# C++基礎　1

2006/05/08

1. C++とは 3
C++とは 3
C 言語とはどう違う？ 3
2. C 言語との違い 4
文法 4
① 新しい型『bool』 4
② 関数に、デフォルト引数を与えることが可能 5
③ new と delete 6
④ 参照型 7
⑤ 変数の宣言場所の自由 12
⑥ 例外処理 13
⑦ 関数オーバーロード 14
⑧ テンプレート関数 15
⑨ 名前空間 17

# 1. C++とは

## C++とは

C++とは、誤解を恐れずに言うと、C 言語をもっと使いやすく機能拡張した言語です。
具体的に言うと、『C 言語をオブジェクト指向プログラミングが出来るよう機能拡張した言語』です。

## C 言語とはどう違う？

C に比べると、出来ることが増えています。何がどのように増えているのかは、これから勉強していきます。
あくまでも「C 言語の機能拡張」なので、今までの C 言語のみのソースを
C++コンパイラでコンパイルすることは(基本的には)可能です。
ちなみに VisualStudio の VisualC++は、名前の通り C++コンパイラです。

# 2. C 言語との違い

## 文法

### ① 新しい型『bool』

真・偽を格納する変数型、bool が新しく導入されました。

```
int main()
{
    bool b1 = true;
    bool b2 = false;
    bool b3 = (1 == 2);

    if(b1){
        printf( "b1 = true ¥n" );
    }
    return 0;
}
```

bool には、true と false のいずれかを格納できます。

主に、関数の戻り値、if 文の判定要素等に使用されます。

```
bool IsPositiveInteger(int num)
{
    bool bPositiveInteger = (num > 0);  // num > 0 が正しいときには true,
                                        // 正しくないときには false が入る。
    return bPositiveInteger;
}
```

### ② 関数に、デフォルト引数を与えることが可能

関数の引数について、『**デフォルト引数**』を与えることが出来るようになりました。

```
int fncKato(int a, int b = 100)
{
    return a - b;
}

int main()
{
    int ret, ret2;
    ret = fncKato(10, 7);    // ret には 7 が入る
    ret2 = fncKato(150);     // ret には 50 が入る
    return 0;
}
```

プロトタイプ宣言の場合は、**プロトタイプ宣言部のみに表記し、実装部には表記しません。**
しかしプロトタイプ部にデフォルト引数を書いたことを忘れないために、コメントで表記しておいた方がいいでしょう。

```
int fncKato(int a, int b = 100);

int main()
{
    int ret = fncKato(10, 7);    // ret には 7 が入る
    int ret2 = fncKato(150);     // ret には 50 が入る
    return 0;
}

int fncKato(int a, int b /* = 100*/ )    // ←忘れないようにコメントを入れて
                                         // おく
{
    return a - b;
}
```

### ③ new と delete

動的に配列を作成する等、動的なメモリ確保をしたい場合、C 言語では malloc 関数を使用していました。C++では malloc 関数よりも簡単な new 演算子が追加されました。new で確保したメモリ領域は、malloc 時の free 関数と同様に、必ず delete で開放する必要があります。

<table>
<tr><th>型</th><th>C 言語</th><th>C++言語</th></tr>
<tr><td>int 型</td><td>int *p = (int*)malloc( sizeof(int) );<br>free(p);</td><td>int *p = new int;<br>delete p;</td></tr>
<tr><td>int の配列</td><td>int *p = (int*)malloc( sizeof(int) * 10 );<br>free(p);</td><td>int *p = new int[10];<br>delete []p;</td></tr>
<tr><td>構造体配列</td><td>struct StTest{<br>   int  num;<br>   char ch;<br>};<br><br>struct StTest *p =<br>(struct StTest*)malloc(sizeof(struct StTest));<br>free(p);</td><td>struct StTest{<br>   int  num;<br>   char ch;<br>};<br><br>StTest *p = new StTest;<br>delete p;</td></tr>
</table>

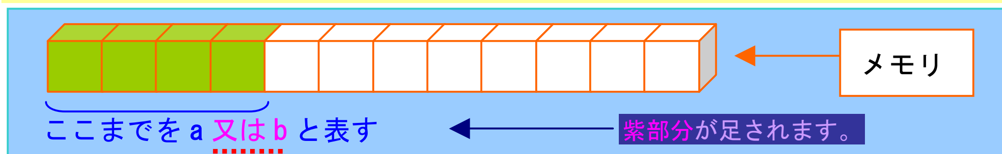
### ④　参照型

変数、定数の別名を意味する『**参照型**』が追加されました。別名とは、**名前だけ違う、同じ変数**、といえるでしょうか。

参照型の宣言の仕方は、以下のとおりです。

・「&」をつけ、必ず同じ型の変数で初期化する。

```
int main()
{
    int a = 0;
    int & b = a;      //「b」が参照型である。
    printf("%d", a);  // 結果は「0」が表示される。
    b = 10;
    printf("%d", a);  // 結果は「10」が表示される。
    return 0;
}
```




上記の例を見ただけでは、一見(ていうか全然)メリットがわかりません。

この**参照型は、関数の引数や戻り値にも使用できます**。参照型を使う上での一番のメリットは、引数に使うときでしょう。

次の例を考えてみます。2 個の引数の内容を交換する、swap 関数です。

```
void swap(int & Arg1, int & Arg2)
{
    int tmp = Arg1;
    Arg1 = Arg2;
    Arg2 = tmp;
}

int main()
{
    int a = 10;
    int b = 999;
    swap (a, b);
    printf("%d", a); // 結果は「999」が表示される。
    printf("%d", b); // 結果は「10」が表示される。
    return 0;
}
```

参照渡しをした仮引数 Arg1 は、a の別名⇒つまり同じメモリ領域を有しているので、Arg1 に対し演算を行うと、当然 a の値も変化します。

要は、**ポインタを使用しなくても、関数内で実引数の値を変えられる**、ということです。

☆　これ以上に素敵なメリットは、C 言語関数の値渡しと違い、メモリ領域を新しく使用しない⇒つまり**無用なコピーを行わない**、ということです。下の例を見てみましょう。

C 言語で作れる普通の関数：

```
int fncA(int a)
{
    return a * 2;
}

int main()
{
    int num = 99;
    int ret = fncA(num);
    return 0;
}
```

上記の main が実行されるとき、実は下の赤い一行のような動きがこっそり行われています。

```
int main()
{
    int num = 99;
    int ret = fncA(num)
    {
        int a = num;
        return a * 2;
    }
    return 0;
}
```

実行時のイメージ

仮引数を作成し、そこに実引数の値を代入している！！
実際に計算に使われるのは、num ではなく、a である！！

これを見ると、別に a を使わず、関数の中で直接 num を 2 倍すればいいじゃん、と思いませんか？

しかし、それは **C 言語の関数では出来ない**のです。

C 言語では、**関数に渡せるのはあくまでも値のみ**で、変数自体を渡す方法は**ありません**。

ん？ポインタは？アレは渡す側の変数値も変更できたじゃん！！と思うかもしれませんが、ポインタも、ポインタの持つ「アドレスという値」を渡しているに過ぎません。関数内部では、仮引数のポインタ変数を宣言し、そこに実引数として渡されたポインタの値（つまりアドレス）をコピーして、関数内で使用しているのです。

```
void fncP(int* a)
{
    *a = *a + 99;
}

int main()
{
    int num = 99;
    fncP(&num);
    return 0;
}
```

```
int main()
{
    int num = 99;
    fncP(&num)
    {
        int* a = &num;
        *a = *a + 99;
    }
    return 0;
}
```

仮引数を作成し、そこに実引数の値（アドレス）を代入している！！

上の例では、「受け取った num のアドレスを利用して、num の値を変更している」のであり、「受け取った num のアドレスを変更しているわけではない」のです。理解できたでしょうか？

このように、C 言語での関数処理のイメージは、

① 必ず実行時に仮引数が宣言され、
② そこに受け取った値を代入、
③ それを使用する。

という流れになっています。

さて、これらの関数について参照を使用すると、次のようになります。

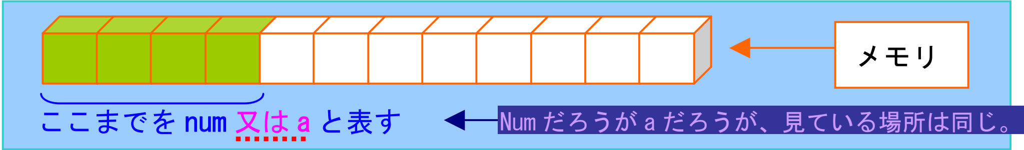
```
int fncA(int& a)
{
    return a * 2;
}

int main()
{
    int num = 99;
    int ret = fncA(num);  // 使う側は、参照でも参照じゃなくても
    return 0;             // 書き方は変わらない。
}
```

```
int main()
{
    int num = 99;
    int ret = fncA(num)
    {
        int& a = num;
        return a * 2;
    }
    return 0;
}
```

実行時のイメージ

実引数をそのまま計算に使用しているのがわかりますか？わからない場合、「a」をゴルビー、「num」を「ゴルバチョフ」と変えて読んでみましょう。「&」って、ほぼそういう意味です。
「ゴルビーっていうのは、ゴルバチョフのことだヨ！！だから、ゴルビーが書記長になった、ということは、ゴルバチョフが書記長になった、というのと同じだヨ！！」




このように、新しいメモリ空間を取ることなく、そのまま変数自体を渡すことが出来ます。
この流れは、次のようなイメージになるでしょうか。

① 実行時に参照として実引数を受け取り、
② 実引数に仮引数という別名（あだ名）をつけ、
③ 別名オブジェクトを操る（あだ名で呼んでつれまわす）

さて、ここまでの参照型のまとめは次のようにいえます。

**※　参照型のメリット**

**①　関数呼び出しで無用なコピーを防ぐ**

**②　ポインタを使用せずとも、関数で実引数の値を変えられる**

だからなんだ！！**別に int 一つくらい、多めに作ってもいいじゃねーか！！**と思うかもしれません。確かに**そのとおり。**「int 一つ」であれば、別に多めに作っても、いまどきのコンピュータで気にする必要はありません。上記関数では、参照型にする必要はないでしょう。

では、int のメンバを 500 個持った構造体についてはどうでしょう？また、1 万回回るループのなかで、関数が呼ばれるごとに新しくコピー⇒廃棄していたらどうなるでしょう？

ほかにも、この後に出てくるクラスというオブジェクトは、一回作る度に必ず勝手に決まった関数を呼んでしまうという代物です。そのようなものを、いちいち関数でちょこっと使う為だけにコピーしていたら、処理時間とメモリの無駄遣いとなってしまいます。

そこで使用するのがこの参照型なのです。

参照型については、クラスの部分でもう一度説明します。

### ⑤ 変数の宣言場所の自由

C 言語では、関数内で使用する変数は、必ず関数の初めに宣言する、という決まりがありました。C++の場合、**関数内のどこでも宣言ができます。**スコープ（使用できる範囲）は、宣言したブロック内（中カッコで挟まれた範囲）のみです。

```
// C 言語の場合
int main()
{
    int i;
    int a, b;
    a = 10;
    for(i = 0; i < a; i++){
        b = a - i;
        printf(“&d¥n”, b);
    }
    return 0;
}
```

```
// C++の場合
int main()
{
    int a = 10;
    for(int i = 0; i < a; i++){  // C 言語だとエラー！！C++だと OK！！
        int b = a - i;           // C 言語だとエラー！！C++だと OK！！
        printf(“&d¥n”, b);
    }// b のスコープはここまで
    return 0;
}
```

## ⑥　例外処理

C++で、新しく『例外処理』という書き方が加わりました。例外処理とは、関数内でなにかが起こったときに、『ヘンなことが起こったぞ！！』と呼び出し元に知らせつつ、決まった処理を行えるという機構です。

書式は次のとおりです。

①　例外を発生させたい場所に throw をおく

②　例外が起こるかもしれない場所を try ブロックで囲む

③　例外時に行う処理を catch ブロックで囲む

```
// 例外を投げる関数
void test(int n){
    if( n == 0 )
        throw n;    // 例外を投げる部分
    double d = n / 2.0;
    if(d < 1.0)
        throw d;    // 例外を投げる部分
}
void main(){
    try{
        // 例外が起こるかもしれないブロック
        int n;
        cin >> n;
        test(n);
    }
    catch( int arg ){
        // 例外 1 が起きた場合に処理したいブロック
    }
    catch( double arg ){
        // 例外 2 が起きた場合に処理したいブロック
    }
    catch(...){
        // その他の例外が起きた場合に処理したいブロック
    }
}
```

## ⑦　関数オーバーロード

C 言語では、同じ名前の関数を複数作成することは出来ませんでした。
例えば、足し算の関数

```
int Add(int A, int B);
```

という関数を作成した場合、実数版を作成するには

```
double Add_Double(double A, double B);
```

などというように、名前を変えなければなりませんでした。
これに対して、C++では
**『引数の種類もしくは数が違えば、同じ名前の関数を宣言できる』**
ようになりました。つまり、一番上の int 型の関数が既にあったとしても

```
double Add(double A, double B);
```

という関数も作れるようになります。

◆　しかし、必ず『引数の種類もしくは数が違う』必要があり、**戻り値のみが違う、といったものについてはオーバーロードできません**。
なぜなら・・

```
int     FncKato(int a, int b); // 戻り値のみ違う関数
double  FncKato(int a, int b); //

int main()
{
    int a = 0;
    int b = 2;
    FncKato(a, b);      // どちらが呼ばれているか、わからない！！
    return 0;
}
```

このように書くと、上記で宣言してある２個の関数のどちらを使用したいのか、コンパイラにはわからないためです。

### ⑧　テンプレート関数

前述のオーバーロードで、同じ名前で変数の型が違う関数、というものを作成できるようになりました。

```
int     GetMax(int a, int b)          { return (a > b) ? a : b; }
double  GetMax(double a, double b)    { return (a > b) ? a : b; }
```

しかし、上記のように、変数の型のみ違い、内容はまるで一緒の関数があったとき、これを何とかして一つにまとめられないだろうか、と考えることがあります。
そんな問題を解決してくれるのが、テンプレート関数です。

```
template<class Type>
Type    GetMax (Type a, Type b) { return (a > b) ? a : b; }
```

このように定義すると、関数 GetMax は、一つの定義で整数にも実数にも使用できるようになります。
では、定義の仕方を見ていきましょう。

テンプレート関数の定義方法は、次のとおりです。

1） 関数の最初に、「template」キーワードをつける。

```
template<class Type>
Type    GetMax (Type a, Type b) { return (a > b)  ? a : b; }
```

2） 仮のデータ型名を<class ...>の...部分に自由に表記（この例だと Type の部分）。

```
template<class Type>
Type    GetMax (Type a, Type b) { return (a > b)  ? a : b; }
```

3） 後はほぼ普通の関数と一緒。型が変わる可能性のある変数を、仮データ型（この例だと Type 型）として宣言、使用する。

```
template<class Type>
Type    GetMax (Type a, Type b) { return (a > b)  ? a : b; }
```

4） こうして定義された関数は、コンパイル時に引数の型によって実体化されます。

```
template<class Type> Type  GetMax (Type a, Type b)
{
    return (a > b)  ? a : b;
}

int main()
{
    int num1 = 100;
    int num2 = 50;
    int nRet = GetMax(num1, num2);

    double d1 = 99.99;
    double d2 = 155.55;
    double dRet = GetMax(d1, d2);
    // GetMax(num1, d1); ← これはダメ！！型が定まらない！！
    // コンパイル時にエラーになります。
    return 0;
}
```

```
int  GetMax (int a, int b)
{
    return (a > b)  ? a : b;
}
```

引数によって、Type が置き換わる

```
double GetMax (double a, double b)
{
    return (a > b)  ? a : b;
}
```

### ⑨　名前空間

オーバーロードという便利な機能が増えた一方、同じ名前の関数が氾濫する、という危険性も増しました。たとえば、下のような宣言を考えてみます。

```
int FncKato (int a, int b = 0)  { return a - b; }
int FncKato(int a)              { return a - 10; }
```

一見、引数の数が違うのでオーバーロード可能に思えます。実際、宣言のみ行い、使用しないでコンパイルする分にはエラーは発生しません。しかし、以下のように実際に使用したとき、思わぬコンパイルエラーが起こります。

```
int FncKato(int a, int b = 0)   { return a - b; }
int FncKato(int a)              { return a - 10; }

int main()
{
    int a = 0;
    FncKato(a);    // どちらが呼ばれているのか、
                   // コンパイラにはわからない！！
    FncKato(10,5);// これなら大丈夫。
    return 0;
}
```

引数 b は省略可能な為、どちらの関数を呼ばれたか、コンパイラには判断できないためです(**名前の衝突**)。このままでは引数一つで関数を呼ぶことが出来ません。
そこで、一方の関数を『**名前空間**』に属させます。

```
namespace Kanawgawa // 名前空間【Kanagawa】
{
    // Kanagawa の加藤さん
    int FncKato(int a, int b = 0)   { return a - b; }
}
// グローバルの加藤さん
int FncKato(int a)                  { return a - 10; }
```

このように宣言すると、引数 2 個の関数は【Kanagawa】という名前空間に属したことになります。**名前空間に属した関数を呼ぶには、『Kanagawa の Kato』と明示的に書かないと使えなくなります。**書き方は、『名前空間名：：関数』と書きます。
（例. Kanagawa::FncKato(10); ）

初めにエラーとなった main 関数にこの宣言を当てはめると、『引数１個の方が呼ばれる』ことになり、コンパイルエラーも起こらなくなります。

```
namespace Kanagawa  // 名前空間【Kanagawa】
{
    // Kanagawa の加藤さん
    int FncKato(int a, int b = 0)   { return a - b; }
}
// グローバルの加藤さん
int FncKato(int a)

int main()
{
    int a = 0;
    FncKato(a);              // グローバルの加藤さんが呼ばれた。
    Kanagawa:: FncKato(a);   // Kanagawa の加藤さんが呼ばれた。
//  FncKato(10,5);  ⇒        このままではコンパイルエラーとなるので・・
    Kanagawa::FncKato(10,5); // このように名前空間を指定
    return 0;
}
```

ちなみに、関数だけでなく、変数についても、名前空間に属させることが出来ます。
この場合も、関数と使用方法は同じです。

```
namespace Kanagawa  // 名前空間【Kanagawa】
{
    // Kanagawa のホゲ
    int hoge;
}
// グローバルのホゲ
int hoge

int main(){
        hoge = 10;       // グローバルのホゲに 10 を代入
        Kanagawa::hoge = 20;     // Kanagawa のホゲに 20 を代入
        return 0;
}
```

ところで、何回も名前空間【Kanagawa】の関数を呼ぶとき、いちいち Kanagawa::をつけるのが面倒なときがあります。

```
namespace Kanagawa  // 名前空間【Kanagawa】
{
    // Kanagawa の加藤さん
    int FncKato(int a, int b = 0)   { return a - b; }
}
// グローバルの加藤さん
int FncKato(int a)                  { return a - 10; }

int main(){
        Kanagawa::FncKato(10,10);
        Kanagawa::FncKato(6);
        return 0;
}
```

そんな場合は、**名前空間使用の宣言**を行うと、いちいち名前空間をつける必要がなくなります。宣言は、『using namespace 名前空間名』と書きます。
すると、名前空間の関数を名前空間指定なしで呼べるようになります。しかし・・

```
namespace Kanagawa  // 名前空間【Kanagawa】
{
    // Kanagawa の加藤さん
    int FncKato(int a, int b = 0)   { return a - b; }
}
// グローバルの加藤さん
int FncKato(int a)                  { return a - 10; }

using namespace Kanagawa;
int main(){
        FncKato(10,10);     // OK！！
        FncKato(6);         // OK・・・？・・・×！！
        return 0;
}
```

引数を省略すると、やはりどちらの関数を呼んでいるのかわからなくなるため、引数一つの関数を呼ぶには、明示的に指定する必要がでてきます。

ちなみに、**グローバルスペースの関数を明示的に呼ぶ場合は、『：：関数名』**という書き方をします。

```
namespace Kanagawa  // 名前空間【Kanagawa】
{
    // Kanagawa の加藤さん
    int FncKato(int a, int b = 0)    { return a - b; }
}
// グローバルの加藤さん
int FncKato(int a)               { return a - 10; }

using namespace Kanagawa;
int main(){
        Kanagawa::FncKato(6);    // ←Kanagawa の関数が呼ばれる
        ::FncKato(20);        // ←グローバルな関数が呼ばれる
        FncKato(10,10);       // ←Kanagawa の関数が呼ばれる
        return 0;
}
```

ここで出てきた：：(コロン 2 個)を、「スコープ演算子」といいます。