プログラミング言語論
第３回

構造化プログラミング

関数/名前とスコープ

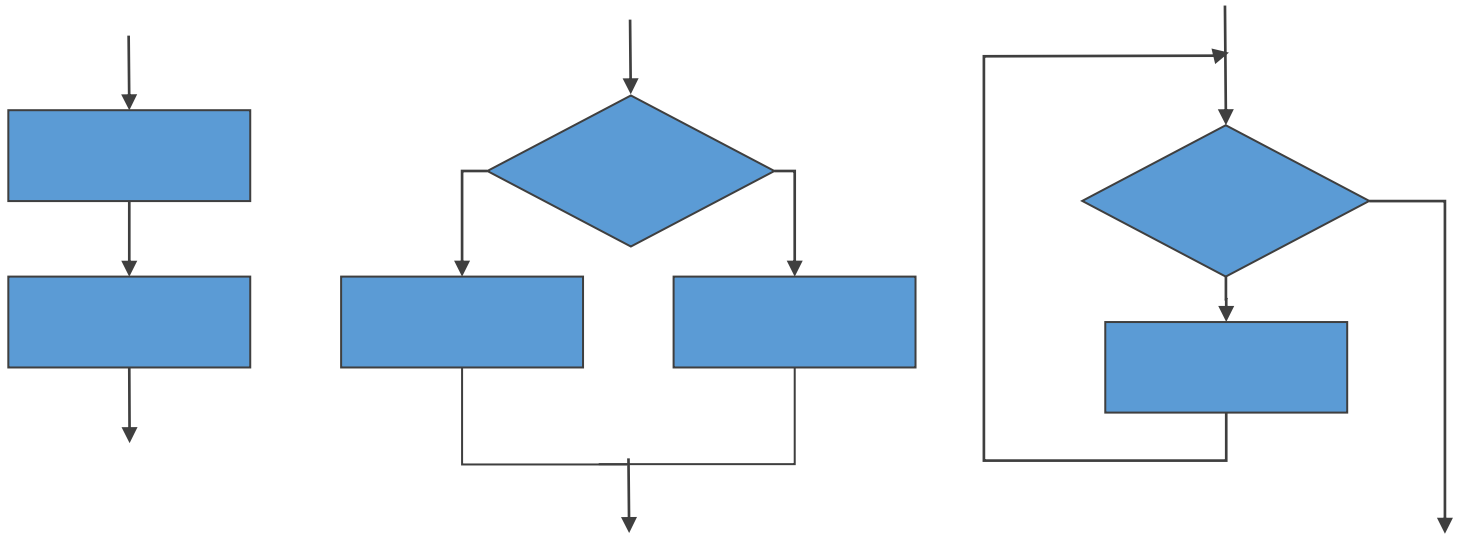
今週の内容

- 構造化プログラミングのおさらい
- 関数による段階詳細化
- 関数
- 名前とスコープ

おさらい: 構造化プログラミング

- 逐次実行（順次実行，順次，順接，順構造）
- 分岐 if-then, if-then-else
- 繰り返し（反復，ループ）　while, repeat, for

の基本構造でプログラムの基本構造を構成する



現代の手続き型言語にはすべてこの考え方が反映されている

関数

関数とは

**「コードの一部を，まとめて切り出し，
それに名前をつける機能」**

- 同じ意味：手続き，プロシージャ，サブルーチン，メソッド
- 何度も同じ命令を繰り返したいという要求は1949年のEDSACの時代から使用
- もちろん，関数がなくてもプログラミング可能
- 何度も同じ命令を繰り返し書くよりもコードが短くなり，理解が容易に（同じ処理の反復部分は関数にまとめる）
- もし無かったら長いコードは理解が大変．小さい部品の積み重ねのほうが理解が容易

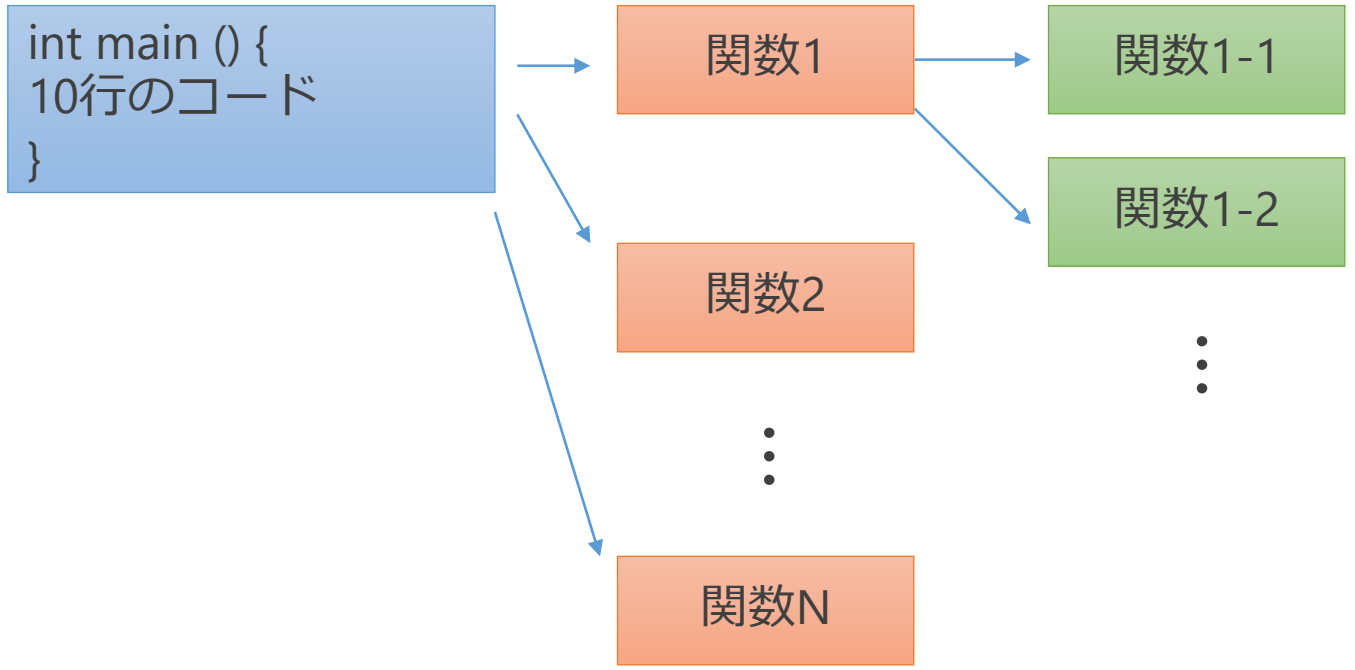
関数による段階詳細化

巨大なmain関数

```
int main () {
…
10000行のコード
…
}
```

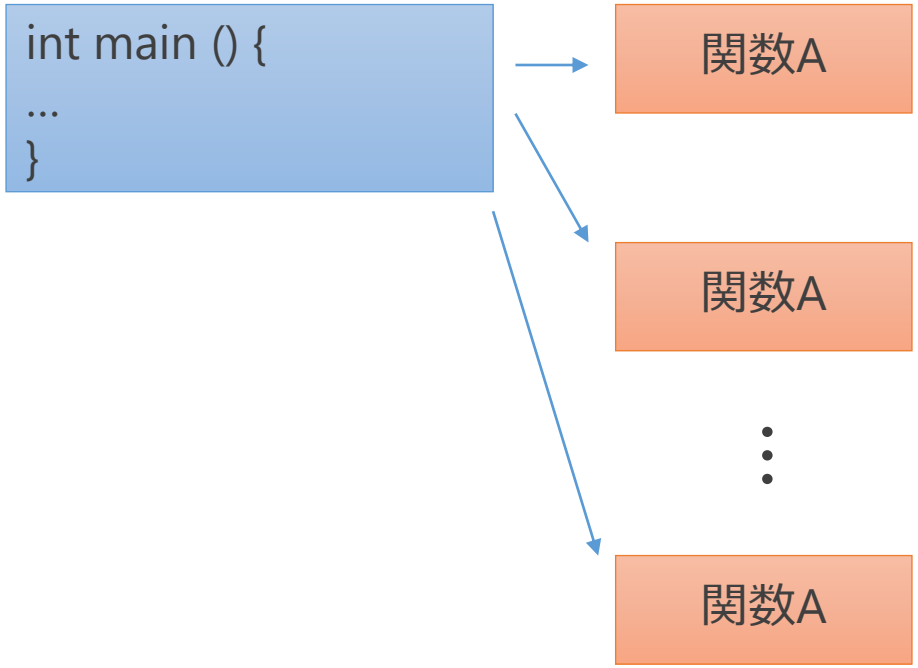
お勧めできない
スタイル

コンパクトなmain関数



関数に基づく段階詳細化

「ソフトウェア部品」としての関数



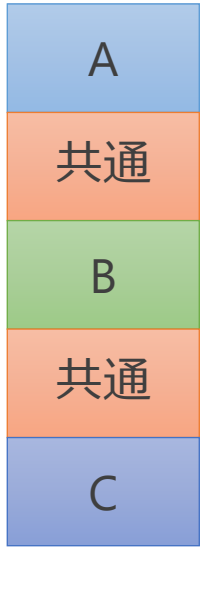
同じ処理をひとつの関数としてまとめる→再利用可能

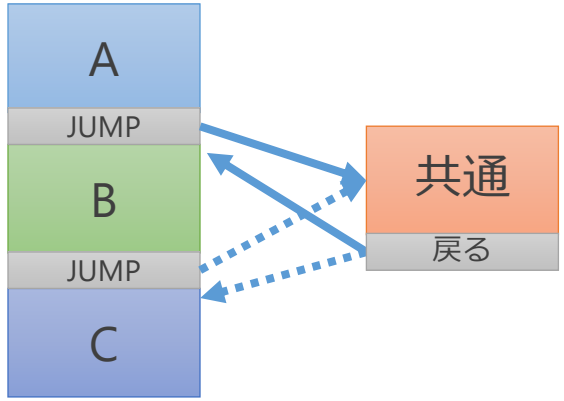
gotoで出来ないこと

- 前回、全ての制御文がgotoで表現可能なことを学んだ
- では，gotoで出来ないことは何？

「元の位置に戻ること」

処理の流れ





戻るgoto先を書き換えないと
同じところに戻ってしまう

goto に基づく関数の実現方法 1

1. 戻る場所を記憶する場所をあらかじめ用意する
2. 関数に入る前に，戻ってほしい場所を登録
3. 目的の関数にジャンプして命令実行
4. 登録先を参照してそこに戻る

問題点：関数Xを呼び出している最中に関数Yを呼び出すと，登録先の場所が上書きされてしまい，戻る場所がわからない迷子状態になる.

関数の実現方法２

1. 戻る場所を記憶する場所を**関数ごと**に用意する
2. 関数に入る前に，戻ってほしい場所を登録
3. 目的の関数にジャンプして命令実行
4. 登録先を参照してそこに戻る

問題点：関数Xから関数 x を呼び出すと迷子状態になる．つまり，再帰呼び出し(recursive call, 自分自身の呼び出し)が出来ない

関数の実現方法 3

1. 戻る場所を記憶する場所を**スタック**で用意する
2. 関数に入る前に，戻ってほしい場所を登録
3. 目的の関数にジャンプして命令実行
4. 登録先を参照してそこに戻る

要点：

- 再帰呼び出しが出来る現代の関数
- 引数もスタックを用意してあげてそこに入れれば完全

現在のコンパイラからみた関数とは

- 関数のコンパイラ的な説明覚えてますか？
  - コンパイラの意味解析あたりで出てきたはず
- アセンブラ的な関数の理解
  - 呼び出し
    - 引数をスタックにプッシュ
    - 関数が宣言してあるアドレスをコール
    - 呼び出し関数はスタックから引数を知る
  - 復帰
    - レジスタに戻り値を記入して復帰命令
    - 呼び出し側はレジスタから戻り値を獲得

命令も変数も，結局は全部メモリのアドレスのどこかに載っているという感覚があれば理解は簡単

再帰呼び出し

- もちろん無くてもプログラミングは出来る
- さまざまな場面でこれが使えると短くコードが書ける
- 関数型プログラミングでこれがないとプログラムがかけない

レポート１：関数

- 関数に引数を渡すとき，値渡しとポインタ渡しはスタックの消費量がどう異なるのか説明せよ．例えば，下記構造体を値渡し，ポインタ渡し，それぞれを用いて，何度か再帰呼び出しをする関数を作った場合，メモリ消費量はどうなるか述べよ．

```
Struct HeavyData
{
double data[10000];
double index[10000];
};
```

  - 調べ方のヒント：プログラムの最後をgetchar()でとめておいてUNIXのプロセスのメモリ消費を調べるコマンドを打てばOK．psなど．

レポート２：関数

- main関数のみでgoto文を使って擬似的に関数の機能を創意工夫して実現せよ．ここでいう関数の機能とは，同じ処理をまとめ，再度使えるようにすることである．
- どちらかのプログラムを作れ
  - 階乗を計算するプログラム，もしくはフィボナッチ数列を計算するプログラム
  - ただし，階乗を計算する場合， for文を使うのを禁止する．F(x)=F(x)*F(x-1)…とすれば関数を使わなくても良いため．
- ヒント：階乗の場合
  1. スタックを配列で作成
  2. スタックに引数をコピー
  3. 関数ラベルにジャンプ
  4. 再帰
  5. 最後にスタックデータの巻上げ
  6. おまけでi<1のエラー処理も作るとなお良い
  - （答え），さっぱりわからないときはここを見ること．自分で作れない場合は，ここのコードの解説を書くこと．
    http://fukushima.web.nitech.ac.jp/fukushima/lecture/prog/index.html

階乗のコード

実現する側のコード

```
#include <stdio.h>

int FAC(int n)
{
	if( n < 0)
		return -1;
	if( n == 0)
		return n;
	return n * FAC(n-1);
}

int main(int argc, char** argv)
{
	int n;
	scanf("%d", &n);
	int ret;
	ret = FAC(n);
	if(ret == -1)
		printf("error¥n");
	printf("%d¥n", ret);
}
```

```
#include <stdio.h>

int main(int argc, char** argv)//for version
{
	int n;
	scanf("%d", &n);
	if(n == -1)printf("error¥n");
	int ret=1;
  for(;int n==0;n--)
	ret*=n;
	printf("%d¥n", ret);
}
```

ヒント：アセンブラ的にはこのコードは答えに近い結構近いので，このコードをfor無しで書いてみてから考える．引数と戻り値がn,mしかないと限定してスタックを作った関数にかなり違いコードになる

名前とスコープ

手続き型プログラミングでの整理整頓

- 一般に人間の認識・理解にとって整理整頓は重要
  - 変数，配列などのデータは構造体(struct)にまとまる
  - 基本構造，数式等は手続きは関数にまとまる

- しかし．．．
  - 構造体と構造体のまとまりは？
  - 関数と関数のまとまりは？
  - データと手続きの関係は分からず，ばらばらのまま
  - 複数の手続きにまたがる変数はグローバル変数に

例：顧客管理システム

- 顧客情報を管理するシステムの構築
  - 名前，住所，所属，電話番号情報は構造体で整理
  - 便利に使うために，きっと作る関数群
    - 確保：init_customer(struct Customer*, char* name, char* addr, char* tel)
    - 解放：delete_customer(struct Customer*)
    - 追加：add_deal_customer(struct Customer*, char* deal_name, char* deal_ammount)

- でも．．．
  - どのタイミングでどの関数を呼べばいいかわからない
    - どんな関数があるかなんて忘れてしまうし，知らないかもしれない．
  - 会社ごとに構造体の中身（追加情報）を変えたいかもしれない
    - 名前だけ確保して使うかもしれない，一部だけ解放して使ってしまうかもしれない
  - 値がいつどこでどのタイミングで書き換わるか全く保証が無い．
    - 初期化していないかもしれない．開放していないかもしれない
  - **どこまで確認すれば，このコードが構造体，関数が関連していて，どうしたら安全に動くのか確認しないといけない範囲が分からない**
  - コードの独立性

整理整頓：モジュール化，部品化の手段

- 統合：構造体，関数化，**名前**
- 分離：アクセス権の制御，**スコープ**

現場の運用

- 命名規則，社内コーディング規約による運用

- 名前の先頭に共通の記号をつける
  - cse_set_student_id()
  - cse_create_student_id()
  - cse_release_student_id()
  - Struct cse_student

- ファイルスコープによる管理
  - 同一ファイル内の関数と構造体は同じグループであると勝手に決める

名前

- 昔は番号ですべての名前をつけていた
  - 何番地が先頭の関数，も何番地の変数など，アドレスの番地で変数，構造体，関数を表現
  - １２３番の関数呼んで！変数４５６番に１足しておいて！
- 変数，関数に名前をつけることで理解が容易に
  - 他にも，構造体，ファイル，ヘッダ，マクロなど
  - **予約語**：main, char, short, int, long, float, double, if, while, elseなどはすでに定義されているので使えない
- 名前の付け方
  - a = b*c
  - Kyuuryo = Jikyuu *hour

名前の衝突問題

- 昔のプログラムは全てグローバル変数
- １つのプログラムに同じ名前は二度と使えない
  - 紙を部屋に貼り付けて管理
- 右のプログラムの
  何がいけない？
- 名前の衝突の管理を全部
  人間がするのは**ツライ**

```
int i;
void loop()
{
        for (i = 0; i < 100; i++)
        {
                process();
        }
}
void process()
{
        printf( “hello¥n” );

        i = 0;
}
```

１００回ハローというプログラムのつもり

名前の衝突の対策

- 長い名前をつける
  - 例：cse_set_student_id_cs_2016()
  - 命名規則を遵守する
  - 衝突の可能性は減るが完全な解決にはなっていない
  - 長すぎると読む気がしない
  - 短い変数名も使いたいが短いと何か分からない
- スコープ
  - 名前に有効範囲を決めて複数の名前を使えるように

命名規則，ネーミング規則・規約

- キャメルケース，キャメル記法
  - 各要素の最初を大文字で記入
  - SetTodayData：アッパーキャメルケース
  - setTodayData：（ローワー）キャメルケース，初めだけ小文字
- スネークケース，スネーク記法
  - 各要素をアンダースコアでつなぐ
  - set_today_data
- チェインケース，チェイン記法
  - ハイフンでつなぐ
  - set-today-data（C言語ではできない）
- ハンガリアン記法
  - 目的を名前に含めるアプリケーションハンガリアン
  - 変数のデータ型を名前に含めるシステムハンガリアン

ハンガリアン記法

- マイクロソフトのハンガリー人，**チャールズ・シモニイ**が考案
- システムハンガリアン
  - bDataflg：ブール型のフラグ
  - iNumber：int型の整数
  - fNumber：float型の整数
  - gNumber：グローバル変数
- アプリケーションハンガリアン
  - iValue：入力変数
  - oValue：出力変数
- 現代では使われなくなり始めている．特にシステムハンガリアン
  IDEが発展しているためマウスオーバーレイで型はすぐ分かる

スコープ：名前の有効範囲

プログラミングを始めたばかりのころ，
一度宣言した名前はどこでも使えると思って
こんな関数を書いたこと無いですか？

```
int main()
{
  int a = 10;
  int out = func(30);
}
int func(int b)
{
  return a+b;//上のaを使う
}
```

「スコープ」が外れているためダメ

スコープとは

- スコープとは，変数や関数の名前が参照可能な範囲
- 範囲外では名前はかぶってもOK
- 見える場所を制限：狭いほど管理しやすい

```
int a=30;
void func()
{
        int a=10;
        printf(“%d¥n”,a);
}
void main()
{
        int a=20;
        func();
        printf(“%d¥n”,a);
}
```

グローバルスコープ

- 全部の変数名，関数名が唯一であり，どこからでもアクセス可能！
  - 初心者が考える一番自然な考え方
  - 関数やポインタなどで悩む必要が一切無い！
    - 昔は名前の対応用が壁に貼られていた.
    - 昔は全部グローバル変数
    - 名前の衝突回避に長い名前をつける
- デメリット
  - 人間は長い名前を覚えられないし，なにより見づらい
    - setSeisekiProgrammingLanguageTheoryPart1Kadai1()
    - for (int index_loop_of_programming_language_theory_exsample=0;
      index_loop_of_programming_language_theory_exsample<100;
      index_loop_of_programming_language_theory_exsample++

「スコープの誕生」

例：関数によるスコープ

下記プログラムの変数aはmainとfunc中で名前が衝突しているように見える．
しかし，これは何も問題がなく，それぞれ別の変数を意味している．

```
int a=30;
void func()
{
        int a=10;
        printf(“%d¥n”,a);
}
void main()
{
        int a=20;
        func();
        printf(“%d¥n”,a);
}
```

スコープの目的

- 名前の衝突の防止
  - 実際にコードを書いていると，プログラム中の関数・変数名は，1日で簡単に100個を超えます．
- 不用意な変数へのアクセスを防ぐ
  - もしグローバル変数にint iなんて宣言があったら．．．
- 適切な名前をつけるため
  - 意味のある適切な長さの名前は重要

スコープにもたくさんの種類が！

- 動的スコープ
  - 静的スコープ＋呼び出し元の親のスコープも参照できる.
  - 関数が呼び出し元で展開されたかのようなスコープが構成される.
  - アクセス可能な変数を全て使えるように展開
- 静的スコープ
  - ブロック内（{}で囲われた場所）で宣言された変数は外部から操作できない
  - コンパイラ型の言語（ C言語も含む）はほとんど静的スコープ
  - 別名構文スコープ（構文だけから決定できるため）
- 名前空間(namespace)
  - 参照範囲を明示的に記述
  - C＋＋で使用可能 「using namespace std;」
- 他にも
  - ファイルスコープ：そのファイルでだけ有効なグローバルスコープ
- 制約の厳しさ
  - グローバルスコープ→動的スコープ→静的スコープ→名前空間

動的スコープ


```
A {
  print x
}

B {
  var x
  call A  // Aの中からxを参照することができる
}

C {
  var y
  call A  // Aの中からxを参照することはできない
}
```

呼び出し元の変数もアクセス可能

スクリプト言語は原理上こうなりやすかった

動的スコープの誕生

- もし関数に引数がなかったら？（昔は無かった）
- スコープの実現方法
  - 関数の初めにグローバル変数からローカル変数にコピー
  - 関数の終わりにグローバル変数へローカル変数コピー

```
void func()
{
    CopyFromGlobal{...}
    何か処理;この間はローカル変数として使える
    CopyToGlobal{...}
}
```

動的スコープの問題点

- 変数を書き換えてから別の関数を呼ぶと，呼び出された関数に影響が及ぶ
- 関数A中である変数iが変わらないように動的スコープを作る
- 関数Aの処理の途中で関数Bを呼び，その中でも動的スコープをつくってiを保護
- でも，関数Bを抜けるときにグローバル変数に値を戻すため，グローバル変数iの値は変更してしまっている.
- （あとで演習）

静的スコープ


```
A {
  var x;
}

B {
  var x; // A内のxとは別物

  C {
    var y; // Cの内側からしか見えない
  }
}
```

みんな知っているスコープはこれ

中括弧によるスコープの作成

- 関数内にスコープが作れる
  - （C++,C#などや最近のC言語C99 ）

```
void func()
{
        int a;
        {
                int a;←上のaと異なる変数

        }
}
```

最近の考え方：**変数のスコープは可能な限り小さく**

最近のC，C++では．．．

- 変数を**利用する直前に宣言**することが推奨されている←変数のスコープの最小化
- 例えば、ループ変数はfor文の中で宣言する

```
for (int i = 0; i < n; i++) {
.....
}
```

もちろんループ変数iのスコープはこのループ内のみ

静的スコープは完成計か？

- グローバル変数にxを宣言し，ローカル変数にもxを宣言したらもうxにはアクセスできない.
- 言語によって対応が違う

```
int x;
void func()
{
 int x;
}
```

名前空間

- スコープに名前がつけられる機能
  - C++，C#など
  - Javaはパッケージにより実現

```
namespace cse
{
        void get(){printf(“cse¥n”);}
}
void get(){ printf(“global¥n”);}
void main()
{
        get();
        cse::get();
}
```

スコープの歴史

- 1991年 Perl4 変数にlocalをつけると動的スコープに
- 1994年 perl5 変数にmyをつけると静的スコープに
  - 現代では，use strict指定して実行すると，動的スコープはコンパイルエラーになる
- 1958年 LISP 動的スコープ
- 1975年 Scheme（LISPの一種） 静的スコープ

- 1994年 JavaScript 基本グローバルスコープ．varをつけると静的スコープ
- 1991年 Python 何もつけなくても静的スコープ
- 1994年 Ruby何もつけなくても静的スコープ

グローバル変数の必要性

- 引数が多いほどコードの把握は困難になる
- 毎回呼び出す共通の引数は，省略したい
  - 例えば？
    - cse_set_data(cse* data, int a);
    - cse_get_data(cse* data, char* name);
    - cse_create_data(cse* data);
    - cse_release_data(cse* data);
- コードを書いている途中に欲しくなる変数
  - 必要か必要じゃないか境界際の引数
  - 必要になったときに引数に渡せば良いがめんどくさい
  - 手っ取り早くコードの修正・テストができる

グローバル変数の弊害

- 名前がぶつかる：スコープの目的
- いつ，どこで，どの関数がグローバル変数を変更したかさっぱり分からない
  - もしグローバル変数の値がおかしくなるバグを発見したら，絶望どの関数が悪さしているか探すのが困難
  - ２人で開発していたらどっちがバグをだしたかすら分からない．．．
  - 間違えて，まったく注意をしていない関数で気付いたらi++していたとか，寝ぼけててグローバル変数の値を変更していたりすると発掘不可能．．．
- その結果，分担してコードが書きづらい

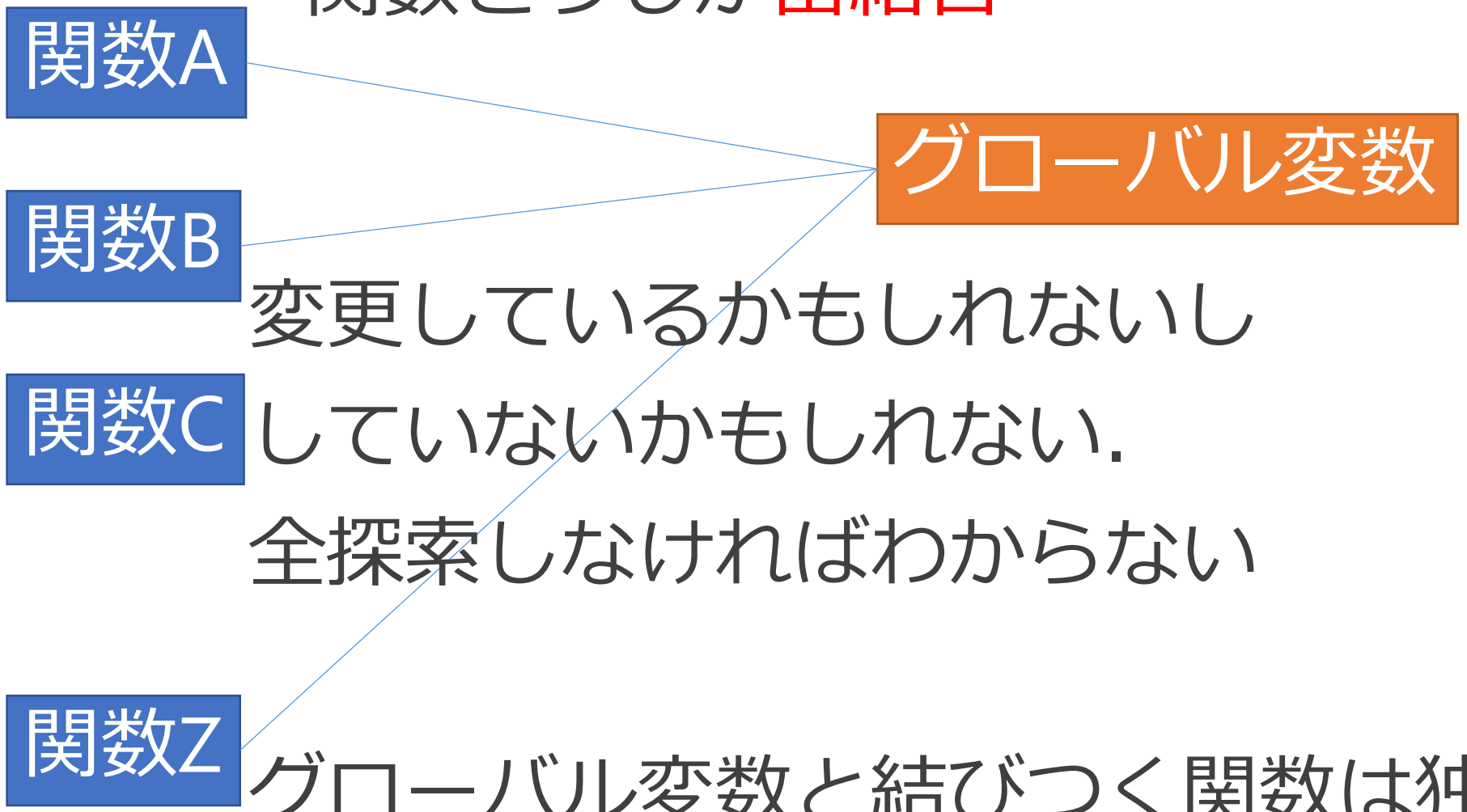
グローバル変数の弊害

- 「いつ」，「どこで」，「どの関数」が
  グローバル変数を変更したかさっぱり分からない
  - もしグローバル変数の値がおかしくなるバグを発見したら，**絶望**．どの関数が悪さしているか探すのが困難
  - 複数人で開発していたら誰がバグを出したかすら分からない．．．

- **デバッグが非常に困難**なプログラムの出来上がり

状態の独立性

関数どうしが密結合



変更しているかもしれないし
していないかもしれない.
全探索しなければわからない

グローバル変数と結びつく関数は独立性が低い

プロジェクトの独立化

- プロジェクトの複雑度はプログラムの長さに従って指数的に大きくなる
- 関数間、クラス間は疎結合になっていることが望ましい
  - 例えば，巨大なプロジェクトを完全に2つに分離できれば複雑度は激減
- **そのためにはグローバル変数はできるだけ使わない**
  - あったほうが便利な時ももちろんある

手続き型プログラミングに残された課題

- グローバル変数が残されている
- 再利用が手続き(関数)しかできない
- データや手続きに関する整理整頓のルールが不足
- 変数へのアクセス制御ができない
- 構造化プログラミングは関数だけの構造化
- データまで整理整頓するには？

オブジェクト指向プログラミングで解決可能

オブジェクト指向言語C++

- C言語をオブジェクト指向言語として拡張する形で提案されたプログラミング言語
  - 基本的にはC言語の文法・関数は全部使える
    - C言語代替としてのC++
  - 適用可能なパラダイム
    - 手続き型
    - オブジェクト指向型
    - ジェネリックプログラミング
  - 新しい言語として学ばなきゃ、と身構えず、まずは「better C」と思い、便利な機能から順番に使ってみるとよい。オブジェクト指向部分を除いてもCよりもずっと書きやすい！（最初はコメント、const, 変数の宣言ぐらいからでもよい）

まとめ: プログラミングにおける整理整頓

- 関数：**段階的詳細化**の手段として、そして再利用可能なソフトウェア部品として
- スタックの利用，再帰計算
- 名前
- スコープ
- ネームスペース

練習：C++

- C++でハローワールドを表示するプログラムをコンパイルして実行せよ.
- 提出は必要ない

C++ Hello world

C言語

```c
#include <stdio.h>

int main(int argc, char** argv)
{
        printf("Hello World¥n");
        return 0;
}
```

C＋＋

```cpp
#include <iostream> //stdioの代わり

int main(int argc, char** argv)
{
        std::cout<<"Hello World"<<std::endl;
        return 0;
}
```

コンパイル方法：g++ main.cpp

gccがg++に，拡張子がcppになっただけ．C言語もg++でコンパイル可能

注意書き


```
#include <iostream> //stdioの代わり
using namespace std;//名前空間

int main(int argc, char** argv)
{
	cout<<"Hello World"<<endl;
	return 0;
}
```

using namespace std;と書くとstd::を省略可能

小テスト３：スコープの確認


```c
#include <stdio.h>
int i= 1000;
int main()
{
  int i= 100;
    for(int i=0;i<100;i++)//for文の中に変数が宣言できる（for文の間だけのスコープ）
    {
      {
        int i = 10;
        printf("%d¥n",i);//さて何が表示される？
      }
    }
  printf("%d¥n",i);
  return 0;
}
```

１．表示される結果をまず予想せよ.

２．プログラムを実行した結果を予想とともに報告せよ.

レポート３－１：名前空間


```
#include <stdio.h>

namespace cse
{
        void get(){printf("cse¥n");}
}
void get(){ printf("global¥n");}

int main()
{
        get();
        cse::get();
        return 0;
}
```

１．プログラムを実行して内容を確認せよ

レポート3－2：動的スコープ

動的スコープを作成し，講義で指摘した問題を再現せよ.

```c
#include <stdio.h>
int global_value= 1000;
void processB()
{
  int value=global_value;
  value++;
  //グローバル変数に戻す処理を書く
}
void processA()
{
  int i;
  //引数相当をコピーする処理を書く
  printf("%d¥n",i);
  processB();
  printf("%d¥n",i);
  //グローバル変数に戻す処理を書く
}
int main()
{
  processA();
  return 0;
}
```

こんな処理を本当は実現したい

```c
void processB(int i)
{
        i++;
}

void processA(int i)
{
        printf("%d¥n",i);
        processB(i);
        printf("%d¥n",i);
}
int main()
{
        int value=1000;
        processA(value)
        processB(value)
        return 0;
}
```

レポート3 – 3

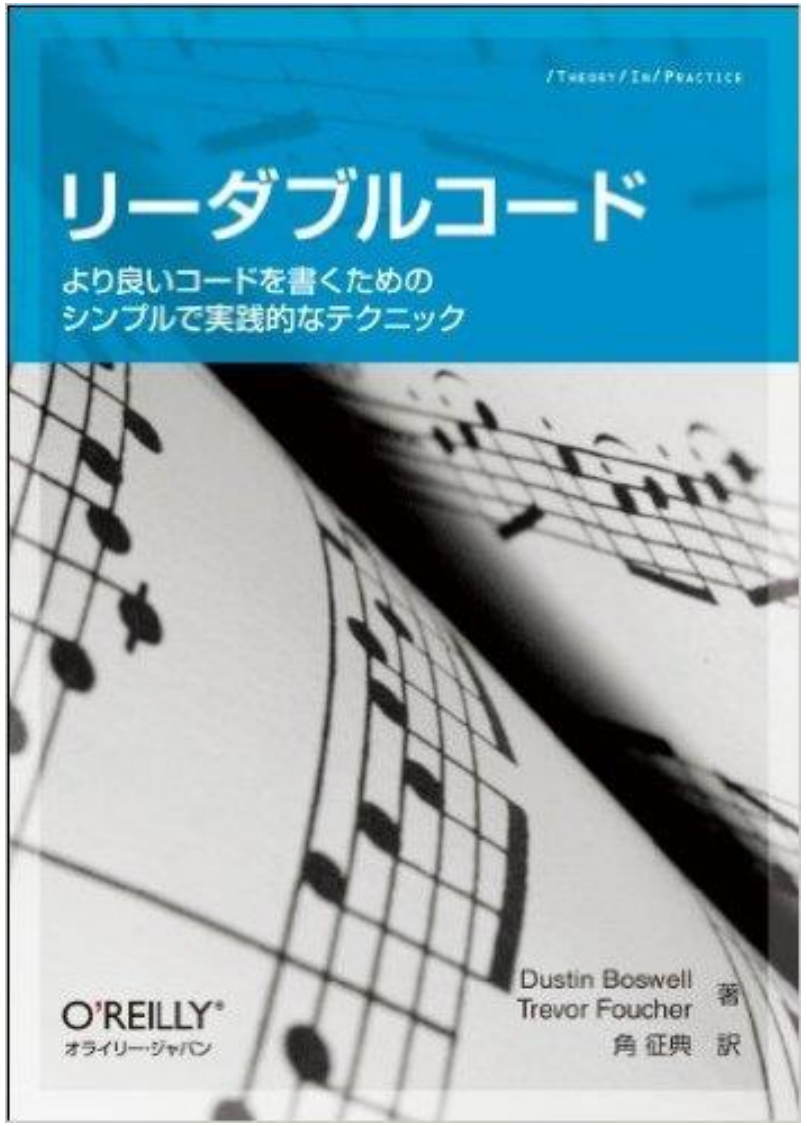
- 変数名，関数名を適切につけることは，**人が読みやすいプログラムを書くために最も重要な事項のひとつ**である．どのようなことに留意すればよいか，先人の知恵を調べ（ネット上に色々な記事がある、Qiitaで調べてもよい）のいくつかを自分なりにまとめよ．単にルールを述べるだけでなく，具体例やその理由も述べること．

  例：スコープの広い変数ほど記述的な名前にし，スコープの狭い変数（ループカウンタ）は単純な名前にする．具体例はｘｘｘ．こうする理由はｙｙｙ．

参考：有名どころではこんな本があります　57

リーダブルコード　変数名
などで検索するとよいかも.

プログラムの読みやすさに
関する技法が述べられている



レポート３－４

次ページのプログラムは，人の名前とその年令をレコードとして持つプログラムの雛形である．名前の設定，年齢の設定，名前の表示，年齢の表示を行う関数群が準備されている．

変数名，関数名が全く良くない．前ページの演習問題で身につけた技法に基づき，適切な変数名，関数名に置き換えたプログラムを作成せよ．

ダメな変数名・関数名を含むプログラム


```
#include <stdio.h>
struct P
{
  char* n;
  int   a;
};
void f(struct P* p, char* _n)
{
  p->n = _n;
}
void g(struct P* p, int _a)
{
  p->a = _a;
}
void u(struct P* p)
{
  printf("%s¥n", p->n);
}
```

```
void v(struct P* p)
{
  printf("%d¥n", p->a);
}
int main(void)
{
  struct P p[2];
  f(&p[0], "Taro");
  g(&p[0], 21);
  f(&p[1], "Hanako");
  g(&p[1], 20);
  for (int i = 0; i < 2; i++) {
    u(&p[i]);
    v(&p[i]);
  }
}
```

エディタのリファクタリング
ショートカットキーを使うと楽

レポート3 – 5 ：ファイルスコープ

- これは，できる人だけでOK
- 下記のファイルスコープを実現せよ
  - 複数のファイルからなるC言語のプログラムを作成し，各ファイルに同じ名前のグローバル変数，関数を作成し，コンパイルエラーが出ないようにせよ.
  - 上の別の説明：他の.cファイルからアクセス不能で衝突をしないグローバル変数と関数を作れ.