

プログラミング言語論

第4回

オブジェクト指向プログラミング
カプセル化

- オブジェクト指向プログラミングとは
- オブジェクト指向プログラミングの歴史
- クラス
- カプセル化 。 。 。 隠蔽とアクセス制御
- つづき（来週以降）
 - 継承（inheritance） 。 。 。 差分プログラミング
 - ポリモーフィズム 。 。 。 ジェネリックプログラミング
 - テンプレート, ライブラリ, STL

オブジェクト指向 イントロダクション

オブジェクト指向プログラミング (**object-oriented programming; OOP**) の定義

- 「**ユーザー定義型**と**継承**を使ったプログラミングである」
 - ビャーネ・ストロヴストルップ (C++の設計者)
- 「データを自律的なオブジェクトとしてとらえ, **オブジェクト間のメッセージ交換**によって計算が進むものの計算モデルによるプログラミングである」
 - アラン・ケイ (Smalltalkの設計者, オブジェクト指向という言葉の発明者)

1954

FORTRAN

1958

ALGOL

LISP

1964

Simula

1971

C

Smalltalk

1983

C++

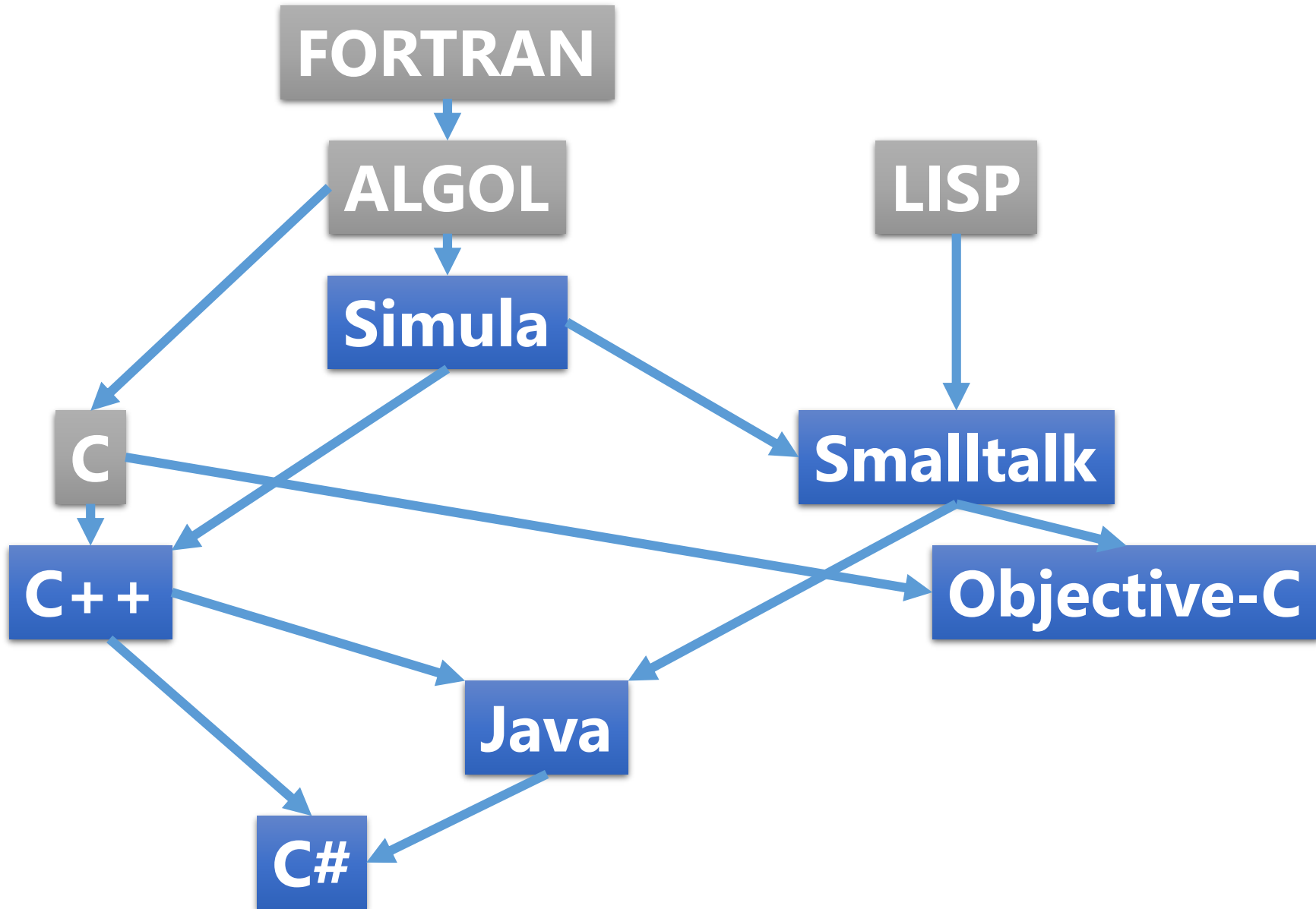
Objective-C

1995

Java

2002

C#



- 開発年：1972年
- 開発者：アラン・ケイ
 - パーソナルコンピュータという概念を提唱した
 - ダイナブック構想の提唱者
 - ・マッキントッシュの大本
 - コンピュータ・リテラシーという単語を造った
- 特徴
 - クラスベースの純粹オブジェクト指向言語
 - オブジェクト指向と初めて命名

- 開発年：1983年
- 設計者：ビャーネ・ストロヴストルップ
- 特徴
 - 手続き型，オブジェクト指向型，ジェネリック
 - 最初の名前はC with Classes
 - 当初はCにクラスが追加されただけの言語。それから仮想関数，多重定義，多重継承，テンプレート，例外処理，ラムダ式、型推論など様々な機能が追加されている
 - 最新の規格はC++20
 - C言語のコードは（ほぼ）そのままコンパイルできるように設計
- なんでもできる（ある意味最強）半面，複雑で難解な言語（過去のしがらみを引きずっている？）であると評価

言語仕様は複雑ではあるが、ある程度理解すれば、巨大かつ高速性の要求されるプログラムを書くための強力な機能を持つ

2003

2011 C++11 (C++0x) : 型推論, foreach, ラムダ式

2014 C++14 : 変数テンプレート, constexpr

2017 C++17 : if constexpr

2020 C++20 : enum classの名前を省略可能に

- 開発年：1995
- 開発者？：サン・マイクロシステムズ→オラクル
- オブジェクト指向
 - SmalltalkやObjective-Cと同様なほうのオブジェクト指向
- 特徴
 - バイトコードにコンパイルされて仮想マシンで実行
 - ビジネスで一番使われている言語
 - C++に比べて厳密で書き方に揺らぎが少ない
 - JavaとC++の最大の違いはポインタの有無

- 1983開発
- ブラッド・コックス
 - スティーブ・ジョブスがObjective-Cを使ってマックの前身を開発していたため知名度が向上→買収
- C言語の上位互換
- 特徴
 - iOSのプログラミング言語
 - Smalltalkから派生した言語
 - Javaよりも古い
 - Swiftに派生

- Objective-C を置き換えるためにApple が発表した言語
- MacOS, iOS のプログラミングに最近はよく用いられる
- オブジェクト指向 + 関数型 (マルチパラダイム)
- C++ と比較して、クリーンかつ現代的な言語仕様
- 安全性への配慮: 静的型検査、変数初期化の強制、オーバーフローの検査

- 最近注目されている
- マルチパラダイム言語
- 速度、並行性、安全性を言語仕様として保証する
- 特徴
 - システムプログラミング: C, C++の置き換え
 - 実行速度はC言語並・メモリ安全性への配慮
 - Rustにはclassがない. 代わりに構造体structとimpl, trait

- クラスをオブジェクトとして扱う言語である
- **クラス（class）** とは **ユーザ定義型** の型である

データと関数をまとめる。
たったそれだけ？

- クラスは、関連する **データと関数** を **まとめて**、**構造体** のような型にして名前をつける仕組み（**抽象化**）

- 構造化プログラミングでの欠点の解消

- **プログラムは、オブジェクトに分割** される

- ・ 参考：構造化プログラミングでは、関数とデータ構造に分割

クラス Example（ユーザ定義型とも解釈できる）



→ インスタンスの生成
Example a;

値を出力するクラス

```
class PrintClass
{
public://publicで公開. カプセル化で説明
    int value;//メンバ変数
    void print()//メソッド
    {
        printf("value is %d\n", value);
    }
}; //セミコロンを忘れない
```

```
int main()
{
    PrintClass v;
    v.value=100;
    v.print();
}
```

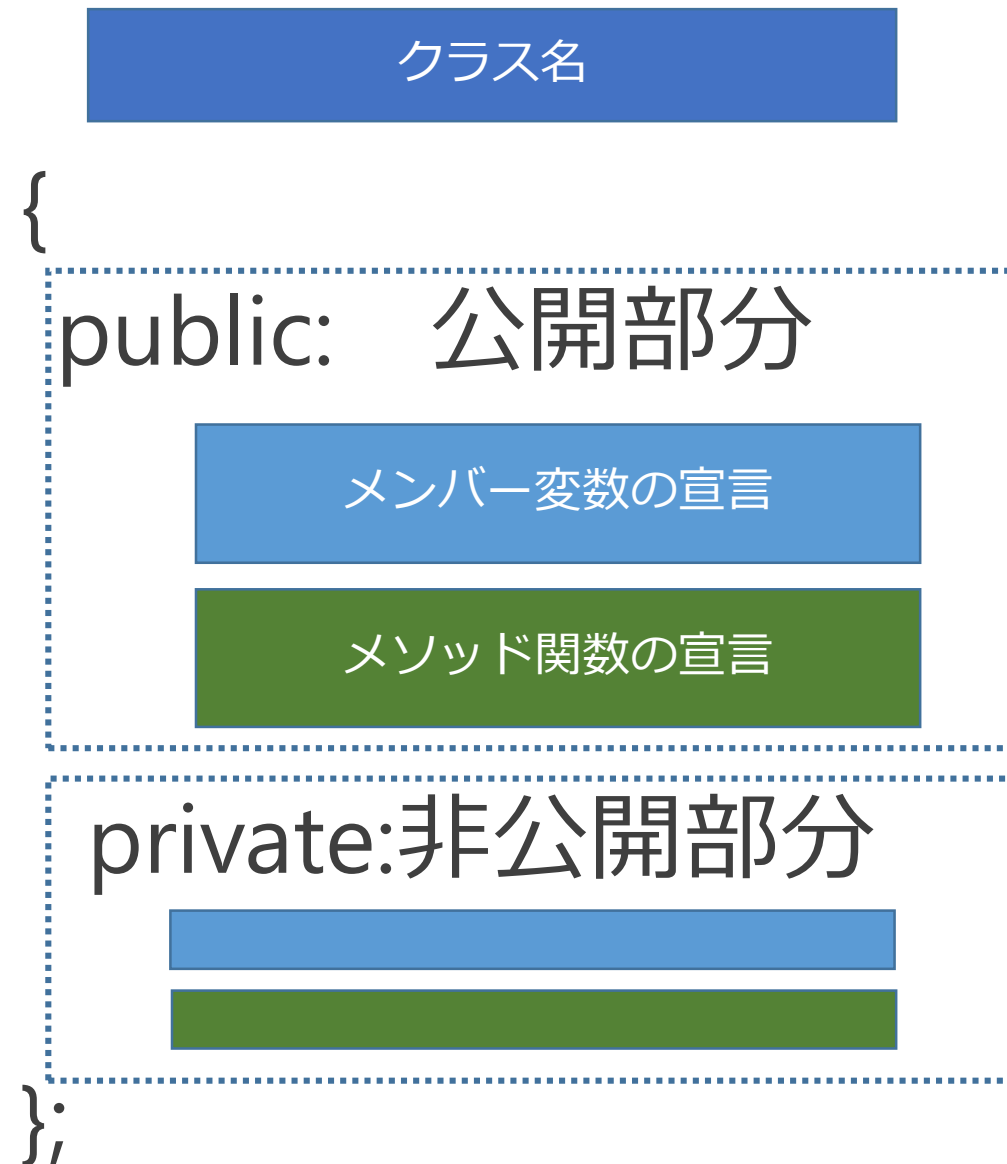
C言語の構造体のように宣言

- **struct**の代わりに**class**
- 中に関数があっても良い

出力結果：value is 100

※C++では、中のクラス内の変数のことをメンバ変数、関数のことをメソッドと呼ぶ

```
class PrintClass
{
public:
    int value;
    void print()
    {
        printf("%d¥n", value);
    }
};
```



```
int main()  
{  
    PrintClass v;  
    v.value=100;  
    v.print();  
}
```

メンバ変数へのアクセス

インスタンス名.メンバ変数名

メソッド関数へのアクセス

インスタンス名.メソッド関数名

ポインタ変数の場合

```
int main()
{
    PrintClass *p;
    p = new PrintClass;
    p->value=100;
    p->print();
}
```

インスタンス実体のメモリ確保が必要（newを利用）

メンバ変数へのアクセス

インスタンス名->メンバ変数名

メソッド関数へのアクセス

インスタンス名->メソッド関数名

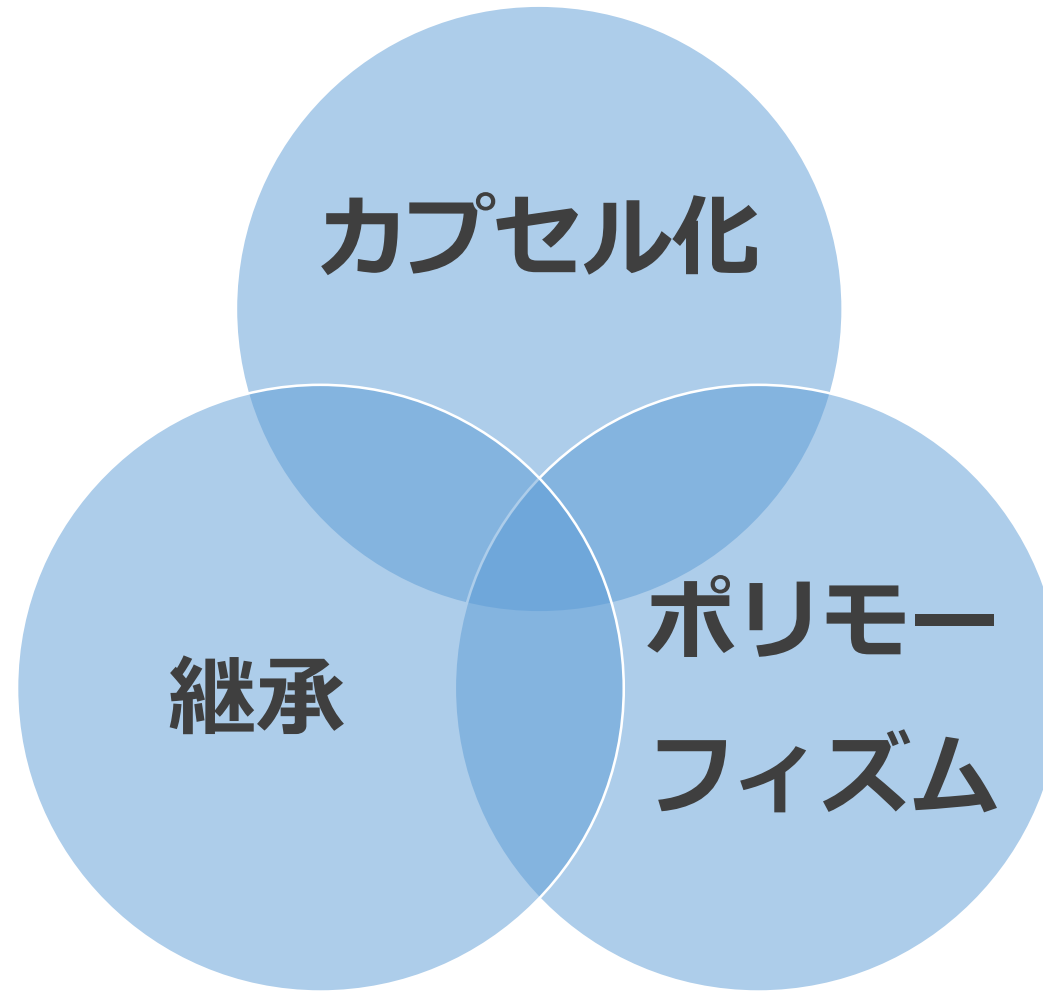
```
class Test
{
    public:
        int value;
        int getValue();
};
```

定義をヘッダかコードの先頭に一度書けば、
クラス名 :: メソッド名でそのメソッドを
どこでも定義可能

```
int Test::getValue()
{
    return value;
}
```

クラスの定義だけ集めたヘッダファイル
(拡張子hpp)を作り、アプリ側でヘッダ
ファイルをinclude するという使い方が多い

オブジェクト指向 カプセル化



※昔は、抽象データ型、派生、仮想関数の3要素

□関数と変数をまとめる

- 長い関数名から解放

- 関数を探しやすく

† 情報隠蔽：外部から使用には不必要な変数，関数を隠すこと

‡ インターフェイス：データ等へのアクセス手段

□情報隠蔽†し，必要な情報は**インターフェイス‡**を明示的に公開する

- 必要な操作が限定されて，中身を知らなくても使いやすく（インターフェイスの理解だけでよい）

- 状態，振る舞いを隠蔽し，プログラム開発の**分担**が可能

- スコープによるアクセス権制御よりも高度な制御が可能

※カプセル化では，使用するのに必要な操作とデータを抽象化することが必要
これは，クラスの大本になった抽象データ型（参考資料）で定義された概念

- publicとprivateでクラスの外からアクセスできるかどうかを制御
 - デフォルトはprivate
 - publicと宣言した場所以降は全部public

```
class TestClass
{
private://コロンを忘れずに
    int i;
    int j;
    int method_a() {printf("func a");};
public:
    int k;
    int l;
    int method_b() {printf("func b %d ", i);};
};
```

隠蔽されたスコープ：外から（このクラスのメソッド関数以外）使えない

公開されたスコープ：外から使える

※protectedとかfriendもあるよ

メンバー変数などをプライベートにすると 24

```
class PrintClass
{
private:
    int value;//メンバ変数
    void print()//メソッド
    {
        printf("value is %d¥n", value);
    }
};

int main()
{
    PrintClass v;
    v.value=100;
    v.print();
}
```

コンパイル時にエラー！

error: 'value' is a private member of
'PrintClass'

- 隠蔽の意味は、あくまでもprivate な変数、関数の
スコープが隠されているということであり、外部から完全にアクセスできないの意味ではない。
- private 変数に対して、読み書きを行う公開メソッドを準備しておけば間接的にprivate 変数にアクセスできる

何を公開して、何を非公開にすればよいの？26

- メンバ変数は基本として、**private** にして外部から**隠蔽することが望ましい**（メンバ変数へのアクセスは公開メソッドからのみとする）←クラスの利用者はメンバ変数の詳細を知る必要なし。
- 外部へ公開する公開メソッドのみを公開
- 将来的にメンバ変数に関する変更があっても、外部からは直接メンバ変数へアクセスがなければ、**このクラスだけの修正**に留めることができる

```
#include <stdio.h>

class Stack
{   メンバ変数隠蔽
private:
    int array[50];
    int pos;

public:
    Stack()
    {   コンストラクタ
        pos = 0;
    }
}
```

```
void push(int x)
{   公開メソッド
    array[pos] = x;
    pos++;
}

int pop()
{
    pos--;
    return array[pos];
}

};
```

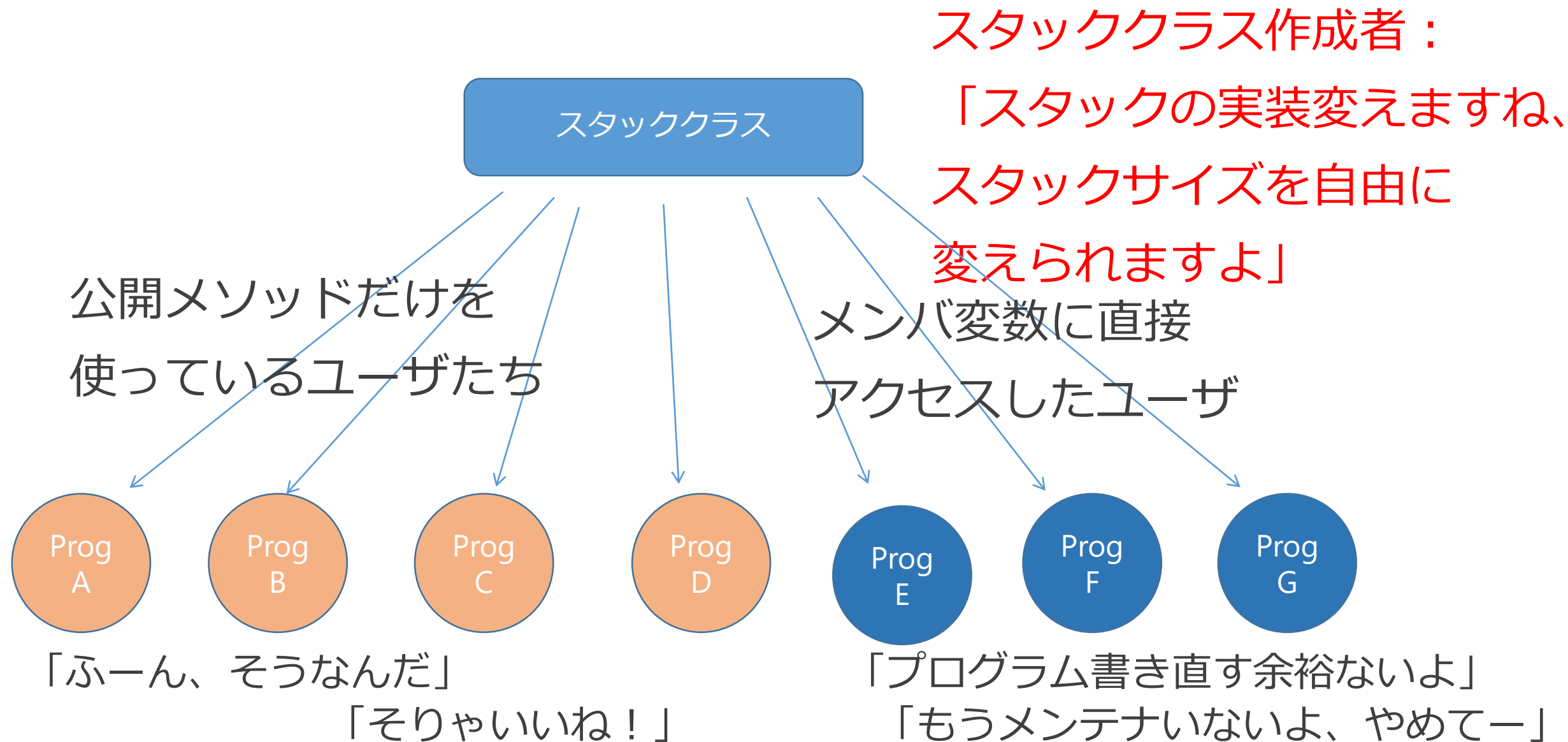
```
int main()
{
    Stack s;
    s.push(1);
    s.push(2);
    s.push(3);

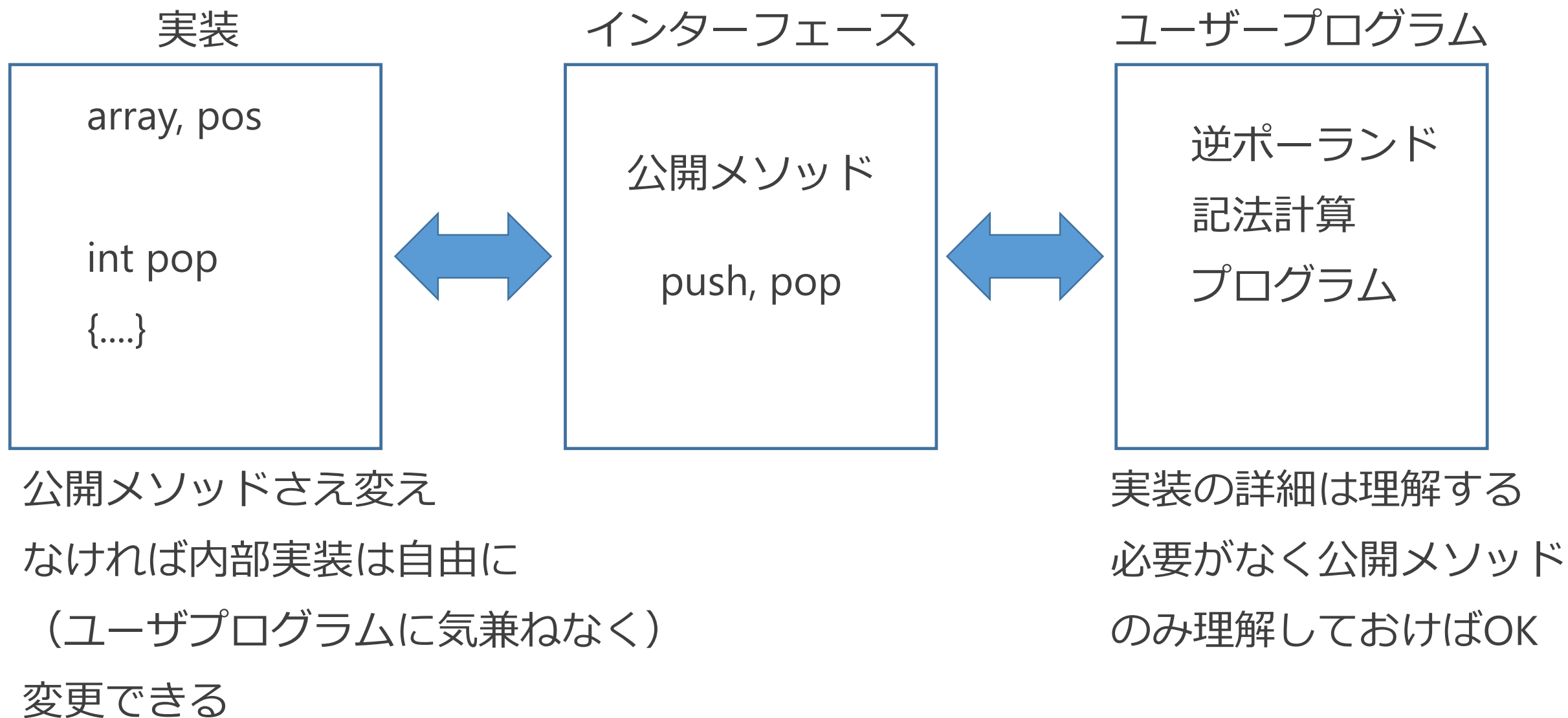
    printf("%d¥n",s.pop());
    printf("%d¥n",s.pop());
    printf("%d¥n",s.pop());
}
```

もし、メンバ変数が隠蔽されていないければどんな問題が起こり得る？

- 隠蔽しなくてもアクセスしなきゃいいんじゃないか？
- 例えば前のページの例で、main側でスタック内に格納されている要素数が知りたいニーズが出た.
- えーい、pos をmain側から直接読んじゃえ、となりませんか？
- pos を直接読むとstackの実装が変わったとき（例えばリストに基づく実装）main側のプログラムを修正する必要があります.
- Cではこのような場合にアクセスを禁止する手段はない.
- 正しいアプローチは、stackクラスにposの値を返す公開メソッドを追加するか、もしくは、そのようなメソッドを持つ派生クラス（後述）をユーザ側が作成するべし.

```
/*  
このstackライブラリを使う何人たりとて、  
arrayとposを直接にアクセスしてはならん。  
もしこれを破りし者あらば、その身に災い  
が降りかからん。  
*/  
  
#include <stdio.h>  
#define STACK_SIZE 50  
  
struct Stack {  
    int array[STACK_SIZE];  
    int pos;  
};
```





```
class TestClass
{
private:
    int i;
    int j;
    int method_a() {printf("func a");};
public:
    int k;
    int l;
    int method_b() {printf("func b %d ", i);};
};
```

プライベートメソッドは、外部に対して非公開。公開メソッド関数がクラス内で利用するクラス内部用の関数。クラスの利用者は、プライベートメソッドのインターフェース・実装など詳細を理解する必要ない。

- C++での情報隠蔽のスライド中のTestClassのインスタンスをmain関数内で作成し, method_b()をコールせよ.
- k,lに値を代入し, 中身をprintfせよ. 値へのアクセス方法は構造体と同じでドット(.)のオペレータである.
- i,jにも同様な操作をし動作を確認せよ.
- method_a()をコールして動作を確認せよ
- method_a()を呼び出すことが出来る関数をクラス内に作成し、その動作を確認せよ.

```
class TestClass
{
private://コロンを忘れずに
    int i;
    int j;
    int method_a() {printf("func a");};
public:
    int k;
    int l;
    int method_b() {printf("func b %d ", i);};
};
```

- 名前と学籍番号をデータとして持ち、showメソッドで名前と学籍番号を表示できるクラスを作成せよ
 - なお、名前と学籍番号を設定するための公開メソッドも必要
- main関数で上記インスタンスを作成し、上のクラスの機能を確認するコードを作成せよ
 - 文字列はstringクラスを利用してもよいし、Cの文字列をつかっててもよい。stringをちょっと調べて使えるようになった方が楽？

- スライド中のスタッククラスを次の形でファイルを分けよ.
 - ヘッダファイル: `stack.hpp` (この中でメソッドの実装は書かない)
 - クラス定義ファイル: `stack.cpp` (すべてのメソッド実装を書く)
 - ユーザプログラム: `main.cpp`(冒頭で `#include "stack.hpp"` とする)
- 分割したファイルをコンパイルして動作を確認せよ.
 - コマンド : `g++ -o stack stack.cpp main.cpp`
- 適切なMakefileを作り, `make` コマンドを利用してコンパイルせよ
 - コマンド : `make` でコンパイルが終わるMakefileを作成する
- スタッククラスに格納されている要素の個数を返す公開メソッドを追加せよ.

参考資料

- 初期化, メモリの確保, メモリの解放をクラスのインスタンスが出来上がったときと消滅するときにやってくれるメソッド
- これがあるだけで使用者はメモリの確保, 解放の動作をしなくてよくなるため非常に楽になる

- malloc, freeの代わりにC++の確保解放関数 new, deleteを使っているので注意.
- クラスはmalloc, freeできない

- 日本語で書けば関数の多重定義：引数が違えば同じ名前で違う挙動をする関数がいくつも書ける
- 例えばfunc(int a)とfunc(short a)で全く違う動作を定義可能（ただし、そのようなトリッキーなコードはバグの要因になるのでオーバーロード時は似たような操作を書くこと。）