

# アクセス指定子とカプセル化

- 教科書P.114~115 Sample303
- C++用フォルダにSample303フォルダを作成  
`mkdir Sample303`  
`cd Sample303`
- sample.h、sample.cpp、main.cppを新規作成  
`copy nul sample.h`  
`copy nul sample.cpp`  
`copy nul main.cpp`

# アクセス指定子とカプセル化

- sample.h

```
#pragma once
```

```
class Sample {  
public:  
    int a;  
    void func1();  
private:  
    int b;  
    void func2();  
};
```

# アクセス指定子とカプセル化

- sample.cpp

```
#include "sample.h"
#include <iostream>
using namespace std;

void Sample::func1() {
    cout << "func1" << endl;
    a = 1;
    b = 2;
    func2();
}
```

# アクセス指定子とカプセル化

- sample.cpp

```
void Sample::func2() {  
    a = 2;  
    b = 2;  
    cout << "a=" << a << ", " << "b=" << b  
    << endl;  
}
```

# アクセス指定子とカプセル化

- main.cpp

```
#include "sample.h"
```

```
int main()
```

```
{
```

```
    Sample s;
```

```
    s.a = 1;
```

```
    //s.b = 2;
```

```
    s.func1();
```

```
    //s.func2();
```

```
    return 0;
```

```
}
```

# アクセス指定子とカプセル化

- コンパイルの仕方

コマンドプロンプトで次のコマンドを入力する

```
cl _/EHsc _main.cpp _sample.cpp
```

重要なのはcppファイルをすべて書くこと

成功したら、main.exeを実行して結果を確認

# アクセス指定子とカプセル化

- sample.h

```
#pragma once
```

```
class Sample {  
public:
```

```
    int a;  
    void func1();
```

クラスの内外からアクセス可能  
→ main.cpp から使用可能

```
private:
```

```
    int b;  
    void func2();
```

クラス内でのみアクセス可能  
→ main.cpp から使用不可

```
};
```

# アクセス指定子とカプセル化

- `private`を使う理由

変数や関数を`private`にすることで、

- 思いもよらない値の変更
- 意図しない関数の実行

を阻止できる！

結果、プログラムのバグの発生を抑制できる

構造体との大きな違いは  
このあたりにある



# アクセス指定子とカプセル化

- sample.cpp

```
#include "sample.h"
#include <iostream>
using namespace std;

void Sample::func1() {
    cout << "func1" << endl;
    a = 1; ← public な変数へ値を代入(代入可)
    b = 2; ← private な変数へ値を代入(代入可)
    func2(); ← private な関数の実行(実行可)
}
```

# アクセス指定子とカプセル化

- main.cpp

```
#include "sample.h"
```

```
int main()  
{
```

```
    Sample s;
```

```
    s.a = 1; ← publicな変数へ値を代入(代入可)
```

```
    //s.b = 2;
```

```
    s.func1(); ← publicな関数を実行(実行可)
```

```
    //s.func2();
```

```
    return 0;
```

```
}
```

# アクセス指定子とカプセル化

- main.cpp

```
#include "sample.h"
```

```
int main()
```

```
{
```

```
    Sample s;
```

```
    s.a = 1;
```

```
    s.b = 2; ← private な変数へ値を代入(不可)
```

```
    s.func1();
```

```
    s.func2(); ← private な関数を実行(不可)
```

```
    return 0;
```

```
}
```

コンパイルエラーが発生

# アクセス指定子とカプセル化

- 教科書P.119~120 Sample304
- C++用フォルダにSample304フォルダを作成  
`mkdir Sample304`  
`cd Sample304`
- sample.h、sample.cpp、main.cppを新規作成  
`copy nul sample.h`  
`copy nul sample.cpp`  
`copy nul main.cpp`

# アクセス指定子とカプセル化

- sample.h

```
#pragma once

class Sample {
public:
    void setNum(int num);
    int getNum();
private:
    int m_num;
};
```

# アクセス指定子とカプセル化

- sample.cpp

```
#include "sample.h"

void Sample::setNum(int num) {
    m_num = num;
}

int Sample::getNum() {
    return m_num;
}
```

# アクセス指定子とカプセル化

- main.cpp

```
#include "sample.h"
#include <iostream>
using namespace std;

int main()
{
    Sample s;
    s.setNum(5);
    cout << s.getNum() << endl;
    return 0;
}
```

# アクセス指定子とカプセル化

- コンパイルの仕方

コマンドプロンプトで次のコマンドを入力する

```
cl _/EHsc _main.cpp _sample.cpp
```

成功したら、main.exeを実行して結果を確認



# アクセス指定子とカプセル化

- sample.h

```
#pragma once
```

```
class Sample {
```

```
public:
```

```
    void setNum(int num); ← セッター(値の代入)
```

```
    int getNum(); ← ゲッター(値の取得)
```

```
private:
```


```
    int m_num; ← privateな変数
```


```
};
```

# アクセス指定子とカプセル化

- sample.cpp

```
#include "sample.h"
```

```
void Sample::setNum(int num) {  
    m_num = num;  privateな m_num に代入  
}
```

```
int Sample::getNum() {  
    return m_num;  privateな m_num を参照  
}
```

# アクセス指定子とカプセル化

- セッター(**setter**)とゲッター(**getter**)  
privateなメンバ変数にアクセスするための  
publicなアクセス関数のこと
- セッターはprivateメンバ変数に値をセット
- ゲッターはprivateメンバ変数から値をゲット

メンバ変数にアクセスできる手段を関数経由に限定することで  
不意の書き換えや読み取りを防止する(カプセル化)

## 【参考】ハンガリアン記法

- プログラミングを行う際に、変数や定数、クラス名や関数名を付ける必要がある
- 統一した指針に則って命名することで、可読性がアップする
- テキストはハンガリアン記法に則って命名している

# 【参考】ハンガリアン記法

- 例

`m_speed` は「`m_`」を付けることでメンバ変数であることを表し、「`speed`」で速度を格納する変数であることを示している

セッター `setSpeed()` は「`set`」+「`speed`」で速度を格納することがわかりやすくなる

# アクセス指定子とカプセル化

- 【練習】 Sample301のカプセル化

コマンドプロンプトで次のコマンドを入力する  
Sample301の内容がSample301cフォルダ内に  
複製される

```
robocopy Sample301 Sample301c  
cd Sample301c
```

# アクセス指定子とカプセル化

- 【練習】 Sample301のカプセル化

- ① `car.h` 内に`public`なセッター(`setSpeed`)を定義する
- ② `public`な`speed`を`private`に移動し、変数名を`m_speed`に変更する
- ③ `car.cpp` にセッターの処理を追加する
- ④ `main.cpp` を変更してセッターを使って値を代入するように書き換える

# Sample301のカプセル化

- car.h (Sample301c)

```
#pragma once
```

```
class Car {  
public:  
    void drive(double hour);  
    void setSpeed(double speed);  
private:  
    double m_speed;  
};
```



# Sample301のカプセル化

- car.cpp (Sample301c)

```
#include "car.h"
#include <iostream>
using namespace std;

void Car::drive(double hour){
    cout << "時速" << m_speed << "kmで" <<
        hour << "時間走行" << endl;
    cout << m_speed * hour << "km移動" << endl;
}

void Car::setSpeed(double speed){
    m_speed = speed;
}
```

# Sample301のカプセル化

- main.cpp (Sample301c)

(略)

```
int main() {  
    Car nbox, tanto;  
    nbox.setSpeed(40);  
    nbox.drive(1.5);  
    tanto.setSpeed(50);  
    tanto.drive(1.0);  
    return 0;  
}
```