ソフトウェア概論 B

数学科 吉開範章, 渡辺俊一 (栗野 俊一)

前回のまとめ 1 : ポインター(型変数)

- ポインター値の計算(属性の変更)
  - アドレス情報 : アドレス演算子 (&) で取り出す
    - 整数を加える (アドレス情報は sizeof(型) だけ変化)
  - 型情報 : 型キャスト 「(type *)」 で変更可能
    - アドレス値からポインターを作ることができる
- ポインター型変数
  - ポインター値を値として持つ変数 (aka ポインター)
    - 「計算した結果としてのポインター値」を持つことができる
- 配列とポインター
  - 配列名は、ポインター(型)定数
  - 配列参照([])は、間接参照(*)
    - array[index] == *(array + index)
- 関数引数とポインター
  - 関数引数で「変数」は渡せないが「ポインター」は渡せる
  - 呼び出し先(関数)内で呼び出し元の変数の値を変更できる
    - scanf の用例

前回のまとめ 2 : 安全なポインター利用

- ポインター(値)は、以前から利用していた
  - 関数引数 (scanf)
  - 配列名 / 配列参照
- 従来と同じ使い方なら安全
  - scanf の場合に利用する
  - 配列名 + offset の代わりに利用する
- ポインター(型変数)の危険性
  - 結局は、「変数」に対応しないアドレスを指す事
    - 常に「配列と対」で利用すれば良い
    - 危険がなくなるわけではないが、配列の利用と同程度の危険性しかない
- ポインター型変数の安全な利用法
  - ポインタ型変数は、必ず、「配列名 + offset」で「初期化」する

本日の概要：ポインター(3)

- ポインターの強力な使い方を学ぶ
  - 多次元配列とポインター
  - ポインター型配列
- ポインターの本質的な利用方法の入門
  - 配列の領域を動的に確保する
    - malloc/free の利用方法
- 課題
  - pascal の三角形のプログラム
    - 20101203-01 : 二次元配列版
    - 20101203-02 : ポインター配列版
    - 20101203-03 : malloc/free 版

多次元配列とメモリモデル

- □ メモリモデル
  - ○ メモリはメモリセル(1 byte)の並び
- □ 変数[*01]
  - ○ 変数は、複数(「sizeof(型)」byte 数)の大きさのメモリセルに対応
- □ 一次元配列
  - ○ 一次元配列は、変数(零次元配列)の連続並び
    - ▹ 一次元配列は、「sizeof(型)」×「配列サイズ」の連続したメモリセルに対応
    - ▹ cf. int a[6] は 「sizeof(int) = 4」×「配列サイズ = 6」 = 24
  - ○ k 番目の要素のアドレス値
    - ▹ 配列の先頭のアドレス値(配列名) + 「sizeof(型)」×「添字」
    - ▹ cf. a[2] は 「a のアドレス値」 + 「sizeof(int) = 4」×「添字 = 2」
- □ n 次元配列
  - ○ n - 1 次元配列の連続並び (a[M][N] は a[M] が N 個ある)
    - ▹ n 次元配列は「sizeof 型」×「各々の次元の積」の連続したメモリセルに対応[*02]
    - ▹ cf. double a[3][5][7] は「sizeof(double) = 8」×「次元の積 = 3 x 5 x 7 」 = 840

多次元配列とポインター

- □ 整数の一次元配列は、整数型へのポインター定数
  - ○ では、整数の二次元配列 (int array[M][N]) は ?
    - ▹ 整数の一次元配列のへのポインター「定数」
    - ▹ array[i] は サイズ N の整数配列 (int [N])の先頭を指す [*03]
- □ メモリモデルからみた多次元配列
  - ○ 実体は、一次元も二次元も同じ
    - ▹ 多次元配列を一次元配列にできる [*04]
    - ▹ サイズさえ同じなら異なる見方もできる [*05]
- □ ポインター配列：ポインター型変数の配列
  - ○ ポインター型変数も変数：変数を並べた配列が作れる
    - ▹ int *parray[M]; // M 個のポインター型配列 [*06]
  - ○ 一次元配列の好きな部分を指すことができる
    - ▹ 矩形以外の形の 2 次元配列 [*07]
  - ○ 二次元配列宣言のサイズ指定
    - ▹ 次の配列の先頭の計算を自動的に行うために必要
    - ▹ ポインター配列を用意し、自分で計算するなら、不要

パスカルの三角形

- 組み合わせの公式

$$ {}_nC_r = {}_{n-1}C_r + {}_{n-1}C_{r-1} $$

  - 並べてみると...

```
                    r=0
                  /  r=1
              n=0 - 1  /  r=2
            n=1 - 1   1  /  r=3
          n=2 - 1   2   1  /  r=4
        n=3 - 1   3   3   1  /
      n=4 - 1   4   6   4   1
                  ...
```

  - これを C 言語で作ってみよう

パスカルの三角形１：二次元配列

□パスカルの三角形の二次元配列への対応

```
      r=0 r=1 r=2 r=3 r=4
       |   |   |   |   |
n=0 - 1    |   |   |   |
n=1 - 1    1   |   |   |
n=2 - 1    2   1   |   |
n=3 - 1    3   3   1   |
n=4 - 1    4   6   4   1
```

パスカルの三角形 1 : 二次元配列

- 二次元配列によるプログラム [*08]
  - nCr のサイズは固定 (ここでは 6)
    - #define PASCAL_SIZE 6
  - nCr の作成と出力は別の関数で行う
    - print_pascal : 三角形の出力
    - make_pascal_triangle : 三角形の作成
  - nCr を二次元配列 pascal_array で表現
    - int pascal_array[PASCAL_SIZE][PASCAL_SIZE];
  - 制約
    - 三角形の大きさ(PASCAL_SIZE)を予め定める必要がある
    - 全ての関数が、PASCAL_SIZE を知らないといけない
    - 配列の一部が無駄になっている

パスカルの三角形 2 : ポインター配列

- ポインター配列よるプログラム [*09]
  - nCr のサイズは固定 (ここでは 6)
    - #define PASCAL_SIZE 6
  - nCr をポインター配列 pascal_triangle で表現
    - int *pascal_triangle[PASCAL_SIZE];
  - 制約
    - 三角形の大きさ(PASCAL_SIZE)を予め定める必要がある
    - 全ての関数が、PASCAL_SIZE を知らないといけない
    - 配列の無駄が減った
- サイズを引数で指定 [*10]
  - 制約
    - 三角形の大きさ(PASCAL_SIZE)を予め定める必要がある
    - PASCAL_SIZE が必要なのは、main だけ
- 任意のサイズを扱うにはどうするか？
  - 予め、大きなサイズの配列を用意し、その一部を利用する？
    - 折角、無駄をなくしたのに..
    - それに、予め、どの位の大きさの領域を用意すれば..？

メモリー管理の話

- メモリーは
  - 確かに大量にはあるが.. しょせん有限な資源
    - 変数を沢山使うと何がおきるか？
- 関数の中で宣言された変数 [*11]
  - 異なる関数内の異なる変数なのに、同じアドレス値をもつ
    - 同じメモリが使い回されている
  - 配列宣言の影響 [*12]
    - 関数の呼ぶ側で沢山の変数を利用すると、ずれてゆく
- メモリーの再利用
  - スタックを使って、自動的に行われる
  - 変数のサイズを予め知る必要がある。
    - 変数宣言/配列の添字宣言の必要性

動的なメモリー管理

- □配列のサイズを後から決めたい
  - ○配列のサイズを、プログラムの実行の後で入力させたい..
    - ▹malloc/free を使う[*13, *14]
  - ○自分でメモリー管理を行う[*15,*16]
    - ▹malloc : 必要なだけのメモリーを確保する
    - ▹free : malloc で確保したメモリーを開放する
    - ▹malloc したものは必ず free する !!
  - ○振舞いの違い
    - ▹静的 : サイズ固定 / 有効範囲が関数内 / 管理不要
    - ▹動的 : サイズ自由 / malloc　free / 管理必要
- □C 言語のデータ構造
  - ○普通の変数 : 構造なし
  - ○配列 : 同じものが並んでいる / サイズは固定
  - ○構造体 : 異なるものが並んでいる / サイズは固定
    - ▹サイズが可変なものはない
  - ○ポインターと malloc/free
    - ▹可変な構造を表現する方法 (ポインターの本質的な利用法)

課題

□課題は、次の Web Page の内容を参照してください。

http://edu-gw2.math.cst.nihon-u.ac.jp/~kurino

おわり

終了