題なく実行できれば、今度はわざと add 関数の引数の個数を増やして（あるいは減らして）実行してください。下のエラーが出るでしょう。

```
$ cc proto1.c
proto1.c: In function 'main':
proto1.c:10: error: too many arguments to function 'add'
$
```

（訳：関数 main について。10 行目：関数 add の引数が多すぎます。）

これはコンパイラが add 関数の呼び出し方が、引数の個数、その型、戻り値などにおいて正しいかどうかチェックしているためです。それが可能なのはコンパイラがそれ以前に add 関数をコンパイルしており、その時に関数の仕様（引数の個数、その型、戻り値など）を記憶しているためです。つまり関数の定義と呼び出しの前後関係が重要なのです。

しかしいつも順序よく書けるとは限りません。プログラムが複雑になると、このような情報を知ることが出来ない位置で関数を使う必要が出てきます。試しに、add 関数と main 関数の定義順を逆（main, add の順）にしてコンパイルしてください。下のエラーが出るでしょう。

```
$ cc proto2.c
proto2.c:10: error: conflicting types for 'add'
proto2.c:5: error: previous implicit declaration of 'add' was here
$
```

（訳：add の型情報が一致しません。それ以前に暗黙に記述された add の定義が存在します。）

これは関数 add の定義より前に add が使われているために、型のチェックができないためです。

> 「この関数は未知のものです」とエラーが出て欲しいところですが、そうはなりません。「未知の場合は暗黙に int 型を仮定する」ためですが、ここでは説明しません。

そこで C 言語では、関数の定義より前に関数について情報を与えるために「関数プロトタイプ（関数の「原型」というような意味）」があります。関数プロトタイプを使うと上のプログラムは右のように書き換えることができます。

```
#include <stdio.h>

double add(double fst, double snd);

int main(void)
{
  printf("answer=%f\n", add(3.0, 4.0) );
  return 0;
}

double add(double a, double b)
{
  return a+b;
}
```

この add の関数プロトタイプ

```
double add(double fst,
           double snd);
```

により、コンパイラは add が double 型の値を返す、また二つの double 型の引数をとる関数であることを事前に知ることができるので、main の中でその戻り値の型や引数の型と個数が一致していることをチェックすることができます。

□ 書式

関数プロトタイプの一般的な形は以下のようになります（引数が二つある場合）。

戻り値型 関数名(引数型 1 仮引数名 1, 引数型 2 仮引数名 2);

関数プロトタイプの仮引数名は、その関数の定義で用いた仮引数名と異なっていても構いません。実際、例ではプロトタイプでの引数名は fst と snd でしたが、関数定義では a と b となっています。

また、仮引数名を省略することもできます。

```
double add(double, double);
```

こちらの方がシンプルですが、上手に仮引数名を付ければ読むときに引数の意味がわかりやすくなる利点もあります。これらの書き方は必要に応じて使い分けましょう。

□ 参考：いくらか特殊な書き方

・関数に引数がない場合、引数型として void と書きます
（この場合は仮引数名は意味がないので書きません）。

```
double foo(void);
```

・printf のように引数の数が可変の関数のプロトタイプは ... を用いて書きます。

```
int printf(char *format, ...);
```

・古いプログラムでは関数プロトタイプの引数の部分が書かれていないこともあります。

```
double add();
```

これは以前の C 言語で用いられていた宣言の仕方で関数宣言と呼ばれます。void と違って引数の有無は不明となり、チェックされません。受講生は関数プロトタイプを使うべきでしょう。

■ 参考：ライブラリ関数の関数プロトタイプ

C 言語でプログラムを開発する場合には入出力関数や数学関数など様々なライブラリ関数を用いますが、それらについても当然ながら型についての情報をどこかから得る必要があります。

ライブラリ関数の関数プロトタイプはまとめて書かれたものが用意されています。よく使っている #include <stdio.h> などのインクルード命令は、これをソースファイルに取り込むものです。ライブラリ関数を使用する前にプロトタイプが必要なので、通常はソースファイルの先頭などでインクルードします。（教科書 p.71 参照）

例えば man sin でライブラリ関数 sin のマニュアルをみると、戻り値や引数の型とともに、使用すべきインクルードの対象ファイル（この場合は#include <math.h>）が書かれています。

```
SYNOPSIS
     #include <math.h>
     double
     sin(double x);
```

ライブラリ関数の使用は、プログラムをいくつかの部分に分けて開発する手法（分割コンパイル）の一種に当たりますが、ここでは説明しません。

インクルードの対象となるプロトタイプ等が書かれたファイルをヘッダファイルと呼びます。拡張子は通常.h を用います。

■ 宿題

プロトタイプに関する宿題はありません。そのぶん課題 2.をじっくりやってください。