

# ソフトウェアテストとデバッグ

---

担当：森田

# 実験内容

- 目的
  - ソフトウェアテストの役割を学習
  - テスト手法, デバッグ技法の習得
- 説明内容
  - ソフトウェア開発とテストについて
  - 具体的なテスト手法について
  - デバッグ技法について
- 課題
  - テストを実施し, 不具合箇所をデバッグする

# ソフトウェアの開発とテスト

---

# テストの重要性

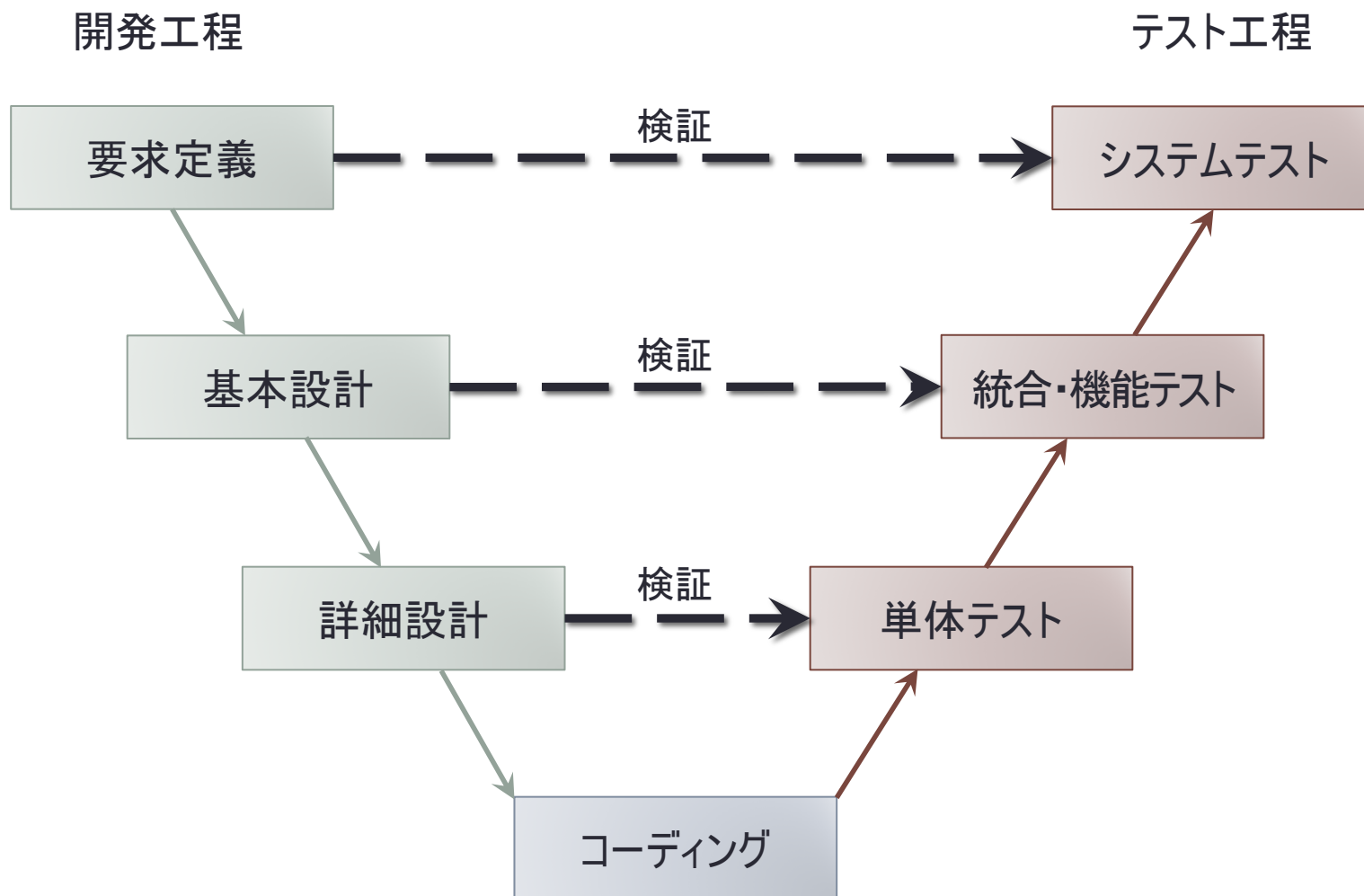
- 一般(市販)の製品においては、欠陥は許されない
- 市場に出す前に十分な検査(テスト)を実施



ソフトウェアについても同じ

- 最初から完全なソフトウェアの開発はほぼ不可能  
➡ テストによって不具合を最小限に抑える努力

# ソフトウェア開発の流れ: V字モデル



# テストの分類

- ホワイトボックステスト
  - ソフトウェアの内部構造に注目したテスト
  - 分岐命令, データ構造などの動作確認
- ブラックボックステスト
  - 内部構造を参照しない(ブラックボックス)
  - 入力と出力のみに注目
  - 要求どおり機能するかを確認

# ソフトウェアテスト手法

---

# ソフトウェアテストの基本

- 起こりうるすべての可能性を網羅的に確認
  - 実際は開発コストとのトレードオフ
- 低コストでより効率的（網羅性の高い）テストが必要



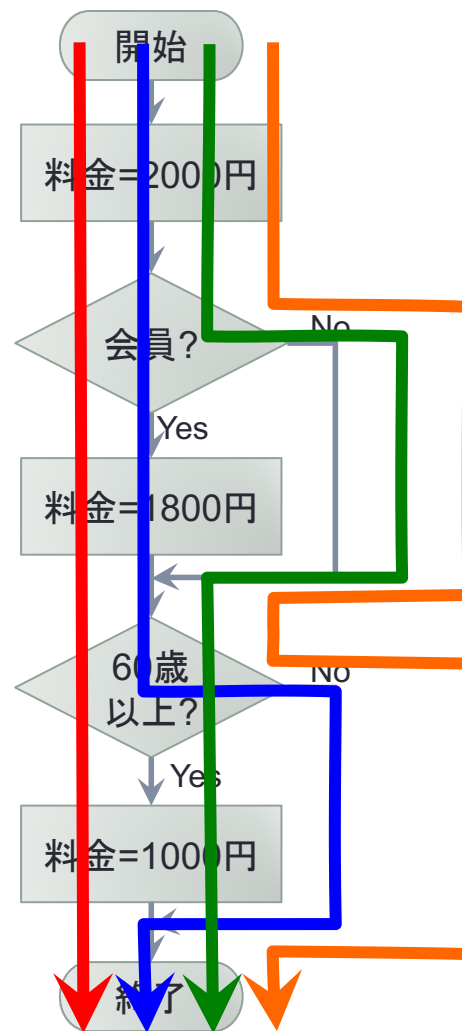
## テスト手法：

- 網羅性を高める目的でテスト項目を洗い出す



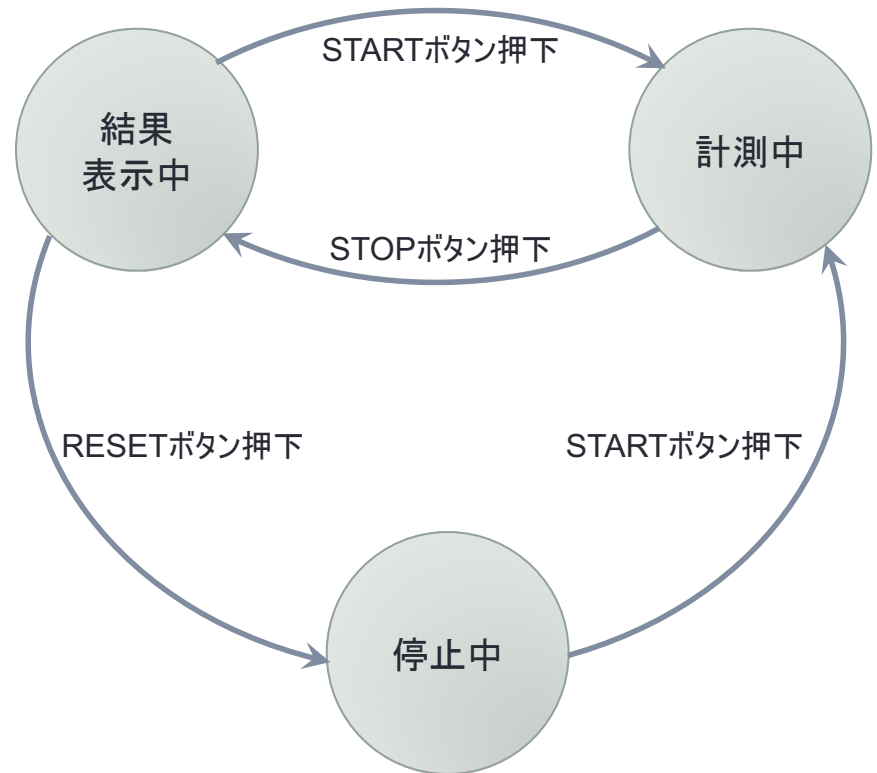
# 制御フローテスト

- 処理の流れ，順序を確認
- 手順
  1. フローチャートを描く
  2. 処理経路を抽出
  3. 抽出経路を通るようにテスト



# 状態遷移テスト

- 動作中の状態に着目
- 状態遷移図を作成
- 以下をテスト
  - 全ての状態を1回は通る
  - 全イベントを1回は発生
  - 全ての遷移を1回は通る



# 同値クラス・境界値テスト

- 入力される値の条件を確認
  - 内部(ソース)では条件に応じて処理を分岐するコードが記述されているため
- 確認項目
  - 同じ動作をする条件の集まりごとにテスト
  - 条件の境界となる値とその隣の値に対してテスト



無効な値の入力でもエラー処理で正常に  
終了するようにコーディングすべき

# デシジョンテーブルテスト

- 複数条件によって  
決定される動作を確認
- 手順
  - デシジョンテーブルを作成
  - 各ルールをテスト

	1	2	3	4	5	6
旧作	Y	n	n	Y	n	n
順新作	n	Y	n	n	Y	n
新作	n	n	Y	n	n	Y
まとめてレ ンタル	Y	Y	Y	n	n	n
60%OFF	Y	Y	n	n	n	n
40%OFF	n	n	Y	Y	n	n
20%OFF	n	n	n	n	Y	n
通常料金	n	n	n	n	n	Y

# テストドキュメントの作成

- 開発工程における情報伝達
- ソフトウェアの品質状況の可視化, 劣化を防ぐ
  - 修正を施すことで別の不具合が埋め込まれることも...
- 種類
  - テストケース・テストログ(講義資料表2参照)
    - テスト実施に必要な情報を記載(条件, 入力, 結果など)
      - 各テスト手法によって洗い出されたテスト項目を記載する
    - 例題:「三辺の長さを入力して三角形の面積を求めるプログラム」の三辺の長さをテストするには, どんな条件が必要?
  - 不具合報告書(講義資料表3参照)
    - 不具合を抽出し, その原因や修正内容などを記載

# デバッグ技法

---

# デバッグ(debug)の基本

- バグ(bug)を取り除くこと
- **確認の原則**
  - どの部分がバグなのかを発見する必要がある
  - ➡「実際に正しい」動作をしているか順番に確認するしかない
- 効率的な作業の指針
  - デバッガ(debugger)を使う
  - トップダウンに調べる
  - 小さく始める
  - 二分探索を用いる

# gdb

- 代表的なデバッガプログラム
  - 実行中のプログラムの状態を制御・確認するツール
  - 任意の場所での一時停止
  - 変数の値の確認, 値の変化の検出
- 使うための準備
  - デバッグ対象のプログラムをコンパイルするときに,  
(デバッグ情報を埋め込むために)  
gccに **-g** オプションをつける
- 起動方法
  - `$ gdb デバッグ対象の実行プログラム`



# gdbの操作方法

- 実行と中断

- (r)un: デバッグ対象のプログラムの実行 (省略形はr, 以下同じ)
- Ctrl+Cキー: プログラムの一時停止 (中断)

- ブレークポイント

- (b)reak *関数名など*: その場所に到達すると自動的に中断

- ステップ実行

- (n)ext: 中断場所から1行分だけ進める
- (s)tep: 1行進めるが, 関数呼び出しがあれば中に入る

# gdbの操作方法

- 実行再開

- (c)ontinue: 中断場所から続きを実行する
- (f)inish: 関数の呼び出し元に戻るまで進める


- 変数の調査など

- (p)rint 式(変数名など): 式の演算結果, 変数の値などを表示
- set 変数名=値: 変数の値を変更する
- (w)atch 変数名: 変数の内容(値)が変わると自動的に中断
- (l)ist: 中断場所周辺のソースを表示

# gdbの操作方法

- 関数呼び出し履歴
  - (back)trace: 関数呼び出し履歴が表示される
  - (f)rame フレーム番号: 履歴内の関数に(確認のために)移動

# coreを伴うデバッグ

- coreとは
  - セグメントエラー(Segmentation fault)で異常終了したときの、プログラムの実行状態を保存したファイル
  - gdbに渡すことで、エラー発生時の状態が確認できる
- 使い方
  - `$ gdb エラーを起こしたプログラム core`  
 エラーを起こした場所で中断したような状態になっている

# エディタとの連携

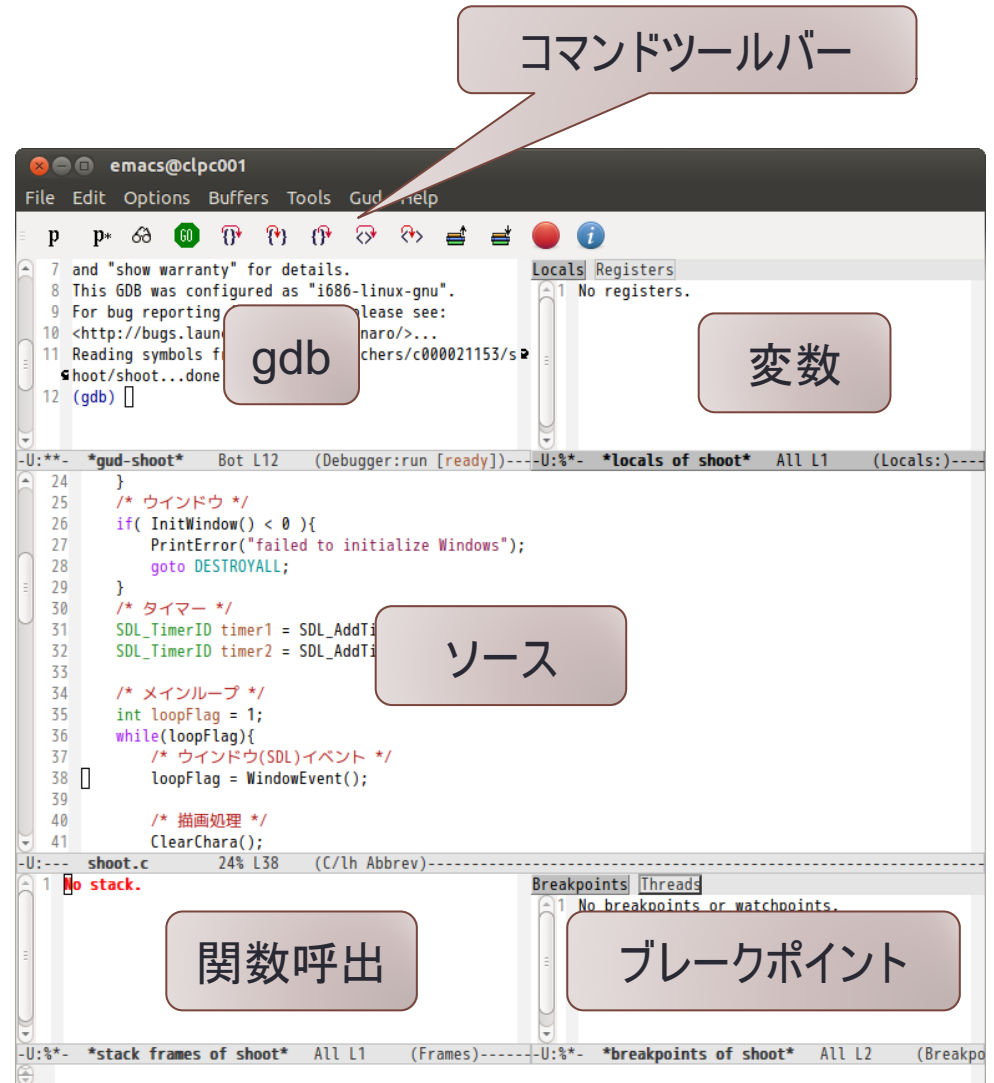
- デバッグはGUI的に使えた方が便利
  - 変数の値などを俯瞰的に確認できる
  - マウスクリックでの実行操作, 変数の確認ができる
- しかしgdbはコマンドラインツール



- エディタ(などGUIのツール)と連携させる
  - emacs
  - Visual Studio Code
  - etc.

# emacsとの連携

- emacsを起動  
(-nw じゃない方がいい)
- **M-x gdb** とタイプ



# Visual Studio Codeとの連携

- Visual Studio Code を起動
- デバッグボタンから開始
  - 事前に設定,  
拡張機能の  
インストールが必要

コマンドツールバー

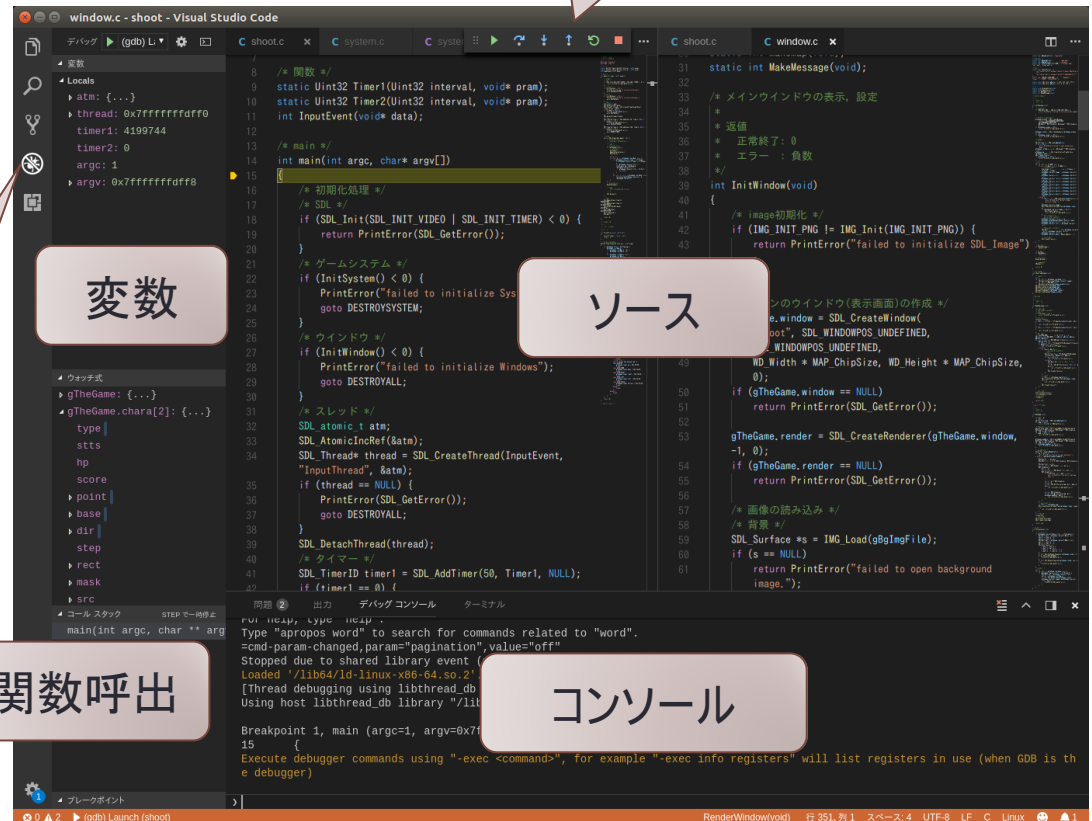
デバッグボタン

変数

ソース

関数呼出

コンソール



# 課題

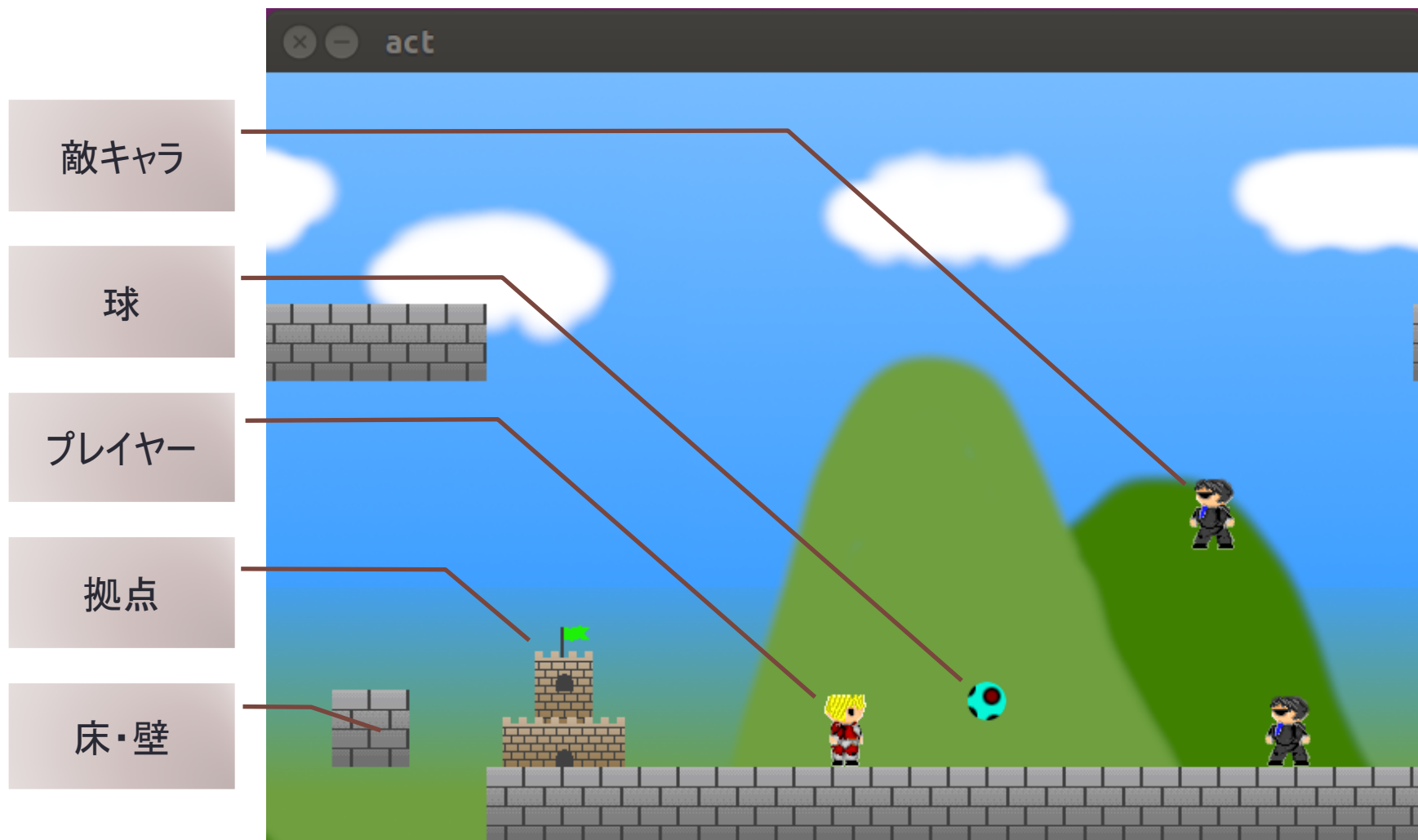
---



# 準備

- ~/.gdbinit に次の内容を記述しておく  
set logging on
- 環境変数 SHELL の内容確認, ない場合追記  
export SHELL=/bin/bash
- vscode.tar.gz をダウンロード, ホームディレクトリに展開
- game.tar.gz をダウンロード
- 適当なディレクトリに展開して make

# 課題のプログラム: アクションゲーム「act」



# 課題(詳細は講義資料参照)

- 課題1

- 課題のプログラムをテストするテストケースを作成せよ
- テストケースを元にテストを実施, テストログをレポートせよ

- 課題2

- テストログから不具合をピックアップし, gdbでデバッグせよ
- デバッグ後, 不具合報告書をレポートせよ.

- 課題3(取り組まなくても受理するが採点には含む. **再提出時は必須**)

- 課題のプログラムを改良せよ
- 改良箇所のテストも実施し, テストログ, 不具合報告書に追記せよ

# レポート提出物 (詳細は講義資料参照)

1. レポート(pdf形式, 以下を結果として記載)
    - テストログ
    - 不具合報告書
    - プログラムの改良内容
  2. デバッグ後のプログラムソース, 画像など一式 (make, 実行に必要なファイルすべて)
  3. デバッグログ gdb.txt
- 圧縮アーカイブにして manaba に提出
  - 提出 ✕ 切 : 6月18日(火) 12:50

# 大規模コーディングの基本

---

コラム

# プログラムの(基本的)構造

- 初期化処理

- メインの処理に必要なデータの準備
- (データ)ファイルの読み込み, メモリ確保, ライブラリの初期化など, コストの高い(時間のかかる)処理は, あらかじめ実施しておく



- メイン処理

- プログラムの目的である処理の実施



- 終了処理

- 初期化などで用意したデータの後始末  
(ファイルを閉じる、メモリ解放, など)

# 構造化

- プログラムの内容を，機能ごとに切り分ける
  - 一般には関数化する
  - 関数は，よく使われる動作をまとめるため，ではなく，機能を分けるために作る
  - プログラム内で一度しか使われなくても，機能が違うなら関数にする．main関数はほとんど関数呼び出しだけ，が理想
- データの取り扱い(データ構造)も機能ごと
  - ある一つの機能や項目を表すのに，複数の変数が必要ならば，まとめる(構造体を駆使する)
  - オブジェクト指向対応言語なら，クラスを駆使する

# エラーハンドリング

- プログラムは、異常が起きても、正常に継続・終了させる
  - セグメントエラーなどで落ちる、ようなことがあってはならない
  - エラーメッセージを表示するなどして、継続可能なら継続、無理なら終了処理に進めて、正常に終わらせる
  - プログラム使用者が間違った入力・操作をしても対処する
  - assertはバグ検出には有用だがエラー処理ではない
- エラーの処理方法
  - 関数の引数、戻り値をチェックして、エラーの場合
    - 値の変更で継続可能なら、適当な値に変更
    - ユーザ操作が原因の場合、再度操作を促す
    - エラー処理ルーチンから終了処理へ遷移



# 好まれないコーディング

- マジックナンバー
  - ソース中に定数(数値)を直書きすること. その数値
    - 数値の意味はわからないがプログラムは正しく動く, まるで魔法の数値
  - 他人が読んで, 意味不明になるような数値には名前を付ける(意味を持たせる)
    - `define`マクロ, `enum`(列挙定数), `const`修飾などを駆使する
- `goto`
  - プログラムの流れを強制的に変えることができるので, 可読性が極めて悪くなる
  - 基本的には使わず, 例外的な処理(エラーなど)にとどめるべき
    - 例外処理機構のある言語では例外処理機構を使い, `goto`は使わない