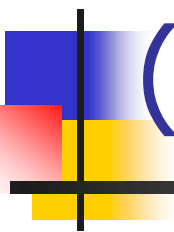


プログラミング及び演習

第14回 C++に触れてみる

(2017/07/28)



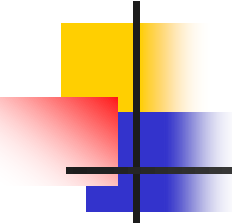
講義担当

大学院情報学研究科知能システム学専攻

教授 森 健策

大学院情報学研究科知能システム学専攻

助教 小田 昌宏



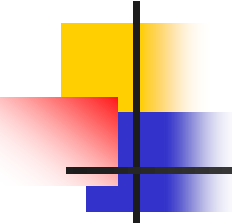
本日の内容

- C++

- 簡単にC++の機能を紹介します。

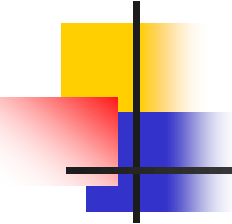
- 今日の2限で「プログラミング及び演習」の講義も終わりです

- この講義はいかがでしたか？
 - きっと皆さんのプログラミングの力もついたかと思います。
 - いつものように出席アンケートに回答してください



さらなる前進

- C++を勉強
 - C言語との違い
 - 変数宣言位置
 - 多重定義(オーバーロード)
 - 名前空間
 - オブジェクト指向プログラミング
 - カプセル化



C言語との違い(ファイル拡張子)

- C言語の場合

- ファイルの拡張子: .c
- コンパイラ: cc, gcc

- C++言語の場合

- ファイルの拡張子: .cpp .cxx .cc .cp
- コンパイラ: c++, g++
 - ※ gccでもコンパイル可能

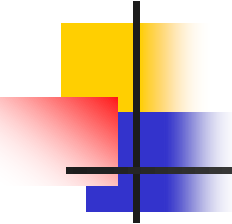
★ C++言語ではC言語の機能をほぼ全て利用可能

C言語との違い(変数宣言位置)

■ C言語の場合

- 関数の先頭に全ての変数を宣言
 - どの変数がどの範囲で利用しているかが分かりにくい

```
int i, j, k, m, n, num = 0, ...;  
for( k = 0 ; k < 100 ; k++ )  
    for( j = 0 ; j < 100 ; j++ )  
        for( i = 0 ; i < 100 ; i++ ){  
            m = n = 0; ← mとnはこの中でのみ使用  
            ...  
        }
```



C言語との違い(変数宣言位置)

■ C++言語の場合

- 変数が必要な所で宣言可能
 - 変数がどの範囲内でのみ有効かが分かる！！

```
int num = 0, ...;
```

```
for( int k = 0 ; k < 100 ; k++ )
```

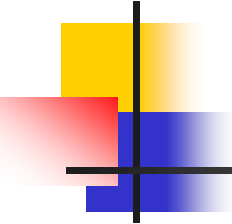
```
    for( int j = 0 ; j < 100 ; j++ )
```

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

```
            int m = 0, n = 0;
```

```
            ...
```

```
        }
```



C言語との違い(多重定義※)

■ C言語の場合

- int, double の値を加算する関数を作成する場合
 - 同じ処理をする関数なのに, 別々の名前が必要

```
int add_int( int a, int b ){ ... }
```

```
double add_double( double a, double b ){ ... }
```

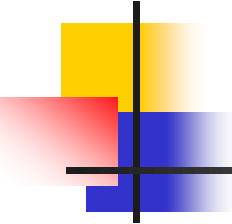
```
int a, b, c;
```

```
c = add_int( a, b );
```

```
double a, b, c;
```

```
c = add_double( a, b );
```

※ オーバーロードとも呼ばれる



C言語との違い(多重定義※)

■ C++言語の場合

- int, double の値を加算する関数を作成する場合
 - **引数の違う同じ名前の関数**を複数定義することが可能
 - ただし, 戻り値のみが違うものは不可

```
int add ( int a, int b ){ ... }
```

```
double add ( double a, double b ){ ... }
```

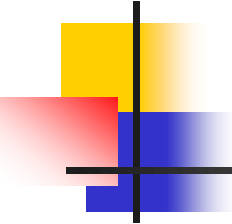
```
int a, b, c;
```

```
c = add( a, b );
```

```
double a, b, c;
```

```
c = add( a, b );
```

※ オーバーロードとも呼ばれる

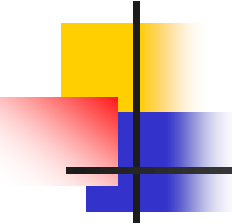


C言語との違い(名前空間)

- 名前の衝突を避けるための機能
 - 同じ名前の関数・構造体を複数定義可能
 - 同じ機能の実装を複数共存させることが可能
 - ※関数の場合, 同じ引数でも可
- 例:AさんとBさんが print_int 関数を作成
 - Aさん:

```
void print_int( int a ){ printf( "%d¥n", a ); }
```
 - Bさん:

```
void print_int( int a ){ printf( "Int Value: %d¥n", a ); }
```



C言語との違い(名前空間)

■ C言語の場合

- AとBの実装に別々の関数名を付ける必要あり
 - 関数の種類が増えると名前の管理がとても大変

```
void print_int_A( int a ){ printf( "%d¥n", a ); }
```

```
void print_int_B( int a ){ printf( "Int Value: %d¥n", a ); }
```

```
print_int_A( 1 );
```

```
print_int_B( 1 );
```

C言語との違い(名前空間)

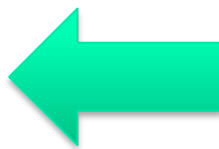
■ C++言語の場合

- 名前空間を用いてAとBの実装を区別可能

```
namespace A{  
    void print_int( int a ){ printf( "%d¥n", a ); }  
}  
namespace B{  
    void print_int( int a ){ printf( "Int Value: %d¥n", a ); }  
}
```

A::print_int(1);

B::print_int(1);



「名前空間 :: 関数名」
の形で利用

C言語との違い(名前空間)

■ C++言語の場合

- 「名前空間 :: 関数名」を毎回書くのが大変な時

➡ 名前空間の一括インポートが可能

```
namespace A{ ... }
```

```
namespace B{ ... }
```

```
{
```

```
    using namespace A;
```

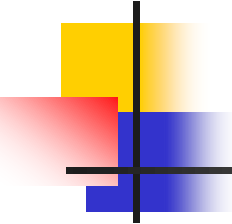
```
    print_int( 1 );
```

```
}
```



```
A::print_int( 1 );
```

と同じ効果



C言語との違い(デフォルト引数)

- 関数呼び出し時の引数を省略可能にする機能
 - パラメータ違いの別バージョンの関数を簡単に実現
- 例: $D = A + B \times C$ を実現する関数
 - 積和演算 (MAC: **M**ultiply and **AC**cumulation)
 - C言語の場合

```
int MAC_1( int a, int b, int c ){ return( a + b * c ); }
```

```
int MAC_2( int a, int b ){ return( a + b ); }
```

```
MAC_1( 3, 2, 1 );
```

```
MAC_2( 3, 2 );
```

} 同じ結果が得られる

C言語との違い(デフォルト引数)

- 例: $D = A + B \times C$ を実現する関数

- C++言語の場合

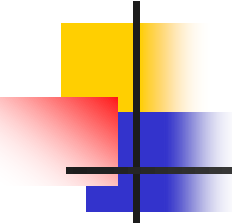
- 省略可能なパラメータの指定が可能

```
int MAC( int a, int b, int c = 1 ){  
    return( a + b * c );  
}
```

```
MAC( 3, 2, 1 );  
MAC( 3, 2 );
```

} 同じ結果が得られる

第3引数の1が省略されている



C言語との違い(入出力機能)

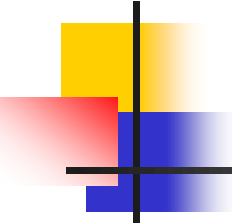
- ストリーム演算子(<<, >>)を使った入出力
 - ヘッダファイル
 - #include <iostream>
 - 標準入力
 - std::cout
 - 標準エラー出力
 - std::cerr
 - 標準入力
 - std::cin
 - 改行
 - std::endl

```
#include <iostream>

int main()
{
    std::cout << 100 << std::endl;

    int a;
    std::cin >> a;
    std::cout << a << std::endl;

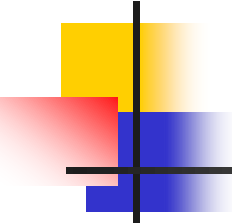
    return( 0 );
}
```



C++超入門 cincout.cpp

```
#include <iostream>
using namespace std;
// Simple example of iostream (cin and cout)

int main()
{
    int i;
    cout << "Atai:";
    cin >> i;
    cout << "Atai" << i << "¥n";
    return 0;
}
```



C++の新機能(クラス)

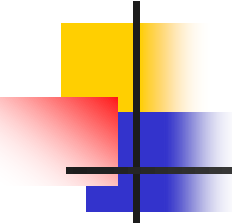
■ C言語の構造体

- 複数の値をまとめて格納できる型
- キーワード: struct

■ C++言語のクラス

- C言語の構造体＋関数を追加した型
 - コンストラクタ, デストラクタ
 - アクセス指定 (public, protected, private)
 - 継承
 - 仮想関数
- キーワード: class※

※ struct を使用しても同様の機能を実現可能



C++の新機能(クラス)

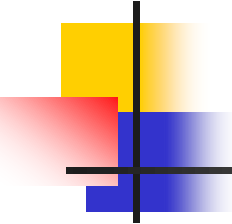
■ C言語の場合

```
struct A{  
    int num;  
};  
  
void init( struct *A){}  
void calc( struct *A ){}  
  
struct A a;  
init( &a );  
calc( &a );
```

■ C++言語の場合

```
class A{  
public:  
    int num;  
    void init( ){}  
    void calc( ){}  
};
```

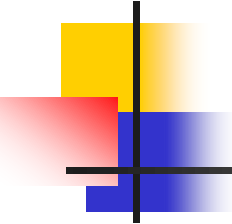
```
A a;  
a.init( );  
a.calc( );
```



C++超入門 simpleclass1.cpp

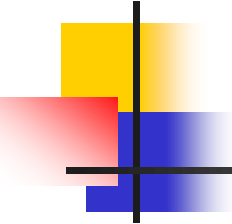
```
#include <iostream>
using namespace std;

class CMyClass{
public:
    CMyClass(){
        cout << "Constructor called¥n";
        value = 10;
    };
    ~CMyClass(){
        cout << "Destructor Called¥n";
    };
};
```



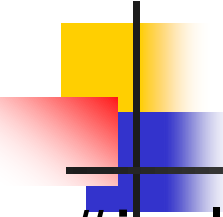
C++超入門 simpleclass1.cpp

```
...  
void num(int in){  
    value = in;  
};  
int num(){  
    return value;  
}  
private:  
    int value;  
};
```



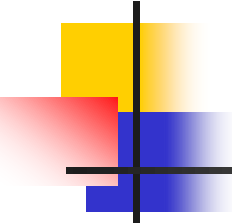
C++超入門 simpleclass1.cpp

```
int main()
{
    CMyClass cl1;
    CMyClass cl2;
    cl1.num(12);
    cl2.num(13);
    cout << "cl1 = " << cl1.num() << "¥n";
    cout << "cl2 = " << cl2.num() << "¥n";
    return 0;
}
```



C++超入門 simpleclass2.cpp

```
#include <iostream>
using namespace std;
class CMyClass{
public:
    CMyClass(){
        cout << "Constructor called¥n";
        value = 10;
    };
    ~CMyClass(){
        cout << "Destructor Called¥n";
    };
};
```



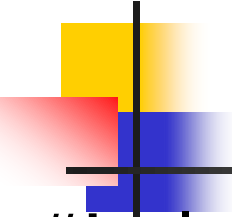
C++超入門 simpleclass2.cpp

```
...  
void num(int in){  
    value = in;  
};  
int num(){  
    return value;  
}  
private:  
    int value;  
};
```



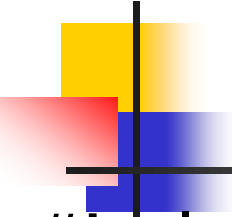
C++超入門 simpleclass2.cpp

```
void func()
{
    CMyClass cl1;
    cl1.num(12);
    cout << "cl1 = " << cl1.num() << "¥n";
}
int main()
{
    cout << "before calling func¥n";
    func();
    cout << "after calling func¥n";
    return 0;
}
```



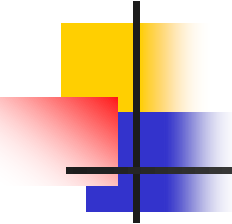
C++超入門 stack.cpp

```
#include <iostream>
using namespace std;
class CStack{
public:
    CStack(int size=100){
        stackSize = size;
        stack = new int[stackSize];
        cout << "alloc size " << stackSize << "¥n";
    };
    ~CStack(){
        delete[] stack;
    }
}
```



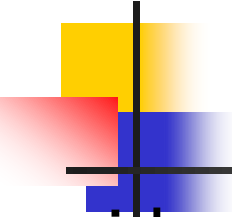
C++超入門 stack.cpp

```
#include <iostream>
using namespace std;
class CStack{
public:
    CStack(int size=100){
        stackSize = size;
        stack = new int[stackSize];
        cout << "alloc size " << stackSize << "¥n";
    };
    ~CStack(){
        delete[] stack;
    }
}
```



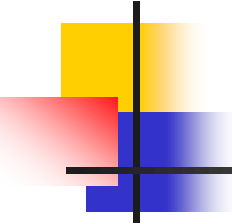
C++超入門 stack.cpp

```
void init();  
void push(int a);  
int pop();  
private:  
    int *stack;  
    int stackSize;  
    int tos;  
};
```



C++超入門 stack.cpp

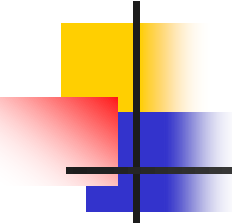
```
void
CStack::init(){
    tos = 0;
}
void
CStack::push(int a){
    if(tos==stackSize){
        cout << "Stack is full\n";
        return;
    }
    stack[tos] = a;
    tos++;
}
```



C++超入門 stack.cpp

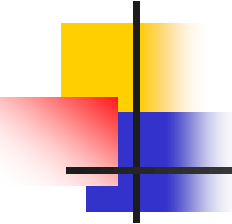
```
CStack::pop(){  
    if(tos==0){  
        cout << "Stack is empty¥n";  
        return 0;  
    }  
    tos--;  
    return stack[tos];  
}
```

```
int main()
{
    CStack s1(10);
    CStack s2;
    int i;
    s1.init();
    s2.init();
    s1.push(1);
    s2.push(2);
    s1.push(3);
    s2.push(4);
    s1.push(5);
    s2.push(6);
    for(i=0; i<3; i++){
        cout << "pop from s1 " << s1.pop() << "¥n";
    }
    for(i=0; i<3; i++){
        cout << "pop from s2 " << s2.pop() << "¥n";
    }
}
```



継承

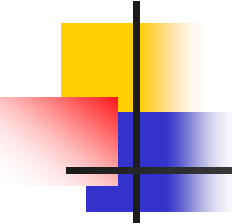
- 他のクラスの性質を受け継ぐための機構
- クラス階層を作成することが可能
 - 汎用的なクラス → 特化されたクラス
- あるクラスを別のクラスが継承する時
 - 継承される側のクラスを基本クラス
 - 継承する側のクラスを派生クラス



継承を使ったプログラムの例

```
#include <iostream>
using namespace std;
class CBase{
private:
    int i;
public:
    CBase(){
        i = 0;
    };
    void set_i(int n){
        i = n;
    };
    int get_i(){
        return i;
    };
};
```

```
class CDerived : public CBase{
private:
    int j;
public:
    CDerived(){
        j = 0;
    };
    void set_j(int n){
        j = n;
    };
    int get_j(){
        return j;
    };
    int add(){
        return j+get_i();
    };
};
```

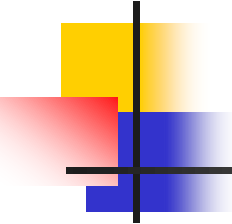


継承を使ったプログラムの例

```
int main()
{
    CDerived ob;
    cout << "add " <<
        ob.add() << '\n';
    ob.set_i(10);
    ob.set_j(50);
    ob.add();
    cout << "add " <<
        ob.add() << '\n';
}
```

■ 実行結果

```
taka{mori}33: g++ -o
    inheri inheri.cpp
taka{mori}34: ./inheri
add 0
add 60
```



派生クラスの作成

■ 記述方法

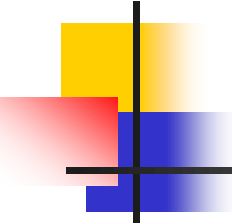
- Cderived が Cbase のすべての要素を継承する場合

```
class CDerived : public CBase{  
    ;  
};
```

■ 一般的な記述

```
class derived-class-name  
    : access-specifier base-class-name  
{ };
```

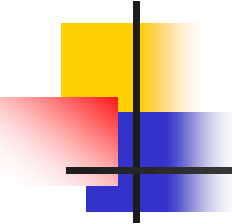
- access-specifier には public, protectedなどを指定
 - public は基本クラスのすべての公開要素を派生クラスで公開



Class(オブジェクト)の配列

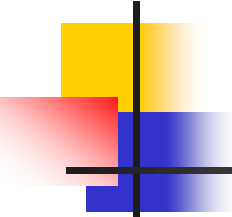
- 以下のようなプログラムも可能

```
int main()
{
    CDerived ob[10];
    int i;
    for(i=0; i<10; i++){
        ob[i].set_i(i);
        ob[i].set_j(i*20);
    }
    for(i=0; i<10; i++){
        cout << "add " << ob[i].add() << '\n';
    }
}
```



参照

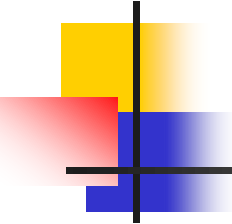
- 変数の別名として動作する暗黙的ポインタ
- 参照の用途は3通り
 - 参照を関数に渡す
 - 関数から参照を返す
 - 独立した参照を作成する



参照呼出しの例

- 参照を使うと呼び出された関数側で値を書き換え可能

```
#include <iostream>
using namespace std;
void func(int &m)
{
    m = m*10; // 注意！！
}
int main()
{
    int m = 5;
    cout << "m= " << m << '\n';
    func(m);
    cout << "m= " << m << '\n';
}
```



オブジェクトの参照渡し

```
void func1(CDerived &ob)  int main()
{
    ob.set_i(100);
    ob.set_j(200);
}
void func2(CDerived ob)   {
    CDerived ob1, ob2;
    func1(ob1);
    cout << "func1 add " <<
                                ob1.add() << '\n';
    func2(ob2);
    cout << "func2 add " <<
                                ob2.add() << '\n';
}
{
    ob.set_i(100);
    ob.set_j(200);
}
```

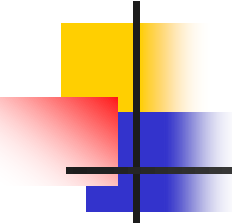
実行結果 -「参照渡し」と「値渡し」の結果の違いに注意

taka@mori: g++ -o inheri3 inheri3.cpp

taka@mori: ./inheri3

func1 add 300

func2 add 0



他にも多数のトピックス

- 関数のオーバーロード
- 演算子のオーバーロード
- 多重継承
- 仮想関数
- テンプレート
- STL (Standard Template Library)
- 興味がある方は
 独習C++ 第3版
 などで勉強してください。