プログラミングTips

－ －

2014年1月20日 初版

板垣 衛

改訂履歴

|  |  |  |  |
| --- | --- | --- | --- |
| 版 | リリース | 担当 | 改訂内容 |
| 初版 | 2014年1月20日 | 板垣 衛 | （初版） |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目次

[ 概略 1](#_Toc378053172)

[ 目的 1](#_Toc378053173)

[ 要件定義 1](#_Toc378053174)

[▼ 基本要件 1](#_Toc378053175)

[▼ 要求仕様／要件定義 1](#_Toc378053176)

[ 仕様の依存関係 2](#_Toc378053177)

[ データ仕様 2](#_Toc378053178)

[ 処理仕様 2](#_Toc378053179)

# 概略

プログラミングの際に知っておくと役に立つことを解説。

# 目的

本書は、ゲームプログラミングを少しでも効率的・効果的にすることを目的とする。

# コーディングに関するTips

## 配列の初期化

C言語で配列を初期化する際は、通常下記のように記述する。

例：配列の初期化

|  |
| --- |
| int arr[5] = { 1, 2, 3, 4, 5 }; |

配列要素数を超える初期化を行うとコンパイルエラーになるが、少なかったときは、残りの要素が 0 で初期化されることをC言語の仕様は保証している。

例：要素数と一致しない配列の初期化

|  |
| --- |
| int arr[5] = { 1, 2, 3, 4, 5, 6 }; //←初期化の数が要素数より多いのでコンパイルエラー  int arr[5] = { 1, 2, 3 }; //←初期値が指定されなかった要素は 0 になる　⇒ { 1, 2, 3, 0, 0 } |

この仕様に基づき、ゼロクリアした配列を用意した場合は、初期値を一つも書かずに初期化を指定するだけでよい。

例：配列要素をすべて 0 で初期化

|  |
| --- |
| int arr[5] = {}; //←初期値を一つも指定しなければ、全ての要素が 0 になる　⇒ { 0, 0, 0, 0, 0 } |

## #defineマクロを活用する

### 便利な「定義済みマクロ」

まず、Visual C++ で使用可能な定義済みマクロを紹介する。ゲームプログラミングでも特に有用なマクロを赤字で示す。

なお、説明内容は、Visual Stduio 2013 のWebページからそのまま引用。（<http://msdn.microsoft.com/ja-jp/library/b0084kay.aspx>）

【ANSI C準拠の定義済みマクロ】

\_\_DATE\_\_ 現在のソース ファイルのコンパイル日付。

日付は Mmm dd yyyy 形式のリテラル文字列です。 月の名前 Mmm は、TIME.H で宣言された asctime ライブラリ関数によって生成される日付のものと同じです。

\_\_FILE\_\_ 現在のソース ファイルの名前。

\_\_FILE\_\_ は二重引用符で囲まれた文字列に展開されます。 ファイルへの完全なパスが表示されるようにするには、/FC (診断時のソース コード ファイルの完全パス) を使用します。

\_\_LINE\_\_ 現在のソース ファイルの行番号。

行番号は 10 進数の整数定数です。 これは #line ディレクティブを使用して変更できます。

\_\_STDC\_\_ ANSI C 標準への完全準拠。

/Za コンパイラ オプションを指定している場合で、C++ コードをコンパイルしないときにのみ、整数定数 1 として定義されます。それ以外の場合は、定義されません。

\_\_TIME\_\_ 現在のソース ファイルの最新のコンパイル時間。

時間は hh:mm:ss 形式の文字列リテラルです。

\_\_TIMESTAMP\_\_ 現在のソース ファイルの最終更新日時。

日時は Ddd Mmm Date hh:mm:ss yyyy 形式の文字列リテラルとして表記します。ここで、Ddd は曜日の略称であり、Date は 1 〜 31 の整数です。

【Microsoft 固有の定義済みマクロ】

\_ATL\_VER ATL バージョンを定義します。

Visual Studio 2012 では、\_ATL\_VER は 0x0B00 として定義されます。

\_CHAR\_UNSIGNED 既定の char 型が unsigned になります。

/J を指定している場合に定義されます。

\_\_CLR\_VER アプリケーションのコンパイル時に使用された共通言語ランタイムのバージョンを定義します。

返される値は次の形式になります。

「Mmmbbbbb 」

指定項目：

「M」はランタイムのメジャー バージョンです。

「mm」はランタイムのマイナー バージョンです。

「bbbbb」はビルド番号です。

\_\_cplusplu\_cli /clr、/clr:pure、または /clr:safe を指定してコンパイルする場合に定義されます。

\_\_cplusplus\_cli の値は 200406 です。\_\_cplusplus\_cli は翻訳単位全体で有効になります。

\_\_cplusplu\_winnt /ZW オプションを使用してコンパイルする場合に定義されます。

\_\_cplusplus\_winrt の値は 201009 です。

\_\_COUNTER\_\_ 0 から始まる整数に展開されます。

ソース ファイルで使用されるか、ソース ファイルのヘッダーに追加されるたびに 1 ずつ増えます。\_\_COUNTER\_\_ はプリコンパイル済みヘッダーを使用するときのその状態を記憶します。

\_\_cplusplus C++ プログラムの場合にのみ定義されます。

\_\_CPPRTTI /GR (ランタイムの型情報の有効化) を指定してコンパイルしたコードの場合に定義されます。

\_\_CPPUNWIND /EH (例外処理モデル) フラグのいずれかを使用してコンパイルしたコードの場合に定義されます。

\_DEBUG /LDd、/MDd、/MTd を使用してコンパイルする場合に定義されます。

\_DLL /MD または /MDd (マルチスレッド DLL) を指定した場合に定義されます。

\_\_FUNCDNAME\_\_ 外側の関数の装飾名を文字列として定義します。

関数でのみ有効です。/EP または /P コンパイラ オプションを使用する場合、\_\_FUNCDNAME\_\_ は展開されません。

\_\_FUNCSIG\_\_ 外側の関数のシグネチャを文字列として定義します。

関数でのみ有効です。/EP または /P コンパイラ オプションを使用する場合、\_\_FUNCSIG\_\_ は展開されません。

64 ビット オペレーティング システムでは、既定の呼び出し規約は \_\_cdecl です。

\_\_FUNCTION\_\_ 外側の関数の非装飾名を文字列として定義します。

関数でのみ有効です。/EP または /P コンパイラ オプションを使用する場合、\_\_FUNCTION\_\_ は展開されません。

\_INTEGRAL\_MAX\_BITS 整数型の最大サイズ (ビット単位) をレポートします。

\_M\_ALPHA DEC ALPHA プラットフォームの場合に定義されます (サポート終了)。

\_M\_AMD64 x64 プロセッサの場合に定義されます。

\_M\_CEE /clr の任意の形式 (/clr:oldSyntax、/clr:safe など) を使用するコンパイルの場合に定義されます。

\_M\_CEE\_PURE /clr:pure を使用するコンパイルの場合に定義されます。

\_M\_CEE\_SAFE /clr:safe を使用するコンパイルの場合に定義されます。

\_M\_IX86 x86 プロセッサの場合に定義されます。x64 プロセッサの場合には定義しません。

M\_IX86 = 600 (既定)

ビルドオプション = Blend (/GB)

※将来のコンパイラは、最も優先度の高いプロセッサを反映する

　別の値を規定に出力する予定です

\_M\_IX86 = 500

ビルドオプション = Pentium (/G5)

\_M\_IX86 = 600

ビルドオプション = Pentium Pro、Pentium II、Pentium III (/G6)

\_M\_IX86 = 300

ビルドオプション = 80386 (/G3)

\_M\_IX86 = 400

ビルドオプション = 80486 (G4)

\_M\_IA64 Itanium プロセッサ ファミリの 64 ビット プロセッサの場合に定義されます。

\_M\_ARM\_FP どの /arch コンパイラ オプションが使用されたかを示す値に展開されます。

/arch オプションが指定されていない場合は 30 ～ 39。ARM の既定のアーキテクチャ (VFPv3) が使用されたことを示します。

/arch:VFPv4 が使用された場合は 40 ～ 49。

\_M\_IX86\_FP どの /arch コンパイラ オプションが使用されたかを示す値に展開されます。

/arch:IA32 が使用された場合は 0。

/arch:SSE を使用した場合は 1。

/arch:SSE2 が使用された場合は 2。

/arch を指定しなかった場合は、これが既定値になります。

\_M\_MPPC Power Macintosh プラットフォームの場合に定義されます (サポート終了)。

\_M\_MRX000 MIPS プラットフォームの場合に定義されます (サポート終了)。

\_M\_PPC PowerPC プラットフォームの場合に定義されます (サポート終了)。

\_M\_X64 x64 プロセッサの場合に定義されます。

\_MANAGED /clr を指定している場合に 1 として定義されます。

\_MFC\_VER MFC バージョンを定義します。

たとえば、Visual Studio 2012 では、\_MFC\_VER は 0x0B00 として定義されます。

\_MSC\_BUILD コンパイラのバージョン番号のリビジョン番号部分に評価されます。

リビジョン番号はピリオド区切りのバージョン番号の 4 番目の部分です。 たとえば、Visual C++ コンパイラのバージョン番号が 15.00.20706.01 ある場合、\_MSC\_BUILD マクロは 1 に評価されます。

\_MSC\_EXTENSIONS /Ze コンパイラ オプション (既定) を使用してコンパイルする場合に定義されます。

その値は定義されると 1 になります。

\_MSC\_FULL\_VER コンパイラのバージョン番号のメジャー番号、マイナー番号、ビルド番号の部分に評価されます。

メジャー番号はピリオド区切りのバージョン番号の最初の部分、

マイナー番号は 2 番目の部分、

ビルド番号は 3 番目の部分です。

たとえば、Visual C++ コンパイラのバージョン番号が 15.00.20706.01 ある場合、\_MSC\_FULL\_VER マクロは 150020706 に評価されます。

コマンド ラインで「cl /?」と入力すると、コンパイラのバージョン番号が表示されます。

\_MSC\_VER コンパイラのバージョン番号のメジャー番号とマイナー番号の部分に評価されます。

メジャー番号はピリオド区切りのバージョン番号の最初の部分、

マイナー番号は 2 番目の部分です。

たとえば、Visual C++ コンパイラのバージョン番号が 17.00.51106.1 である場合、\_MSC\_VER マクロは 1700 に評価されます。

\_\_MSC\_RUNTIME\_CHECKS /RTC コンパイラ オプションのいずれかを指定している場合に定義されます。

\_MT /MD または /MDd (マルチスレッド DLL)、あるいは /MT または /MTd (マルチスレッド) を指定している場合に定義されます。

\_NATIVE\_WCHAR\_T\_DEFINED

/Zc:wchar\_t を使用している場合に定義されます。

\_OPENMP /openmp を使用してコンパイルする場合に定義されます。

Visual C++ に実装されている OpenMP 仕様の日付を表す整数が返されます。

\_VC\_NODEFAULTLIB /Zl を使用している場合に定義されます。

\_WCHAR\_T\_DEFINED /Zc:wchar\_t を使用している場合、またはプロジェクトのシステム ヘッダー ファイルで wchar\_t 型を定義している場合に定義されます。

\_WIN32 Win32 および Win64 用のアプリケーションの場合に定義されます。

これは、常に定義されます。（Win32かWin64かに関わらず、Windows用のビルドかどうかの判定に使用できます）

\_WIN64 Win64 用のアプリケーションの場合に定義されます。

\_Wp64 /Wp64 を指定している場合に定義されます。

幾つかのマクロの表示例を示す。

例：定義済みマクロ表示プログラムサンプル

|  |
| --- |
| //定義済みマクロ表示  int printDefinedMacro(const int argc, const char\* argv[])  {  printf("\n【ANSI C準拠の定義済みマクロ】\n");  printf("\_\_FILE\_\_ = \"%s\"\n", \_\_FILE\_\_);  printf("\_\_LINE\_\_ = %d\n", \_\_LINE\_\_);  printf("\_\_DATE\_\_ = \"%s\"\n", \_\_DATE\_\_);  printf("\_\_TIME\_\_ = \"%s\"\n", \_\_TIME\_\_);  printf("\_\_TIMESTAMP\_\_ = \"%s\"\n", \_\_TIMESTAMP\_\_);  printf("\n【Microsoft 固有の定義済みマクロ】\n");  printf("\_\_COUNTER\_\_ = %d\n", \_\_COUNTER\_\_); //←カウントアップを確認するために３回使用  printf("\_\_COUNTER\_\_ = %d\n", \_\_COUNTER\_\_); //←  printf("\_\_COUNTER\_\_ = %d\n", \_\_COUNTER\_\_); //←  printf("\_\_FUNCDNAME\_\_ = \"%s\"\n", \_\_FUNCDNAME\_\_);  printf("\_\_FUNCSIG\_\_ = \"%s\"\n", \_\_FUNCSIG\_\_);  printf("\_\_FUNCTION\_\_ = \"%s\"\n", \_\_FUNCTION\_\_);  printf("\_INTEGRAL\_MAX\_BITS = %d\n", \_INTEGRAL\_MAX\_BITS);  printf("\_M\_IX86\_FP = %d\n", \_M\_IX86\_FP);  printf("\_MSC\_FULL\_VER = %d\n", \_MSC\_FULL\_VER);  printf("\_MSC\_VER = %d\n", \_MSC\_VER);  return 0;  }  //関数オーバーロード用のダミー関数（\_\_FUNCDNAME\_\_テスト用）  void printDefinedMacro()  {} |

↓（実行結果）

|  |
| --- |
| 【ANSI C準拠の定義済みマクロ】  \_\_FILE\_\_ = "e:\work\github\public\test\program\c++\rtti\src\main.cpp"  \_\_LINE\_\_ = 431  \_\_DATE\_\_ = "Jan 21 2014"  \_\_TIME\_\_ = "10:04:38"  \_\_TIMESTAMP\_\_ = "Tue Jan 21 10:04:38 2014"  【Microsoft 固有の定義済みマクロ】  \_\_COUNTER\_\_ = 0  \_\_COUNTER\_\_ = 1  \_\_COUNTER\_\_ = 2  \_\_FUNCDNAME\_\_ = "?printDefinedMacro@@YAHHQAPBD@Z"  \_\_FUNCSIG\_\_ = "int \_\_cdecl printDefinedMacro(const int,const char \*[])"  \_\_FUNCTION\_\_ = "printDefinedMacro"  \_INTEGRAL\_MAX\_BITS = 64  \_M\_IX86\_FP = 2  \_MSC\_FULL\_VER = 180021005  \_MSC\_VER = 1800 |

### 定義済みマクロのデバッグ活用：アサーション

よく使われるテクニックとして、アサーション違反発生時などに \_\_FILE\_\_ や \_\_LINE\_\_ などのマクロの情報を表示すると、ランタイム時に問題発生箇所を確認し易くなる。

例：定義済みマクロの活用サンプル

|  |
| --- |
| #include <stdio.h>  #include <assert.h>  //Visual C++ 固有の定義済みマクロ「\_\_FUNCSIG\_\_」を標準化するために、  //「\_\_FUNC\_\_」として再定義（他のビルド環境でも \_\_FUNC\_\_ を用いればソースコードを共通利用できるようにする）  #define \_\_FUNC\_\_ \_\_FUNCSIG\_\_  //ファイル名取得関数  const char\* getFileName(const char\* path)  {  const char\* p = path + strlen(path) - 1;  while (p > path)  {  if (\*p == '\\' || \*p == '/')  return p + 1;  --p;  }  return path;  }  //定義済みマクロを活用したアサーションマクロ  #define MY\_ASSERT(expr, msg) \  if (!(expr)) \  { \  fprintf(stderr, "------------------------------------------------------------\n"); \  fprintf(stderr, "Assertion failed!\n"); \  fprintf(stderr, "%s(%d): %s\n", getFileName(\_\_FILE\_\_), \_\_LINE\_\_, \_\_FUNC\_\_); \  fprintf(stderr, "%s\n", #expr); /\* ←exprを # で文字列化 \*/ \  fprintf(stderr, "%s\n", msg); \  fprintf(stderr, "------------------------------------------------------------\n"); \  assert(expr); /\* ←処理比較とabortのために、標準で用意されているassert()を実行 \*/ \  }  //アサーションテスト  int testAssert(const int argc, const char\* argv[])  {  struct STRUCT  {  char memberA;  short memberB;  int memberC;  char memberD;  };  MY\_ASSERT(sizeof(STRUCT) == 8, "STRUCT構造体のサイズが8バイトではありません！");  return 0;  } |

↓（実行結果）

|  |
| --- |
| ------------------------------------------------------------  Assertion failed!  main.cpp(498): int \_\_cdecl testAssert(const int,const char \*[])  sizeof(STRUCT) == 8  STRUCT構造体が8バイトではありません！  ------------------------------------------------------------  Assertion failed: sizeof(STRUCT) == 8, file e:\work\github\public\test\program\c++\rtti\src\main.cpp, line 498 |

### 定義済みマクロのデバッグ活用：文字列連結と文字列化のテクニック

前述のサンプルでもアサーションの条件として指定した「式」を表示するために、「#」（文字列化演算子）を使用して文字列化している。

これを利用して、前述の「main.cpp(498): int \_\_cdecl testAssert(const int,const char \*[])」のように、\_\_FILE\_\_ + \_\_LINE\_\_ + \_\_FUNC\_\_ をつなげた一つの文字列を生成し、デバッグ情報のプールを効率化する。

例えば、独自のメモリマネージャを作成して、一つ一つのメモリ管理情報の中に、呼び出し元を識別するための情報を保存しておくと、後からメモリリークを追跡する際などに役立つ。この時、保存したい情報を一つの文字列に連結しておくことで、デバッグ情報のためのメモリを効率化できる。（プログラム中の文字列リテラルが増えて、プログラムサイズが大きくなる点には注意）

文字列を連結する際、\_\_LINE\_\_ のような整数を返すマクロを単純に文字列化できないので注意が必要である。以下にその問題点と解決策を示す。

例：文字列連結のサンプル①：NGバージョン

|  |
| --- |
| //ファイル名取得関数  const char\* getFileName2(const char\* path, const size\_t path\_len)  {  const char\* p = path + path\_len - 1;  while (p > path)  {  if (\*p == '\\' || \*p == '/')  return p + 1;  --p;  }  return path;  }  //処理行情報作成マクロ  #define MAKE\_FUNC\_LINE\_INFO() getFileName2(\_\_FILE\_\_ "(" #\_\_LINE\_\_ "):" \_\_FUNC\_\_, strlen(\_\_FILE\_\_))  //文字列連結テスト  int testConcatenateStr(const int argc, const char\* argv[])  {  const char\* func\_line\_info = MAKE\_FUNC\_LINE\_INFO();  printf("MAKE\_FUNC\_LINE\_INFO() = \"%s\"\n", func\_line\_info);  return 0;  } |

↓（コンパイル結果）

|  |
| --- |
| error C2121: #' : 無効な文字です : マクロ展開が解決できません。 |

C/C++言語では、コンパイル時に文字列を連結したい場合、ただ単に文字列リテラルを並べるだけで良い。

単純に \_\_LINE\_\_ に # を付けて文字列化しようとするとコンパイルエラーになってしまう。

これを回避するために、文字列化のマクロを用意する。

例：文字列連結のサンプル②：NGバージョン　※サンプル①との差分のみ

|  |
| --- |
| //文字列化マクロ  #define TO\_STRING(x) #x  //処理行情報作成マクロ  #define MAKE\_FUNC\_LINE\_INFO() getFileName2(\_\_FILE\_\_ "(" TO\_STRING(\_\_LINE\_\_) "):" \_\_FUNC\_\_ TO\_STRING( [テスト!]) TO\_STRING(123), strlen(\_\_FILE\_\_)) // ←TO\_STRINGの動作チェック用にテストコードも追加 |

↓（実行結果）

|  |
| --- |
| MAKE\_FUNC\_LINE\_INFO() = "main.cpp(\_\_LINE\_\_):int \_\_cdecl testConcatenate(const int,const char \*[])[テスト!]123" |

TO\_STRING() マクロはおおむね意図通りに動作しているが、\_\_LINE\_\_ マクロがそのまま文字列化されてしまっている。この問題を解決するために、一度 \_\_LINE\_\_ マクロを展開してから TO\_STRING() マクロに渡すようにする。

例：文字列連結のサンプル③：OKバージョン　※サンプル①＋②との差分のみ

|  |
| --- |
| //マクロ展開付き文字列化マクロ  #define TO\_STRING\_EX(x) TO\_STRING(x)  //処理行情報作成マクロ  #define MAKE\_FUNC\_LINE\_INFO() getFileName2(\_\_FILE\_\_ "(" TO\_STRING\_EX(\_\_LINE\_\_) "):" \_\_FUNC\_\_ TO\_STRING( [テスト!]) TO\_STRING(123), strlen(\_\_FILE\_\_)) // ←TO\_STRINGの動作チェック用にテストコードも追加 |

↓（実行結果）

|  |
| --- |
| MAKE\_FUNC\_LINE\_INFO() = "main.cpp(535):int \_\_cdecl testConcatenate(const int,const char \*[])[テスト!]123" |

以上により問題を解決。

改めて、修正版のサンプルプログラム全文を示す。

例：文字列連結のサンプル：OKバージョン全文　※テスト部分は削除

|  |
| --- |
| //文字列化マクロ  #define TO\_STRING(x) #x  //マクロ展開付き文字列化マクロ  #define TO\_STRING\_EX(x) TO\_STRING(x)  //ファイル名取得関数  const char\* getFileName2(const char\* path, const size\_t path\_len)  {  const char\* p = path + path\_len - 1;  while (p > path)  {  if (\*p == '\\' || \*p == '/')  return p + 1;  --p;  }  return path;  }  //処理行情報作成マクロ  #define MAKE\_FUNC\_LINE\_INFO() getFileName2(\_\_FILE\_\_ "(" TO\_STRING\_EX(\_\_LINE\_\_) "):" \_\_FUNC\_\_, strlen(\_\_FILE\_\_))  //文字列連結テスト  int testConcatenateStr(const int argc, const char\* argv[])  {  const char\* func\_line\_info = MAKE\_FUNC\_LINE\_INFO();  printf("MAKE\_FUNC\_LINE\_INFO() = \"%s\"\n", func\_line\_info);  return 0;  } |

↓（実行結果）

|  |
| --- |
| MAKE\_FUNC\_LINE\_INFO() = "main.cpp(535):int \_\_cdecl testConcatenate(const int,const char \*[]) |

### コードの連結：トークン連結演算子

前述のサンプルにて、文字列の連結方法と「#」（文字列化演算子）による文字列化を説明しているが、C/C++言語には、もう一つ似たような演算子に「##」（トークン連結演算子）というものがあるので、違いを説明しておく。

トークン連結演算子「##」は、「プログラムコードとしての文字列」を連結するためのものである。これもマクロと組み合わせて使う事が多い。以下、具体的な使用例を示す。

例：トークン連結のサンプル

|  |
| --- |
| //幾つかの関数  void functionForTokenConcatenateTestAAA01(){ printf("TEST: AAA-01\n"); }  void functionForTokenConcatenateTestAAA02(){ printf("TEST: AAA-02\n"); }  void functionForTokenConcatenateTestBBB01(){ printf("TEST: BBB-01\n"); }  void functionForTokenConcatenateTestBBB02(){ printf("TEST: BBB-02\n"); }  //トークン連結演算子を使ったマクロ  #define TEST(ID, NO) functionForTokenConcatenateTest ## ID ## NO ()  //トークン連結テスト  int testConcatenateToken(const int argc, const char\* argv[])  {  TEST(AAA, 01);  TEST(AAA, 01);  TEST(BBB, 02);  TEST(BBB, 02);  return 0;  } |

↓（実行結果）

|  |
| --- |
| TEST: AAA-01  TEST: AAA-01  TEST: BBB-02  TEST: BBB-02 |

### プリプロセッサ

ここまでに説明した「文字列化」「トークン連結」は、「コメント除去」や「#include展開」、「マクロ展開」などと同様に、「プリプロセッサ」で処理される。

「プリプロセッサ」とは「前処理」の事である。実際にコンパイルする前の前処理として、上記の処理が実行され、純粋なプログラムのテキストになったものをコンパイルするのである。

Visual C++ の場合、コンパイルの設定を変えて「プリプロセッサ」設定の「ファイルの前処理」を「はい」にすると、拡張子「.i」のファイルが出力されて、プリプロセッサの結果を確認することができる。

なお、この設定を適用すると、「.obj」ファイルが作成されず、ビルドできなくなるので注意。

例：「.i」ファイルのサンプルの一部

|  |
| --- |
| #line 1 "c:\\work\\test\\program\\c++\\rtti\\src\\main.cpp"  #line 1 "c:\\program files (x86)\\microsoft visual studio 12.0\\vc\\include\\stdio.h"  #pragma once  #line 1 "d:\\program files (x86)\\microsoft visual studio 12.0\\vc\\include\\crtdefs.h"int  testConcatenateStr(const int argc, const char\* argv[])  {  const char\* func\_line\_info = getFileName2("c:\\work\\test\\program\\c++\\rtti\\src\\main.cpp" "(" "538" "):" \_\_FUNCSIG\_\_, strlen("c:\\work\\test\\program\\c++\\rtti\\src\\main.cpp"));  printf("MAKE\_FUNC\_LINE\_INFO() = \"%s\"\n", func\_line\_info);  return 0;  }  void functionForTokenConcatenateTestAAA01(){ printf("TEST: AAA-01\n"); }  void functionForTokenConcatenateTestAAA02(){ printf("TEST: AAA-02\n"); }  void functionForTokenConcatenateTestBBB01(){ printf("TEST: BBB-01\n"); }  void functionForTokenConcatenateTestBBB02(){ printf("TEST: BBB-02\n"); }  int testConcatenateToken(const int argc, const char\* argv[])  {  functionForTokenConcatenateTestAAA01 ();  functionForTokenConcatenateTestAAA01 ();  functionForTokenConcatenateTestBBB02 ();  functionForTokenConcatenateTestBBB02 ();  return 0;  } |

### メタプログラミング：意識すべき問題点

マクロは、「#define CALC(x) (x + 2)」のように定義して、「a = CALC(b);」のように、一種の関数のように使う事ができる。この処理は、プリプロセッサによって「a = (b + 2);」という式に置き換わる。

また、「a = CALC(3);」のようにリテラル値が指定された場合、プリプロセッサによって「a = (3 + 2);」という式に置き換わると、コンパイル時に計算を済ませることができ、「a = 5;」という定数値に置き換わる。

このように、コンパイルの時点で（プログラムが実行可能になる前に）、プログラムを生成し、ある程度処理を済ませてしまう手法を「メタプログラミング」と呼ぶ。

マクロは一見して関数のように便利に使うことができるが、プリプロセッサの時点で「文字列が展開されているだけに過ぎない」という点には十分に注意が必要。以下、それが問題になるケースを示す。

例：問題になるマクロの使い方

|  |
| --- |
| //最大値判定マクロ  #define MAX(a, b) (a > b ? a : b)  //問題になるマクロの使い方テスト  int test (const int argc, const char\* argv[])  {  int x = MAX(1, 2); //結果：x=2  int y = MAX(4, 3); //結果：y=3  int z = MAX(++x, ++y); //結果：z=5（期待値）⇒ z=6（実際）  printf("x=%d, y=%d, z=%d\n", x, y, z);  return 0;  } |

↓（実行結果）

|  |
| --- |
| x=3, y=6, z=6 |

これは、プリプロセッサを通した時に、下記のようなプログラムに展開されるためである。

例：問題になるマクロの使い方

|  |
| --- |
| int test (const int argc, const char\* argv[])  {  int x = (1 > 2 ? 1, 2);  int y = (4 > 3 ? 4 : 3);  int z = (++x > ++y ? ++x, ++y);  printf("x=%d, y=%d, z=%d\n", x, y, z);  return 0;  } |

「MAX()」マクロへの引数に指定した「++」演算子がそのまま展開されていることが問題の原因である。

このように、マクロを使用する際は、「引数の中にヘタに式を書かないように」などの配慮が必要になる。

### メタプログラミング：マクロ以外の方法

上記のような問題は、マクロではなくテンプレートを使用することで解決できる。テンプレートもメタプログラミングのための強力な手段である。

テンプレートを使用したメタプログラミングの手法は、別紙の「効果的なテンプレートテクニック」にて詳述する。

## #pragma （プラグマ）を活用する

「#pragma」（プラグマディレクティブ）は、コンパイラに対して、コンパイルのための何らかの指示・指定に使用する。例えば、「#pragma optimize(“ts”, on)」は、コンパイラに最適化手法を指定するためのプラグラマである。インクルードファイルを一回しかインクルードしないようにするための「#pragma once」などが特に有名。

プラグマは、一時的なコンパイルオプションの変更にとても効果的である。

例えば、通常はプログラムサイズが大きくなりすぎないように、「インライン関数の展開」を「規定」や「inlineのみ」にしていところを、特にパフォーマンスが要求される特定のコードでのみ「拡張可能な関数すべて」に変えるといったことができる。

プラグマの内容・用法はコンパイラによって異なるため、各プラットフォーム向けの共通コードで利用するのは若干難しい。その回避策として、プラグマを#define マクロを使って「#define OPTIMIZE\_ON #pragma optimize(“ts”, on)」のようなことをしようとすると、「#」が文字列化演算子とみならされてうまくいかない。「#define OPTIMIZE\_ON optimize(“ts”, on)」「#pragma OPTIMIZE\_ON」のようにするとうまくいくが、プラグマの指定自体は省略できなくなる。省略したい場合は、「message」などの何の影響もない無難なプラグマを指定しておくとよい。

なお、存在しないプラグマが指定されてもエラーにはならないため（警告にはなる）、警告前提で複数プラットフォーム向けのプラグマを列挙するようなことは可能。

どのようなプラグマがあるかは各プラットフォームで調べること。

【重要】プラグマを適用する際は、個人の判断で自由に行わず、開発プロジェクトの方針に従う必要があるため、よく相談してから使用しなければならない。

# 分析に関するTips

## 集計の考え方

## プロファイラ／パフォーマンスアナライザ

# 処理に関するTips

## フェード処理の考え方

■■以上■■

索引

**索引項目が見つかりません。**

以　上