ソフトウェア概論 A/B

-- データ構造 (4) --

(メモリモデルとポインタ値)

数学科 栗野 俊一 / 渡辺 俊一 (TA: 栗原 望 / 小嶋 仁子 [M2])

2018/12/21 ソフトウェア概論

伝言

私語は慎むように !!

- 出席パスワード : 20181221
- 色々なお知らせについて
  - 栗野の Web Page に注意する事

    http://edu-gw2.math.cst.nihon-u.ac.jp/~kurino
- 廊下側の一列は遅刻者専用です(早く来た人は座らない)
- 講義開始前に済ませておく事
  - PC の電源を入れておく
  - ネットワークに接続しておく
  - 今日の資料に目を通しておく
- 講義前の注意
  - 講義前は、栗野は準備で忙しいので TA を捕まえてください
- やる気のある方へ
  - 今日の資料は、すでに上っています
    - どんどん、先に進んでかまいません

今後の予定

- 今後の予定(後ろから)
  - 2018/01/25 (講議最終日)
    - 試験を行う
  - 2018/01/18 (講議最終日前)
    - 模擬試験を行う (予定) / FILE 入出力
  - 2019/01/11
    - データ構造 (5)
  - 2018/12/28, 2019/01/04
    - 冬期休暇期間中：この講議はない
  - 2018/12/21 (本日)
    - データ構造 (4)：メモリモデルとポインタ値

前回(2018/12/14)の内容

- 前回 (2018/12/14) の復習
  - 共用体：二つの型のどちらか一方を表現する (和集合を作る) 型を作る
    - 構文 : union タグ { 型1 メンバ名1; 型2 メンバ2; .. };
    - 意味 : この型の変数は、メンバ名i で参照すると型i を持つ変数として振る舞う
    - データ構造としては、「条件構文」に対応 (どれか一つ)
  - 配列と文字列：「文字列」は「文字配列」として実現されている
    - C 言語では、EOS ('\0') で終わる文字配列を「文字列」として扱う
  - 配列を関数引数
    - 配列名を関数の引数に指定できる(サイズは無意味)
    - 関数の呼出し側と呼ばれる側(関数内) では、配列要素が共有される

お知らせ

- 本日の予定
  - データ構造 (4)
    - メモリモデルとポインタ値
- 本日の目標
  - 演習
    - 課題の提出

先週 (2018/12/14) の課題

- □ 先週 (2018/12/14) の課題
  - ○ 課題 先週-01:
    - ▹ ファイル名 : 先週-01-QQQQ.c (QQQQ は学生番号)
    - ▹ 内容 : 配列内の浮動小数点数の合計を求める Sum 関数
  - ○ 課題 先週-02:
    - ▹ ファイル名 : 先週-02-QQQQ.c (QQQQ は学生番号)
    - ▹ 内容 : 文字配列に入った文字列の途中に文字を挿入する
  - ○ 課題 先週-03:
    - ▹ ファイル名 : 先週-03-QQQQ.c (QQQQ は学生番号)
    - ▹ 内容 : 一行分の文字列を入力して、その中の文字列を全て大文字に変換する
- □ ※
  - ○ ファイル形式は、いずれもテキストファイル(C 言語プログラムファイル)

今週 (2018/12/21) の課題

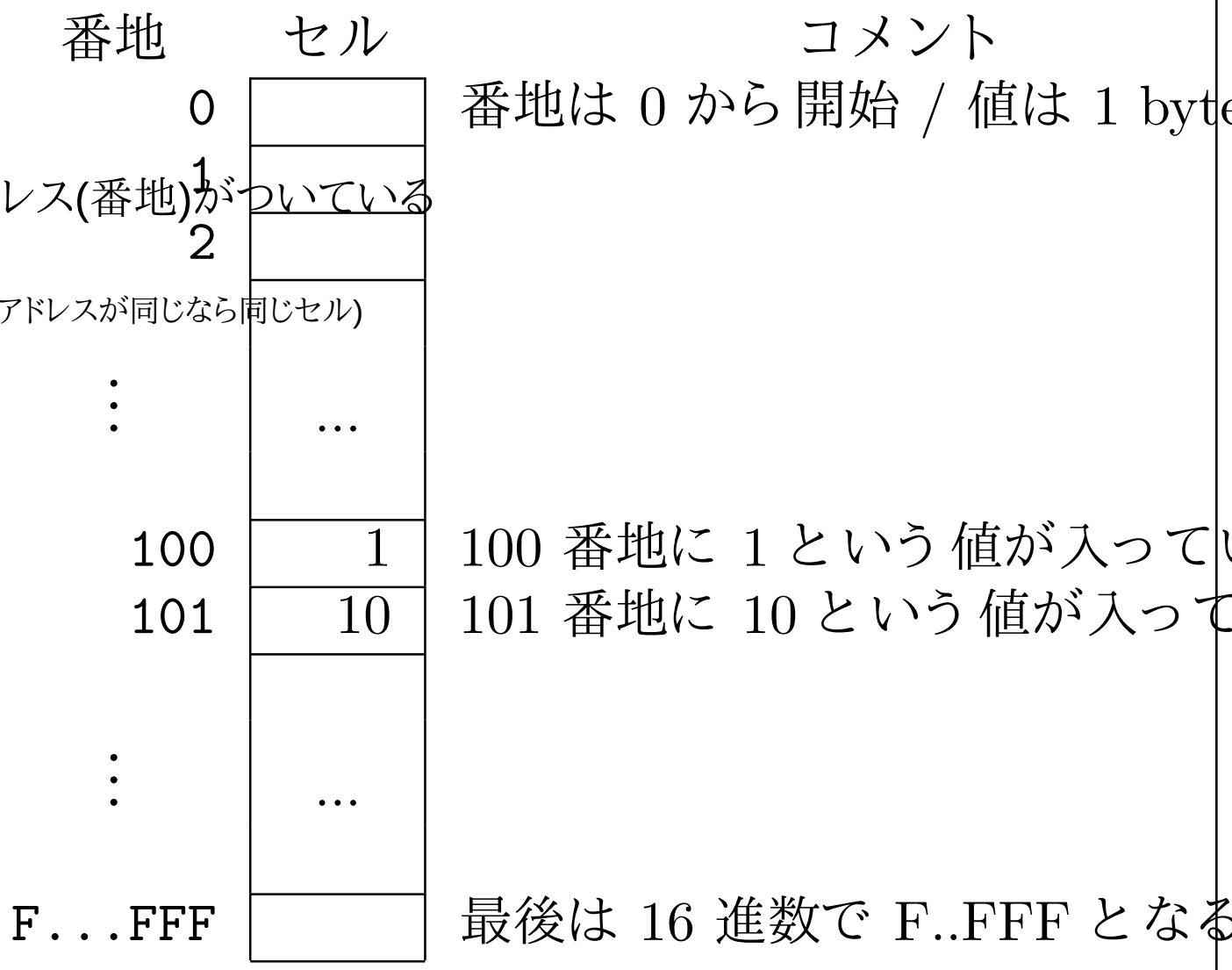
- 今週 (2018/12/21) の課題
  - 課題 今週-01:
    - ファイル名 : 今週-01-QQQQ.c (QQQQ は学生番号)
    - 内容 : union の応用 (整数型変数のメモリ構造の出力)
- ※
  - ファイル形式は、いずれもテキストファイル(C 言語プログラムファイル)

文字列の入力

- 文字列の入力
  - 「文字列」は、「文字」の配列(の途中に EOS が入ったもの)
  - 「文字列を入力する」とは？：複数の文字を入力して配列に収める
- 文字列の入力方法
  - 方法 1 : scanf の「%s」を使う -> 危険なのでやってはいけない
  - 方法 2 : gets を使う -> 禁止されている
    - 方法 1 / 方法 2 は、「バッファオーバーフロー」の根源
    - 結果的にセキュリティホールの山
  - 方法 3 : fgets を使う -> 安全な方法(推奨)

メモリのイメージ

- メモリ
  - セルの並んだ物
    - セルのサイズは 1 byte
  - 個々のセルには、別々のアドレス(番地)がついている
    - アドレスは 0..0 ～ F..F (16 進)
    - アドレスは、「セルの名前」として働く(アドレスが同じなら同じセル)
  - セルの機能 (変数と同じ)
    - 情報を記録できる
    - 情報を取り出せる



メモリ・モデル

- メモリ・モデル
  - C 言語の変数のモデルの一つで、「変数をメモリセルの組み合わせ」として理解する
    - C 言語の「変数の振舞い」を「考えるための仕組み(モデル)」
    - !! 「『何か』のモデル」とは『何か』を理解するために利用可能な、「より簡単な仕組み」の事
    - !! 「C 言語の変数」を「メモリモデル」を通じて理解する/ 簡単なの理解しやすい
- 実は..
  - 多くの場合、「C 言語の変数」は実際に「メモリセルの組み合わせ」になっている
    - 変数の性質(代入)はメモリの性質(記憶能力)から説明できる
- char 型変数とメモリモデル (sample-005.c)
  - char 型変数は、一つのメモリセルだと考える事ができる
    - char 型変数は address を持つ
  - char 型変数をメモリセルと同様に扱う事ができる

メモリモデルと配列

- 文字列とメモリモデル (sample-006.c)
  - 文字列は、文字の並び
    - 文字は char 型変数で記録できるので、文字列は char 型変数の並び
- 文字変数の並びと文字列 (sample-007.c)
  - アドレスが判れば、変数の内容をアドレス経由で操作できる
- 配列宣言 (sample-008.c)
  - 配列とは
    - 「複数の変数の並び」の事 (個々の変数を「配列の要素」と呼ぶ)
  - 一次元の配列宣言 (sample-008.c)
    - 変数と同様に型名の後ろに「配列名[サイズ]」の形で宣言
    - 「サイズ」の個数だけの変数が宣言される。
    - 配列の要素は「配列名[0] ～ 配列名[サイズ-1]」という「名前」になる
    - 添字：「[」と「]」の間には整数値が指定でき、配列の何番目の要素かを表す

文字列と文字型の一次元配列の関係

- 文字列と文字型の一次元配列の関係
  - C 言語の文字列
    - 文字の並んだ物 (文字コードが連続に記録されている)
  - C 言語の文字型変数
    - 文字コードを一つだけ記憶できる
  - C 言語の文字型の一次元配列
    - 複数の文字型変数が並んだもの
  - C 言語の文字列型の一次元の配列で文字列を記憶する事ができる
- 文字列の一部の操作方法 (sample-009.c)
  - 文字配列の要素を変更すればよい
- 文字列を利用した文字配列の初期化 (sample-010.c)
  - 文字配列の要素を文字列を利用して初期化できる

型のサイズ

- 型のサイズ
  - データ(情報)はサイズを持つ
    - 例 1: char 型 のサイズ : 8 bit = 1 byte
    - 例 2: int 型のサイズ : 64bit = 4 byte
  - サイズ S byte のデータは 2^(8S) = 256^S の状態を表現できる
    - 例1 char 型 は 0 ～ 255 (256 通り) の状態 : 半角文字は表現できるが全角文字は無理
    - 例2 int 型 は -2^{63} (-2147483648) ～ 2^{63} - 1 (2147483647) までの 2^64 通り
    - cf. /usr/include/limits.h
  - その型のデータのサイズ
    - その型の状態数を表現 / その型の情報を記録するために必要な記憶領域サイズ
    - より多くの状態を表現したければ、より多くのサイズ(の記憶領域)が必要

sizeof 演算子

- sizeof 演算子
  - 前置演算子で、その後ろにあるデータのサイズを byte 単位で答える
    - 引数に「型名」を記述する事もできる
  - C 言語では、型に対するデータのサイズはシステムによって異る
    - cf. /usr/include/limits.h
    - 例 : int は、その計算機(32bit/64bit)で最適なサイズになる (sizeof(char) は 1)
    - 個々の計算機で「最適」なコードが作られる(可能性が高い):利点
    - (サイズが異るので..) 同じプログラムが、システムによって異る振舞をする:欠点
    - sizeof 演算子は、その「違い」を吸収する必要がある場合に利用
- C 言語における型情報
  - 型 : 表現形式 x 操作方法
  - 表現形式 : サイズ x 情報との対応形式
    - サイズは、表現対象の集合のサイズ(有限の場合)より大きくする(char)
    - 表現対象の一部としか対応していな場合がある(無限の場合:整数、実数等)

暗黙の型変換(型の昇格)

- char 型から int 型への型の昇格
  - 「計算」の場合、char 型の値は int 型に「(無条件に)昇格」する
    - char 型のサイズは 1 (= sizeof(char))
    - 'A' の値は、整数値 65 (ASCII Code) になる
    - cf. sizeof(char) == 1 / sizeof('A') == sizeof(int)
- int 型から double 型への型の昇格
  - int 型同士の計算は int 型のまま
  - double 型と int 型の混在式では、int 型から double 型への「型の昇格」が起きる
- 代入における型変換
  - 変数への代入では、値の型が、変数の型に変換される
    - 関数の「実引数(値)」は、関数の「仮引数(引数変数)」への代入となる
  - サイズの小さい方から大きい方の変換は問題ない
    - その逆(大きい方から小さい方)は「危険」!! (オーバーフローする)

配列の添字, 間接(参照)演算子,アドレス演算子

- □ 文字列の操作 (復習)
  - ○「*」:間接(参照)演算子：文字列の先頭の文字を取り出す (sample-011.c)
    - ▹ *"abc" == 'a'
  - ○「[]」:添字演算子：「[n]」で n (整数値)で「n+1番目の文字」意味する
    - ▹ "abc"[0] == 'a', "abc"[1] == 'b', ..
  - ○「*」と「[]」の関係；文字列[n] == *(文字列+n)
    - ▹ "abc"[0] == *("abc"+0) == *("abc") == *"abc" == 'a'
- □ アドレス演算子「&」
  - ○ アドレス演算子「&」は 間接演算子「*」の逆演算を行う
    - ▹ (&(*("abc"))) == "abc"
  - ○ 変数に関しては逆が成立する
    - ▹「*(&var)」==「var」
  - ○ アドレス演算子「&」の正体
    - ▹ 変数に対応したメモリセルの「アドレス」を得る演算子

多次元の配列

- 二次元の配列
  - 一次元の配列を二次元的に利用する事ができる (sample-012.c)
    - 二つの添字からアドレスを計算すればよい (sample-013.c)
  - 始めから二次元の配列を宣言する事ができる (sample-014.c)
    - 配列の要素のアドレス計算は、自動的に行われる (sample-015.c)
    - 実態は「(一次元)配列の(一次元)配列」だが、慣例により「二次元配列」と呼ぶ
  - 応用例
    - 行列は、二次元配列で表現可能 (sample-016.c)
- 多次元の配列
  - 一つの型から新しい配列型を作るだけなのでいくらでも大丈夫(?)

整数型とメモリモデル

- 整数型とメモリモデル
  - 文字型変数はメモリセル一つに対応
    - では、整数型変数は ... ? / 実は、連続したメモリセルに保存される
- sizeof 演算子 (sample-017.c)
  - その型の変数が、何個(byte)のセルを占めるかを教えてくれる
    - sizeof(char) == 1
    - sizeof(int) == 4 (32 bit の場合)
  - int 型の変数は 4 つのセルで表現される

ポインタ値とポインタ型

- ポインタ値
  - 「&」の作る値は実は、単なるアドレス値 *だけ* ではない
    - アドレス値も持つが、それと、「型情報」も持つ
    - 型情報 : サイズ + 処理の仕方
    - !! 型情報は、コンパイル時だけ、実行時には解らない(解るのはアドレスだけ)
- ポインタ値の型
  - 「型名 *」:「～型へのポインタ型」と読む
    - 「* をつけると「型」と同じになる」の意味
    - char * :「文字列」ではなく、「char 型へのポインタ型」だった

ポインタ値の計算

- ポインタ値
  - 二つの情報をもつ
    - 型情報：何型の情報が入っているものか？
    - アドレス値：どこに入っているか？
- ポインタ値の計算
  - 整数値 n を加える事ができる
    - 型情報は変らず、アドレス値だけが変化
    - アドレス値は n × sizeof (型) だけ変化 (n は負の数でもよい)
  - 同じ型のポインタ同士なら引き算もできる
    - 結果は整数値で、(アドレス値の差) / sizeof (型) となる
    - p, q が同じポインタ型なら「p + (q - p) == q」が恒等的に成立
  - 「キャスト」を利用して、型を変更できる
- ポインタ値と添字
  - 恒等的に「 p[n] == *(p + n) 」が成立する

配列とポインタ型

- 配列とポインタ値
  - 「～型一次元の配列名」は、「～型へのポインタ型定数」となる
- 一次元の配列宣言とポインタ型変数宣言
  - 一次元の配列宣言「char a[N];」
    - char 型の変数 a[0] ～ a[N-1] の N 個の変数を宣言
    - 個々の要素変数の型は char 型
    - 配列名「a」は 要素の先頭を指すポインタ型定数(a == &a[0])
  - ポインタ型変数宣言「 char *p; 」
    - 「char *」型の変数 p の 1 個の変数を宣言
    - 変数 p の値は不定 (だから *p の値も宣言時点では不定)
    - !! p の値は適切に初期化して利用する必要がある
  - 配列名によるポインタ変数の初期化
    - 代入文「p = a」を行うと...
    - 「p[k]」と「a[k]」は全く、同じように振る舞う