プログラミング演習II
デバッガ演習

2026/4/30

担当：月川 淳

tsuki@is.Utsunomiya-u.ac.jp

本演習で実施すること

- デバッガの基本的な使用方法の復習
- デバッガを利用しながら、ポインタの理解を深める

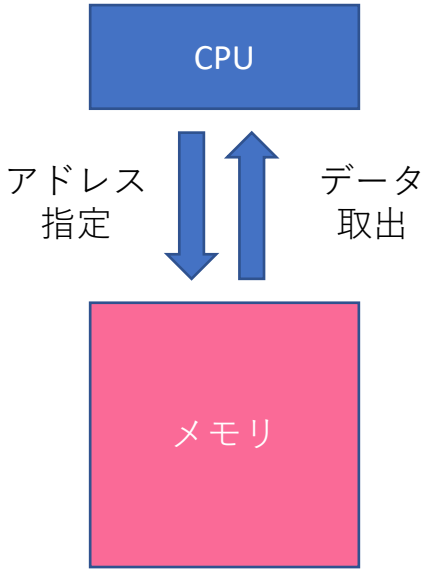
※解説は黒字、作業する部分は赤字にて記載しています

事前知識の確認：計算機で扱うビット列データ

- ビット列
  - 1と0 (On/Off)が連続したデータ
  - 計算機では主に**バイト(byte)** 単位で扱う
- 単位
  - 1バイト ＝ 8 ビット
  - 1ワード (word)　→ 環境依存 ・処理系依存(ex CASL では16ビット)
- 表記
  - 16進数で表すことが多い
    - 4ビットずつ区切ってそれぞれを16進数表記
  - 1バイトは16進数表記で2桁
- 言語処理系での16進数定数
  - **`0x1234`**　(C/C++, Java, python)
  - **`#1234`**　(CASL)
  - **`1234h`**　(多くのアセンブリ言語)

事前知識の確認：メモリ(主記憶)について

- 複数の多量のビットデータを保存する記憶素子
- アドレス(番地)でデータを区別
- １アドレスにつき **1 byte**のデータを格納
  - (※ CASL等では、1アドレスにつき１word)
- アドレス空間
  - CPUがアクセスできるメモリの範囲・メモリの量
    - 8ビットのアドレスでアクセスできるアドレス空間は？⇒2^8
    - 4GBのメモリにアクセスするために必要なアドレスのビット数は？⇒　???

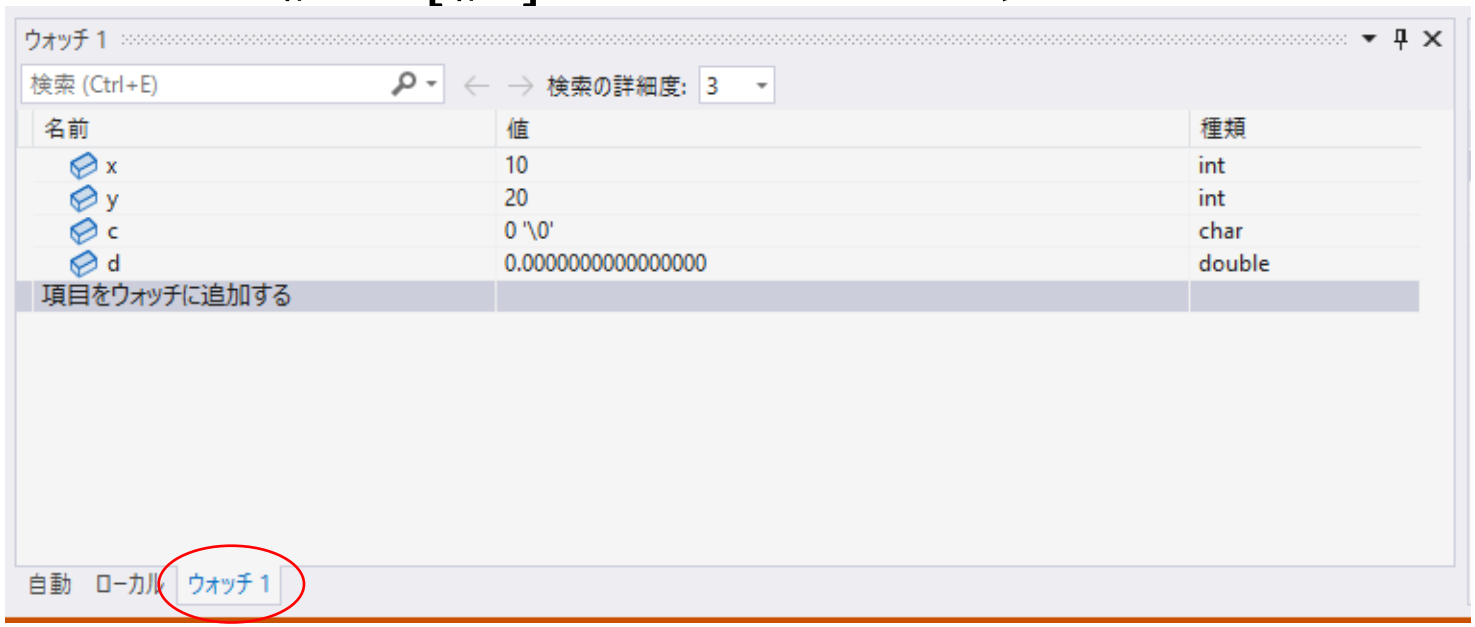


準備

1. pointer フォルダ内の pointer.sln をダブルクリックして開く
   - ※必ずzドライブからのパスで開くこと
2. ビルド
3. ここにブレークポイントを置く　のコメント位置にブレイクポイントを置く
4. デバッガを起動　（[デバッグ]→[デバッグの開始]）
5. ウォッチウインドウにx y c d を入力して値を確認
6. メモリウインドウを表示[デバッグ]→[ウインドウ]→[メモリ]→[メモリ１]
7. メモリウインドウのアドレスに **0x19FE5C** と入力

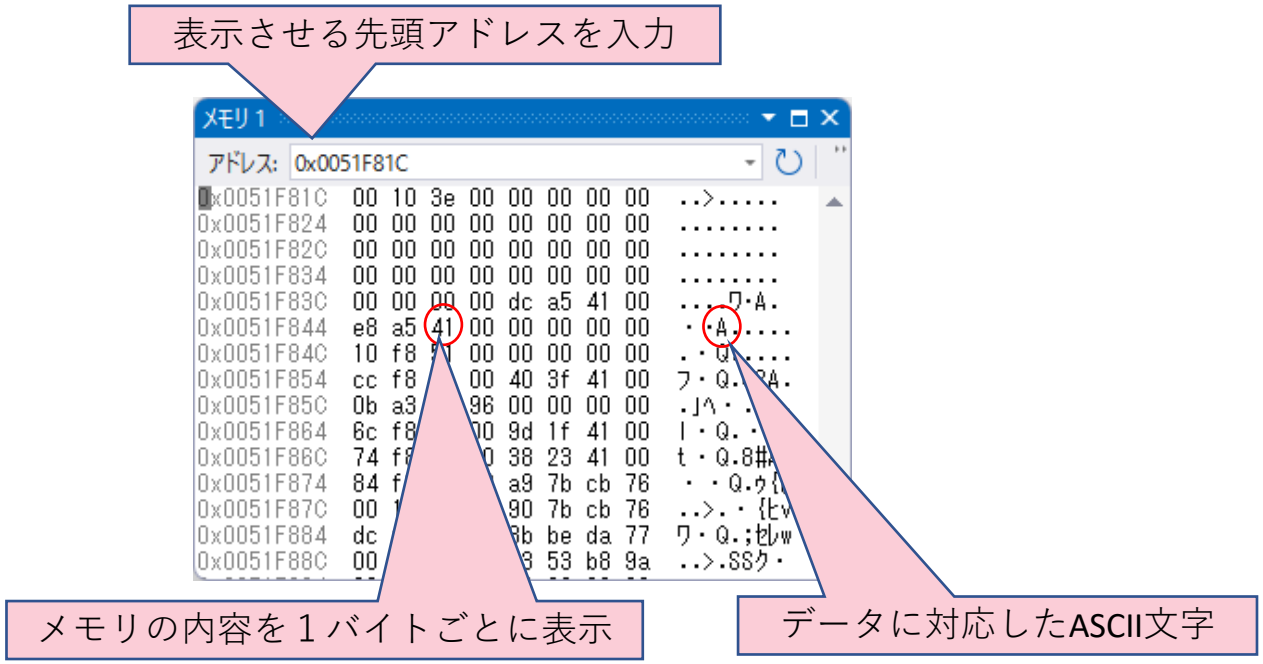
ウォッチウインドウ

- [名前]フィールドに式を入力すると、それが評価された値が[値]フィールドに表示される



メモリウインドウ

- デバッガ実行中に [デバッグ]→[ウインドウ]→[メモリ]→[メモリ 1]
- メモリマップの状態を観察することができる



変数のサイズを確認する

- 変数はメモリ上に配置される
  - 変数の型により、必要とするメモリ量が異なる
- ウォッチウインドウから、**`x,y,c,d`** の値を変更してみる
  - Ex) x を **`0xffffffff`**, y を **`0xeeeeeeee`**, c を **`97`**, d を **`3.14`**
- メモリウインドウでは変更された部分が赤字で表示され、どの部分が使用されているか確認できる
- それぞれの変数について、何バイトの領域を使用しているか確認

sizeof 演算子(C言語)

- **sizeof (型名)，　sizeof 変数名** の書式で、その型、変数のバイト数を表す
- ウォッチウインドウに以下の式を入力して値を確認
  - **sizeof(int)，sizeof(char), sizeof（x), sizeof(array)**
  - **sizeof x, sizeof y, sizeof c, sizeof d**

(発展)：エンディアン

- ウォッチウインドウより、x の値を **0x12345678** に変更
  - メモリ上にどのように格納されるかを確認する
- エンディアン
  - メモリにデータを配置するときのバイトの並び順（バイトオーダー）
  - プロセッサごとに異なる
    - リトルエンディアン：データの下位バイトを下位メモリに配置
    - ビッグエンディアン：データの上位バイトを下位メモリに配置

- 単一の計算機で動作させるプログラムを作成する際にはそこまで意識する必用はないが、他のシステムとデータを交換するようなプログラムでは注意が必要

アドレス演算子(C言語)

- 変数はメモリ上に構成される→かならずそのアドレスがある
- 変数のアドレスを得るのに **アドレス演算子(&)** を使用する
  - **`&x`** と記述すると、それは変数**`x`** のアドレスを表す

- ウオッチウインドウに &x, &y を入力して，その値を確認する

ポインタ変数(C言語)

- ポインタとは、変数のアドレスを格納するための変数
  - ポインタ変数p に x のアドレスが格納されているとき、「p は x を指している」という
- ポインタ変数の**宣言**

差している先の変数の型　　ポインタであることを表す＊

```
int *p;
```

- プログラムでは、ブレイクポイントの直前に **p=&x;** を実行している

1. ウォッチウインドウで p の値を確認する
2. ポインタ変数 p, pc, pd のサイズを確認し、なぜそうなるかを考察する

間接参照演算子(*) による値の参照(C言語)

- ポインタが指している変数の値を参照するには、関節参照演算子(＊）を用いる
  1. ウォッチウインドウに *p と入力してその値を確認
  2. xの値を変更し *p の値がどのようになるか確認

- ポインタは指す先を変更できる
  - プログラム中で p=&y; を実行したときの挙動をデバッガで確認
  1. p の値に &y と入力し、確定した値がyのアドレスであることを確認 (ウォッチウインドウの[名前]に p = &y と入力してもよい)
  2. このとき *p の値はどうなっているか？
  3. また、y の値を変更したときに *p の値はどのようになるか？

ポインタ変数の型について

- ポインタ変数はアドレスを格納する型⇒サイズは一定
  - int* も char* もfloat* もアドレスを格納しているだけ

- メモリ上の特定のアドレスを参照
  - ➡　アドレスだけでなく、そのメモリ領域をどのように評価する必要があるかの情報が必要

間接参照による変数の書き換え(C言語)

- 間接参照演算子を用いて、ポインタの指している変数を変更することができる
  - プログラム中で　*p = 1000; を実行した場合の挙動を、デバッガで確認する
  1. p の値を確認し、そのアドレスから指している変数を確認
  2. *p の値を変更してその挙動を確認する
  3. pの値を &x や &y に変更してから *p の変更し、その挙動を確認する

- C言語ではポインタが指す先が有効な変数であるかはチェックされない
  1. pの値に0を入力し、この時の*pの値を確認する
  2. この状態で実行ボタンを押してプログラムの*p=0の行を実行し、その挙動を確認する
- ポインタ変数は、それを経由して意図しない変数や領域の変更をしてしまう場合があるので注意して使用する必要がある

(発展)：const ポインタについて

- 変数の参照のみが目的で、変更する必要のないポインタ変数には、その旨を定義することでコンパイル時にエラーを発生させ、実行時の事故を減らすことができる
  - バグがあったときに、なるべくコンパイル時に発見できるようなコード記述を目指すべき
- constポインタの宣言例
  - **const** int* ip;
- 以下のコードを x=10 の行の下あたりに記述しビルドを行いコンパイラの挙動を観察する
  (確認が終わったら追加したコードを削除してもとにもどす)

```
const int* cp;
cp = &x;
y = *cp;     //間接参照
*cp = 200;   //ポインタ経由の変数書き換え
```

配列(C言語)

- 配列：添え字を使ってアクセスできる変数の集合
  - プログラム中では **`int array[10];`** と宣言されている
- ウォッチに以下のシンボルを入力し、その値を確認する
  - **`&array[0]`**
  - **`array`**
  - **`&array`**
- 上記のいずれも配列の先頭要素のアドレスを示す
- 以下を入力し、配列の各要素のアドレスを確認する
  - **`&array[1]`**
  - **`&array[2]`**
  - **`&array[3]`**
- アドレスがいくつおきに配置されているか？
  - メモリウインドウでも確認してみる

配列とポインタ(C言語)

- p=array; を実行し、配列のアドレスをポインタに代入した状態をデバッガで確認してみる

1. p の値に array と入力し、pに配列の先頭アドレスを入れる
2. この時、以下のシンボルがどのように評価されるか確認する
   - `*p`
   - `p[3]`
   - `p[9]`

- ポインタ変数に添え字を使って配列のようにアクセスすることは可能

- ウォッチウインドウで arrayの値を&x に変更して動作を確認する

アドレスの加減(C言語)

- ウォッチウインドウで以下の式がどのように評価されるか調べる
  - X + 1
  - array + 1
  - array + 2
  - p + 1
  - p + 2
- ポインタの加減は、その型により**スケーリング**される

- 同様の操作を char, double へのポインタである pc, pd についても行い結果を確認しその理由を考察する

配列のポインタによる表現

- array[3]　⇒配列の３番目の要素　これをポインタ加算と間接参照で表す
  - ３番目の要素のアドレス⇒　array + 3
  - 上記アドレスを間接参照⇒　*(array + 3)
- **array[3]** と ***(array +3)** は同じ意味となる

- ※ 実際にコンパイラ内では、A[B] という構文は *(A+B) と解釈されている

配列とポインタ

- ウォッチウインドウで p = array と設定した後に、以下のシンボルの評価をしてみる
  - array, array[2], *array
  - p, p[2], *p
  - &p, &array
- どちらも同一？（見た目的には）
- 以下の式も試してみる
  - p = &y
  - array = &y

- ポインタ変数 p は変数 ⇦任意の値を代入可能
  ＝メモリ上に存在（番地がある）

- array が表すものは、配列の先頭アドレス⇦ 定数
  - （配列の各要素 array[0],array[1] などには任意の値を代入可能）

各演算子(*, &, [])を使用した際の型の確認

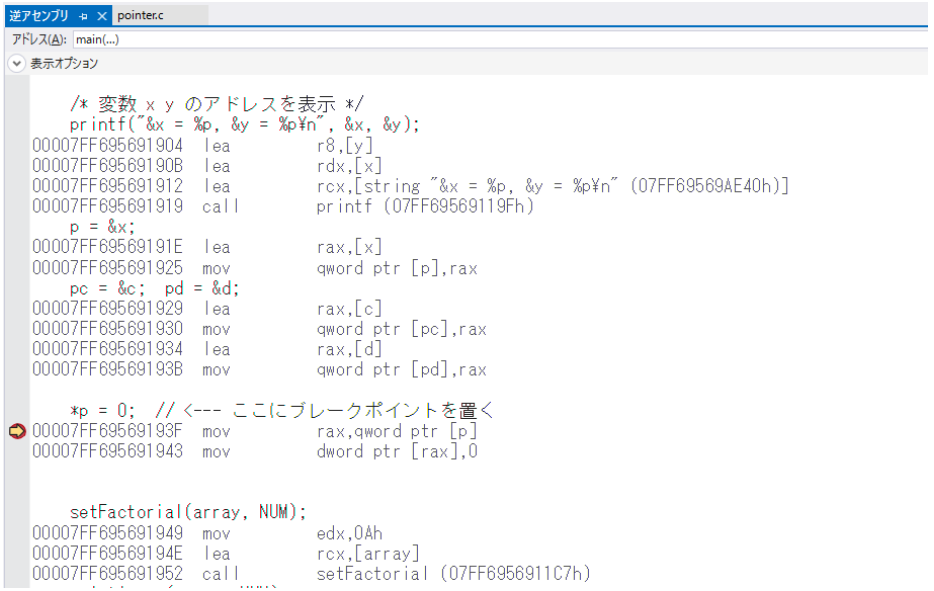
- int x; int array[10]; int *p; のとき、以下の式のの表す「型」を確認して表の空欄に入れる
  （ウォッチウインドウの[種類]フィールドを参照）
  - x, &x, *x, x[2], array, &array, array[1], *array, p, &p, *p,p+1, p[4]

int**	int*	int	評価不能（エラー）
		x	
	p		
		array[1]	

(発展) 関数名の評価

- ウォッチウインドウに main, printArray 等の関数名を入力し、どのように評価されるか調べる

- 逆アセンブルの表示
  - デバッグ中に、[デバッグ]メニュー→[ウインドウ]→[逆アセンブリ]を選択
  - 逆アセンブリウインドウで、コンパイル後の機械語命令を確認できる



発展：配列の関数へのひきわたし

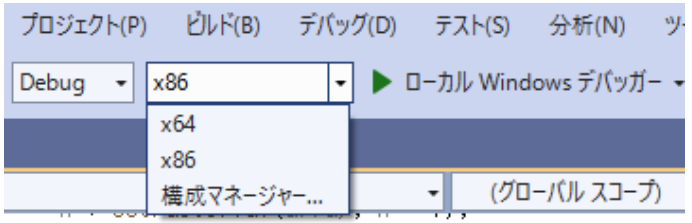
- このプログラムで使用されている２つの関数の引数は以下のようになっている
  - int setFactorial(int array[], int n)
  - void printArray(int* p, int n)

⇨それぞれの関数にブレークポイントを置き、引数の値について調べてみる

- C言語の関数の引数では、配列・ポインタのどちらの書き方をしても、ポインタとしてコンパイルされる
- C言語では配列全体を値渡しすることはできない
  - 先頭アドレスのみが渡される
  - （配列を構造体のメンバにすれば値渡し可能）

32ビットアプリケーションと64ビットアプリケーション

- 現在のWindows OS は64ビットOS
  - 4Gを超えたアドレス空間を扱える
- 64ビット Windows上では32ビット・64ビット双方のアプリケーションを起動することが可能
  - 双方の違いは、扱えるアドレス空間の大きさ
- Visual Studioでは、ビルドの種類を変更することで生成するアプリケーションの種類を変更できる
  - X64: 64ビット　　x86：32ビット



- ビルドの種類を x86からx64に変更してリビルドしたのちに、以下の変数の大きさを調べよ
  - **x　y　c　d　p　pc　pd**

文字列と関数(C言語)

- C言語には**文字列型**という型は存在していない
  - ⇒文字配列へのポインタを文字列として扱っている
    つまりメモリ上の文字の列の先頭アドレスのみの情報
    文字列がどこで終端するかの目印が必要：null文字 ('¥0')
- 関数への文字列の引き渡しの確認
  - 以下の部分に新規にブレークポイントを設定し、指示に従いながらトレース実行を行い挙動を確認する

```
55    //以下、文字列演習用
56    char msg1[] = "hello";
57    char* pmsg = msg1;  //ここにブレークポイントを設定
58
59
60    print(msg1);  //配列先頭アドレスの引き渡し
61    print(pmsg);  //ポインタを媒介したアドレスの引き渡し
62    print("konnichiwa");    //文字列リテラルの引き渡し
63    print(NULL);
64
```