プログラミング演習I
デバッガ演習

バグとデバッグ

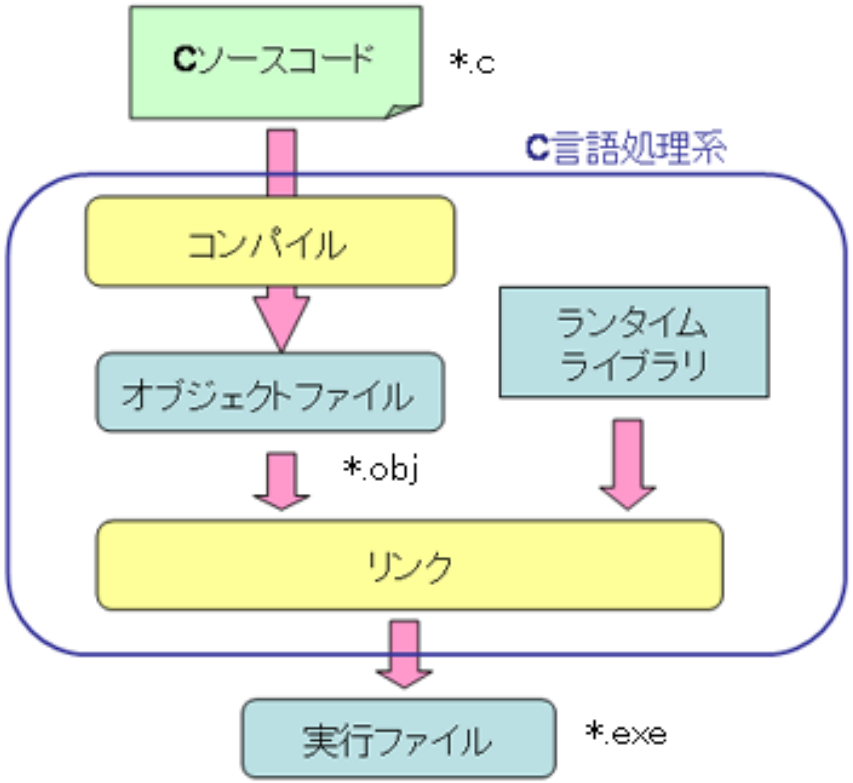
- コンピュータプログラムに潜む不具合のことをバグと呼ぶ。実行すると強制終了してしまうように、非常に明確に具現化するバグもあれば、正常に動作しているようで実は計算結果が誤っていたり、普段は正常に動作しても特定の入力に対しては正しく処理できなかったりといった、なかなか発見しにくいバグもある。

- プログラムの開発は、プログラム設計、コーディング、コンパイル、リンク、そして実行の手順を踏んで行われる。おのおのの段階で発生したエラーがそのまま実行ファイルに残ってしまうと、それがバグとなり、当然のことながらプログラムは正しく動作せず、大切なデータを破損するなどの最悪の結果を招くこともある。

- プログラムからバグを取り除く作業をデバッグと言い、それを支援するプログラムがデバッガである。

ビルドから実行までの過程

バグの無いプログラムを作成することは、とても重要なことであり、かつとても大変な作業である。やみくもに場当たり的な処置を施しても事態は改善しない。まず、プログラム開発の過程を整理した上で、どの段階でどのようなエラーが生じるかを、しっかりと理解しておいてほしい。



エラーの種類

- **設計時のエラー**
  - 仕様のエラー

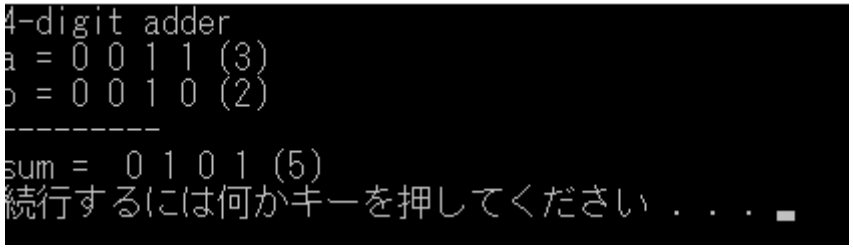
    プログラムの外部仕様を設計する段階で発生するエラー。入力される値の範囲の予想を誤っているなどの場合である。
  - アルゴリズムのエラー

    正しい仕様を設計したものの、その仕様を実装するのに用いるアルゴリズムにエラーが含まれる場合がある。
- **コンパイル時のエラー**
  - コーディングを終えたプログラムを、コンパイラが翻訳する時に発生するエラー。構文のエラーや、代入や関数呼び出し時の変数型の不整合などがある。
  - コンパイラは言語仕様に基づいた構文規則にしたがって解釈・翻訳していくので、記述したソースコードがこれにのっとっていない場合にはエラーメッセージを出力し、翻訳を中断する。コンパイルエラーを防ぐには、まずコンパイルに使用する言語の仕様を、しっかりと理解しておく必要がある。
  - また、C言語処理系でのコンパイルは、プリプロセス、コンパイルの 2 段階で行われる（課題 2 の資料参照）ことを理解しておくことも重要
- **リンク時のエラー**
  - 呼び出す関数名の記述を間違えていたり、プログラムに必要なライブラリをリンカに指定していない場合に発生する。前者の場合には、関数プロトタイプ宣言を正しく使用することにより、防ぐことができる。
- **実行時エラー**
  - 完成した実行ファイルを動作させたときに生じるのが実行時エラーである。

デバッガ実習１　（青字の部分を操作します）

- **準備**
- 以下のサンプルをディレクトリごとコピーする
  - **P:¥学部授業関連¥2017年度前期¥プログラミング演習I¥debugger**

- Adderプログラム
  - 課題１で作成したプログラムと類似した４ビット加算器のサンプルです
  - adder フォルダ内の adder.sln をダブルクリックしてソリューションを開きます
  - 正しくビルド・実行ができることを確認します



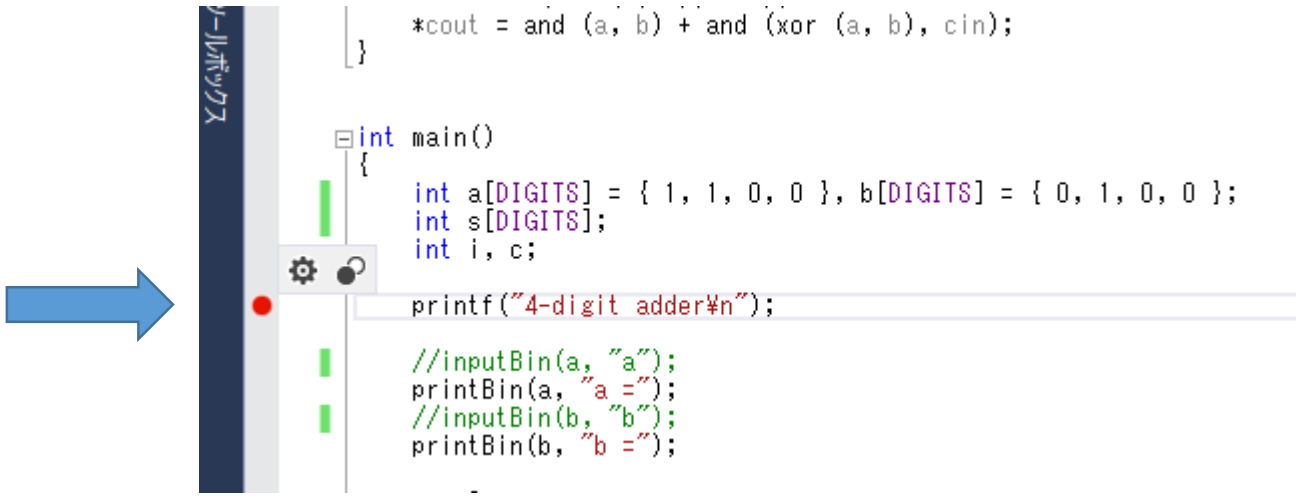
プログラムのトレース実行

- プログラムを，ソースコードレベルで１行ずつ実行する機能
- プログラム実行中の変数の値や，メモリの内容などを確認しながら進めることができます

ブレークポイント

- プログラムの実行を一時停止させるポイントのこと
- VSソースコードエディタの左側，グレーの部分をクリックすることで，ブレークポイントの設定/解除ができる

- ソースコードの任意の場所に，ブレークポイントを設定/解除できることを確認
- 図の位置にブレークポイントを設定



デバッグの開始

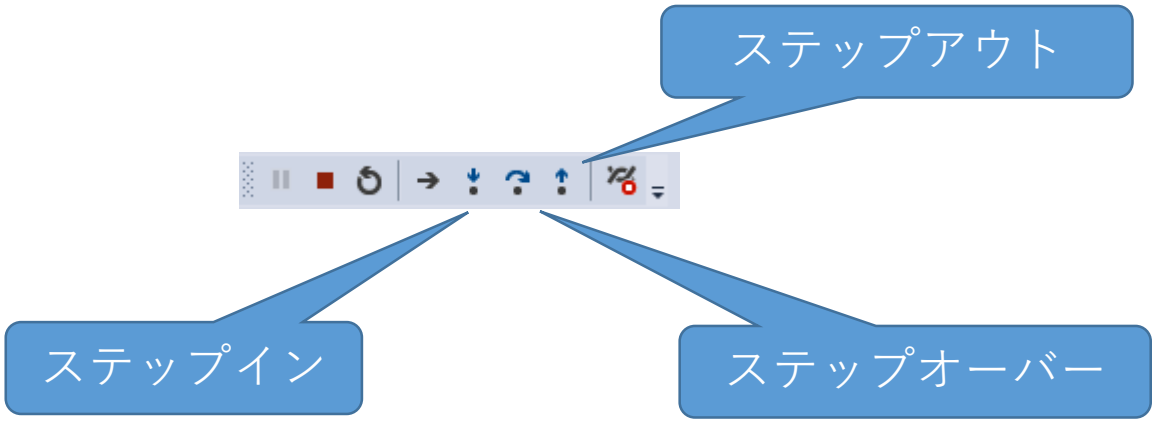
- ブレークポイントを設定したら，デバッグを開始します
  - [デバッグ]メニュー-> デバッグの開始
- プログラムが実行され，ブレークポイントで一時停止していることを確認
  - 次に実行する行に，黄色い矢印が表示されるのを確認

プログラムのトレース

- [デバッグ]ツールバーが表示されていることを確認

  - 表示されていない場合，ツールバー領域の空いている部分を右クリックして，[デバッグ]にチェックを入れる

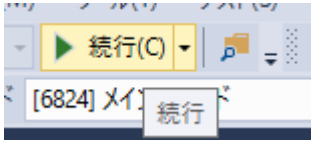
[デバッグ]ツールバー



- **ステップ実行 （ステップイン）[F11]**
  - 実行を一行すすめる。関数呼出の場合には、関数内部に入る。
- **関数単位で実行（ステップオーバー）[F10]**
  - 実行を関数単位で実行する。
- **関数から抜ける （ステップアウト）(Shift+F11)**
  - 現在実行している関数から抜ける。

トレース実行

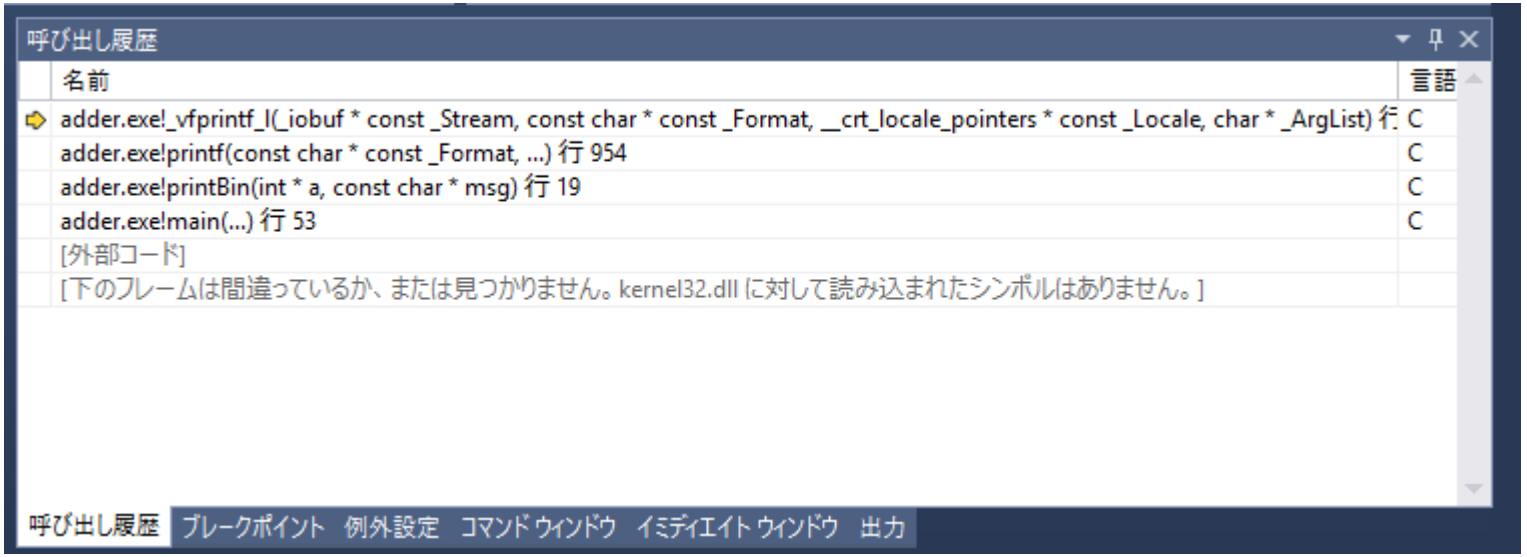
- adder プログラムをトレース実行してみましょう
  - ステップオーバーを使ってプログラムをトレース
    - printBin 関数を実行すると，コンソールに表示されるのを確認
    - main関数の最後まできたら，続行ボタンを押してプログラムを最後まで実行させるか，[デバッグの中止]をクリックしてデバッグを終了させる



  - ステップインを使ってプログラムをトレース
    - printf 関数にステップインするとどうなるかを確認
  - ステップアウトの動作を確認
  - 自由に何度もトレース実行をして，プログラムの挙動を確認してください

呼び出し履歴（コールスタック）

- 現在実行中の関数が，どのような順序で呼び出されているかを確認できます
  - （表示されていない場合には，[デバッグ]メニュー->[ウインドウ]から[呼び出し履歴]をクリック
- 関数名をクリックすると，エディタは呼び出し位置へジャンプ

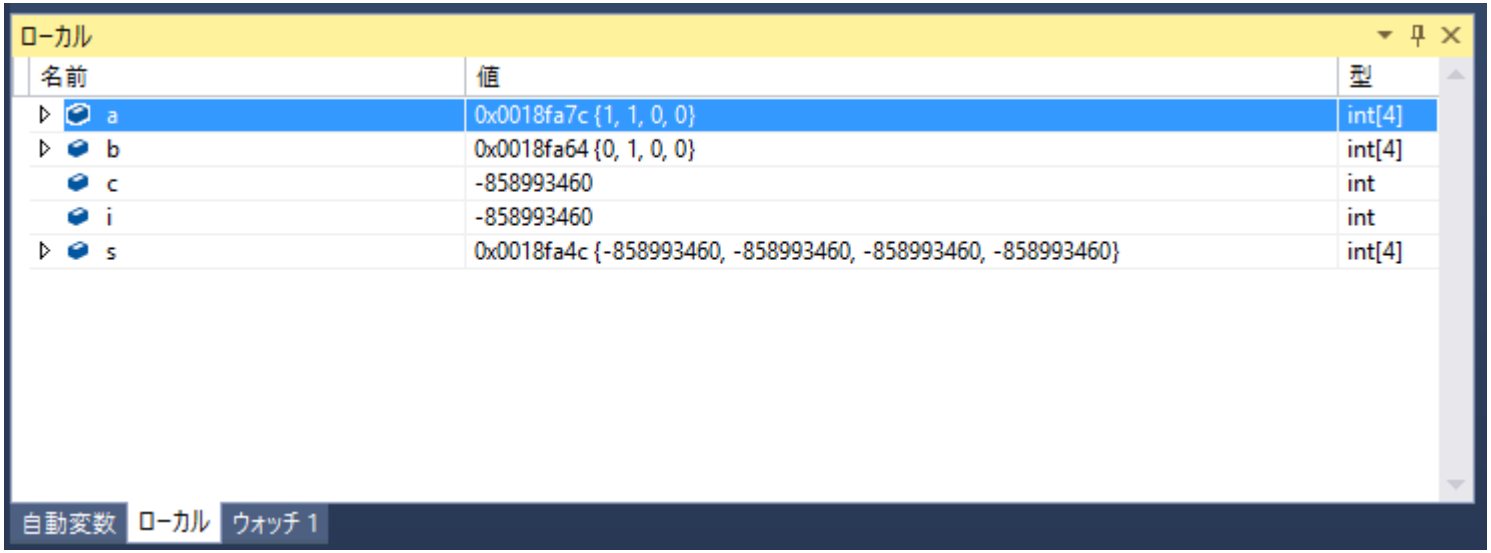


mainも関数，ではだれが呼び出すの？(課題1 テキスト 9ページ)

- 呼び出し履歴ウインドウを右クリック
  - -> [外部コードの表示]　にチェックを入れる
  - main 関数がどこから呼び出されているのか確認しよう

変数の値を確認

- 自動変数ウインドウ，ローカルウインドウ
  - 実行位置のスコープにある変数の型や値を表示



- 変数の値の変化を確認しながら，トレース実行をしてみよう

ウォッチウインドウ

- 自動変数・ローカルウインドウと同様の機能
- 自分で任意の式を入力して，それがどのように評価されるかを確認することができる
- ウォッチウインドウに以下の式(赤字)を入力して，どのように評価されるか調べてみよう. なぜそのように評価されるか考えてみよう

  - 変数名
    - i
    - &i
    - a[0]
  - 定数式
    - 2 / 3
    - 2 / 3.
    - (double)2/3
    - 1 / 3 * 6
  - sizeof 演算子
    - sizeof i
    - sizeof(int)
    - sizeof(double)
    - sizeof(char)
  - #define した値
    - DIGITS
  - 三項演算子
    - 1 ? 2 : 3
    - 0 ? 2 : 3
  - 関数名
    - main
    - fulladder

三項演算子

- □ ? △ : ○
  - □が true なら△，false なら○

```
a = i > 0 ? 4 : 10;
```

// i>0 なら a = 4，そうでなければ a = 10

以下と同値

```
if (i > 0) {
    a = 4;
} else {
    a = 10;
}
```

ブロッキング関数のトレース

- scanf 関数を呼び出すと，ユーザーが入力を行うまで呼び出し側に制御は戻ってきません
- デバッガで実行中でも同様で，scanf関数にステップインを行ったら，コンソールウインドウから入力をする必要があります

- inputBin 関数のコメントアウトを外してリビルドし，デバッガ上でトレースしてみましょう

再帰を用いて階乗を求めるプログラム

- ここから別のプログラムをデバッガで動かしながら次のことを学びます
  - 実行時エラーの対応
  - スタックフレームについて

- debugger フォルダ内の factor から factor.sln を開く
  - ビルドが通ることを確認する

  - factor は，再帰呼び出しを用いて階乗を求めるプログラムです

- factor プログラムのデバッグを開始
- コンソールに「階乗を計算する値を入力：」と表示されるので，９と入力

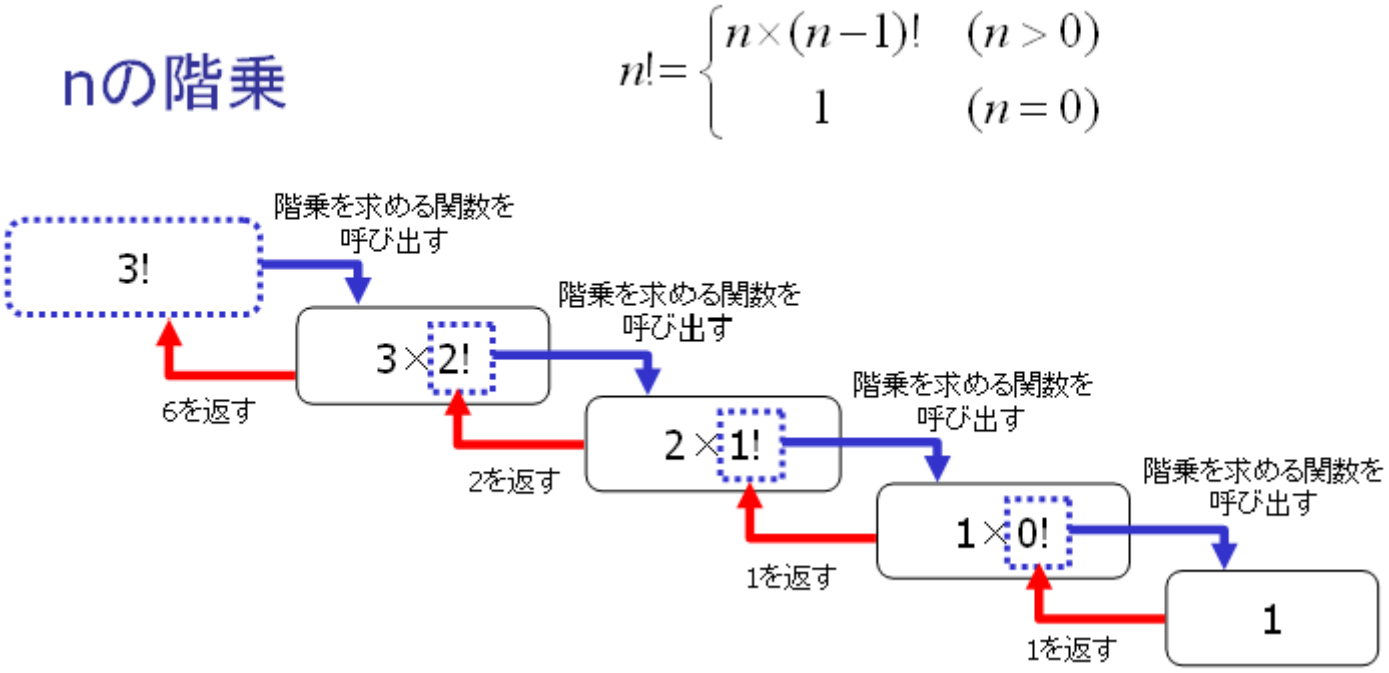
実行時エラーの対応

- 実行時エラーが出た場合の処理
  - ランタイムがエラーメッセージ（アサーション）を出している場合，**[再試行]**を押してデバッガに処理を戻す
  - デバッガが，ブレークポイントが発生したメッセージを出すので，**[中断]**を押してプログラムを一時停止させる
  - [呼び出し履歴]を参照して，エラーの発生個所を特定する

- 上記の手順で，バグのあるところを探し，修正する．
  （スクリーンで説明します）

再帰呼び出し

- 関数が，直接あるいは間接的にその関数自身を呼び出すこと
- 複雑な処理を単純なコードで表すことができる場合がある



呼び出し履歴による再帰の確認

- 「ここにブレークポイントを設定」の部分にブレークポイントを設定
- 呼び出し履歴を参照しながら，ステップイン実行を行い，factor関数が再帰的に呼び出されている様子を確認する
- 呼び出し履歴ウインドウで右クリックを押し，[パラメーター値の表示]にチェックを入れると，引数の値も確認できる

スタックフレームの観察

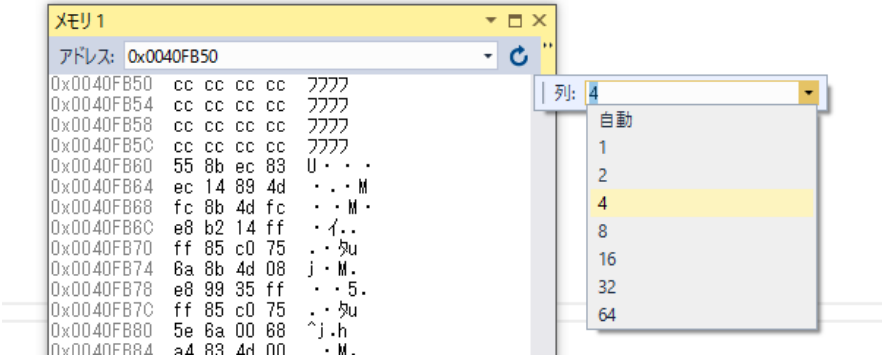
- スタックフレーム
  - 関数が呼び出された際にスタックに積まれ，関数のローカル変数などのが格納される
  - 課題１テキスト 10ページ Cのプログラム領域 (4)スタックフレーム

- スタックポインタ
  - スタックの先頭アドレスが格納されているレジスタ
  - Intel x86系プロセサでは、espレジスタが使用されている

- Factorプログラムを再起動しブレークポイントで停止させる

スタックフレームの観察

- メモリの状態を参照するメモリウインドウを表示させる
  - [デバッグ]メニュー->ウインドウ->メモリ->メモリ1
  - 表示を見やすくするために
    列の幅を 4 に設定
  - [アドレス]欄に esp と入力して
    スタックポインタの指している
    アドレスを表示させる



  - 表示をスクロールさせて、スタックポインタよりも下位のメモリ領域を表示させる
  - ステップイン実行を行い、スタックフレームの変化を確認する
- 関数が呼び出されるごとに、引数やローカル変数がスタックに積まれて利用されている様子を観察しよう

スタックフレームの観察（補足）

- 関数呼び出しが発生する際のスタックの変化は一般的には以下のとおり
    1. 引数をプッシュ
    2. call 命令により、戻り番地がスタックにプッシュ
       呼び出された側にジャンプ
    3. ベースポインタ(ebs)の値をプッシュ
       （その後、esp の値をebsにコピー、引数やローカル変数にはebsからのオフセットを用いてアクセスする）
    4. ローカル変数で使用する分だけスタックポインタを進める（減算する）

- 関数から制御を戻る際には、上記と逆の変化がおこる
    1. ローカル変数の分だけ、スタックポインタを戻す（加算する）
    2. ベースポインタへポップ
    3. ret 命令により、戻り先番地がポップされ、制御がこの関数を呼び出した位置にジャンプ

    - 戻り値は、EAXレジスタに格納されて返される

- デバッグ->ウインドウ->逆アセンブル を表示させると、マシン語レベルでステップ実行できるので、余力のある人は実行してみる

本日のキーワード

- 意味のわからない単語は無いか，再確認

- デバッグ
- ブレークポイント
- トレース実行
- 呼び出し履歴
- ウォッチウインドウ
- sizeof 演算子
- 三項演算子
- 再帰呼び出し
- スタックフレーム
- ローカル変数
- スタックポインタ