THE POSSIBILITIES ARE INFINITE FUJITSU

【JaSST'09 Tokyo テクノロジーセッション D3-1】

C++の知っておくべき言語仕様と安全なコーディングのススメ

2009年1月28日

株式会社富士通ソフトウェアテクノロジーズ
AP基盤サービス事業部 第一AP基盤サービス部
佐々木 孝次

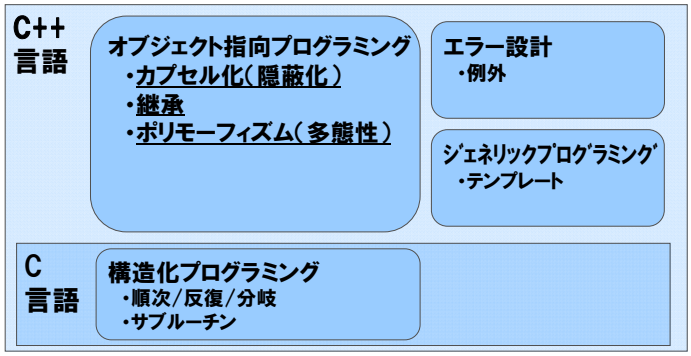
目次

FUJITSU

1. C++言語とは

C++言語とは

C言語に、オブジェクト指向プログラミング、エラー設計、ジェネリックプログラミングなどの手法を拡張した言語



FUJITSU

2. 誤りやすいコーディング例

- 問題
- 解答

問題(1) ～カプセル化～

プログラムの実行結果は？
① メンバの複写が正しく行われ、"123"と3回表示される。
② メンバの複写が正しく行われず、3回目の表示に失敗する。

```
class A {
  public:
    //コンストラクタ、
    //引数の長さの領域を確保し、メンバへ複写
    A( const char* p ) :
        m( new char[ strlen(p) + 1]) {
            strcpy( m , p ) ;
    }
    //デストラクタ、メンバの領域を解放
    ~A() {
        delete m ;
    }
    //文字列のポインタ取出し
    const char * getm() { return m; }
  private:
    char* m;
};
```

```
void main() {
    A a1("123") ;
    //1回目の表示
    cout << a1.getm() << endl;
    {
        A a2(a1) ;//a2をa1で初期構築
        //2回目の表示
        cout << a2.getm() << endl;
    }
    //3回目の表示
    cout << a1.getm() << endl;
}
```


解答（１） ～カプセル化～

FUJITSU

解答：②メンバの複写が正しく行われず、3回目の表示に失敗する。

```
class A {      //誤ったコーディング例
  public:
    A ( const char* p ) :
        m ( new char [ strlen (p) + 1]) {
            strcpy ( m , p ) ;
    }
    ~A ( ) {
        delete m ;
    }
  private:
    char* m;
};
        :
    A a2 ( a1 ) ;
```

デフォルトコピーコンストラクタで暗黙に代入が実施される a2.m=a1.m;//アドレス複写 一方のデストタクタ実行後に破棄された領域を参照する

```
class A {      //正しいコーディング例
  public:
    A ( const char* p ) :
        m ( new char [ strlen (p) + 1]) {
            strcpy ( m , p ) ;
    }
    A ( const A&a ) { //コピーコンストラクタ
        m = new char [ strlen (a.m) + 1 ] ;
        strcpy ( m , a.m ) ;
    }
    ~A ( ) {
        delete m ;
    }
  private:
    char* m;
};
        :
    A a2 ( a1 ) ;
```

コピーコンストラクタを定義し、ポインタ型データメンバを正しく複写

コーディング時の注意点： ポインタ型データメンバが含まれるクラスは、コピーコンストラクタ定義し、ポインタ型データメンバを適切に複写しましょう。
コピー代入演算子も同様です。

問題（２） ～カプセル化～

FUJITSU

プログラムの実行結果は？
① メンバ初期化リストの記述順で初期化されるため、”1,1”と表示される。
② メンバ変数の宣言順で初期化されるため、”不定な値,1”と表示される。

```
class A {
  public:
    A ( int x ) :
        m_y ( x ),
        m_x ( m_y ) {
    }
    void print () {
        cout << m_x << "," << m_y << endl;
    }
  private:
    int m_x;
    int m_y;
};
```

```
void main ( ) {
    A a (1) ;
    a.print () ;
}
```


解答（２） ～カプセル化～

FUJITSU

解答：②メンバ変数の宣言順で初期化されるため、”不定な値,1”と表示される。

```
class A {      //誤ったコーディング例
  public:
    A ( int x ) :
        m_y ( x ),
        m_x ( m_y ) {
    }
    void print () {
        cout << m_x << "," << m_y << endl;
    }
  private:
    int m_x;
    int m_y;
};
    :
  A a (1) ;
  a.print () ;
```

m_xは未初期化のm_yの値が設定される

m_x、m_yの順で初期化される

```
class A {      //正しいコーディング例
  public:
    A ( int x ) :
        m_x ( x ),
        m_y ( m_x ) {
    }
    void print () {
        cout << m_x << "," << m_y << endl;
    }
  private:
    int m_x;
    int m_y;
};
    :
  A a (1) ;
  a.print () ;
```

メンバ変数の宣言順でメンバ初期化リストを記述

コーディング時の注意点：
- メンバ初期化リストの記述順とメンバ変数の宣言順を揃えて記述しましょう
- メンバ初期化リストで初期値として参照する変数は、初期化されている変数を使用しましょう

問題（３） ～ポリモーフィズム～

FUJITSU

プログラムの実行結果は？
① ポリモーフィズムな関数呼出が行われず、”A::f () ”と表示される。
② ポリモーフィズムな関数呼出が行われ、“B::f () ”と表示される。

```
class A {
  public:
    void f ( ) {
        cout << "A::f () " ;
    }
} ;
class B : public A {
  public:
    void f ( ) {
        cout << "B::f () " ;
    }
} ;
```

```
void main ( ) {
    A* a = new B ;
    a->f ( ) ;
}
```


解答（３） ～ポリモーフィズム～

FUJITSU

解答：①ポリモーフィズムな関数呼出が行われず、”A::f () ”と表示される。

```
class A {      //誤ったコーディング例
  public:
    void f ( ) {
        cout << "A::f () " ;
    }
} ;
class B : public A {
  public:
    void f ( ) {
        cout << "B::f () " ;
    }
} ;
        :
    A* a = new B ;
    a->f ( ) ;   //”A::f () ”が呼ばれる
```

基本クラスの非仮想関数を再定義しても、ポリモーフィズムな関数呼び出しが行われない

```
class A {      //正しいコーディング例
  public:
    virtual void f ( ) {
        cout << "A::f () " ;
    }
} ;
class B : public A {
  public:
    void f ( ) {
        cout << "B::f () " ;
    }
} ;
        :
    A* a = new B ;
    a->f ( ) ;   //”B::f () ”が呼ばれる
```

動作の変更を許す場合は、仮想関数（virtual付き）として定義する

コーディング時の注意点：
非仮想関数の再定義はやめましょう

アドバイス（仮想関数の使い分け）：
- 純粋仮想関数（virtual=0付き）
  インターフェイスの宣言時に使います
- 仮想関数（virtual付き）
  デフォルトの動作を定義する時に使います
- 非仮想関数
  変更できない動作を定義する時に使います

問題（４） ～ポリモーフィズム～

FUJITSU

プログラムの実行結果は？
① 派生クラス、基本クラスの順にデストラクタが呼ばれ、”BA”と表示される。
② 派生クラスのデストラクタが呼ばれず、”A”と表示される。
③ 未定義な動作のため、”BA”、 ”A”のどちらが表示されるかはわからない。

```
class A {
  public:
    A ( ) { }
    ~A ( ) {
        cout << “A”;
    }
    virtual void f ( ) { }
} ;
class B : public A {
  public:
    B ( ) { }
    ~B ( ) {
        cout << “B”;
    }
    void f ( ) { }
} ;
```

```
void main ( ) {
    A* a = new B ;
    delete a ;
}
```

解答（４） ～ポリモーフィズム～

FUJITSU

解答：③未定義な動作のため、”BA”、”A”のどちらが表示されるかはわからない。

```
class A {     //誤ったコーディング例
  public:
    ~A() {
       cout << "A";
    }
};
class B : public A {
  public:
    ~B() {
       cout << "B";
    }
};
        :
   A* a = new B;
   delete a;
```

基本クラスのデストラクタが非仮想関数のため、派生クラスのデストラクタ呼び出しは未定義な動作になる

```
class A {     //正しいコーディング例
  public:
    virtual ~A() {
       cout << "A";
    }
};
class B : public A {
  public:
    ~B() {
       cout << "B";
    }
};
        :
   A* a = new B;
   delete a;  //"BA"が表示される
```

基本クラスのデストラクタは仮想関数（virtual付き）として定義する

コーディング時の注意点：

基本クラスのデストラクタを仮想関数として宣言しましょう。

アドバイス：

virtualを付けると、オブジェクトサイズが大きくなったり、性能劣化の可能性があります。
仮想関数の必要があるか、よく考えましょう。

問題（５） ～継承～

FUJITSU

プログラムの実行結果は？
① 呼出関数と異なる関数のデフォルト引数が有効になり、”B::f (a) ”と表示される。
② 呼出関数と同じ関数のデフォルト引数が有効になり、”B::f (b) ”と表示される。

```
class A {
  public:
    virtual void f( char c='a' ) {
       cout << "A::f (" << c << ") ";
    }
};

class B : public A {
  public:
    void f( char c='b' ) {
       cout << "B::f (" << c << ") ";
    }
};
```

```
void main() {
   A* a = new B;
   a->f();
}
```


解答（５） ～継承～

FUJITSU

解答：①呼出関数と異なる関数のデフォルト引数が有効になり、”B::f (a) ”と表示される。

```
class A {     //誤ったコーディング例
  public:
    virtual void f( char c='a' ) {
       cout << "A::f (" << c << ") ";
    }
};
class B : public A {
  public:
    void f( char c='b' ) {
       cout << "B::f (" << c << ") ";
    }
};
        :
   A* a = new B;
   a->f();
```

デフォルト引数はこっち

呼び出しはこっち

格納されている型で定義されている仮想関数のデフォルト引数になる

```
class A {     //正しいコーディング例
  public:
    virtual void f( char c='a' ) {
       cout << "A::f (" << c << ") ";
    }
};
class B : public A {
  public:
    void f( char c='a' ) {
       cout << "B::f (" << c << ") ";
    }
};
        :
   A* a = new B;
   a->f();
```

デフォルト引数の値を同じにする

コーディング時の注意点： 同じ仮想関数のデフォルト引数の値は、同じ値にしましょう。

FUJITSU

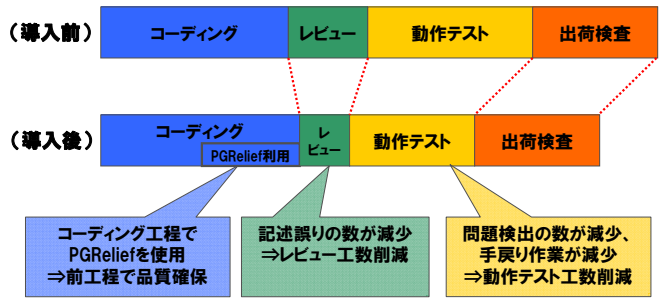
3. PGReliefによる自動検出

- PGReliefの導入効果
- PGReliefの特徴

PGReliefの導入効果

FUJITSU

**コーディング工程で利用することで、問題検出の自動化と、
前工程での問題検出が可能になり、工数削減・品質向上が図れます**



PGReliefの特徴

FUJITSU

■ 高いヒット率

データフロー解析・ロジックフロー解析により、プログラム欠陥の可能性を的確に指摘します。

■ 充実したドキュメント

各々の指摘メッセージについて、詳細な解説が用意されています。
簡単な例を交えて、そのメッセージが指摘された理由や対処例なども記載されていますので的確に対処することができます。

■ 安心なサポート

開発およびサポートは、日本国内で行っています。
問題対応、お問い合わせの回答は、迅速に行っています。
また、ご要望もお聞きし、製品へ反映しています。

FUJITSU

THE POSSIBILITIES ARE INFINITE

FUJITSU

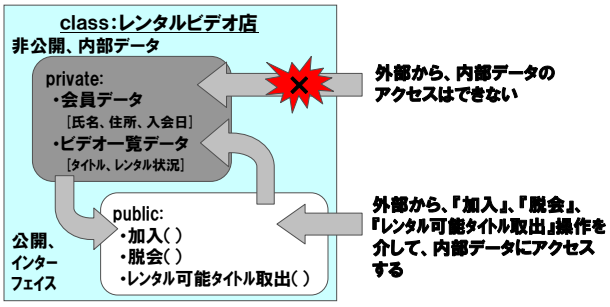
付録

・オブジェクト指向プログラミングとは
・誤りやすいコーディング例（本日説明しなかった例）

オブジェクト指向プログラミングとは（カプセル化）

FUJITSU

動作と状態を一体化し、ブラックボックス化する仕組み

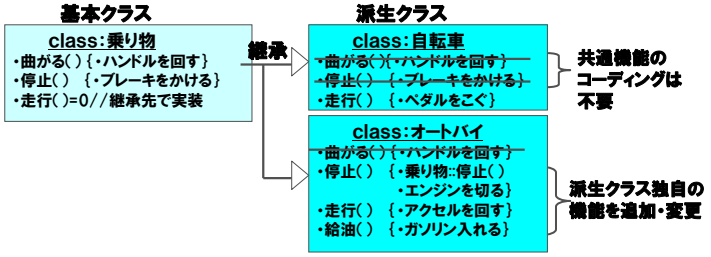


効果
- 内部の動作や状態を隠蔽できる
- 単純で使い易いインターフェイスを提供できる

オブジェクト指向プログラミングとは（継承）

FUJITSU

あるクラスの機能（動作/状態）を他のクラスに引き継ぎ、さらに機能を追加したり変更したりする仕組み



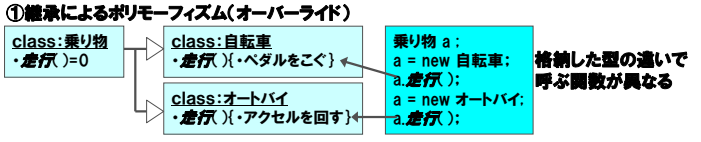
効果
- コーディング量を減らすことができる
- 機能追加や変更が容易にできる

オブジェクト指向プログラミングとは（ポリモーフィズム）

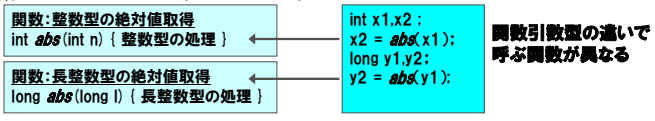
FUJITSU

同じ目的の複数の動作を、同じ名前の関数でさせる仕組み

①継承によるポリモーフィズム（オーバーライド）



②引数によるポリモーフィズム（オーバーロード）



効果
- 関数名を減らすことできる
（使う人が覚える関数名が少なくて済む）

問題（６） ～カプセル化～

FUJITSU

プログラムの実行結果は？
① コンストラクタと変換関数が呼ばれ、”123”と表示される。
② コンストラクタと代入演算子が呼ばれ、“1230”と表示される。

```
class A {
 public:
    A( int x ) : m(x) {     //コンストラクタ
    }
    A& operator= ( int x ) { //代入演算子
        m = x*10 ;
        return *this;
    }
    operator int() {        //変換関数
        return m;
    }
 private:
    int m;
};
```

```
void main ( ) {
    A a = 123;
    cout << a << endl;
}
```

解答（６） ～カプセル化～

FUJITSU

解答：①コンストラクタと変換関数が呼ばれ、”123”と表示される。

```
class A {     //誤ったコーディング例
  public:
    A( int x ) : m(x) {     //コンストラクタ
    }
    operator int() {        //変換関数
      return m;
    }
  private:
    int m;
};
  A a = 123;
  cout << a << endl;
```

コンストラクタが呼ばれる

変換関数が呼ばれる

暗黙的な呼び出しは危険
・意図しない箇所から呼び出される
・影響範囲が見つけにくい

```
class A {     //正しいコーディング例
  public:
    explicit A( int x ) : m(x) {
    }
    int getm() {
      return m;
    }
  private:
    int m;
};
  //A a = 123;  ←コンパイルエラー
  A a(123);
  cout << a.getm() << endl;
```

explicitを付ける
⇒暗黙呼出でコンパイルエラー

変換関数は使わず、明示的に定義する

コーディング時の注意点：
・単一引数のコンストラクタには、explicitを指定しましょう
・変換関数は使わず、呼び出しが明示的に分かる通常のメンバ関数を使いましょう

問題（７） ～カプセル化～

FUJITSU

プログラムの実行結果は？
① 全メンバが複写され、”10,20”と表示される。
② 一部メンバが複写されず、“10,2”と表示される。
③ 一部メンバが複写されず、” 0,2”と表示される。

```
class A {
  public :
    //コンストラクタ
    A( int n1,int n2 ) :
      m_n1( n1 ) , m_n2( n2 ) {
    }
    //コピー代入演算子
    A & operator=( const A & a ) {
      m_n1 = a.m_n1 ;
      return *this ;
    }
    //メンバ変数表示
    void print() {
      cout << m_n1 << "," << m_n2 << endl;
    }
  private :
    int m_n1, m_n2 ;
} ;
```

```
void main( ) {
    A a1(1,2);
    A a2(10,20);
    a1 = a2;
    a1.print();
}
```


解答（７） ～カプセル化～

FUJITSU

解答：②一部メンバが複写されず、“10,2”と表示される。

```
class A {     //誤ったコーディング例
        :
    //コピー代入演算子
    A & operator=( const A & a ) {
      m_n1 = a.m_n1 ;
      return *this ;
    }
    private :
      int m_n1, m_n2 ;
} ;
        :
    a1 = a2;
```

m_n2の複写漏れ

```
class A {   //正しいコーディング例
        :
    //コピー代入演算子
    A & operator=( const A & a ) {
      m_n1 = a.m_n1 ;
      m_n2 = a.m_n2 ;
      return *this ;
    }
  private :
    int m_n1, m_n2 ;
};
        :
    a1 = a2;
```

全てのメンバを複写

コーディング時の注意点： コピー代入演算子では、staticメンバ変数とconstメンバ変数を除く全てのメンバ変数と、基本クラスのメンバ変数を複写しましょう。
コピーコンストラクタも同様です。

問題（８） ～継承～

FUJITSU

プログラムの実行結果は？
① ”1”が10回表示される。
② ”1”が5回表示され、その後の5回は何が表示されるか不明。

```
class A {
  public:
    A() : m(1) {}
    int m;
} ;
class B : public A {
} ;
//基本クラスのポインタを受け取り、
//配列数分の”m1”を表示する。
void f( A *pa, int n ) {
    for( int i = 0 ; i < n ; i++ )
        cout << pa[i].m << endl ;
}
```

```
void main( ) {
    A a[5];
    f( a, 5 ) ;  //基本クラスの配列を
                 //基本クラスとして処理する
    B b[5];
    f( b, 5 ) ;  //派生クラスの配列を
                 //基本クラスとして処理する
}
```


解答（８） ～継承～

FUJITSU

解答：② ”1”が5回表示され、その後の5回は何が表示されるか不明。

```
class A {  //誤ったコーディング例
  public:
    A() : m(1) {}
    int m;
} ;
class B : public A {
} ;
void f( A *pa, int n ) {
    for( int i = 0 ; i < n ; i++ )
        cout << pa[i].m << endl ;
}
      :
    A a[5];
    f( a, 5 ) ;
    B b[5];
    f( b, 5 ) ;
```

派生クラスと基本クラスでは型サイズが異なるため、ポインタを進めても正しいアドレスを示さない

```
//正しいコーディング例(その1)
void f( A *pa, int n ) {
    for( int i = 0 ; i < n ; i++ )
        cout << pa[i].m << endl ;
}
void f( B *pb, int n ) {
    for( int i = 0 ; i < n ; i++ )
        cout << pb[i].m << endl ;
}
    B b[5];
    f( b, 5 ) ;
```

基本クラスと派生クラスの引数とする関数を定義する

```
//正しいコーディング例(その2)
template< class T > void f( T &x ) {
  for( T::const_iterator it = x.begin() ;
       it != x.end() ; it++ )
       cout << ( *it ).m << endl ;
}
    B b[5];
    vector<B> arrayB;
    for( int i = 0 ; i < 5 ; i++ )
        arrayB.push_back( b[i] ) ;
    f( arrayB ) ;
```

コンテナを使ったオブジェクト操作にする

コーディング時の注意点： 派生クラスの配列を、基本クラスのポインタに渡すことはやめましょう

問題（９） ～例外・継承～

FUJITSU

例外引数のポインタ渡しと値渡しで発生する問題とは？

例1（ポインタ渡し、ポインタ捕捉）

```
class A {
};
void f1() {
    A* p = new A;
    throw p;      //確保した領域をポインタ渡し
}
void f2() {
    A a;
    throw &a;  //ローカル領域をポインタ渡し
}
void main() {
    try {
        f1();
        f2();
    }
    //ポインタで捕捉
    catch(A* a) {
    }
}
```

例2（値渡し、値捕捉）

```
class A {
};
class B : public A {
};
void f3() {
    //派生クラスの領域を値渡し
    B b;
    throw b;
}
void main() {
    try {
        f3();
    }
    //基本クラスの値で捕捉
    catch(A a) {
    }
}
```

注意）各関数は、例外発生時のコードのみが記述されています。

解説（９） ～例外・継承～

FUJITSU

解答：例１：メモリの解放漏れが発生する。
　　例２：捕捉したオブジェクトがポリモーフィズムな動作をしない。

```
//誤ったコーディング例
class A { }; //例1（ポインタ渡し、ポインタ捕捉）
void f1 ( ) {
    A* p = new A;
    throw p;
}
void f2 ( ) {
    A a;
    throw &a ;
}
        :
    catch (A* a) {
```

捕捉したポインタの領域が、動的かどうかを判断できない

```
//誤ったコーディング例
class A { }; //例2（値渡し、値捕捉）
class B : public A { };
void f3 ( ) {
    B b;
    throw b ;
}
        :
  catch (A a) {
```

派生クラスのオブジェクトを、基本クラスの値として捕捉すると、派生クラスの部分が切り捨てられる

```
//正しいコーディング例
class A { };
class B : public A { };
void f ( ) {
    A a;
    throw a;
    B b;
    throw b;
}
        :
  catch ( A& a ) {
```

値を投げる

基本クラスの参照で捕捉する

コーディング時の注意点：

例外は値を投げ、参照で捕捉しましょう

問題（１０） ～例外～

FUJITSU

プログラムの実行結果は？
① 再スローすると基本クラスになるため、”BA”が表示される。
② 再スローしても派生クラスのままで、”BB”が表示される。

```
class A {
    virtual void print ( ) { cout << "A" ; }
};
class B : public A {
    void print ( ) { cout << "B" ; }
};
void f ( ) {
    B b;
    throw b; //派生クラスをスロー
}
void f2 ( ) {
    try {
      f ( );
    } catch (A& a) { //基本クラスで捕捉
        a.print ( ); //1回目の表示
        throw a; //基本クラスを再スロー
    }
}
```

```
void main ( ) {
    try {
        f2 ( );
    } catch ( A &a ) { //基本クラスで捕捉
        a.print ( ); //2回目の表示
    }
}
```


解答（１０） ～例外～

FUJITSU

解答：①再スローすると基本クラスになるため、”BA”が表示される。

```
class A {//誤ったコーディング例
    virtual void print ( ) { cout << "A" ; }
};
class B : public A {
    void print ( ) { cout << "B" ; }
};
void f ( ) {
    B b;
    throw b;
}
void f2 ( ) {
    try {
      f ( );
    } catch (A& a) {
        a.print ( );
        throw a; //再スロー
    }
}
void main ( ) {
    try {
        f2 ( );
    } catch ( A &a ) {
        a.print ( );
    }
```

実引数を記述すると、宣言型の例外オブジェクトがスローされる

```
class A { //正しいコーディング例
    virtual void print ( ) { cout << "A" ; }
};
class B : public A {
    void print ( ) { cout << "B" ; }
};
void f ( ) {
    B b;
    throw b;
}
void f2 ( ) {
    try {
      f ( );
    } catch (A& a) {
        a.print ( );
        throw ; //再スロー
```

実引数を書かない

コーディング時の注意点：

再スロー先でポリモーフィズムな動作をさせるためには、実引数がないthrow式を記述しましょう