

システムプログラミング序論

第1回

オリエンテーション， C言語の基本

大山恵弘

科目の概要（1）

- まずはシラバスを参照のこと
- 前半：C言語プログラミング
- 後半：アセンブリ言語（機械語）プログラミング
- 場所：講義 3A202，演習 3C113
 - 情報科学類教育用計算機システム（COINS）のアカウントが必要
持っていない人は，至急，以下の Web ページを読んで申請，取得すること
<https://www.coins.tsukuba.ac.jp/ce/>（要認証）
- 教員：大山恵弘 おおやまよしひろ
- TA:
 - 河野 匠 かわの たくみ
 - 前田 優人 まえだ ゆうと
 - 安藤 洸将 あんどう こうすけ

科目の概要 (2)

- 講義や演習の内容についての質問は，教員やTAのメールアドレスではなく，下記のメールアドレスに送ること
 - isyspro2019@syssec.cs.tsukuba.ac.jp
 - 教員とTAの両方に届く
- Web ページ
 - <https://www.cs.tsukuba.ac.jp/~oyama/isyspro2019/>
- 成績
 - 演習課題のレポート，中間試験，期末試験で評価
- 2年次春学期までの必修科目は既習であることを前提とする
 - コンピュータリテラシ，プログラミング入門A，B，論理回路，データ構造とアルゴリズムなど
 - 復習的な内容は適宜入れるので，内容の重複はありうる

manabaの利用

- 以下の目的で利用する
 - 出欠確認
 - 授業資料配付
 - 演習課題の提示
 - 演習レポートの提出
- さっそく，manaba から出席カードを提出して下さい
(履修登録が済んでいない人は登録も)

科目名： システムプログラミング序論
コースコード： GB11954
受付番号：

講義（前半）資料

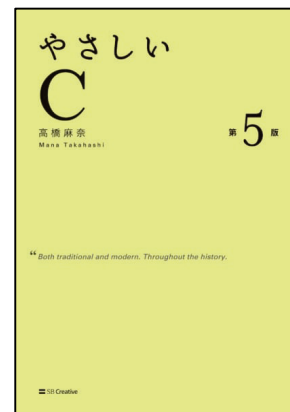
- manaba 上で配布
- 再配布は禁止
- 授業前に事前に各自がダウンロードすること

- 2014年度まで担当していた佐藤三久先生が作成した資料がベース
- 2015～2016年には阿部先生が改訂

参考図書（≠教科書）

- 追加の教材が欲しい人には下記の書籍を薦める
 - 必須ではない

- やさしいC 第5版
 - 高橋麻奈，SBクリエイティブ
- 新・明解C言語 入門編
 - 柴田望洋，SBクリエイティブ
- 新・明解C言語 ポインタ完全攻略
 - 柴田望洋，SBクリエイティブ



C言語の作者によるバイブル的書籍

- プログラミング言語C 第2版（通称 K&R）
 - B. W. カーニハン / D. M. リッチー（石田晴久 訳），共立出版
 - 推薦する人も多いが，初心者向きではない
 - むしろ，初心者は買ってはいけない



参考Webサイト

- 具体的なサイトは挙げないが，良さそうなサイトが無数にある
 - 書籍とほぼ同じコンテンツが無料で公開されていることもある
 - 使わない手はない
- 検索して自分に合ったサイトを見つけ，利用しよう

C言語の基本

講義の目的

- C言語と機械語によるシステムプログラミングや低レベルプログラミングを通じて、コンピュータやプログラムの低層の構造や基礎概念を理解する

システムプログラミング

- システムソフトウェア
 - オペレーティングシステム (OS)
 - デバイスドライバ

などの、ハードウェアを直接操作するソフトウェア、
および、OS カーネルを直接呼び出すソフトウェア

- libc などのライブラリ
- これらシステムソフトウェアのプログラミングを一般的に、システムプログラミングと呼ぶ
 - C言語や機械語が使われることが多い

なぜ C を使うか

- システムプログラミング
 - 実行環境が単純であり，OS なしで実行可能にすることが容易
 - ハードウェアの直接操作に必要なメモリアクセスの記述が可能
 - 機械語と組み合わせることが容易
- 速度や省資源を重視するプログラミング
 - 実行時のオーバヘッドや資源消費が小さく，効率が良い

Java との違い

- Java

- オブジェクト指向
- 強い型付け
- ポインタは無い（参照そのものはある）
- GC（ゴミ集め）がある（メモリ管理は Java 任せ）
- Java VM 上で実行（基本的には）

- C

- 型付けは弱いので、何でもあり
- ポインタで何でもできる
- GC がない（メモリ管理は自分で気をつける）
- 機械語にコンパイルして CPU が直接実行

C言語の歴史

- 1973年頃に Ken Thompson と Dennis Ritchie が開発
 - 当初の目的は UNIX オペレーティングシステムの記述
- その後、UNIX と共に大学や研究所に普及
- さらにその後、UNIX 以外の開発にも普及し、1990年頃には ISO で標準化
- 2019年現在でも “Top Programming Languages” で第3位 (IEEE Spectrum 調べ)
 - <https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019>
 - 1位から順に Python, Java, C, C++, R, JavaScript, C#, Matlab, Swift, Go

C言語の仕様

- ISO 規格が存在
 - C89, C99, C11, ...
 - 改訂され続けている
 - この科目で「説明に使う」 コードは C89 に準拠
 - 新しい機能を使わず，互換性を重視
 - 課題レポートでは，演習の想定環境でサポートされているなら，新しい仕様を使ってもよい
- コンパイラによって異なる
 - gcc, clang, Intel C++, Microsoft Visual C++, ...
 - 各コンパイラが独自に拡張している
 - この科目の演習では，計算機室の macOS の gcc を仮定

Hello world!

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    /* Printing a message */
```

```
    printf("Hello world!¥n");
```

```
    return 0;
```

```
}
```


コンパイルと実行

- コンパイル

- プログラムが書かれたファイルから実行型ファイルを作る

```
$ gcc -o hello hello.c
```

- 実行

- シェルのコマンドラインで実行型ファイルを指定し，必要に応じて引数を与える

```
$ ./hello
```

```
Hello world!
```

```
$
```

基本文法 (1)

- プログラムは `int main(void) { ... }` から始まる
 - `main` 関数と呼ぶ
 - 関数とは何かについては後述
- プログラムは，書かれた順（上から下）に実行される
- `;` で終わる単位が文であり，プログラムは文を単位に実行される
- 初めに実行される文は `printf(...);`
 - `printf` は ... の文字列を画面に出力する関数
- ダブルクォーテーションで囲まれた部分は文字列
 - 日本語を入れても良いが，コンパイラやシステムによっては日本語が使えないので，この講義では常に英文を入れる

基本文法 (2)

- `printf` の文字列の中の `\n` は改行を意味する
 - よって `Hello world!` と出力した後、改行を出力する
- `\` から始まる文字列はエスケープシーケンスという
 - 特殊な文字 (列) を表す
 - `\t` はタブ, など
 - `\` を出力したい場合には `\\` と書く
- 空白や改行を文の間にいくつ入れてもかまわない
 - 無くてもかまわない
 - ただ, 見やすくなるように適当に改行することを勧める
 - `printf` と (の間や, (と " の間にも, 空白を入れてかまわない
 - とはいえ, 見にくくなるのでやめたほうが良い
 - `printf` などの名前の中に空白を入れてはいけない
 - 例えば, `pri nt` は `printf` とは別の名前と解釈される

基本文法 (3)

- `/*` と `*/` の間の部分をコメントという
 - 空白と同じに扱われて、無視される
- `printf` の次には `return 0` が実行される
 - `main` 関数を終了するという文
 - Cプログラムの実行は `main` 関数の先頭から始まり、`main` 関数が終了すると終了する
- `printf` を使うプログラムには `#include <stdio.h>` を書く
 - `printf` を使う場所よりも上に
(典型的にはプログラムの先頭に)
 - `stdio.h` というファイルをそこに読み込むという指示
 - 今は「おまじない」とっておけば良い

printf 関数

```
printf("Hello!¥nThe number is %d¥n", 10);
```

- 端末に文字を出力
- %d は、文字列の後に与えられた整数をその場所に表示
しなさい、という意味

```
Hello!
```

```
The number is 10
```

計算をさせてみる

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int x, y;
```

```
    x = 1;
```

```
    y = x + 13;
```

```
    printf("y = %d\n", y);
```

```
    return 0;
```

```
}
```

変数

- 変数とは，値を入れておく箱のようなもの
- 変数は関数の最初などで「宣言」することが必要
- 数学の「変数」とは違うので注意

変数への代入

- 「代入」とは，変数の箱に値をいれること

変数の名前 = 式

- C言語の = は，数学の = とは違う
 - $x = 1$ は「 x に 1 を入れる」であり「 x は 1 に等しい」ではない
- 式の中の変数が現れる場所ではその変数の値が使われる
 - x に 1 が入っているとき， $x + 2$ の計算結果は 3 になる
 - ここは数学と同じ

算術式

- $+$ は足し算
- $-$ は引き算
- $*$ は掛け算
- $/$ は割り算
- $\%$ は剰余算

変数の宣言の仕方

- 通常，関数の最初で宣言する

変数の型 変数名, 変数名, ...;

- 変数の型には，`int` 型（整数型） ， `double` 型（実数型） などがある
 - 実数型はしばしば浮動小数点型とも呼ばれる
- 例
 - `int a, b, c;` `int` 型の変数 `a, b, c` を宣言
 - `double x;` `double` 型の変数 `x` を宣言

scanf 関数

- 出力は `printf`
- 入力は `scanf`

`scanf ("%d", &整数型の変数)`

`scanf ("%f", &実数型の変数)`

- `&` を忘れないで！

端末からの数の入力

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int x;
```

```
    printf("Please input? ");
```

```
    scanf("%d", &x);
```

```
    printf("Input is %d\n", x);
```

```
    return 0;
```

```
}
```

色々なデータ型

- 整数と実数が区別されている
 - 整数 `int`
 - 実数 `double`
 - 混ぜて使う場合に注意
- `scanf` や `printf` の第一引数に与える文字列は、整数を使うか実数を使うかで違うので注意

条件分岐

if 文

- 基本： 上から順に実行する
- 「判断」して，その後に実行する文を変える

```
if (条件式) {  
    条件式が真の時に実行されるプログラム  
} else {  
    条件式が偽の時に実行されるプログラム  
}
```

- **else** 以降の部分は，必要がなければ書かなくても良い
- 上の「プログラム」が一文だけから成る場合には { や } を省略可能

if 文の使用例

```
#include <stdio.h>

int main(void)
{
    int x;
    printf("Please input? ");
    scanf("%d", &x);
    if (x > 0) {
        printf("Input is positive\n");
    } else {
        printf("Input is negative or zero\n");
    }
    return 0;
}
```


条件式

$A > B$	A が B よりも大きい場合に真
$A < B$	A が B よりも小さい場合に真
$A \geq B$	A が B よりも大きい，または等しい場合に真
$A \leq B$	A が B よりも小さい，または等しい場合に真
$A == B$	A が B と等しい場合に真
$A != B$	A が B と等しくない場合に真

条件式についての細かい話

- 条件文には任意の式を入れることができる
 - 式の値が0の場合は偽，それ以外は真として扱われる

```
if (3 - 2 - 1) {  
    printf("true");  
} else {  
    printf("false");  
}
```

- 実は，`x > 1` などの等号や不等号の条件式も，0か1を返す，式の種類

```
printf("%d\n", 1 == 1);
```

複文：複数の文からなる文

- 複数の文を { と } で囲んだものも文
 - これを複文という
- 例
 - `printf("Hello¥n");` は文
 - `{ x = 1; y = x + 2; printf("y = %d¥n", y); }` も文
- if文の正確な文法：
`if (式)`
文1
`else`
文2

絶対値を求める

```
#include <stdio.h>
int main(void)
{
    int x;
    printf("Please input? ");
    scanf("%d", &x);
    if (x < 0){
        printf("Input is negative\n");
        x = -x;
    }
    printf("Absolute number is %d\n", x);
    return 0;
}
```

もっと複雑なif文

`if` (式1)

式1が真の時に実行する文

`else if` (式2)

式2が真の時に実行する文

`else if` (式3)

式3が真の時に実行する文

`else if` (式4)

式4が真の時に実行する文

`else`

それ以外の時に実行する文

もっと複雑な条件式

- 論理演算子

$A \ \&\& \ B$	A が真，かつ B が真ならば真
$A \ \ B$	A が真，または B が真ならば真
$!A$	A が偽ならば真

否定演算子 (!) の正確な意味

- **!x は**
 - x が0ではないならば0
 - x が0ならば1
- **ISO/IEC 9899:1999 6.5.3.3-5**

“The result of the logical negation operator ! is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. The result has type int. The expression !E is equivalent to (0==E).”

複雑な式

- 演算子には優先度がある
- * と / は + と - よりも先に計算される
- 算術式は論理演算子よりも先に計算される
 - `x == 1 && y == 2` は
 `(x == 1) && (y == 2)` と同じ意味の式であって
 `(x == (1 && y)) == 2` や
 `x == (1 && (y == 2))` と同じ意味の式ではない
- 迷ったらカッコをつけよう！

ここまでの内容

- コンパイルと実行
- C言語の基本文法
- `printf` 関数による文字列の出力
- 変数，代入，式
- `scanf` 関数による数の入力
- `int` 型と `double` 型
- 条件分岐
 - `if` 文
 - 条件式
 - 論理演算子

繰り返し

コンピュータはどのくらい速いか？

- クロック速度がコンピュータの速さの目安
 - Intel Core i9-9900（2019年夏発売）のクロックは最大5 GHz
 - CPU は，通常，数クロックに1命令ずつ実行できる
 - 仮に，2.5クロックに1命令実行できるとすると，1秒間に2G 命令，つまり， $2,000,000,000 = 20$ 億命令
- 百万回繰り返しても、あっという間！
 - コンピュータのパワーの源

goto 文：実行の順番を変える

- “万能”の制御文
- 何もしないと，プログラムは上から下に順に実行される
- goto 文は，その引数に与えたラベルが書かれているプログラムの地点に制御を移す
- あまり推奨されない
 - goto 文を使いすぎのプログラムは，良くないプログラム

goto 文

- 繰り返す．永遠に！

```
#include <stdio.h>
int main(void)
{
    int x;
    x = 0;
again:
    printf("x=%d\n", x);
    x = x + 1;
    goto again;
    return 0;
}
```

goto 文

- 99で止めてみる

```
#include <stdio.h>
int main(void)
{
    int x;
    x = 0;
again:
    if (x >= 100) {
        goto stop;
    }
    printf("x=%d\n", x);
    x = x + 1;
    goto again;
stop:
    return 0;
}
```

goto 文のむやみな使用は推奨されない．なぜか？

- 色々なところに飛べるので，飛び先（や飛び元）を探すのが大変
- いきあたりばったりに制御してしまうプログラムを書きがちになる
- プログラムの構造が見えない
 - 繰り返しのなかそうでないのかが，一目ではわからない

構造が見える制御文を使おう！

- while, for, do

while 文：条件が成立している間は繰り返す

- while 文は，ある条件が成立している間，文を繰り返し実行する

```
#include <stdio.h>
int main(void)
{
    int x;
    x = 0;
    while (x < 100) {
        printf("x=%d¥n", x);
        x = x + 1;
    }
    return 0;
}
```


while 文の書き方

```
while (条件式)  
    条件が成立している間実行する文
```

文が複文である場合には以下のような形になる

```
while (条件式) {  
    条件が成立している間実行する文  
}
```

goto で書き直してみれば...

```
#include <stdio.h>
int main(void)
{
    int x;
    x = 0;
again:
    if (x < 100) {
        printf("x=%d¥n", x);
        x = x + 1;
        goto again;
    }
    return 0;
}
```

for 文：初期化式と繰り返し式を必ず書く繰り返し

- 変数の値をある値からある値まで変化させるのはよく現れるパターン
- この時、便利なのが for 文

```
for (i = 0; i < 100; i++) { ... }
```

for (初期化のための式; 条件式; 繰り返し式)
条件が成立している間実行する文

文が複文である場合には以下のような形になる

```
for (初期化のための式; 条件式; 繰り返し式) {  
    条件が成立している間実行する文  
}
```

for 文で書き換えてみると...

```
#include <stdio.h>
int main(void)
{
    int x;
    for (x = 0; x < 100; x = x + 1) {
        printf("x=%d¥n", x);
    }
    return 0;
}
```

for 文の利点

- ある変数を順に増やしていく時には，while 文よりも簡単に書ける
 - 簡単 ⇒ わかりやすい
 - コンパクト
- 1 ずつ増やす場合だけでなく，初期化，繰り返し，停止条件というパターンに合う場合には使える
 - よく現れるパターン

記述を簡単にする演算子

- 「1ずつ足す」 $x = x + 1$ は、よく現れるパターン
- $x = x + 1$ は $x++$ と書くことができる

```
for (x = 0; x < 100; x++) {  
    ...  
}
```

インクリメント・デクリメント 演算子

変数++	ポストインクリメント．変数に1を加える． 式としての値は1を加える前の値となる．
++変数	プリインクリメント．変数に1を加える． 式としての値は1を加えた後の値となる．
変数--	ポストデクリメント．変数から1を減らす． 式としての値は1を減らす前の値となる．
--変数	プリデクリメント．変数から1を減らす． 式としての値は1を減らした後の値となる．

プリインクリメントとポストインクリメントの違い

```
a = 3;  
b = a++ + 5;
```

a の値は4に,
b の値は8になる

```
a = 3;  
b = ++a + 5;
```

a の値は4に,
b の値は9になる

代入演算子

- 「変数 演算子= 式」は
「変数 = 変数 演算子 式」と同じ
 - $x += 10$ は $x = x + 10$ と同じ
 - $x -= 20$ は $x = x - 20$ と同じ

break 文と continue 文： ループの停止と中断

- break が実行されると，ループの実行を中断する
- continue が実行されると，現在実行中の回のループの実行をやめて，次の回のループに移る
 - for 文の中で continue が実行された場合には，次の回が実行される前に繰り返し式が実行されることに注意

break の使用例

```
#include <stdio.h>
int main(void)
{
    int x, sum;
    sum = 0;
    for (x = 1; x < 100; x++) {
        sum += x;
        if (sum > 1000) {
            break;
        }
    }
    printf("1+2+...+%d = %d > 1000¥n", x, sum);
    return 0;
}
```

continue の使用例

```
#include <stdio.h>
int main(void)
{
    int x;
    printf("Numbers that cannot be divided by 2 or 3: ");
    for (x = 1; x < 100; x++) {
        if ((x % 2 == 0) || (x % 3 == 0)) {
            continue;
        }
        printf("%d ", x);
    }
    printf("\n");
    return 0;
}
```

do 文： while 文の変形

do

条件が成立している間実行する文

while (条件式)

文が複文である場合には以下のような形になる

do {

条件が成立している間実行する文

} while (条件式)

配列

配列


- データを入れる箱を並べて，「何番目か」（インデクス）でデータを参照するためのデータ型
 - 「何番目か」を，実行時に計算した値で指定できるのがポイント

配列の宣言

- 文法

データ型 配列の名前 [要素の数]

- 例： 10個の整数の配列の宣言

`int a[10];` a 

- 要素の数（何個のデータが入るか）を，配列のサイズと言う

- C89 の仕様では，コンパイル時に決まっている値でなくてはならない（最近の仕様では異なる）

配列の参照

- 文法

配列の名前 [何番目かを指定する数]

- 注意！ 配列の要素は0から数える

- 初めの要素は0番目

- 参照（取り出し）

$a[i * 2 + 1] + 10$

- = の左辺に配列を書くと代入

$a[i + 3] = 100$

配列を使ってみる

- 10個の数を入力として受け取り，合計を計算するプログラム

```
#include <stdio.h>
int a[10];
int main(void)
{
    int i, k, s;
    printf("Please input 10 numbers? ");
    for (i = 0; i < 10; i++){
        scanf("%d", &k);
        a[i] = k;
    }
    s = 0;
    for (i = 0; i < 10; i++) {
        s += a[i];
    }
    printf("Sum is %d.¥n", s);
    return 0;
}
```

変数と配列の初期化

- あらかじめ、変数に最初に値をセットしておく
（変数を初期化しておく）と便利なことがある
- 変数 `a` の宣言と同時に `a` に10を入れる

```
int a = 10;
```

- 配列の宣言と同時に、サイズ4の配列に順番に
1, 3, 5, 1を入れる

```
int test[4] = { 1, 3, 5, 1 };
```

関数

関数

- 数学の関数：

- パラメータに対して，パラメータで決まる値を対応させるもの

$$f(x) = 2x + 1$$

- C言語の関数：

- 似ているが少し違う
- パラメータを受け取って値を返す一連の手続き
- 数学の関数と同じ働きをさせることができる

```
int f(int x)
{
    return 2 * x + 1;
}
```

関数の役割

- 数学の関数のような働き
 - 値を受け取って，値を計算する
 - 関数 “function”
- ある特定の処理を実行するためのコードをまとめたもの
 - 手続き “procedure”
 - 呼び出すことによって，その処理を実行することができる
 - `printf` 関数や `scanf` 関数
- プログラムは関数の集まりでできている

関数の定義

• 文法

```
関数が返す値のデータ型 関数名 (パラメータのデータ型 パラメータ名, ...)  
{  
    /* あれば, 変数の宣言 */  
    関数のプログラム...  
    return 関数が返す値の式;  
}
```

- 呼び出されると, 上から順に実行される
- return 文で, 呼び出し元に戻る

関数の呼び出し

- 文法

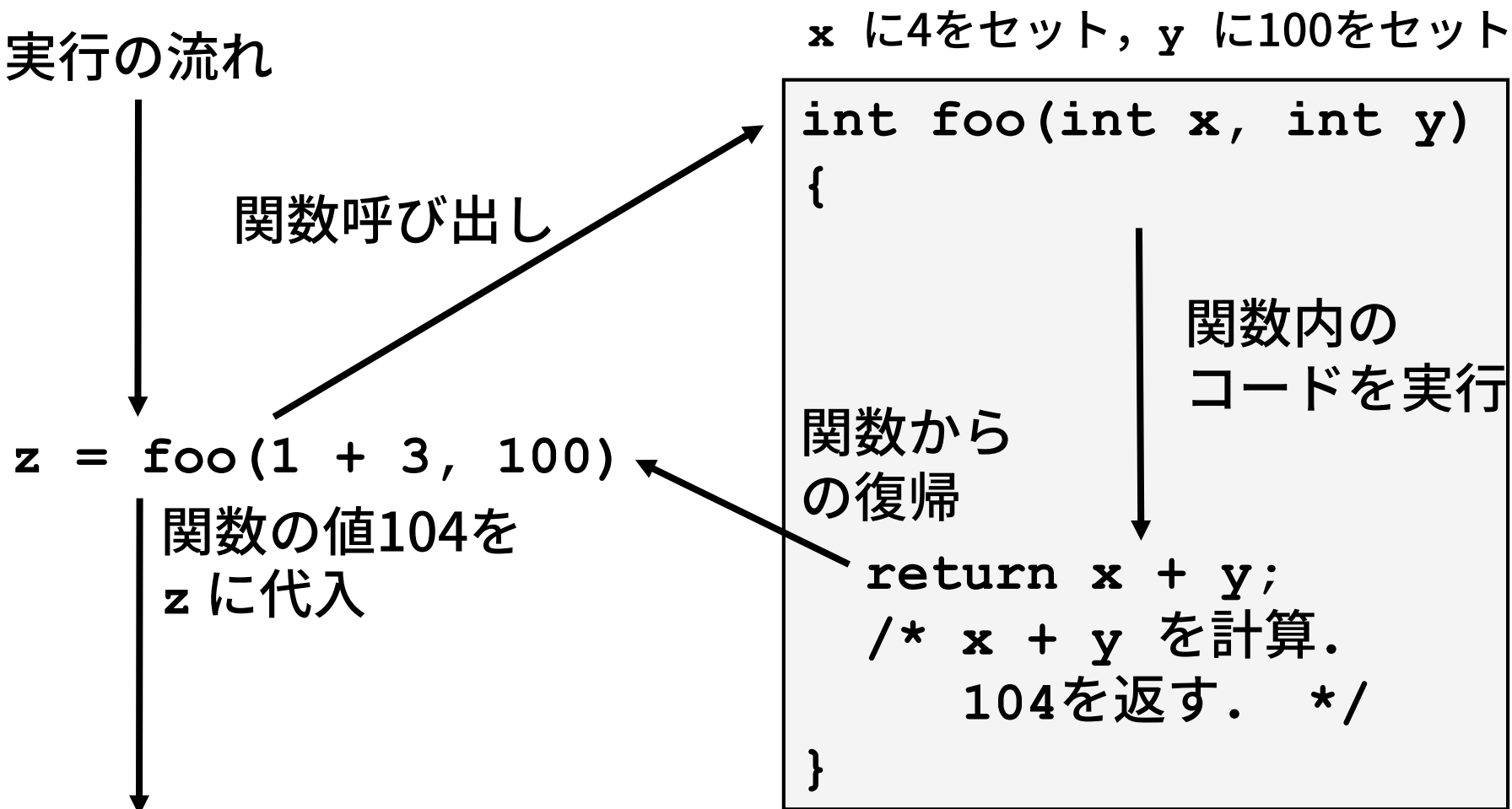
関数名 (式, ...)

- 式の値が計算されて，関数に渡される
 - この値を（関数の）引数と呼ぶ
- 関数呼び出しを式の一部として使うこともできる

```
x = foo(1, y) + 1;
```


関数の実行の流れ

実行の流れ



値を返さない（手続きとしての）関数

- 文法

```
void 関数名 (パラメータのデータ型 パラメータ名, ...)  
{  
    /* あれば、変数の宣言 */  
    関数のプログラム...  
    return;  
}
```

- 関数が返す値（返り値）の型を `void` とする
- `return` には値が無くても良い（あってはいけない）

関数の宣言

- 関数を呼び出す前には，関数の型の宣言が必要
 - 関数のプロトタイプ宣言と言う
- 文法

関数が返す値のデータ型 関数名 (パラメータのデータ型 パラメータ名, ...) ;

- 関数定義の本体の部分を除いたもの

宣言が不要な場合と必要な場合

```
int f(int x, int y)
{
    ...
}

double g(double z)
{
    ...
    f(1, 2);
    ...
}
```

```
int f(int x, int y);

double g(double z)
{
    ...
    f(1, 2);
    ...
}

int f(int x, int y)
{
    ...
}
```

宣言の書き方

- 関数呼び出しの前（上）で、関数を定義するか、関数のプロトタイプ宣言を書く
- プロトタイプ宣言を書けば、関数の本体はどこで定義しても良い

```
#include <stdio.h>
int imax(int a, int b);
int main(void)
{
    int x, y, z;
    scanf("%d", &x);
    scanf("%d", &y);
    z = imax(x, y);
    printf("Max is %d.¥n", z);
    return 0;
}

int imax(int a, int b)
{
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

コマンドライン引数

コマンドライン引数の使い方

- プログラムにはコマンドライン引数を与えることができる
 - 文字列の並び
 - 例：\$./a.out 1 abc 3.4 pqr
- プログラムはそれらを main 関数の引数として受け取る

```
int main(int argc, char *argv[])  
{  
    ...  
}
```

- `argc` はコマンドライン引数の数
- `argv` はコマンドライン引数（文字列）の配列
 - 配列の先頭（0番目の）要素はプログラム名

コマンドライン引数を表示する プログラム

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    printf("The number of the arguments is %d¥n", argc);
    for (i = 0; i < argc; i++) {
        printf("The argument %d is %s¥n", i, argv[i]);
    }
    return 0;
}
```


コマンドライン引数からの整数の受け取り

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int a, b;
    if (argc != 3) {
        fprintf(stderr, "Error: specify two integers.¥n");
        exit(1); /* Terminate this program */
    }
    a = atoi(argv[1]); /* Convert string to integer */
    b = atoi(argv[2]); /* Convert string to integer */
    printf("%d + %d = %d¥n", a, b, a + b);
    return 0;
}
```

色々な変数

- 局所変数
- 大域変数
- 静的変数

局所変数

- 関数の中で宣言されている
 - 関数の引数部分の変数も含まれる
- 関数の作業領域として使われる
- 宣言した関数から復帰すると無効になる
- 違う関数の中の局所変数は使えない
- 使用例：

```
int sum(int x, int y)
{
    int r;
    r = x + y;
    return r;
}
```

大域変数

- 関数の外で宣言されている
- 複数の関数の間で共有するデータを入れる
- 使用例：

```
int counter;

void increment(void)
{
    counter += 1;
}

void decrement(void)
{
    counter -= 1;
}
```

大域変数の好ましくない使用例

```
int t;
```

```
int foo(int x, int y)
{
    t = x + y;
    return t;
}
```

- foo も bar も，返り値以外に対して影響を及ぼす
 - この影響を副作用（side effect）と言う
 - 副作用がない関数のほうが理解しやすいし，誤りも入りにくい

```
int bar(int x, int y)
{
    t = x - y;
    z = foo(x, y);
    ...
    return t;
}
```

- 局所的にだけ使う一時的なデータにはできるだけ局所変数を使いましょう
- 必要なときだけ大域変数を使いましょう

大域変数の使用はできるだけ避けたほうが良い．なぜか？

- 局所変数に比べ，プログラムのどこで読んだり書いたりしているかを把握しにくい
- 同じ名前の大域変数を使うプログラムを結合すると，本来別の変数が同じ変数名でアクセスされる
- マルチスレッドプログラムで排他処理が必要になる，など

Aさんのプログラム

```
int c;  
void age_plus1(void)  
{  
    c++;  
}
```

Bさんのプログラム

```
int c;  
void score_add1(void)  
{  
    c++;  
}
```

大域変数と同じ名前の局所変数を使うとどうなるか？

```
#include <stdio.h>
int x = 1;
void f(void)
{
    int x;
    printf("%d\\n", x);
    x = 2;
    printf("%d\\n", x);
}
int main(void)
{
    printf("%d\\n", x);
    f();
    printf("%d\\n", x);
    return 0;
}
```

```
$ gcc -Wall conflict.c
conflict.c:6:18: warning: variable
'x' is uninitialized when used here
[-Wuninitialized]
    printf("%d\\n", x);
                    ^

conflict.c:5:8: note: initialize the
variable 'x' to silence this warning
    int x;
        ^
        = 0

1 warning generated.
$ ./a.out
1
0
2
1
$
```

ここまでの内容

- 繰り返し
 - while, for, break, continue, do, (goto)
 - goto は，必要に迫られない限り，使わないほうが良い
- インクリメント・デクリメント演算子，代入演算子
- 配列
 - 宣言，参照，代入，初期化
- 関数
 - 数学の関数との関係，定義，呼び出し，宣言
- コマンドライン引数
- 色々な変数
 - 局所変数
 - 大域変数
 - （静的変数）

演習の準備

準備

- 各自でやっておく
- レポートの提出は不要
- 情報科学類の端末の macOS 環境にログインする
 - 端末からログインでも，リモートログインでもよい
- Hello world! を端末に表示するC言語のプログラムを作成し，ファイルに保存する
- そのファイルを gcc でコンパイルし，実行する
- うまくいかない場合には，状況を教員に連絡する

自宅や外出先で課題をやりたい人のための準備

- リモートログインの準備
 - 自分の公開鍵と秘密鍵のペアを作る
 - 公開鍵のファイルを情報科学類計算機環境における所定のディレクトリに置く
 - 公開鍵認証によるログインができるかどうかをテストする

自宅や外出先で課題をやりたい人のための準備

- 自分のローカル環境への UNIX+gcc の導入
 - macOS が動くマシンを入手
 - PC を入手して Linux をインストール
 - Raspberry Pi なら5000円以下
 - 仮想化ソフトウェアを用いて仮想マシンを作成し、そこに Linux をインストール
 - VMware, VirtualBox, QEMU, Docker などを利用
 - Linuxにも色々あるが、Ubuntu, CentOS, Debian などが人気
- Windows 上に Linux またはそれっぽいものを導入
 - Windows Subsystem for Linux (WSL)
 - Cygwin

10月の定期全学停電に注意

- 10月26，27日（土，日）は全学的に停電になります
- 情報科学類コンピューティング環境のサーバも端末も使えません
- 全学計算機システムの中には，全学停電中にも利用できるものがあります
 - 詳しくは学術情報メディアセンターの Web ページを参照