ゲームプログラミング応用／ゲームプログラミング実践

10日で分かる C++言語 LEVEL③

■「値渡し」「ポインタ渡し」「参照渡し」

関数の引数には、C言語にあった「値渡し」「ポインタ渡し」に加え「参照渡し」があります。

1. 値渡し
   値そのものを渡す。渡す変数の内容をコピーして、コピーされた変数を使用する。
   変数の中身を書き換えても呼び出し元の変数には影響しない。

2. ポインタ渡し
   アドレスを渡す。変数の置いてある場所（アドレス）を渡して、自由に使ってもらう。
   変数の中身を書き換えると呼び出し元の変数の中身も書き変わる。

3. 参照渡し
   ポインタと同じ・・だが、より安全で制約の厳しいポインタである。

では、なぜポインタと参照の2種類に分かれているのか？
ポインタ渡しの問題点・・
- 宣言だけで領域確保がされていない場合がある。
- 関数呼び出し側と呼ばれる側で表記が異なる。（例：player.x、player->x） ※問題という訳ではないが・・

＜ポインタ渡しの場合＞
- ポインタ配列を渡せる。
- 配列の要素数のコンパイルチェックができない。
- ポインタの演算ができる。
- nullが渡せる。（とりあえず実体がなくても入れ物だけ渡せる）

＜参照渡しの場合＞
- ポインタ配列が渡せない。
- 配列の要素数のコンパイルチェックができる。（要素数を明示する必要がある）
- ポインタの演算ができない。
- nullが渡せない。（参照元が有効である事が前提）

参照の例

参照は、変数に別名をつけてどちらからでもアクセスできる様にする事ができます。
参照を使って別名の変数を宣言する場合は「＆」を付けて宣言します。

```
int num;                    // int 型の変数 num
int &newNum = num;          // 元となる変数と同じ型で newNum に参照する

newNum = 100;               // 参照の方に 100 を代入すると、元の num の値が 100 になる
```

同様の事をポインタで行う例

C 言語でもポインタを使えば参照と同様の事ができます。
ポインタの場合は別名の変数の値にアクセスする時に「*」を付ける必要があります。「間接演算子」

```
int num;            // int 型の変数 num
int* newNum;        // int 型へのポインタ変数 newNum

newNum = &num;      //  ポインタ型変数 newNum によって、変数 num を間接参照する。
*newNum = 100;      //  ※num の値が 100 になります。アクセス時には「*」が必要 ！
```

□関数の「ポインタ渡し」

ポインター渡しの例

```
// ---------- int 型変数の場合
int num;                    //  変数
int *numPtr;                //  ポインタ変数

void Pointer(int *a)
{
    *a = 100;               // num に 100 が代入される
}

void main1()
{
    Pointer(&num);          //  通常の変数の場合はアドレスを示す「＆」を付ける
    Pointer(numPtr);        //  ポインタ変数の場合はそのまま渡す
}

// ----------  構造体などの型の場合
Player player;              // Player 型(例：構造体)の変数
Player *playerPtr;          // Player 型(例：構造体)のポインタ変数

void Pointer(Player *p)
{
    p->メンバ変数など;      //  構造体などは「->」でアクセスする
}

void main2()
{
    Pointer(&player);       //  通常の変数の場合はアドレスを示す「＆」を付ける
    Pointer(playerPtr);     //  ポインタ変数の場合はそのまま渡す
}
```

□関数の「参照渡し」

参照渡しの例

```
// ---------- int 型変数の場合
int num;                    // 変数
int *numPtr;                // ポインタ変数

void Reference(int &a)
{
    a = 100;                // 通常の変数の様にそのままアクセス可能（num に 100 が代入される）
}

void main1()
{
    Reference(num);         // 通常変数の場合はそのまま渡す
    Reference(*numPtr);     // ポインタ変数の場合は「*」を付けて実体モードとして渡す
}

// ---------- 構造体などの型の場合
Player player;              // Player 型(例：構造体)の変数
Player *playerPtr;          // Player 型(例：構造体)のポインタ変数

void Reference(Player &p)
{
    p.メンバ変数など;       // 構造体などは通常の変数の様に「.」でアクセスできる
}

void main2()
{
    Reference(player);      // 通常変数の場合はそのまま渡す
    Reference(*playerPtr);  // ポインタ変数の場合は「*」を付けて実体モードとして渡す
}
```

□「ポインタ」「参照」まとめ

「変数」「ポインタ」を関数に渡す場合、3種類の渡し方によってそれぞれ、渡し方、受け取り方が違います。
状況別の違いを一覧表にまとめましたので、上手に活用していきましょう。

渡す方法	関数で受け取る方	渡す値	関数に渡す方
値渡し	関数名(型　変数名)	変数	関数名(変数名);
		ポインタ変数	関数名(*変数名);
ポインタ渡し	関数名(型 *変数名)	変数	関数名(&変数名);
		ポインタ変数	関数名(変数名);
参照渡し	関数名（型 &変数名）	変数	関数名(変数名);
		ポインタ変数	関数名(*変数名);

ポインタ渡しは、呼び出す際に引数もポインタである必要がありますが、参照渡しの場合は、変数の名前をそのまま記入すれば良いという点にあります。また、単に値を見るだけで値の変更がない場合は「const修飾子」を付ける事で、不用意な変更を出来ない様にしてより安全にする事もできます。

```
Player player;                          // Player 型の変数
Player* playerPtr;                      // Player 型のポインタ変数

// ----- ポインタ
void Pointer(Player*){}
void Func1()
{
    Pointer(&player);                   // 「ポインタ」で渡す
    Pointer(playerPtr);                 // 「ポインタ」で渡す
}

// ----- 参照
void Reference(const Player&){}
void Func2()
    Reference(player);                  // 「参照」で渡す
    Reference(*playerPtr);              // 「参照」で渡す
}
```

```
// ----- ポインタ渡し
void Pointer(Player* p)
{
    p->メンバー     // メンバーにアクセスする場合はアロー演算子(->)で行う
}

// ----- 参照渡し
void Reference(const Player& p)
{
    p.メンバー      // メンバーにアクセスする場合はドット(.)で行う
}
```

とりあえずC++での関数の引数は「constの参照渡し」をしましょう！

これにより、引数の状態が変更されない事を保証する事ができます。

■const修飾子

constは通常、定数を表す修飾子として使用されます。

1. 定数の定義の場合

```
const int max = 1000;     //  定数の定義(変更不可)

例)
  const int ENEMY_MAX = 100; //  定数 100 で定義
  ENEMY_MAX = 10;                    //  コンパイルエラー
```

2. 関数の引数の場合

```
void func(const Player* p);   //  ポインタの場合
void func(const Player& p);   //  参照の場合
```

引数として、ポインタや参照を受け取った場合に、pの値を変更しようとした場合にはエラーが発生します。

3. メンバ関数での使用例

```
class Sample
{
public:
      int posX = 10;
      int getPosX() const;     //  メンバ関数の const 化
};
```

```
int Sample::getPosX() const     // const 化されたメンバ関数
{
      posX = 0;          // const 指定した引数の値を変更しようとしたのでビルドエラーとなる
      return posX;     // posX の値を返す
};
```

関数内でのメンバ変数の変更不可の指定をしているので、関数内で値の変更が行えない。

□const修飾子の使われ方一覧

使用タイプ	使用例	意味
①変数の前	const int a = 100;	定数の定義 変更できない変数(定数)となる)
②メンバ関数の引数内	void func(const int a);	引数の変更不可 関数内では、引数の状態を変更できない。
③メンバ関数の後ろ	int getParam() const;	メンバ変数の変更不可 関数内では、メンバ変数の状態を変更できない。

■STL（Standard Template Library）とは？

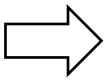
C++言語の標準ライブラリに含まれるテンプレートによるコンテナとアルゴリズムのライブラリです。

テンプレートとは？

データ型にとらわれずにコードを書く事を可能にした機能の事。
例えば、一つの関数で扱う形式がintでもfloatでも可という様な、どの様な型でも当てはめて対応できる様な関数を作る事ができる。※オーバーロードとはちょっと違います。
このような変幻自在な形式を「ジェネリック(汎用性)プログラミング」と言います。

```
template <typeneme T>

T max(T x, T y)
{
    if(x < y){
        return y;
    }else{
        return x;
};
```



```
float f = max( 5.5f, 3.2f );
※f に 5.5f が入る

int i = max( 5, 10 );
※i  に 10 が入る
```

どんな型が来ても対応 OK

コンテナとは？

動的にメモリを確保する要素の集合体（配列みたいなもの）を管理する時に、確保したメモリの解放忘れてメモリーリークを起こしたりする事を防ぐ為に、確保した領域を自動的に開放してくれるなどの機能を持たせた動的確保が可能な集合体（型みたいなもの）。
用途に合わせて色々なコンテナの種類があります。

コンテナの種類

・vector(ベクター)・・・・可変長配列
・list（リスト）　・・・・双方向連結リスト
・map(マップ)　・・・・連想配列

他にもあります。

イテレータとは？

配列にとってのポインタの様に、コンテナの各要素にアクセスする為のクラスになります。それぞれのコンテナのどの位置かを保存して、要素毎に前後に進めたり、先頭や終端をチェックしたりの機能があります。各要素に関しては”*”や”->”で参照する事ができます。

尚、コンテナのイテレータを取得する方法は、コンテナ毎にメンバ関数「begin()」「end()」などの専用の関数が用意されているので、用途に合わせて使用していきます。

■STL（vector） ※可変長配列

通常の配列は、int* a = new int[要素数]; という宣言で要素数を確定しないといけませんが、vectorは、あらかじめサイズを指定しなくても利用できる様にした配列を拡張したクラスです。その為、添え字を使って要素にアクセスしたり、最後の要素に新しい要素を追加したりできる様になっています。

基本的には「任意の場所に要素を追加する」「任意の場所を削除する」事は目的としていません。

①使う前に・・

```
#include <vector>
```

vectorを利用する為には、vectorクラスをインクルードします。

②動的配列の宣言の仕方

```
std::vector<型名> 変数名;

例） 配列の型だけ確保する。
 vector<int> intVect;            // int 型の配列
 vector<ACTOR_BASE*> enemyVect;   // ACTOR_BASE*型の配列
```

```
＜番外編＞配列の場合は少しトリッキーです

  std::vector<std::vector<型名>> 変数名(要素 1, vector<型名>(要素 2) );
  std::vector<std::vector<型名>> 変数名(要素 1, vector<型名>(要素 2, 値);
  std::vector<std::vector<型名>> 変数名{ {要素 1, 要素 2}, {要素 3, 要素 4} };

  例） 二次元配列の例
    std::vector<std::vector<int> > array( 10, std::vector<int>(5) );      // array[10][5]
```

③要素の追加の仕方

```
変数名. push_bask(要素);

例） それぞれの型に合わせて要素を追加する
 intVect. push_back(5);          // int 型の配列に 5 を追加
 intVect. push_back(10);         // int 型の配列に 10 を追加

 ACTOR_BASE* e1, e2;
 enemyVect. push_back(e1);     // ACTOR_BASE*型の配列に e1 を追加
 enemyVect. push_back(e2);     // ACTOR_BASE*型の配列に e2 を追加
```

④-1添え字によるアクセスの仕方

```
for(int i = 0; i < intVect.size(); i++)
{
      int tmp = intVect[i];   // 値を別の変数に代入したり・・
}
```

```
for(int i = 0;  i < enemyVect. size();  i++)
{
      enemyVect[i]->Draw();    // メンバ関数を実行したり・・
}
```

④-2　イテレーターによるアクセスの仕方　※イテレーターとはコンテナ内の1要素を示すオブジェクトの事です。

・begin()関数・・・・一番初めの要素

・end()関数・・・・・一番後ろの要素の次へイテレータを示す。

```
std: :vector<CHARA_BASE*>: :iterator itr;
for(itr = enemyVect. begin();  itr != enemyVect. end();  ++itr)
{
      (*itr)->Draw();     // (*itr)とは・・*を付けて itr の内容を実体化しています【間接演算子】
}
```

⑤全要素を削除する場合

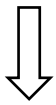
・clear()関数・・・・全要素を削除する

```
intVect. clear();
```

⑥省略記述

・auto型・・・・・・「vector<型名>::iterator」と書く所を「auto」で省略できる。

```
// itr の定義を省略しない場合
for(vector<int>: :iterator itr = intVect. begin();  itr != intVect. end();  ++itr)
{
}
```



```
// itr の定義を auto で省略した場合
for(auto itr = intVect. begin();  itr != intVect. end();  ++itr)
{
}
```

vectorの主なメンバ関数

関数名	機能
push_back()	要素の追加
clear()	要素のクラス
size()	配列の大きさを得る
capacity()	動的配列に追加できる要素の許容量
empty()	要素が空がどうか調べる

■STL（list） ※双方向連結リスト

vectorと似ていますが、vectorとの違いはその名の通り、「任意の場所に要素を追加（挿入）・削除する」という使い方を想定しています。その為、インデックスでは管理できずにイテレータで管理する事になります。

①使う前に・・

```
#include <list>
```

listを利用する為には、listクラスをインクルードします。

②宣言の仕方

```
std::list<型名> 変数名;

例） 配列の型だけ確保する。
 std::list<int> intList;                    // int 型の配列
 std::list<ACTOR_BASE*> enemyList;    // ACTOR_BASE*型の配列
```

③要素の追加の仕方

```
変数名.push_bask(要素);   // 終端に要素を追加

例）それぞれの型に合わせて要素を追加する
 intList.push_back(5);          // int 型の配列に 5 を追加
 intList.push_back(10);         // int 型の配列に 10 を追加

 CHARA_BASE* e1, e2;
 enemyList.push_back(e1);     // ACTOR_BASE*型の配列に e1 を追加
 enemyList.push_back(e2);     // ACTOR_BASE*型の配列に e2 を追加
```

④要素の削除の仕方

```
変数名.pop_front();   // 先頭を削除
変数名.pop_back();    // 終端を削除
変数名.erase(itr);    // イテレータのある場所を削除

例）要素を削除する
intList.pop_front();        // 先頭を削除
intList.pop_back();         // 終端を削除

for(auto itr = intList.begin(); itr != intList.end(); ++itr)
{
      itr = intList.erase(itr);     // 要素を削除し itr は次を示す
}
```

⑤全要素を削除する場合

・clear()関数・・・・全要素を削除する

```
intList. clear();
```

⑥イテレーターによるアクセスの仕方

・begin()関数・・・・一番初めの要素

・end()関数・・・・・一番後ろの要素の次へイテレータを示す。

```
std::list<int>::iterator itr;
for(itr = intList. begin();  itr != intList. end();  itr++)
{
	int tmp = *itr;	// int 値は*itr そのもので取得できる
}

std::list<ACTOR_BASE*>::iterator itr;
for(itr = enemyList. begin();  itr != enemyList. end();  itr++)
{
	(*itr)->Draw();	// メンバー関数は(*itr)->でアクセスする
}
```

⑦要素を消す場合（要注意！）

リストの途中の要素を削除する場合は、削除したあとのイテレータの扱いに注意が必要です。

ポイントは、削除した瞬間にitrは次の項目に移動している為、そのままfor文などでループさせるとリストの項目を超えてアクセスしてしまう事です。ですので削除した場合はイテレータを進ませず、削除しなかった場合はイテレータを進めるといった2種類の処理が必要になります。

■for 文の処理の場合

```
std::list<Shot*>::iterator itr;
for (itr = mShotList.begin();  itr != mShotList.end();  /*「it++」を削除*/  )
{
	if ((*itr)->RemoveIf() == 1)  // 消して良し？
	{
		delete(*itr);
		itr = mShotList.erase(itr);		// 消すと自動的次のイテレータに移動
		continue;
	}
	itr++;  // 消さなかった時に次のイテレートに移動
}
```

■while 文の処理の場合

```
std::list<Shot*>::iterator itr = shotList.begin();
while (itr != shotList.end())
{
	// 消す必要があればリストとshotの実体を削除
	if ((*itr)->RemoveIf() == 1)
	{
		delete (*itr);
		itr = shotList.erase(itr);
	}
	else
	{
		itr++;  // itrを進める
	}
}
```

listの主なメンバ関数

関数名	機能
push_front()	先頭に要素を追加する
push_back()	末尾に要素に追加する
pop_front()	先頭の要素を削除する
pop_back()	末尾の要素を削除する
insert()	要素を挿入する
erase()	要素を削除する
clear()	全要素を削除する

■STL（map） ※連想配列

vector同様配列の一種で、vectorは要素のアクセスを0,1,2といったインデックスで行うのに対し、mapはキーといういわゆる文字列で配列にアクセスするものです。番号でなく名前でアクセスができるので「何番目に入っている」などを管理する必要がなく動的配列としてどんどん追加していく事ができるイメージです。

①使う前に・・

```
#include <map>
```

mapを利用する為には、mapクラスをインクルードします。

②map配列の定義

```
std::map<キーの型名, 値> 変数名;
```

③-1map配列への追加

[ ]演算子でアクセスした時に、存在しない要素だったら自動的に追加される。

```
例） string で指定した文字列をインデックスとして map 変数に値を入れる。
 std::map <std::string, int> hpMap;

 hpMap["player"] = 10;   // インデックス名 player、値 10 を追加
 hpMap["enemy"] = 20;    // インデックス名 enemy、値 20 を追加
 hpMap["shot"] = 30;     // インデックス名 shot、値 30 を追加
```

③-2キーでmapを検索する方法

指定したキーの要素が存在するかをチェックしながら追加する。

```
例） 指定した要素がなかった場合？

if( hpMap.find("boss") == hpMap.end())
{
    // end()まで来ているのでなかった事になる
    hpMap["boss"] = 50;   // インデックス名 boss、値 50 を追加
}
```

④map配列の使用例

```
例） map 配列より string の文字列をキーにした値を利用する。

 playerHp = hpMap["player"];
 enemyHp = hpMap["enemy"];
 shotHp = hpMap["shot"];
 bossHp = hpMap["boss"];
```

mapの主なメンバ関数

関数名	機能
begin()	先頭を示すイテレータを取得する
end()	末尾を示すイテレータを取得する
clear()	全ての要素をクリアする
empty()	map配列が空ならtrue、そうでない時はfalseを返す
erase()	指定した要素をクリアする
size()	マップの中の要素数を返す
fined()	指定したキーと一致する要素をイテレータで返す

④map配列の活用した画像管理関数の例

文字列のID名と、文字列のファイル名を引数にして画像の登録を試みる。

→画像が新規だったら

LoadGraphでファイル名で読み込んだID番号を取得

map配列に「LoadGraphで返すID番号」と「文字列のID名」をセットで登録する

登録したLoadGraphで返すID番号を関数から返す

→画像が登録済みだったら

登録したLoadGraphで返すID番号を関数から返す

```
// ①キーとなる文字列とファイル名で保存する imageMap という Map 配列を作成
std::map<std::string, int> imageMap;

int LoadImg(std::string id_name, std::string f_name)
{
    // ②画像が登録済みか？
    if(imageMap.find(id_name) == imageMap.end())
    {
        // ③新規の文字列だったら、要素を imageMap に追加保存
        //    LoadGraph で返す ID を imageMap に追加
        //    c_str()は、C 言語の文字列に変換(DxLib 対策)
        imageMap[id_name] = LoadGraph(f_name.c_str());
    }
    return imageMap[id_name];   // ④LoadGraph で返す ID を返す
}
```

■std::string

C++で標準的に利用できる文字列のクラスです。これまでC言語でのsprintf()などの文字列を作成する関数などはあったのですが、動的な操作などに不向きだった為、C++では動的にサイズ変更が可能な文字列クラスとしてstd::stringが導入されました。

①使う前に・・

```
#include <string>
```

stringを利用する為には、stringクラスをインクルードします。

②stringの定義

```
std::string 変数名;      // string 型の変数として定義する
```

③文字列の初期化

```
std::string str = "abcdef";     //  "abcdef"で初期化

std::string str("abcdef");       // "abcdef"で初期化

char[] data = "abcdef";          // char*で設定
※もしくは const char* data = "abcdef";
std::string str(data);           // char*をコンストラクタ時に代入。"abcdef"で設定
```

④文字列の足し算

```
std::string str = "abcdef";     // "abcdef"で初期化
str +="ghijk";                  // str の中身は"abcdef"+"ghijk"となる。

※str の中身は"abcdefghijk"となる
```

⑤文字列の比較

```
std::string str = "abcdef";     //  "abcdef"で初期化
if(str == "abc")
{
    // "abcdef"と"abc"の比較
}

std::string str1 = "abcdef";     //  "abcdef"で初期化
std::string str2 = "abc";     //  "abcdef"で初期化
if(str1 == str2)
{
     // "abcdef"と"abc"を値同士の様に比較できる
}
```

□std::stringの文字列とC言語との連携は？

折角の文字列ですので、DxLibなどにも積極的に利用したいですね。では実際はどうでしょう？

①文字列をDxLibで使用してみる

```
std::string str = “image/player.ng”;

int image = LoadGraph(str);     // ダメです！
```

残念ながら、そのまま使用できません。C言語の関数には「C言語としての文字列表現」が必要です。

②C言語としての文字列表現の仕方

```
std::string str = “image/player.png”;

int image = LoadGraph(str.c_str());     // .c_str()を使用する事で可能となります
```

実に簡単です、この「c_str()関数」を使用するだけでC言語としての表現（char*）に変換してくれます。

③他にも様々な関数があります。

stringの主なメンバ関数

関数名	機能
c_str()	C言語の文字表現を返す（charへのポインタ）
find()	文字列の中の指定文字を検索
empty()	文字列が空かどうかを判定する(bool)
size()	文字数を返す（==0は空と同等）
length()	size()と同様
clear()	文字列を空にする
resize()	文字列を指定サイズに設定する

※まだまだありますので、興味のある方はリファレンスを見てみましょう。

■まとめ

このドキュメントで出て来た用語の一覧です。各キーワードの理解はいかがでしょうか。

・値渡し、ポインタ渡し、参照渡し
・const 修飾子
・STL とは
・STL（vector） 可変長配列
・STL（list） 双方向連結リスト
・STL(map) 連想配列