華麗なる C++テクニック!

～マクロ・テンプレート編～

あいさつ

こんにちは。kinokkory こと松下です。今回、C++の素晴らしさをみなさんにお教えしたくて、この部誌を執筆することとしました。
ある程度プログラミングをしたこともない人も意識していますが、基本的には C++中級者～上級者向けに書きました。ま、分からないことがあったら是非インターネットなんかで調べてみてください。もちろん僕がいれば直接聞いてもいいですよ。

序章　C++とは

C++（シープラスプラス）はプログラミング言語のひとつです。C 言語の進化版です。僕にとっては「自由すぎて速すぎるプログラミング言語」です。「++」はインクリメント演算子というもので 1 増やすという意味です。ちなみに C 言語は B 言語を改良して作られたのが名前の由来だそうです。[1]

第 1 章　とりあえず、入門から

まずは開発環境が要ります。無料のものとしては、純粋なコンパイラの GNU C++ Compiler (g++)と Borland C++ Compiler (BCC)や、高機能な統合開発環境 (IDE)[2]である Visual C++ Express[3] (VC++ Express)があります。ど

[1] さらに補足すると B 言語は A 言語の改良版というわけではなく、BCPL という言語がもとになっている可能性が高いそうです。
[2]コンパイラ、エディタ、デバッガなどをひとつの対話型操作環境（多くは GUI）から利用できるようにしたもの。IDE は Integrated Development Environment の略。

れでも構いません。
ただ僕は VC++を使っているのでこれから VC++固有の便利な機能も紹介していきます。もし迷われているのでしたら、**VC++のインストールをおすすめします**。
（なお、これから書いていくコードは VC++でしか検証していません。ご了承ください。）
Visual C++ Express のインストールはここから。
Microsoft Visual Studio Express
http://www.microsoft.com/japan/msdn/vstudio/express/
入門はここなんかがいいと思います。
Visual C++ 2010 Express プログラミング入門
http://cvwww.ee.ous.ac.jp/vc10prog.html
C++入門 http://wisdom.sakura.ne.jp/programming/cpp/
Programing Place http://www.geocities.jp/ky_webid/index_old.html

＊DreamSpark
ちなみに、僕の使っている IDE（統合開発環境）は Visual Studio 2010 Professional です。これは Visual C++ Express の上位版で、普通に買ったら 100,000 円ほどするものです（桁を見間違えないでね）。もちろん、こんなノートパソコン 1 台分ほどの大金は払えません。[4]でも僕はこれを**無料**で入手しました。
DreamSpark というサイトを利用したのです。これは「高等教育機関での授業や研究に対して、低経費にて、学生の教育を支援する」目的で作られたもので、とりあえず灘校生はみんな登録されているみたいです。DreamSpark に登録すると、Microsoft のいろんなツールが非商業・非営利に限って無料でダウンロードできるそうです。ぜひ学校の先生にも聞いてみてください。

---

[3]販売されている Visual Studio パッケージを C++に関する機能だけに制限した無料版。コンパイラ本体は cl.exe。
[4] ちなみにレベルがまったく違いますが、さらに高級な Visual Studio Ultimate OPEN Business は 128 万円します(笑)

第 2 章　ちょっと遠回り

本題に移る前にちょっと遠回りして C++の特徴について書きます。これを読めばあなたも C++でプログラミングしたいと思うはずですよ。
C++の主な特徴としては以下のようなことが挙げられます。

1) 実行速度が速い

中間言語を経由しないため、また言語の構造上高速化が容易なため、実行速度がとにかく速いです。Java[5]や C#[6]やインタプリタ言語[7]よりもちろん速いです。アセンブラ言語[8]や C 言語に次ぐ速さだと思ってもらって構いません。やはりこれほどの高機能な言語にもかかわらずこの速さというのはほかの言語に代えがたいです。

2) 柔軟な文法

C 言語の文法を受け継いでいるので、goto や三項演算子やコンマ演算子なんかも使えます。

3) マルチパラダイムプログラミング言語である

複数のプログラミングパラダイム（プログラミングスタイル）に対応しているということです。具体的に見たほうがわかりやすいと思うので、それぞれのパラタイムについて説明します。

a. 手続き型プログラミング

端的に言うとグローバル関数を使えるということです。例えば Java は完全なオブジェクト指向なのでグローバル関数は使えません。

b. オブジェクト指向プログラミング

---

[5] Java: オブジェクト指向プログラミング言語のひとつ。Java で開発されたソフトウェアは Java 仮想マシンのもとで動作するため、プラットフォームに依存しないアプリケーションソフトウェアの開発と配備を行うことができる。構文は C および C++から多くを引き継いでいるが、より単純になっており、学習しやすいのが特徴。C++の永遠のライバル。（最後の文は僕の妄想）

[6] C#(シーシャープ): マイクロソフト社によって開発されたオブジェクト指向プログラミング言語。共通中間言語(CIL)にコンパイルされて実行される。構文は Delphi に準じておりまた C 言語風でもある。

[7] ソースコードやそれをコンパイルした中間表現を逐次解釈しながら実行するプログラミング言語。

[8] コンピュータを動作させるための機械語を逐語的に変換した言語で低級言語のひとつ。

要するにクラスを使えるということです。特にカプセル化や継承、多態が重要な機能です。

c. ジェネリックプログラミング

テンプレートクラスやテンプレート関数を使えるということです。実をいうと Java でも似たような機能であるジェネリクスが導入されていますが、C++は特にテンプレート引数やテンプレートの特殊化という強力な機能があります。テンプレートについてはあとでたっぷり解説するのでお待ちください。

4) マクロが使える

マクロを使うことで、いろいろな面白いコードを作れます。これは C++(もしくは C 言語)の大きな醍醐味です。まあマクロについてもあとでたっぷり解説します。

5) 演算子オーバーロードが使える

演算子オーバーロードを使うと可読性が低くなるとか結構言われていますけど（特に Java の影響大）、やっぱりどう考えても演算子オーバーロードがなかったら困ります。iostream のシフト演算子なんかは賛否両論あるのですが、かなり直観的ではあります。

こういうわけで C++は「なんでもあり」な言語なんです。ある意味そのせいで学習者は苦労するので、より分かりやすくスマートな設計を目指し、Java などが作られました。

でもやっぱり、この速さと自由さ。このふたつをもってすれば、**玄人プログラマ[9]にとって C++は最高に素晴らしい言語に違いないでしょう!!**

もうひとつ補足。C++では実は最近、従来の C++03 という規格に代わって C++11[10]という新しい規格が採用されました。Visual C++ 2010 でもごく一部ですが対応しているので、これからの解説では C++11 の新機能も積極的に使用していきます。

[9] 僕が玄人プログラマだという意味ではありません。

[10] 正式名称は ISO/IEC 14882:2011。2011 年時点で最新のプログラミング言語 C++の ISO 標準。規格の策定中は 2009 年中の標準化を目指していたため、C++0x という仮称で呼ばれていた。Visual C++ 2010 では正確にはこの製作途中の C++0x が一部採用された。

ではここから本題に入っていきます。

第 3 章　マクロテクニック

マクロというのはプリプロセッサディレクティブ（コンパイラがソースコードをコンパイルする前にプリプロセッサによって実行される命令）のひとつです。これを用いることで C++のコーディングの性格はガラリと変わります。

第一節　マクロの基礎

マクロには次のような種類があります。

1) オブジェクト形式マクロ

これは単純な文字列の置換です。形式は

```
#define [マクロ名] [置換文字列]
```

です。次のような使い方があります。

```
#define BUFFER_SIZE 1024 /*定数宣言 const とはまた違った使い道がある*/
#define PLUS + /*演算子でももちろん OK*/
#define MYNAME yusuke matsushita /*途中にスペースやタブが入っても OK*/
#define UKNAME United Kingdom\
                                of Great Britain\
                                and Northern Ireland
                                /*途中に改行が入る場合はバックスラッシュを入れます*/
```

そのほかインクルードガードや_DEBUG などの使われ方があります。インクルードガードについては#pragma once で代用できます（今やほとんどのコンパイラが対応しています）し、_DEBUG のような定数は自分で定義することは少ないと思うので割愛しておきます。

2) 関数マクロ

関数に似たマクロです。結局は置換です。

```
#define [マクロ名](引数リスト) [置換文字列]
```

まずごく基本的な使い方から。

```
#include <cstdio> /*以降のソースコードではこの include は省略します。*/
```

```
/*a、b のうち大きい方を返すマクロ*/
#define MAX(a,b) ((a)>(b)?(a):(b))
int main()
{
    printf("%d\n",MAX(1*3,1+3)*2);
    /*MAX(1*3,1+3)はプリプロセッサに
                ((1*3)>(1+3)?(1*3):(1+3))
    と展開される。*/
    /*Debug モードの時は自動的にコンソールが閉じてしまう。
      これを防ぐにはこの文を追加する。
      printf("Press Enter Key.");
      getchar();*/
    return 0;
}
```

実行結果

```
8
```

まず少し補足的な話から。「?」は三項演算子といいます。「[条件式]?[真式]:[偽式]」という形式をとります。このように三つの項をとるのが三項演算子の名前の由来です。この構文自身も**式**であるので条件式が真の場合は真式、偽の場合は偽式を返します。ここら辺が if-else **文**と違うところです。なお、「式」と「文」というのは、マクロにおいてかなり重要な概念となってくるので覚えておいてください。

で、マクロの話に戻ると、コメントに書いてあるように a と b がそれぞれ素直に展開されています。MAX マクロが a、b のうち大きい方を返すのは自明ですね。（また型に依存しません!）で、置換文字列でなぜ a と b、そして全体に丸括弧が付けられているのか。わかりますね。付けないと展開されて次のようになってしまうからです。

```
printf("%d\n",1*3>1+3?1*3:1+3*2);
```

で、コンソールには 1*3+2 つまり「7」と表示されます。いろいろ予測できないことが起こってしまうので、引数や置換文字列全体に括弧をつけるのです。それでも「マクロの副作用」というものは起きてしまいます。

```
//MAX は先ほどと同じ定義。
int n=7;
int Function()
```

```
{
    printf("N increased by 2!\n")
    return n+=2; //n を 2 増やし、その n を返す。
}
int main()
{
    printf("%d\n",MAX(Function(),8));
    /*((Function())>(8)?(Function()):(8))と展開される。
      コードを書いた人はおそらく
                N increased by 2!
                9
    と表示されるのを期待している。*/
    return 0;
}
```

実行結果

```
N increased by 2!
N increased by 2!
11
```

このように、関数などが二回呼び出され、予期せぬ動作が起こってしまいます。そこでこのような場合はインライン関数を使うのが普通です。MAX マクロでは型に依存していないことが特徴でしたので、テンプレートを使ってこのように書けます。

```
template<typename T>
inline T& max(T& a, T& b)
{
    return a>b?a:b;
}
```

こちらのほうが作るのも使うのもよっぽど楽です。よってインライン関数を使える場面ではマクロではなくなるべくインライン関数を使いましょう。

第二節　for とマクロ

ここまででではごく普通のマクロ解説です。ここからは、インライン関数ではできなくてマクロでしかできないことをこれから紹介していきます。まず、かなりポピュラーな使い方として for ループを短く書くマクロがあります。

```
#define REP(i,x) for(int i=0; i<(x); ++i)
/*マクロの引数には変数などの名前を指定する役割を持たせることもできます。*/
```

```
/*REP は repeat の略だよ*/
int main()
{
	REP(cnt,10){
		printf("cnt is %d.\n", cnt);
	}
	return 0;
}
```

実行結果

```
cnt is 0.
cnt is 1.
cnt is 2.
(中略)
cnt is 9.
```

こんなことはインライン関数ではできません。for とマクロを組み合わせるとなかなか面白いですよね。

でもこのマクロだと i の型が int だけになってしまいます。これだと、例えば x が unsigned int だった場合などでは signed と unsigned の数値の比較をすることになり危険なので、どんな型でも出来るようにしたいところです。そこでこのように出来ます。

```
#define REP(type,i,x) for(type i=0; i<(x); ++i)
/*type を()で囲ってはいけないことに注意。i は囲ってもいいが
  囲う意味がないので 囲っていない。*/
int main()
{
	REP(unsigned int,cnt,10u){ //10u は unsigned int 型のリテラル
		printf("cnt is %d.\n", cnt);
	}
	return 0;
}
```

これでかなりよくなりました。では同様にイテレータのバージョンも作ってみましょう。

```
#include <list>
#include <string>
#define REPit(type,it,container) \
		for(type::iterator it=(container).begin();\
		it!=(container).end(); ++it)
int main()
{
	std::list<std::string> textlist;
```

```
        textlist.push_front("ありがとう");
        textlist.push_back("おおきに");
        textlist.push_front("サンキュー");
        REPit(std::list<std::string>,it,textlist){
                printf("感謝の言葉、%s。\n", it->c_str());
        }
        return 0;
}
```

実行結果

```
感謝の言葉、サンキュー。
感謝の言葉、ありがとう。
感謝の言葉、おおきに。
```

うん、ちゃんと出来ました。でも std::list<std::string>ってわざわざ書くのはめんどくさいですよね。特に std::list<std::map<std::string,std::string> >なんていう長い型なら尚更です（また今のままではコンマが入っている型は使えません）。さあ、そこでです。C++11 の新機能、auto と decltype の登場です。

```
#include <list>
#define REP(i,x) for(decltype(x) i=0; i<(x); ++i)
#define REPit(it,container) for(auto it=(container).begin();\
                                        it!=(container).end(); ++it)
int main()
{
        REP(cnt,10u) printf("cnt is %d.\n", cnt);
        std::list<std::string> textlist;
        textlist.push_front("ありがとう");
        textlist.push_back("おおきに");
        textlist.push_front("サンキュー");
        REPit(it,textlist) printf("感謝の言葉、%s。\n", it->c_str());
        getchar();
        return 0;
}
```

実行結果

```
cnt is 0.
(中略)
cnt is 9.
感謝の言葉、サンキュー。
感謝の言葉、ありがとう。
感謝の言葉、おおきに。
```

ちゃんと動きます。では説明していきます。auto と decltype は次のようなものです。

```
auto [変数名] = [式];
```

```
decltype(式) 変数名;
```

auto は初期化の際、自動的に型を判別してくれるものです。REPit マクロの場合、it は begin()の戻り値の型になっています。

decltype（多分 declared type の略）は auto とよく似たもので、式と同じ型の変数を宣言できます。REP マクロの場合 i は x と同じ型になり、さらに 0 に初期化されます。わかりましたね。（なお、decltype の括弧の中の式に関数なんかを入れてもその式が評価されることはありません。見られるのは型だけです。）

どうでしたか? マクロを上手に使うとこんなにも洗練されたコードが書けるんです。実際僕もこのマクロは多用しています。

ふう。少し疲れました。でもまだまだですよ～。

第三節　マクロの中の#[11]たち

マクロでしか使えない二つの演算子を紹介します。

まず、文字列化演算子#から。

```
#include <iostream> //以降のコードでは省略します。
#include <string>
#define DEBUG_VAR(var) \
                    std::cout<<"the value of " #var " is: "<<var<<std::endl
/*var は variant（変数）の略。*/
/*”abc”“def”のような文字列は”abcdef”と解釈される。*/
/*急に iostream を使ったのは、表示が型に依存しないから。*/
int main()
{
        int a=123;
        std::string b="hello!";
        DEBUG_VAR(a);
        DEBUG_VAR(b);
        return 0;
}
```

実行結果

```
the value of a is 123
the value of b is hello!
```

[11] この記号はシャープではなく、ナンバーサイン（番号記号）である。電話機にあるのもこの記号。ナンバーサインは横棒が水平で縦棒が斜めだが、シャープ♯は縦棒が垂直で横棒が斜めになっている。ただし、C#、J#などではあえてシャープをナンバーサインで代用しているのでややこしい。

このように、#演算子をつけるとダブルクオーテーションをつけた文字列として解釈されます。この演算子はデバッグなどにおいてかなり有用です。なお、VC++独自の機能として#@演算子（文字定数化演算子、charizing operator）もあります。こちらはシングルクォーテーションをつけるものですが、使う機会はほぼないでしょう。
もうひとつはトークン連結演算子##です。さっきとは打って変わって特に意味のないプログラムです。

```
#define MAKE_NAME(name,id) name##_##id
int main()
{
	int MAKE_NAME(locomotive,d51) = 100;
	/*MAKE_NAME(locomotive,d51)は locomotive_d51 と展開される*/
	printf("%d\n", locomotive_d51);
	return 0;
}
```

実行結果

```
100
```

というわけで、何でもくっつけてしまうのが##です。これもかなり有用です。

第四節　可変長引数マクロ

マクロはなんと、可変長引数をとることができます。これは関数の可変長引数より低機能ですが使いやすいです。

```
#define DEBUG_PRINTF(format, ...) \
		printf("FILE:%s\nLINE:%d\n" format, \
		__FILE__, __LINE__, __VA_ARGS__)
int main()
{
	int a=1+2+3+4+5,b=1*2*3*4*5;
	DEBUG_PRINTF("a:%d b:%d\n",a,b);
	/*printf("FILE:%s\nLINE:%d\n" "a:%d b:%d\n",
	ファイル名,行番号,a,b)と展開される。*/
	return 0;
}
```

実行結果

```
FILE:（ファイルの完全パス）
LINE:（DEBUG_PRINTF("a:%d\n",a);がある行の番号）
a:15 b:120
```

このように、__VA_ARGS__が単純に...に入れられた引数に置換されます。ちなみに急に登場してきたこの__FILE__と__LINE__は「定義済みマクロ」といって、最後に展開されます。__FILE__や__LINE__が「__FILE__, __LINE__, __VA_ARGS__)」のある行で展開されると思うかもしれませんが、そうではなく、最後に展開されます。そういう仕様です。

さて、さっきの main 関数をちょっと変えてみましょう。

```
int main()
{
	DEBUG_PRINTF("I have nothing special to write.");
	/*printf("FILE:%s\nLINE:%d\n"
	  "I have nothing special to write.\n",ファイル名,行番号,)
	   と展開される*/
	return 0;
}
```

というわけで、妙なコンマが残ってしまうのでコンパイルエラーとなります!!!

…と思っていたのですが、VC++2010 でコンパイルしたところ、最後のコンマが消されてコンパイルが通りました。どうやら進化したみたいです。知らなかった。なんでもやってみるもんですね。

ちなみに gcc では##__VA_ARGS__と書けば最後のコンマは除去してくれます。

第五節　マクロを消す

マクロのひとつ大きな欠点が、名前空間がないということです。名前空間がないせいで使用するときに名前が被って邪魔になることがあります。その場合はマクロをなかったことにします。それが undef です。オブジェクト形式マクロの場合でも関数マクロの場合でもマクロ名のみを書きます。

```
#undef MYNAME
#undef MAX
```

また、マクロの名前を変えたい場合には#pragma push_macro と#pragma pop_macro を利用します。これは VC++でも g++でも使えます。例を示しておきます。

```
/*CreateWindow というマクロを WinapiCreateWindow というマクロに変える*/
#pragma push_macro(“CreateWindow”)
#undef CreateWindow
#define WinapiCreateWindow
#pragma pop_macro(“WinapiCreateWindow”)
```

なお VC++では warning C4602 が表示されますが気にしなくていいです。

第六節　複文マクロ

マクロは一つの式として表せることが多いのですが、どうしても一つの文もしくは複数の文となってしまう場合があります。変数宣言や for 文・while 文などの繰り返しが必要な場合です。（if 文や switch 文は頑張れば三項演算子だけで出来ます。）たとえばこれ。

```
#define REPEAT_HELLO(n) for(int i=0; i<n; ++i)printf("hello!\n");
/*n 回 hello!と表示するというだけのマクロ(笑)*/
int main()
{
	REPEAT_HELLO(3);
	return 0;
}
```

実行結果

```
hello!
hello!
hello!
```

とりあえず問題ないようです。ではこうしたらどうでしょうか?

```
int main()
{
	int n=3;
	if(n<10)REPEAT_HELLO(n);
	else printf("TO MUCH");
	getchar();
	return 0;
}
```

途端に「error C2181: else 文が if と一致しません。」というコンパイルエラーが出ます。よく見てください。REPEAT_HELLO(n);は一見文ですが、for 文自体文でそれにセミコロンで空文を追加しているので、二文です。

```
if(条件式)文 [else 文(なくてもよい)]
```

というのが正しい形式なので、else 文に対応する if 文はないことになってしまうのです。なお、こう書けば大丈夫です。

```
if(n<10){
	REPEAT_HELLO(n);
}else{
	printf("TO MUCH");
}
```

{}の中に複数の文を入れると一つの文ということになります。

REPEAT_HELLO(n)の最後のセミコロンを取り除いてもうまくいきます。

しかし、こんな風に気にせずに普通の関数と同じように使いたいものです。そこで、あるイディオムが役に立つのです。REPEAT_HELLO をこう書き換えましょう。

```
#define REPEAT_HELLO(n) \
		do{\
			for(int i=0; i<n; ++i)printf("hello!\n");\
		}while(0)
```

こうすると、セミコロンをつけてちょうど一文になるのです。

do-while 文を知らない人のために説明します。

```
do 文 while(条件式);
```

①文を評価(実行)する

②条件式が真ならば①に戻る。偽ならば終わる

というのが do-while で、while とは文の評価と条件式のチェックの順番が逆です。

では、この do-while(0)イディオムの意味はわかりましたね? 0 は偽なので文が一度だけ評価されるのです。最適化コンパイラは大抵 while(0)のような無意味なテストを取り除くので、時間の心配はしなくてよいでしょう。

というわけで、複文マクロを作るときには必ず do-while(0)で囲みましょう。

最後に実用的な例を紹介しておきます。これは Win32 アプリケーションでの例です。

```
#define OutputStringF(...) \
	do{ \
		TCHAR text[256]; \
		wsprintf( text, __VA_ARGS__ ); \
		OutputDebugString( text ); \
	}while(0)
```

OutputDebugString はデバッグウィンドウに出力する関数で、これと wsprintf を組み合わせました。こういう処理は何回も繰り返すのでまとめてしまったほうが楽です。また、可変長引数はマクロのほうが楽です。はあ。やっと知識的なことが終わった。ではここからはより実用的な話をしていきます。

第七節　ちょっと使えるマクロコレクション

ではこれまで勉強したことからいくつか実用的なマクロを作ってみましょう！

```
/*配列の要素数を求めるマクロ*/
#define ARRAY_SIZE(x) (sizeof(x)/sizeof(x[0]))
int main()
{
	int piarray[]={314,159,265,358,979,323,846,264,338,327};
	printf("piarray has %d elements.\n", ARRAY_SIZE(piarray));
	return 0;
}
```

実行結果

```
piarray has 10 elements.
```

これはインライン関数ではできません。配列はテンプレートではポインタとして解釈されてしまうのです。次も変わったマクロです。

```
/*ある数が範囲内かを調べるマクロ */
#define BETWEEN(a,x,b) (a (x))&&((x) b)
int main()
{
	int x=17;
	if(BETWEEN(-100<,x,<=100))printf("範囲内です。\n");
	/*( (-100<(x))&&((x)<=100) )と展開される*/
	return 0;
}
```

これも変わったマクロです。使い方は見ればわかると思います。結構変数二回書いて&&を書くのが面倒なのでこんなのもありだと思います。

さてお次は

```
#define S_CASE(x) break; case (x):
#define S_DEFAULT() break; default:
#define OR_CASE(x) case (x):
#define OR_DEFAULT() default:
```

```
int main()
{
	for(int i=1; i<=4; ++i){
		switch(i){
			S_CASE(1)OR_CASE(2){
				printf("1 or 2\n");
			}
			S_CASE(3){
				printf("3\n");
			}
			S_DEFAULT(){
				printf("neither 1 nor 2 nor 3\n");
			}
		}
	}
	return 0;
}
```

実行結果

```
1 or 2
1 or 2
3
neither 1 nor 2 nor 3
```

これは switch 文で break を忘れてしまうことが多いので無理やりマクロでまとめちゃえ、ということで作ってみました。それにこうすると if 文に似て見た目の上で{}と相性がよくなるんですよね。switch 文でここまでやってしまうのも賛否あるでしょうが僕は気に入っています。

さて、最後にとっておきシンプルなものをどうぞ。

```
#include <map>
/*コンマが入ったものをマクロに渡す際、一引数にまとめる*/
#define BIND(...) __VA_ARGS__
/*auto を使わない REPit マクロ*/
#define REPit(type,it,container) \
		for(type::iterator it=(container).begin();\
		it!=(container).end(); ++it)
int main()
{
	std::map<int,int> m;
	REPit(BIND(std::map<int,int>::iterator),it,m){
		/*何らかの処理*/
	}
	return 0;
}
```

わかりますか? std::map<int,int>::iterator をそのまま指定するとコンマの部分が勝手に引数を区切るコンマだと勘違いされて引数二つぶんだと思われてエラーになってしまいます(マクロでは<>は普通に比較演算子と解釈されます)。でも BIND()でくくるといくらコンマを入れても引数一つぶんだと解釈されます。すごく華麗でしょ?
これでマクロコレクションは終わりです。ぜひ全部使ってみてくださいね。

第八節　マクロ展開の罠

__COUNTER__という興味深いマクロがあります。

```
int main()
{
	printf("%d\n",__COUNTER__); //0 と展開される
	for(int i=0;i<2;++i){

	printf("%d,%d\n",__COUNTER__,__COUNTER__,__COUNTER__*2);
		//1,2,3*2 と展開される
	}
	return 0;
}
```

実行結果

```
0
1,2,6
1,2,6
```

このように、__COUNTER__は展開される度に 1 増えます。というわけで、これを利用すればユニークな名前の変数を作成可能ですね。やってみましょう。

```
#define MAKEUNIQUENAME() veryuniquename##__COUNTER__
int main()
{
	int MAKEUNIQUENAME(); //veryuniquename0 と展開される?
	double MAKEUNIQUENAME(); //veryuniquename1 と展開される?
	return 0;
}
```

あれれ、「'veryuniquename__COUNTER__' : 再定義されています。異なる基本型です。」ってエラーが出ましたね。どうやらどちらも

veryuniquename__COUNTER__と展開されたようです。
これこそが世にも恐ろしい**マクロ展開の罠**なのです。
これは##とか#とかを使っていると結構すぐにはまる罠です。では回避方法をお教えします。**一旦別のマクロを経由するのです。**

```
#define COMBINE(x,y) x##y
#define COMBINE2(x,y) COMBINE(x,y)
#define MAKEUNIQUENAME() COMBINE2(veryuniquename,\
                                                    __COUNTER__)
int main()
{
        int MAKEUNIQUENAME();
        /*COMBINE2(veryuniquename,__COUNTER__)
          →COMBINE(veryuniquename,0)
          →veryuniquename0
          と展開される*/
        double MAKEUNIQUENAME();
        /*同様に veryuniquename1 と展開される*/
        return 0;
}
```

わかりましたね。これ結構高等なテクニックです。マクロの展開の順番は慣れたら簡単です。
でも、いちいちこういうマクロつくるのって面倒ですよね。ではこうします。

```
#define EXPANSION0_2(macro,x,y) macro(x,y)
#define EXPANSION1_2(macro,x,y) EXPANSION0_2(macro,x,y)
#define COMBINE(x,y) x##y
#define MAKEUNIQUENAME() \

        EXPANSION1_2(COMBINE,veryuniquename,__COUNTER__)
int main()
{
        int MAKEUNIQUENAME();
        double MAKEUNIQUENAME();
        return 0;
}
```

ね、華麗でしょ? この EXPANSION マクロはマクロ展開の罠を回避するために汎用的に使えるマクロなんです。

```
//EXPANSION[展開する階層]_[引数の数](マクロの名前,引数リスト)
#define EXPANSION0_1(macro,x) macro(x)
```

```
#define EXPANSION1_1(macro,x) EXPANSION0_1(macro,x)
#define EXPANSION2_1(macro,x) EXPANSION1_1(macro,x)
#define EXPANSION3_1(macro,x) EXPANSION2_1(macro,x)
#define EXPANSION0_2(macro,x,y) macro(x,y)
#define EXPANSION1_2(macro,x,y) EXPANSION0_2(macro,x,y)
#define EXPANSION2_2(macro,x,y) EXPANSION1_2(macro,x,y)
#define EXPANSION3_2(macro,x,y) EXPANSION2_2(macro,x,y)
```

同じ要領でどんどん拡張できますね。これ、結構便利です。ぜひ使ってみてくださいね。

さあ、マクロはこれで終わりです。なんか長かったですね。ここまで勉強すればあとは実践あるのみです。
では、お待ちかねのテンプレート編です！

第 4 章　テンプレートテクニック

テンプレートはあっさり解説します。そのかわり最後にエラトステネスの篩というネタコード（書き下ろしです！）を載せておきました。

第一節　基本

```
/*最大公約数*/
template <class T> T Gcd(T a, T b) //a,b は正整数
{
	アルゴリズムはユークリッドの互除法
	if(b==0) return a;
	return Gcd(b, a%b);
}
/*行列*/
template <class T=double, int WIDTH=3, int HEIGHT=3> class Matrix{
	T elements[WIDTH][HEIGHT];
public:
	template<int X,int Y>T& At()const{return elements[X][Y];}
};

int main()
{
	/*テンプレート関数の場合、引数から「型推論」することができる。*/
	int gcd1=Gcd(42,63);
```

```
	int gcd2=Gcd<int>(42,63);//これも同じこと

	Matrix<int,5,7> mx1;
	Matrix<> mx2; //Matrix<double,3,3>の型　デフォルト値を使用
	int value=mx.At<3,0>();
	return 0;
}
```

このように**クラス**や**構造体**や**関数**にはテンプレート引数を指定することができます。テンプレート引数には**型**（**ポインタ型**や**参照型**や**関数ポインタ型**も含みます）や整数型の定数（**列挙型**や**ポインタ型**や**関数ポインタ型**の定数も可）を使用できます。また、デフォルト値を指定することもできます。

基本、終わっちゃいました。テンプレートはこんなもんです。使い方がすべてです。

第二節　テンプレートの特殊化

```
/*行列*/
template <class T, int WIDTH, int HEIGHT> class Matrix{
	T elements[WIDTH][HEIGHT];
public:
	void WhatAmI(){
		printf("I am a matrix but not a square matrix\n");
	}
};
/*正方行列*/
template <class T, int N> class Matrix<T,N,N>{
	T elements[N][N];
public:
	void WhatAmI(){printf("I am a square matrix\n");}
};
/*なんか int だけ特別にしてみた*/
template <int WIDTH, int HEIGHT> class Matrix<int,WIDTH,HEIGHT>{
	int elements[WIDTH][HEIGHT];
public:
	void WhatAmI(){printf("I am an integer matrix\n");}
};
int main()
{
	Matrix<double,2,4> mx1;
	Matrix<double,3,3> mx2;
	Matrix<int,2,3> mx3;
```

```
        //Matrix<int,2,2> mx4; は曖昧なのでエラーとなる
        mx1.WhatAmI();
        mx2.WhatAmI();
        mx3.WhatAmI();
        return 0;
}
```

実行結果

```
I am a matrix but not a square matrix
I am a square matrix
I am an integer matrix
```

こんなふうにテンプレートは特殊化できます。フィーリングでわかってください。多分これで事足りるでしょう。

第三節　テンプレートの限界

テンプレートの再帰の上限数を調べてみましょう。

```
template <int N> struct loop {enum{value = loop<N - 1>::value+1};};
template <> struct loop<0> {enum{value = 0};};
const int value = loop<499>::value;
int main(){return 0;}
```

ここでは 499 としましたが、この 499 の部分に数字を入れてコンパイルが通る上限の数がそのコンパイラのテンプレートの再帰の上限です。

VC++では 500 以上の数を入れると「fatal error C1202: 再帰的な型の指定または関数の依存関係が複雑すぎます。」と出て、そのあとに恐ろしく長いエラーを吐き出します。**たったの 500 でですよ**。なんという制約でしょうか。積極的に使いたい側としてはなかなか困ったものですが、実際コンパイラの側としても、制約を設けなければ無限ループを起こしてしまい、スタックオーバーフローしたりメモリが足りなくなったりするので、制約を設けざるを得ないのです[12]。テンプレートメタプログラミングをする際にはこういったことを意識しなければなりません。

第四節　エラトステネスの篩はさよならの代わり

では最後にエラトステネスの篩(ふるい)をコンパイル時に行うプログラムを書きます[13]。たぶんこれまでに解説したことでギリギリ読めます。頑張

[12]コンパイラはあらかじめテンプレートの展開が無限になるかどうかは予測することはできない。詳しくは「停止性問題」を参照。

[13] これはネタプログラミングです。普通にやるなら素数表をハードコードするかあるいは別ファイルを作りましょう。

ってください。一応 enum ハックというテクニックを使っていますが見れ
ばわかるでしょう。
コンパイル時間は思ったよりは速いです。ぜひ実行してみてください。

```
template<int Size,int Depth> struct EratosthenesFirstPrime;
template<int N> struct SquareRoot;

//Eratosthenes
template<int N,int Size,int Depth>
struct EratosthenesB{
	enum{
		PrevIsPrime=EratosthenesB<N,Size,Depth-1>::IsPrime,
		FirstPrime=EratosthenesFirstPrime<Size,Depth-1>::Ans,
		End=EratosthenesFirstPrime<Size,Depth-1>::End,
		/*素数と確定したら 1、合成数と確定したら 0、未確定な
ら-1。*/
		IsPrime=(PrevIsPrime!=-1)?PrevIsPrime:End?1
			:(N==FirstPrime)?1:(N%FirstPrime==0)?0:-1
	};
};
template<int Size, int Depth>
struct EratosthenesB<1,Size,Depth>{
	enum{IsPrime=0};
};
template<int N,int Size>
struct EratosthenesB<N,Size,0>{
	enum{IsPrime=-1};
};
//これが本体
template<int N,int Size>
struct Eratosthenes{
	enum{IsPrime=EratosthenesB<N,Size,SquareRoot<Size>::Ans>
                                                    ::IsPrime};
};

//SquareRoot
template<int N,int M>
struct SquareRootB{
	enum{Ans=M*M<=N?M:SquareRootB<N,M-1>::Ans};
};
template<int N>
struct SquareRootB<N,1>{
	enum{Ans=1};
};
```

```
template<int N>
struct SquareRoot{
	enum{Ans=SquareRootB<N,N>::Ans};
};

//EratosthenesFirstPrime
template<int Size,int Depth,int N>
struct EratosthenesFirstPrimeB{
	enum{
	Ans=(EratosthenesB<N,Size,Depth>::IsPrime==-1)?N:
			EratosthenesFirstPrimeB<Size,Depth,N+1>::Ans};
};
template<int Size,int Depth>
struct EratosthenesFirstPrimeB<Size,Depth,Size>{
	enum{Ans=Size-1};
};
template<int Size,int Depth>
struct EratosthenesFirstPrime{
	enum{
			Ans=EratosthenesFirstPrimeB<Size,Depth,
				EratosthenesFirstPrime<Size,
					Depth-1>::Ans+1>::Ans,
			End=(Ans>SquareRoot<Size>::Ans)
	};
};
template<int Size>
struct EratosthenesFirstPrime<Size,0>{
	enum{Ans=2,End=0};
};

int main()
{
	const int x=389;
	/*400 までの範囲でエラトステネスの篩を行い、 389 が素数かどう
かを判定する。*/
	if(Eratosthenes<x,400>::IsPrime) printf("%d is a prime number.\n",x);
	else printf("%d is not a prime number.\n",x);
	return 0;
}
```

華麗でしょ? みなさんも好きなものをテンプレートメタプログラミングで作ってみてください。（組むのは案外簡単です。一番大変なのはデバッグです...）結構やってみるとはまりますよ!!!

これでテンプレートも終わりました。全部終わりです。ばんざーい！

最後に

え、枚数多かった割にまだ物足りないって？　ずいぶん意欲的ですね。
じゃあ僕のおすすめのサイトをいくつか紹介したいと思います。

C++マニアック http://homepage2.nifty.com/well/Index.html
　中級者向けのサイトですが、なかなか痒いところに手の届くいいサイトです。解説も大変わかりやすいです。

トリッキーコードネット http://tricky-code.net/nicecode/code06.php
　C 言語/C++を中心にちょっと面白い&使える（ときどき使えない）コードをいろいろ掲載しています。まあ、なんかプログラミングっていいな、って気持ちになれます。

More C++ Idioms http://ja.wikibooks.org/wiki/More_C%2B%2B_Idioms
　結構面白いことがたくさん書いてあります。内容としては上級者向けだと思います。僕自身このサイトを見てかなり「へえ」と思うことがありました。
　英語版(http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms)もぜひご覧ください。

ちなみに、僕のブログは（今のところほとんど何も書いてないのですが）「非現実的非日常的非常識的非日記」(http://d.hatena.ne.jp/Kinokkory/)です。

まだ書きたいことは山ほどあるのですが、ひとまずこれで終わりにしたいと思います。この文章をきっかけに C++プログラマが増えてくれると本当にうれしいです。
最後まで僕の文章を読んでくださって本当にありがとうございました。