QA C++

高级用户手册

目　录

1 个性配置（Personalities）

配置文件包含一组配置选项，它们决定了 QA C++分析源代码和显示注释源代码的方式。工程中的每个文件夹都可以有自己的个性配置，在文件夹参数对话框中设置。

GUI 中个性配置的表象之下是一套命令行的配置选项，保存在个性文件里。在下面对个性配置的描述中，给出了这些命令行选项的参考，它们在将**第 2 章：配置选项**中做详细说明。然而，个性设置和配置选项之间的关联并非总是显而易见的。只有 QA C++的命令行用户才有必要理解这些配置选项。

Message Personality

Message Persoanlity 控制警告消息在注释源代码中的显示。

每条警告消息根据其类型包含在一个消息组当中，每个消息组分配在一个消息级别里。消息级别从 0 到 9。

默认的个性设置显示所有检查到的警告消息，你可以抑制某些消息或消息级别的显示，这有助于聚焦重要问题。抑制消息的方法见**“QA C++用户手册”**。

消息级别的标准配置描述如下。级别中的消息分配在某种程度上是随意的，但它为我们提供了区分问题重要性的框架。

Information（0）

级别的重要性最低。

Style Guidelines（1）

内含关于编程风格的信息。

Maintainability（2）

可能的维护性问题，诸如“未使用的变量”。

C++ Language Features（3）

其中的信息提供对 C++语言使用方面的指导，包括 C 到 C++的转换、C++库的使用、以及语言的有效使用等。

Local Standards（4）

关于对企业内部编程标准的违背问题。

Design Problems（5）

指出源代码中带有设计问题的领域，诸如类设计、资源泄漏、行为问题等。

Portability（6）

在特定编译器环境下，某些遗留的 C++代码得不到编译器的正确解释时给出警告。

Complexity（7）

内含缺省的警告消息（4700），不论何时当 Analyser Personality->Metrics->Metrics Thresholds 中的设置被超出，都会给出该警告。见**“QA C++用户手册”第 8 章**。

ISO Standards（8）

指出不符合 ISO C++标准中语言定义的代码结构。

Errors（9）

这些消息指出了 QA C++无法分析的情况，可能是因为配置错误、语法错误或者是 QA C++无法识别的语言变化。

Errors（9）、ISO Standards（8）以及 Design Problems（5）中的消息是更为有用的。其他级别的重要程度依赖于本地环境，但在需要的情况下，可以使用用户消息文件改变 QA C++的消息结构，见**“QA C++用户手册”第 10 章之定制消息系统**。

个性设置	选项
Warning Messages	
Message Groups	
Information	-su 0
Style Guidelines	-su 1
Maintainability	-su 2
C++ Language Features	-su 3
Loacal Standards	-su 4
Design Problems	-su 5
Portability	-su 6
Complexity	-su 7
ISO Standards	-su 8
Errors	-su 9
Advanced Settings	

User Message Files	
User Message File	-up
	-usr
Post-Analysis Command String	N/A
Save Message Settings By	
Suppressed Messages	-n
Selected Messages	-o
Display	
Message Display	
Annotated source display format	
Messages Only	-m
Source Lines with Warnings	-one
Full Source with Warnings	-m-, -one-
Message Filtering	
Create Analysis Summary	-summ
Display Header Warnings	-hdr
Show Suppressed Warnings	-hw
Display Line Number	-l
Maximum Message Occurrence	-max
Message Format	-format

Analyser Personality

个性设置	选项
Project Headers	
Header Includes	-i
Suppress Output	-q
Project Macros	

Project Macro Defines	-d
Style	
Coding Style	
Source Code Tab Spacing	-t
Code Indent Level	-liw
Layout Configuration File	-lf
Ignore code layout in headers	-lih
Labels create hanging indent	-luhi
Analysis Processing	
Processing Options	
Encoding	-en
Preprocessed Output	-ppl
Include filename and line number in preprocessed listing	-ppf
Only Perform Preprocessing Analysis	-ppo
Macro Warning Processing	
Hide Macro Warnings	-hmw
Warning Calls	
Warning Calls	-wc
Metrics	
Metrics Thresholds	-thresh

Compiler Personality

个性设置	选项
System Headers	
System Header Includes	-si
Suppress Output	-q
System Macros	

System Macro Defines	-sd
Data Types	
Intrinsic Types	
size_t	-it size_t
ptrdiff_t	-it prtdiff_t
wchar_t	-it wchar_t
Type Size and Alignment	
Size	-s
Alignment	-a
Compatibility	
Parsing Extensions and Language Compliance	
Lookup in Dependent Base Classes	-sdep
Propagate Template Parameters	-ptp
Delay parsing of function template bodies	-dpft
Introduce Friend Names	-ifn
Template argument deduction succeeds where deduced types differ by const/volatile	-accd
Allow temporaries to be bound to non-const references	-atncrb
Treat std as a namespace alias to the global scope	-sig
Use non-C99 literal promotion rules	-duc99dilt
Always Use Direct Initialization	-audi
Allow Null Pointer Conversions for non-type Arguments	-atanpc
Allow Implicit Conversion from Member to Pointer-to-Member	-aicpm
Maximum recursive template nesting path	-mrtnd
Preprocessor	
Additional Include File	-fi
Ignore whitespace between '\' and new line	-sl

2 配置选项

大多数配置选项可以简写，在下面的描述中给出了它们的全称和简写形式。配置选项的名字不是大小写敏感的。例如：

-quiet
-QUIET
-q

是等同的。

下面的描述中也给出了任何参数的语法，以及选项在 GUI 中设置的上下文环境。GUI 的语法是相同的，除了没有包含的选项名字。

-a, -align

语法：	-align [*type*]=[*bytes*]
默认：	见下表
程序：	qacpp.exe
GUI：	**Compiler Personality->Data Types->Type Size and Alignment->Alignment**
功能：	决定每个 C++语言基本类型的数据排列

数据类型的排列受到硬件和编译器的限制。比如，你的环境可能会把 char 类型分配到任何内存地址，而对 int 类型的排列需要 4 字节的界限。这样的限制加强了对指针类型转换的约束。例如，把 int*类型强制转换为 char*类型是有效的，但反过来却是无效的。QA C++对危险的类型转换产生警告。

在 GUI 中，数据类型的排列由滚动条和编辑框设置。

QA C++还使用类型的排列值计算结构的大小。排列通常跟数据类型的大小相关，见-size 选项。

类型	大小（位数）		规则	默认排列（字节数）
	最小值	默认值		
char	8	8	<=short	1
short	16	16	<=int	2
int	16	32	<=long, <=8bytes	4

long	32	32	<=long long <=16bytes	4
long long	32	64	<=16bytes	8
float	32	32	<=16bytes	4
double	64	64	<=16bytes	8
ldouble	64	64	<=16bytes	8
codeptr		32	如 int(*)()	4
dataptr		32	如 char*	4

-accd, -allowconstconvdeduction

语法：	-allowconstconvdeduction {+|-}
默认：	-allowconstconvdeduction-
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Template argument deduction succeeds where deduced types differ by const/volatile**
功能：	使用-allowconstconvdeduction+，那么在进行模板实参推演（template argument deduction）时允许被推演的参数类型不带有 const 或 volatile 限定。根据 ISO C++ 标准，当同一个模板实参被推演成两个不同的函数参数时，模板实参推演就是失败的。

-acipm, -allowimplicitconvptrmbr

语法：	-allowimplicitconvptrmbr {+|-}
默认：	- allowimplicitconvptrmbr -
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Allow Implicit Conversion from Member to Pointer-to-Member**
功能：	使用- allowimplicitconvptrmbr+，QA C++允许从成员函数到成员函数指针的非法转换。

-atanpc, -allowtmplargnullptrconv

语法：	-allowtmplargnullptrconv {+|-}

默认：	- allowtmplargnullptrconv-
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Allow Null Pointer Conversions for Non-type Arguments**
功能：	使用- allowtmplargnullptrconv+，QA C++允许从实参(arguments)的空指针到具有指针类型的非类型模板参数（non-type template parameters）的非法转换。

-atncrb, -allowtempnonconstrefbind

语法：	-allowtempnonconstrefbind{+|-}
默认：	- allowtempnonconstrefbind -
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Allow temporaries bo be bound to non-const references**
功能：	使用- allowtempnonconstrefbind +，允许把临时变量绑定到非 const 引用。根据 ISO C++标准，临时变量只能绑定到一个 const 引用。

-audi, -alwaysusedirectinitializer

语法：	-alwaysusedirectinitializer{+|-}
默认：	- alwaysusedirectinitializer -
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Always Use Direct Initialization**
功能：	使用- alwaysusedirectinitializer +，QA C++认为对声明“T x = e”的初始化始终是直接进行的，而不是采用拷贝初始化的方法。

-cmaf, -crossmoduleanalysisfile

语法：	-cmaf *filename*
默认：	无
程序：	qacpp.exe、errdsp.exe、pal.exe
GUI：	**N/A**
功能：	对产生型的工具而言，如 qacpp.exe 和 pal.exe，该选项指定一个输出文件保存基于工程的 err 和 met 信息；对显示型的工具而言，它指定这些信息的位置。

	在 GUI 中操作时，-cmaf 入口从根文件夹的输出路径和根文件夹的名字创建。

-d, -define

语法：	-define *ident*[=[*value*]]
默认：	无
程序：	qacpp.exe
GUI：	**Analyser Personality->Project Macros**
功能：	定义 QA C++分析时会遇到的宏。

源代码的编译通常由编译器提供的宏控制，这里有三种定义宏的方式：

配置选项	等价的源代码语句
-d *ident*	#define ident 1
-d *ident=value*	#define ident value
-d *ident=*	#define ident

编译器供应商为了实现语言扩展经常引入新的关键字。针对这些扩展，可以使用**_ignore** 宏来指导 QA C++忽略某个关键字或某段代码。具体方式有以下四种：

-d *identifier*=_ignore　　忽略指定的 identifier 及其后面的符号

-d *identifier*=_ignore(n)　　忽略指定的 identifier 及其后面的 n 个符号。这种情况只适用于确切知道要忽略多少个字符的情况

-d *identifier*=_ignore_paren　　忽略指定的 identifier 及其后面的代码块。代码块可以是任意长度，是由(…)、[…]或{…}括起来的

-d *identifier*=_ignore_semi　　忽略指定的 identifier 直到下一个分号（包括分号）

ignore 操作在其它任何宏之前进行预处理。

注：通过配置系统传递-defne 参数，总是会剥离引号，因此不能以这种方式定义带引号的字符串。解决该问题的方法是通过函数宏，如：

```
-d STR(x)=#x
-d STRNG=STR(abcd)
```

-dpft, -delayparsefntemplates

语法：	-delayparsefntemplate{+|-}

默认：	- delayparsefntemplate-
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Delay parsing of function template bodies**
功能：	使用-delayparsefntemplate+，将延迟对函数模板的解析，直到它们被实例化的时候。在分析 MFC 或 ATL 源文件时需要它。

-duc99dilt, -DontUseC99DecIntLitTypes

语法：	-DontUseC99DecIntLitTypes{+|-}
默认：	- DontUseC99DecIntLitTypes +
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Use non-C99 Literal Promotion Rules**
功能：	使用-DontUseC99DecIntLitTypes-，QA C++接受当前 C++编译器对大型无后缀的十进制数的类型处理方式。特别地，根据 C99 标准，大于 LONG_MAX（通常是 2^31）的数据具有“long long”类型。然而一些 C++编译器在这种情况下会把数据视为“unsigned long long”类型。这会影响到函数的重载解析（function overload resolution）和模板实例化。

代码示例：

```
void foo (int i);
void foo (unsigned long ul);
void test_duc99dilt_OFF ()
{
    //
    // qac++ uses long long as the type of literal in accordance
    // with the C99 standard
    //
    // ambiguous: no overload with an argument long long
    // and the conversions to unsigned long and int are equal
    foo (4294967295);  // ambiguous
}

void test_duc99dilt_ON ()
{
    //
    // qac++ uses unsigned long as the type for the literal.
    //
    // therefore foo(unsigned long) is chosen from the overload set
```

```
    foo (4294967295);  // calls foo(unsigned long)
}
```

-e, -echo

语法：	-echo *text*
默认：	无
程序：	qacpp.exe、errdsp.exe、prjdsp.exe
GUI：	**N/A – 忽略控制台输出**
功能：	指定回写给控制台的 *text*。该选项用于显示正在引用的配置文件。

-emhm, -expandmultihomedmessages

语法：	-expandmultihomedmessages *message_number_list*
默认：	无
程序：	errdsp.exe、prjdsp.exe
GUI：	**Message Personality->Advanced Settings->Message Expansion List**
功能：	展开所选的消息集，消息的实例就会出现在每一个子消息列表出现的地方，而不是只在基本位置上产生单一的消息。对这些消息的每个实例来说，它承载了一个子消息列表以及原始消息中所有其它的子消息。 该选项的目的是直接在注释源代码中高亮显示同源文件相关的问题。 分析器没有关于它的默认选项，在 GUI 中打开该选项时的候选消息是： -emhm 1500, 1501, 1506, 1507, 1508, 1509, 1510, 1512, 1513, 1515, 1516, 1517, 1520, 1525, 1526, 1527, 1528, 1529

-en, -encoding

语法：	-encoding {*ASCII, EUC, NEWJ, OLDJ, NECJ, SJ*}
默认：	-encoding *ASCII*
程序：	qacpp.exe、errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**Analyser Personality->Analysis Processing->Encoding**
功能：	指定字符集。该选项只应用在宽字符格式的环境中。

-fi, -forceinclude

语法：	-forceinclude *filename*
默认：	无
程序：	qacpp.exe
GUI：	**Compiler Personality->Preprocessor->Additional Include File**
功能：	使得文件 *filename* 显式地包含在每个文件的头部。这种强制包含的方法有时用于定义宏和其它工程设置。

-file

语法：	-file *filename*
默认：	无
程序：	errdsp.exe、prjdsp.exe
GUI：	**N/A**
功能：	产生输出文件 *filename*。如果没有给出文件的全路径，该文件创建在环境变量 QACPPOUTPATH 或-op 选项指定的路径中。

-forget, forgetall

语法：	-forgetall *option*
默认：	无
程序：	qacpp.exe、errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**N/A**
功能：	复位先前的配置 *option*。 某些选项在使用时是有累积效果的，它们在之前设置的基础上增加而不是覆盖。见-i、-d、-n、-o、-su、-q 和-wc。

例如，

qacpp –forget i –iC:\mypath source.cpp

该命令使分析器忽略先前设置的包含路径，只使用当前的 C:\mypath 路径。

-format

语法：	-format *string*
默认：	无

程序：	errdsp.exe、viewer.exe
GUI：	**Message Personality->Display->Message Format**
功能：	决定警告消息在注释源代码中显式的格式。在使用-format 选项时，将忽略-ref 和-text 选项，同时限制使用-line 选项。 见**“QA C++用户手册”第 10 章之格式化消息输出**。

如果没有指定-format 选项，则使用默认的格式。

对 Message Browser（viewer.exe），默认为：

%?u==0%(%?C%(%^%)%)%?u==0%(%?h%(Err%:Msg%)%:-->%) (%g:%N) %R(%u,)%t

对 errdsp.exe，默认格式依赖于其它选项：

```
format = MessageOnly ? “” : “%?C%(%^%)”;
if (MessageOnly || Line)
{
   format += “%F(%l, %c) : “;
}
format += TextOnly ? “” : “%?u==0%(%?h%(Err%:Msg%)%:-->%) (%g:%N)”;
format += “%R(%u, )%t”;
format += Reference ? “%?v%(\n%v%)” : “”;
```

-h, -help

语法：	-help
默认：	无
程序：	qacpp.exe、errdsp.exe、prjdsp.exe
GUI：	**N/A**
功能：	显式所有选项的帮助信息

-hdr, -hdrsuppress

语法：	-hdrsuppress {+|-}
默认：	-hdrsuppress-
程序：	errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**Message Personality->Display->Display Header Warnings**
功能：	使用-hdrsuppress+时，头文件中检查出的所有警告被抑制，除了第 9 级以外（不

	能被抑制）。 默认设置包含所有来自头文件分析的消息（没有被-n、-o、-q 或其它诊断性抑制手段关闭的消息）。在注释源文件输出中，它们紧跟在#include 之后出现。

经常会有大量的警告消息产生自头文件。它们可能是库文件或工程的头文件，产生的警告不是很重要。该选项允许你抑制这些消息在注释源文件中的显示。

注：-q 选项也抑制头文件的警告消息，它与-hdr 的区别在于，它不仅抑制消息的显示，也不在.err 文件中产生消息。-q 选项可以有选择地用在特定的路径或文件中。

-hm, -hiddenmessage

语法：	-hiddenmessage {+|-}
默认：	-hiddenmessage-
程序：	errwrt.exe
GUI：	**N/A**
功能：	-hiddenmessage+在错误的记录上标记被抑制的诊断消息。

它相当于在#pragma PRQA_MESSAGES_OFF 的控制下产生的消息的行为。

注：隐藏的消息可以由带有-hw+选项的实用程序显示。

消息抑制系统不使用这种技术标记被抑制的消息，它只由 PRQA_MESSAGES_OFF 替代使用。

-hmw, -hidemacrowarnings

语法：	-hidemacrowarnings {+|-}
默认：	- hidemacrowarnings -
程序：	qacpp.exe
GUI：	**Analyser Personality->Analysis Processing->Hide Macro Warnings**
功能：	- hidemacrowarnings +将抑制代码中由宏的扩展产生的警告。这些消息可以稍后用-hiddenwarnings+来显示。

-html

语法：	-html {+|-}
默认：	-html-
程序：	errdsp.exe、prjdsp.exe

GUI：	**Options->Annotated Source->HTML Annotated Source**
功能：	默认情况下，注释源代码以文本方式（-html-）创建。设置为-html+时，则以 HTML 格式创建。这样做是使得 **Warning Listing** 报告可以链接到单独的注释源文件输出。 GUI 的菜单可以按要求产生各种格式的注释源文件输出。

-hw, -hiddenwarnings

语法：	-hiddenwarnings {+|-}
默认：	-hiddenwarnings-
程序：	errdsp.exe、prjdsp.exe、viewer.exe、errsum.exe
GUI：	**Message Personality->Display->Show Suppressed Warnings**
功能：	使用-hiddenwarnings+时，使用消息抑制技术抑制的警告消息被显示。见**“QA C++用户手册”第 3 章之消息的抑制**。

消息诊断可以有多种方式抑制（代码注释中的抑制指令、-hiddenmessage+、-hidemacrowarnings+、或#pragma 指令）。设置该选项会重新打开所有被抑制的消息。Message Browser 能够交互地打开这些诊断。

注：如果相关消息号被-n 或-o 抑制，则不能被显示。

-i, -include

语法：	-i *path*
默认：	无
程序：	qacpp.exe
GUI：	**Analyser Personality->Project Headers->Header Includes**
功能：	指定头文件的搜索路径

-ifn, -introducefriendnames

语法：	-introducefriendnames{+|-}
默认：	- introducefriendnames-
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Introduce Friend Names**

功能：	-introducefriendnames+把友元的名字引入到最内层嵌套的名字空间，这样，友元的名字即使没有做过事先声明，它们在名字搜索过程中也是可见的。根据 ISO C++标准，友元名字在使用前必须先经声明。如果没有声明，则只能通过实参依赖（argument dependent）的搜索（lookup）找到友元函数的声明。

-it, -intrinsictype

语法：	-intrinsictype {*size_t*|*ptrdiff_t*|*wchar_t*}=*datatype*
默认：	-intrinsictype “size_t=unsigned int” -intrinsictype “ptrdiff_t=long” -intrinsictype “wchar_t=unsigned char”
程序：	qacpp.exe
GUI：	**Compiler Personality->Data Types->Intrinsic Types**
功能：	决定 sizeof（size_t）操作符、指针减法（ptrdiff_t）和宽字符类型（wchar_t）的基本类型。 如果这些设置与源代码中的 typedef 不匹配时，QA C++会产生第 9 级消息。见**“QA C++用户手册”第 3 章之设置数据类型**。

-lf, -layoutfile

语法：	-layoutfile *file*
默认：	无
程序：	qacpp.exe
GUI：	**Analyser Personality->Style->Layout Configuration File**
功能：	指定配置代码布局的文件的所在位置

在编程过程中保持缩进的一致性是很好的编程实践，同时它也通常是编程标准或指南的一部分，因为它提供了良好的可读性和逻辑性。QA C++通过检查花括号风格、缩进以及空格等等的一致性，使代码保持一致的布局。如果布局风格前后不同，则产生警告。

为了最好匹配源代码的布局风格，该选项允许你指定一个布局配置文件。QA C++预装了三个这样的配置文件，它们是 exdented.layout、indented.layout 和 knr.layout，分别实施下面的花括号风格：

Exdented

```
if ( )
{
```

```
        …
    }
```

Indented

```
    if ( )
        {
        …
        }
```

K&R (Kernighan & Ritchie)

```
    if ( ) {
        …
    }
```

另外还有其他三个文件， exdented-spacing.layout 、 indented-spacing.layout 和 knr-spacing.layout。除了上述的花括号风格外，这三个文件还包含了关于空格“容限”的规则，于是代码中的合理数量内的空格都是可接受的。这些文件都位于 personalities 路径下。

-lih, -layoutignoreheaders

语法：	-layoutignoreheaders{+|-}
默认：	-layoutignoreheaders-
程序：	qacpp.exe
GUI：	**Analyser Personality->Style->Ignore code layout in headers**
功能：	在执行布局分析时，-layoutignoreheaders+不会检查头文件的代码。如果头文件是经过完整测试的，则可以不必检查其代码布局。

-liw, -layoutindentwidth

语法：	-layoutindentwidth *value*
默认：	- layoutindentwidth 0
程序：	qacpp.exe
GUI：	**Analyser Personality->Style->Code Indent Level**
功能：	设置代码缩进时的空格数量。 在进行布局分析时，如果代码缩进的空格数量与此设置不同，QA C++就给出警告。如果没有设置该选项，或者设置为 0，则产生 1982 号警告消息。

-luhi, -layoutusehangingindent

语法：	-layoutusehangingindent {+|-}
默认：	-layoutusehangingindent-
程序：	qacpp.exe
GUI：	**Analyser Personality->Style->Labels create hanging indent**
功能：	在进行布局检查时，-layoutusehangingindent+要求标签后面的语句要有缩进。关于标签的布局风格一般是，以标签为基准，缩进其后面的所有语句。

如果设置了该选项，而代码中标签后面的语句没有以标签为基准进行缩进，QA C++就给出警告。如果没有设置该选项，QA C++只强制缩进紧跟在标签后面的语句，相对于 switch 语句。

以下例来说，应该设置-luhi，这样 QA C++就能识别标签后语句的缩进是相对于该标签的。而如果没有设置，缩进则应当是相对于 switch 语句。

```
switch (value)
{
    case 1:             // the switch label
        statement;
        break;
}
```

而在下例中，不需要设置-luhi，因为语句的缩进不是相对于标签的。

```
switch (value)
{
case 1:
    statement;
    break;          // correctly indented with respect to switch
}
```

-l, -line

语法：	-line {+|-}
默认：	-line-
程序：	errdsp.exe、viewer.exe
GUI：	**Message Personality->Display->Display Line Numbers**
功能：	使用-line+，那么文件名字、行号和列号同警告消息一起显示。 如果指定了-format 选项，则忽略-line 选项。

	Message Browser 使用-line+，把行号显示在源代码旁边。

-list

语法：	-list filelist.lst
默认：	无
程序：	prjdsp.exe、viewer.exe
GUI：	**N/A**
功能：	为实用程序和外部过程，包括 **Message Browser** 和 **Warning Listing** 报告，传递文件列表。

在 GUI 中，filelist.lst 是为 **Message Browser** 和 **Warning Listing** 报告自动生成的，由所选文件、或包含文件和子文件夹的文件夹组成。在 **Custom Reports** 和 **CMA** 分析中如果使用了%L 扩展关键字，它也会自动生成。对每个文件夹来说，它包含文件夹的路径，后跟源文件列表。典型如下：

-op “<output_path1>”
“<source_path_and_file1>”
“<source_path_and_file2>”
“<source_path_and_file3>”
“<source_path_and_file4>”
-op “<output_path2>”
“<source_path_and_file5>”
“<source_path_and_file6>”
“<source_path_and_file7>”

-outputpath 选项用于定位每个源文件的.err 和.met 文件。

-m, -messageonly

语法：	-messageonly {+|-}
默认：	-messageonly-
程序：	errdsp.exe
GUI：	**Message Personality->Display->Messages Only**
功能：	-messageonly+使得注释源代码中只显示警告消息，虽然记录了每条消息发生的位置，但并不显示源代码行。 如果指定了-format 选项，所有消息的格式遵循-format 的指定。

-max, -maxcount

语法：	-maxcount *value*
默认：	-maxcount 0
程序：	errdsp.exe、prjdsp.exe
GUI：	**Message Personality->Display->Maximum Occurrences of each Message**
功能：	给出每条警告消息出现的次数，默认的 0 表示显示所有。

-met, -metrics

语法：	-metrics {+|-}
默认：	-metrics+
程序：	qacpp.exe、errdsp.exe
GUI：	**Analyser Personality->Metrics->Display Metrics**
功能：	该选项用于两种不同的目的： 分析时（qacpp.exe）， 1. -metrics+使得分析数据写入度量输出文件 2. -metrics-使得产生度量输出文件，但是为空 显示注释源代码时（errdsp.exe）， 1. -summary-和-metrics+增加一个表，显示每个警告消息数量和密度（即每行代码出现多少次），以警告级别分组 2. -summary+和-metrics+显示每条消息出现的次数

-mrtnd, -maxrecursivetemplatenestingdepth

语法：	-maxrecursivetemplatenestingdepth *value*
默认：	-maxrecursivetemplatenestingdepth 200
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Maximum Recursive Template Nesting Depth**
功能：	指出 QA C++能够允许的模板递归实例化的最大深度。

代码实例：

```
template <typename T>
```

```
struct A
{
   enum { value = T::non_existing};
};

template <int I>
struct B
{
   static const int value = B<I-1>::value;
};

template <>
struct B1<1>
{
};

void foo ()
{
   B <A<int>::value> b;
}
```

由于在 T 中不存在成员 non_existing，所以不能确定 value 的值，而使用 0 值。然而，对 B 来说这是有问题的，因为停止条件为 1，所以当以 0 作为 A::value 的恢复时，将产生无穷递归。该选项限制了递归的程度。

-n, -no, -nomsg

语法：	-nomsg *message_number_list*
默认：	无
程序：	errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**Message Personality->Advanced Settings->Save Message Settings By->Suppressed Messages**
功能：	抑制指定消息的显示

消息号可以单独指定，或者指定为一个列表或范围：

```
-nomsg 2356
-nomsg 1234, 1236, 1240
-nomsg 2111, 2200-2300
```

-nosort

语法：	-nosort {+|-}

默认：	-nosort-
程序：	errdsp.exe
GUI：	**N/A**
功能：	控制消息的排序。 以-messageonly+产生注释源代码时，-nosort+按照检测到消息的次序进行显示。在 **Analysis Status** 窗口中的错误列表中的消息按照这种方式排序。

-nrf, -namerulefile

语法：	-namerulefile *rulefile*
默认：	无
程序：	pal.exe
GUI：	**Message Personality->Advanced Settings->Secondary Analysis Setup**
功能：	指定后续分析中含有命名规范检查规则的文件。见第 9 章。

-o, -only

语法：	-only *message_number_list*
默认：	无
程序：	errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**Message Personality->Advanced Settings->Save Message Settings By->Selected Messages**
功能：	只显示指定的消息。 消息号可以单独指定，也可以指定为列表或范围： -o 2356 -o 1234, 1236, 1240 -o 2111, 2200-2300

-one, -onelineonly

语法：	-onelineonly {+|-}
默认：	-onelineonly-
程序：	errdsp.exe

GUI：	**Message Personality->Display->Source Lines with Warnings**
功能：	指定为-onelineonly+时，注释源代码中只包含具有警告消息的代码行。

-op, -outputpath

语法：	-outputpath *path*
默认：	%OUTPUTPATH%，或者没有
程序：	qacpp.exe、errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**Folder Parameters->Output File Path**
功能：	指定输出文件产生的路径。 分析过程中，输出路径决定了在哪里产生所有的输出文件。在显示 **Message Browser**、注释源代码或产生 **Warning Listing** 报告时，输出路径决定了.err 文件的所在位置。

注：在命令行应用中，输出路径通常由环境变量 QACPPOUTPATH 决定，除非使用了这个选项。

-ppf, -ppfilename

语法：	-ppfilename {+|-}
默认：	-ppfilename-
程序：	qacpp.exe
GUI：	**Analyser Personality->Analysis Processing->Include filename and line numbers in preprocessed listing**
功能：	设置为-ppfilename+时，预处理文件中会加入注释，指示代码来自的源文件和行号。该选项只和-pplist 一起使用。

-ppl, -pplist

语法：	-pplist {+|-}
默认：	-pplist-
程序：	qacpp.exe
GUI：	**Analyser Personality->Analysis Processing->Preprocessed Output**
功能：	设置为-pplist+时，在输出路径中产生以.i 为扩展名的预处理文件。

-ppo, -pponly

语法：	-pponly {+|-}
默认：	-pponly -
程序：	qacpp.exe
GUI：	**Analyser Personality->Analysis Processing->Only Perform Preprocessing Analysis**
功能：	-pponly+使得分析在预处理之后就终止。此时的输出文件（.err 和.met）只包含预处理阶段的信息：文件度量、隐式声明、宏以及其他预处理消息。该选项因此只用于调试的目的，特别是为了确定预处理已经完成而没有进行 C++方面的解析和分析。

-ptp, -propagatetemplateparameters

语法：	-propagatetemplateparameters {+|-}
默认：	-propagatetemplateparameters-
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Propagate Template Parameters**
功能：	设置-propagatetemplateparameters+，初始模板定义中的模板参数名字可以用在模板的特例定义中。

-q, -quiet

语法：	-quiet {*filename*|*path*}
默认：	无
程序：	qacpp.exe
GUI：	**Compiler Personality->System Headers->Suppress Output** **Analyser Personality->Project Headers->Suppress Output**
功能：	对指定头文件或指定路径中的头文件，抑制其输出到.err 文件和.met 文件。在抑制警告消息的同时也显著地减小了输出文件的大小。见**“QA C++用户手册”第 4 章之抑制头文件的分析输出**。

-ref, -references

语法：	-references {+|-}

默认：	-references-
程序：	errdsp.exe、prjdsp.exe
GUI：	**N/A**
功能：	设置为-references+时，消息文件中的详细引用文本被显示出来。消息可以包含附加的解释性文本或 ISO 标准的引用，而用户消息文件可以包含本地编程标准的引用。 如果指定了-format，-references 选项将不起作用。所有消息的格式都由-format 字符串控制，其中的%v 限定符用于显示详细文本或引用文本。

-rem, -remark

语法：	-remark *comment*
默认：	无
程序：	qacpp.exe、errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**N/A**
功能：	-remark 在配置文件中写入注释。另外一种方法是，在配置文件中以星号开头写入注释。

-s, -size

语法：	-size [*type*]=[*bits*]
默认：	见-option 选项中的表
程序：	qacpp.exe
GUI：	**Compiler Personality->Data Types->Type Size and Alignment->Size**
功能：	设置编译环境中数据类型的大小（以位为单位）。

-sa, -showalias

语法：	-showalias {+|-}
默认：	-showalias+
程序：	qacpp.exe
GUI：	**Compiler Personality->Data Types->Miscellaneous**
功能：	决定类型名字显示在注释源代码中的方式。设置为-showalias+，任何经由

	typedef 声明的类型名称都将取代相应的 C++基本类型参与显示。

-sd, -systemdefine

语法：	-systemdefine *ident*[=[*value*]]
默认：	无
程序：	qacpp.exe
GUI：	**Compiler Personality->System Macros**
功能：	设置系统宏

-sdep, -searchdependentbaseclasses

语法：	-searchdependentbaseclasses{+|-}
默认：	-searchdependentbaseclasses-
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Lookup in Dependent Base Classes**
功能：	-searchdependentbaseclasses+使能了在依赖基类（dependent base class）中对名字的搜索（lookup）。某些编译器下，在类模板中搜索到的名字可以在依赖类型（dependent type）的基类中找到。这样的搜索不是标准行为，但如果能得到它们的定义，QA C++可以在相关的初始类模板中搜索。

-set, -settings

语法：	-settings {+|-}
默认：	-settings-
程序：	qacpp.exe、errdsp.exe、prjdsp.exe
GUI：	**N/A**
功能：	设置为-settings+时，所有配置选项的当前值列出在标准输出中。

-si, -systeminclude

语法：	-systeminclude *path*
默认：	无
程序：	qacpp.exe
GUI：	**Compiler Personality->System Headers->System Header Includes**

功能：	指定系统头文件的搜索路径。

-sig, -stdisglobal

语法：	-stdisglobal {+|-}
默认：	-stdisglobal-
程序：	qacpp.exe
GUI：	**Compiler Personality->Extensions->Treat std as a namespace alias to the global space**
功能：	设置为-stdisglobal+时，名字空间 std 中的声明被当做是如同在全局域中做的声明，而且全局域中的声明也被当做是如同在名字空间 std 中做的声明。

对限定名字的搜索规则是，如果未能在限定域中查找到，那么就是一个错误。在名字空间 std 和全局域上对该规则的放宽，是旧版本 gcc 的隐式行为。它允许用户代码在操作 ISO C++标准库时可以不用 std::前缀。

例如，在-stdisglobal+情况下，

```
// header code
void foo ();
namespace std
{
    void bar ();
}

// user code
int main ()
{
    std::foo ();    // OK，尽管 foo () 没有声明在 std 中
    bar ();         // OK，尽管 bar () 没有声明在全局域中
}
```

-sl, -slashwhite

语法：	-slashwhite {+|-}
默认：	-slashwhite-
程序：	qacpp.exe
GUI：	**Compiler Personality->Preprocessor->Miscellaneous**
功能：	-slashwhite+使得在分析中忽略\和行尾之间的空格。

一些源代码使用续行时在\和本行结尾之间含有空格。当源文件在不同的操作系统间移植时会出现这种情况。如果不打开这个选项，QA C++就会产生第 9 级错误。

-su, -suppresslvl

语法：	-suppresslvl *message_level*
默认：	无
程序：	errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**Message Personality->Warning Messages**
功能：	抑制指定级别的所有消息。欲抑制多个级别，可用短划线选择一个范围，如-su 4-7，或者使用逗号分隔的列表，如-su 1, 2, 5。

-summ, -summary

语法：	-summary {+|-}
默认：	-summary-
程序：	errdsp.exe
GUI：	**Message Personality->Display->Create Analysis Summary**
功能：	-summary+会产生一个总结，替代注释源代码，显示每个级别检测到的消息数量。 如果设置了-metrics 选项，总结报告会按级别列出每条警告消息以及它们出现的次数。

-t, -tab, -tabstop

语法：	-tabstop *columns*
默认：	-tabstop 8
程序：	qacpp.exe
GUI：	**Analyser Personality->Style->Source Code Tab Spacing**
功能：	指定 Tab 字符占有的空格数。Tab 空格数与缩进的深度无关（见-layoutindentwidth 选项）。

-text, -textonly

语法：	-textonly {+|-}

默认：	-textonly-
程序：	errdsp.exe、prjdsp.exe
GUI：	**N/A**
功能：	-textonly+会抑制通常出现在消息文本之前的消息级别号和消息号。 如果指定了-format，-textonly 将不起作用。所有消息的格式由-format 决定。

-thresh, -threshold

语法：	-threshold *metric*{<|<=|=|>|>=}*value*{:*msg_no*}
默认：	无
程序：	qacpp.exe
GUI：	**Analyser Personality->Metrics->Metric Thresholds**
功能：	设置该选项之后，如果指定度量 *metric* 的门限条件（来自{<|<=|=|>|>=}）超出时，则产生 *msg_no* 号消息。度量由其名字指定，度量名出现在.met 文件中。见**第 6 章之度量的记录**。这里没有默认值，如果省略了*:msg_no*，则默认地使用 4800 号消息。

-total

语法：	-total {+|-}
默认：	-total-
程序：	errdsp.exe
GUI：	**N/A**
功能：	在多个文件上运行 errdsp.exe，同时使用了-summary 或-tablesummary 选项时，-total+会产生一个额外的记录，包含所有文件的累积结果。

-tsumm, -tablesummary

语法：	-tablesummary {+|-}
默认：	-tablesummary-
程序：	errdsp.exe
GUI：	**N/A**
功能：	设置为-tablesummary-时，产生的总结报告中显示每个级别出现的消息数量。这

	种格式代替了注释源代码的显示。它等同于-summary 选项，只是消息级别由 tab 分隔，而不是以冒号分隔。 如果设置了-metrics+，总结报告按消息级别的组织方式，列出每条警告及其出现的次数。

-up, -usrpath

语法：	-usrpath [*path*]
默认：	%QACPPBIN%
程序：	errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**Message Personality->Advanced Settings->User Message File**
功能：	指定用户消息文件的路径。 在 GUI 中，只有当你在其它的路径而不是 bin 路径中指定用户消息文件时，才设置该选项。 如果没有设置该选项，观察工具使用 QACPPBIN 定义的路径。 QA C++还在指定的路径中搜索消息帮助文件，这是在搜索由 QACPPHELPFILES 定义的路径之前完成的。在这两种搜索情况下，都会添加一个\messages 子路径。

-usr, -usrfile

语法：	-usrfile *extension*
默认：	无
程序：	errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**Message Personality->Advanced Settings->User Message File**
功能：	指定用户消息文件的扩展名

QA C++的消息都包含在 qacpp.msg 文件中。但可以通过用户消息文件指定新的消息或更改已有消息文本，用户消息文件的名称格式必须要采用 *qac.usr.extension* 的形式，它位于用-usrpath 或 QACPPBIN 指定的路径中。

在 GUI 中，点击 **Refresh Message View**，用户消息文件中的消息和消息文件中的消息一同显示在消息列表中。

-ver, -version

语法：	-version {+|-}

默认：	-version-
程序：	qacpp.exe、errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**N/A**
功能：	-version+显示执行程序的版本号

-via

语法：	-via *file*
默认：	无
程序：	qacpp.exe、errdsp.exe、prjdsp.exe、viewer.exe
GUI：	**N/A**
功能：	指定包含配置选项的文件

-wc, -warncall

语法：	-warncall [*function*[=*msg_no*]]
默认：	无
程序：	qacpp.exe
GUI：	**Analyser Personality->Warning Calls**
功能：	如果 QA C++在分析过程中遇到 *function* 调用，则显示 *msg_no* 号消息。

如果没有声明消息号，则显示如下消息：

```
Msg (4:3999) Function 'func_name' is called.
```

3 函数结构的组成

函数结构图显示了函数的控制流结构。在结构中，判断分支（if、switch、以及循环的条件部分）以分叉的方式显示。其路径中的最右边部分，表示在什么地方控制回到主流中。循环结构显示的是个虚线的环。

函数结构图显示下列代码结构：

- 直接代码
- if
- if else
- switch
- while 循环
- for 循环
- 嵌套结构
- 循环中的 break
- 循环中的 return
- 循环中的 continue
- 不可达代码

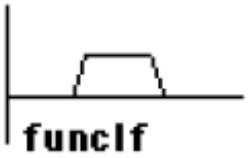
每个组成部分都是随着控制流在源代码中的流向而从左到右的。

直接代码


```
static int code = 0;
void funcStraight (void)
{
    code = 1;
}
```

x 轴表示该函数只有一条路径。

if




```
static int code = 0;
void funcIf (void)
{
    if (code > 0)
    {
        code = 1;
    }
}
```

函数有两条路径，第一条路径是行径 if 语句，显示为上升的线段；第二条路径由 x 轴表示，代表 if 条件为假的情况。

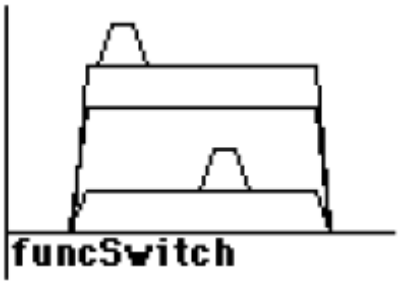
if-else


```
static int code = 0;
void funcIfElse (void)
{
    if (code > 0)
    {
        code = 3;
    }
    else
    {
        code = 4;
    }
}
```

函数有两条执行路径，第一条是 if 子语句，表示为上升的线段；第二条是 x 轴表示的 else 子语句。

该结构体要长于 if 结构体。如果 if-else 的两臂不是直接代码，那么 if 分支出现在上升线段的左边结尾处，else 分支出现在较低线段的右侧。

switch




```
static int code = 0;
void funcSwitch (void)
{
    switch (code)
    {
    case 1:
        if (code == 1)
        {
        /* block of code */
        }
        break;
    case 2:
        break;
    case 3:
        if (code == 3)
        {
        /* block of code */
        }
        break;
    default:
        break;
    }
}
```

在 switch 结构中，x 轴代表 default 分支，每个 case 分支由一条上升线段表示。两条简短的上升线段表示 case 1 和 case 3 中的 if 分支。

上图显示了 if 语句是如何交错绘制的，case 1 中的 if 显示在左边，而 case 3 中的 if 显示在右边。

while 循环


```
static int code = 0;
void funcWhile (void)
{
    while (code > 0)
    {
        --code;
    }
}
```

在 while 循环中，x 轴表示直接通过函数的路径，就好像 while 并没有执行一样。上升的实线显示 while 结构体的路径，虚线显示到 while 语句开始处的循环。

for 循环


```
static int code = 0;
void doSomethingWith (int);


void funcFor (void)
{
    for (int i = 0; i > code; ++i)
    {
        doSomethingWith (i);
    }
}
```

for 循环类似于 while 循环，因为 for 可以重写为 while 循环的形式。如上例可以重写为：

```
void funcFor (void)
{
    int i = 0;
    while (i > code)
    {
        ++i;
```

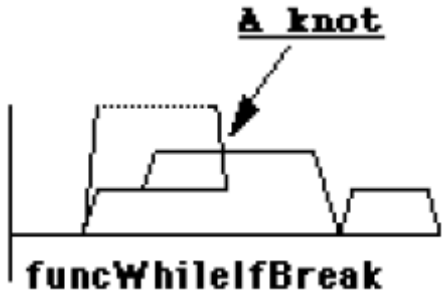
```
    }
}
```

嵌套结构


```
static int code = 0;
void funcWhileIfElse (void)
{
    while (code > 0)
    {
        if (code == 1)
        {
            code = 0;
        }
        else
        {
            --code;
        }
    }
}
```

这是 if-else 包含在 while 循环中的嵌套结构。第一条上升的实线代表 while 循环，而其内部的上升实线代表 if-else 结构。其它的函数结构可以类似地嵌套进来。

循环中的 break

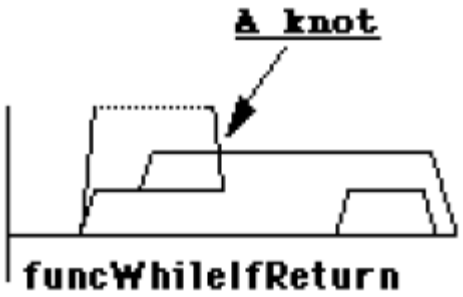



```
static int code = 0;
void funcWhileIfBreak (void)
{
    while (code > 0)
```

```
        {
            if (code == 3)
            {
                break;
            }
            --code;
        }
        if (code == 0)
        {
            code++;
        }
    }
```

break 跳到 while 语句的结尾，打一个结。打结是控制流穿越其它语句块的边界，表明非结构化的代码。在这个例子中，break 跳到 while 的结尾，执行程序的下一个部分 if。

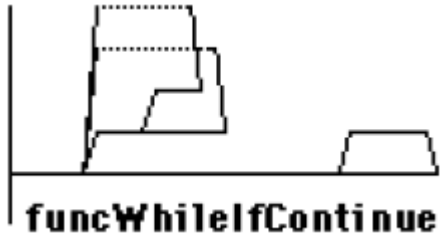
循环中的 return




```
static int code = 0;
void funcWhileIfReturn (void)
{
    while (code > 0)
    {
        if (code == 3)
        {
            return;
        }
        --code;
    }
    if (code == 0)
    {
        code++;
    }
}
```

return 语句导致程序跳到函数的结尾，它打断了控制流，在代码中打了一个结。

循环中的 continue




```
static int code = 0;
void funcWhileIfContinue (void)
{
    while (code > 0)
    {
        if (code == 3)
        {
            continue;
        }
        --code;
    }
    if (code == 0)
    {
        code++;
    }
}
```

continue 语句导致程序跳回到 while 循环的开始处，不打结。

不可达代码


```
void funcUnReach (int i)
{
    if (i)
    {
        i = 1;
    }
    goto Bad;
```

```
        if (i)
        {
            i = 2;
        }

        Bad:
        if (i)
        {
            i = 3;
        }
        return;
    }
```

图中红色的被抬升的段表示不可达的代码。

图中显示了不可达代码的结构和近似位置。它发生在第一个 if 之后和最后一个 if 之前。它在主结构之上的位置表示控制流错过了中间的 if 语句。

4 度量的计算

QA C++计算三组度量。函数度量是为所有具有完整定义的函数产生的，文件度量是为每个被分析的源文件产生的，类度量为每个类产生的。

函数度量

每个单独的函数都会产生下面的这些度量。这里按字母顺序对它们进行描述。

STCYC	圈复杂度
STGTO	goto 语句数量
STLIN	可维护代码行数
STMIF	控制结构中的最大嵌套层次
STPAR	函数参数个数
STPTH	静态路径数估计
STSUB	函数调用数
STXLN	可执行行数

STCYC 圈复杂度

圈复杂度计算为判断的数量加 1。

圈复杂度高时表明函数的模块化不充分，或者具有许多的逻辑判断。对软件度量的研究表明，圈复杂度大于 10 的函数存在复杂性问题。

McCabe[1]对这个问题做了重要的探讨，而且他也引入了这个度量。

例 1：

```
int divide (int x, int y)
{
   if (y != 0)                  // 1
   {
        return x / y;
   }
   else if (x == 0)             // 2
```

[1] *McCabe, T.J. (1976) A Complexity Measure, IEEE Transactions on Software Engineering, SE-2, pp. 308-320.*

```
    {
        return 1;
    }
    else
    {
        printf (stderr, 'div by zero\n');
        return 0;
    }
}
```

上面的代码中，因为函数做了两次判断，因此它的圈复杂度等于 3。注意，*正确缩进*的代码并非总是反映了代码的*嵌套结构*，特别是“else if”结构的使用，它增加了嵌套的层次，但通常不用额外的缩进来书写，因此嵌套不是明显可见的。

例 2：

```
void how_many (int n)
{
    switch (n)
    {
    case 0:    printf (stdout, "zero\n");          // 1
               break;
    case 1:    printf (stdout, "one\n");           // 2
               break;
    case 2:    printf (stdout, "two\n");           // 3
               break;
    default:   printf (stdout, "many\n");
               break;
    }
}
```

上面代码的圈复杂度等于 4，因为 switch 语句等价于一系列的逻辑判断。

STGTO　　goto 语句数量

某些 goto 的使用只是为了简化错误处理，然而应该尽量避免。

STLIN　　可维护代码行数

这是函数定义中（在花括号之间但不包含花括号）的代码行总数，包括空行和注释行。这是在原始代码上计算的。某些函数，如含有#include 的代码，或者含有带花括号的宏定义，是没有 STLIN 度量的。

下例函数的 STLIN 值为 5。

```
int test ()
```

```
{
   int x;                  // 1
   int y;                  // 2
                           // 3
   return (x + y );        // 4
   /* comment here */      // 5
}
```

较长的函数难以阅读，因为它们不能全部列在一页中。建议该值的上限为 200。

STMIF　　控制结构中的最大嵌套层次

测量源代码中最大的控制嵌套。

通过把嵌套拆分为独立的函数，可以把该度量的值降低。这种做法通过减少嵌套和平均圈复杂度，增加了代码的可读性。

下例代码的 STMIF 值为 3：

```
int divide (int x, int y)
{
   if (y != 0)                    // 1
   {
         return x / y;
   }
   else if (x == 0)               // 2
   {
         return 1;
   }
   else
   {
         printf (stderr, “div by zero\n”);
         while (x > 1)                        // 3
               printf (“x = %i”, x);
         retrun 0;
   }
}
```

STMIF 值的增加是随着 switch、do、while、if 和 for 语句的。值得注意的是，代码的嵌套层次并非总是能通过观察其缩进而肉眼可见的。特别地，“else if”结构增加了控制流的嵌套，但它的书写通常不增加额外的缩进。

STPAR　　函数参数个数

该度量计算函数参数列表中声明的参数个数。

STPTH　　静态路径数估计

它类似于Nejmeh[2]的NPATH统计，并给出了函数控制流中可能路径数的上限。它是函数中非循环（non-cyclic）执行路径的数目。

在相同的嵌套层次内，语句序列的 NPATH 值是每条语句和内嵌结构的 NPATH 值。

- NPATH（非控制语句序列）= 1
- NPATH（if）= NPATH（then 结构体）+ NPATH（else 结构体）
- NPATH（while）= NPATH（while 结构体）+ 1
- NPATH（do while）= NPATH（while 结构体）+ 1
- NPATH（for）= NPATH（for 结构体）+ 1
- NPATH（switch）= Sum（ NPATH（case 1 结构体） … NPATH（case n 结构体））

注：else 和 default 不管是否在代码中出现，都会计算在内。

在 switch 语句中，同一分支上的多个 case 选项只被计算一次。如：

```
swtich (n)
{
case 0: beak;          // NPATH of this branch is 1
case 1:
case 2: break;         // NPATH for case 1 & case 2, combined is 1
default: break;        // NPATH for this default is 1
}
```

如果函数中存在 goto 语句，则不能计算 NPATH。

下例代码的静态路径数为 26。

```
int n;
if (n)
{
} // block 1, paths 1
else if (n)
{
   if (n)
   {
   } // block 2, paths 1
   else
   {
```

[2] ***Nejmeh, B.A. (1988), NPATH: A Measure of Execution Path Complexity and its Applications, Comm ACM, 31, (2), p.188-200***

```
    } // block 3, paths 1
    // block 4, paths block2 + block3 = 2
    switch (n)
    {
    case 1: break;
    case 2: break;
    case 3: break;
    case 4: break;
    default: break;
    } // block 5, paths 5
} // block 6, paths block 4 * block 5 = 10/
else
{
    if (n)
    {
    } // block 7, paths 1
    else
    {
    } // block 8, paths 1
} // block 9, paths block 7 + block 8 = 2
// block 10, paths block1 + block6 + block 9 = 13
if (n)
{
} // block 11, paths 1
else
{
} // block 12, paths 1
// block 13, paths block 11 + block 12 = 2
// outer block, paths block 10 * block 13 = 26
```

条件为真的路径统计通常遵循不等式：圈复杂度 <= 真值路径统计 <= 静态路径统计。

STSUB 函数调用数

含有大量函数调用的函数难以理解，因为其功能体现在多个组件中。STSUB 的计算是基于函数调用，而非被调用的不同函数数量。

STSUB的值很大时，说明可能存在设计问题。见Brandl（1990）[3]关于设计复杂度的论述以及如何用调用树显示。

下例代码的 STSUB 值为 4：

```
extern dothis (int);
```

[3] ***Brandl, D.L., (1990) , Quality Measures in Design, ACM Sigsoft Software Engineering Notes, vol 15, 1.***

```
extern dothat (int);
extern dotheother (int);
void test ()
{
   int a, b;
   a = 1;
   b = 0;
   if (a == 1)
   {
        dothis (a);          // 1
   }
   else
   {
        dothat (a);          // 2
   }
   if (b == 1)
   {
        dothis (b);          // 3
   }
   else
   {
        dotheother (b);      // 4
   }
}
```

STXLN　　可执行行数

该度量是函数体中代码行的统计。注释、花括号以及所有声明标记都不作为代码标记。下例的 STXLN 值为 12：

```
void fn( int n )
{
   int x;
   int y = 1;                          // 1

   if ( x )                            // 2
   {
        x++;                           // 3
        for (;;)                       // 4
        /* Ignore comments */
        /* Ignore braces */
        {
             switch ( n )              // 5
```

```
                {
                case 1 : break;       // 6
                case 2 :              // 7
                case 3 :              // 8
                    break             // 9
                                ;     // 10
                }
            }
        }
        else                          // 11
        {
            y++;                      // 12
        }
    }
```

文件度量

文件度量是针对源代码的整体进行计算的。按字母顺序的完整列表如下：

STCDN	代码注释率
STOPN	Halstead 相异操作数个数
STOPT	Halstead 相异操作符个数
STTLN	预处理代码总行数
STTOT	记号出现总次数
STTPP	未预处理的源文件总行数
STVAR	标识符数量
STCCA	字符总数
STCCB	代码字符总数
STCCC	注释字符总数

STCDN 代码注释率

该度量定义为注释内的可见字符数被注释外的可见字符数除，忽略注释的定界符。字符串中的空格也作为可见字符。

该值太大说明注释太多，会造成模块难以阅读。该值太小则说明注释不充分。

下例代码中的注释包含 28 个可见字符，而代码包含 33 个可见字符，因此其注释率为

0.85。

```
int test ()
// This is a test
{
   int x;
   int y;
// This is another test
   return (x + y);
}
```

度量的计算：STCDN = STCCC / STCCB。

STOPN　　Halstead 相异操作数个数

文件中使用的相异操作数的数量。相异操作数定义为唯一的标识符及其文字的每次出现。

多数文字，除了 0 和 1，在程序中通常都是相异的。由于经常使用宏来表示固定的成功或失败的值（如 TRUE 和 FALSE），统计策略之间的差异是相当小的。下例代码的 STOPN 的值为 11：

```
extern int result;          // 1 -> result
static int other_result;    // 2 -> other_result

main()                      // 3 -> main
{
    int x;                  // 4 -> x
    int y;                  // 5 -> y
    int z;                  // 6 -> z

    x = 45;                 // 7 -> 45
    y = 45;                 // 8 -> 45
    z = 1;                  // 9 -> 1
    result = 1;             // 10 -> 1
    other_result = 0;       // 11 -> 0

    return ( x + other_result );
}
```

STOPT　　Halstead 相异操作符个数

该度量覆盖了源代码中不是由用户给出的任意标记，如关键字、操作符、标点符号等。STOPT 用于许多度量的计算。

下例代码的 STOPT 值为 11：

```
extern int result;          // 1, 2, 3 -> extern, int, ;
```

```
static int other_result;          // 4 -> static

main()                            // 5, 6 -> (, )
{                                 // 7 -> {
    int x;
    int y;
    int z;

    x = 45;                       // 8 -> =
    y = 45;
    z = 1;
    result = 1;
    other_result = 0;

    return ( x + other_result );  // 9, 10 -> return, +
}                                 // 11 -> }
```

STTLN 预处理代码总行数

该度量计算翻译单元在预处理之后的总行数。经过预处理的文件反应了头文件的预处理、预处理指令以及对注释的剥离。

STTOT 记号出现总次数

该度量是源文件中非相异标记的标记总数。下例代码的 STTOT 值为 19：

```
int test ()                       // 1, 2, 3, 4
{                                 // 5
    int x;                        // 6, 7, 8
    int y;                        // 9, 10, 11
    // excluded comment/
    return (x + y );              // 12, 13, 14, 15, 16, 17, 18
}                                 // 19
```

STTPP 未预处理的源文件总行数

预处理之前的源文件总行数。

STVAR 标识符数量

相异标识符总数。下例代码的 STVAR 值为 5：

```
int other_result;                 // 1
extern int result;                // 2
int test ()                       // 3
{
```

```
    int x;                          // 4
    int y;                          // 5

    return ( x + y + other_result);
}
```

STCCA　字符总数

该度量统计文件中的字符总数。这里只计算可见的字符，除了字符串或字符数组。因为在后者的情况下，所有字符都计算在内，同时 tab 只作为一个字符计算。在计算注释的字符时，注释的分隔符也计算在内。

STCCB　代码字符的总数

该度量统计文件中的代码字符总数。这里只计算可见的字符，除了字符串或字符数组。因为在后者的情况下，所有字符都计算在内，同时 tab 只作为一个字符计算。组成注释的字符，包括注释的分隔符，都不被计算。

STCCC　注释字符的总数

该度量统计文件中可见的注释字符总数。注释的分隔符不计算在内。

类度量

计算代码中类的度量。其中某些度量的运算需要首先执行 CMA 分析，见 **QA C++用户手册第 4 章之 CMA 分析**。完整列表如下：

STCBO	对象间的耦合（Coupling）
STDIT	继承的最深层次
STLCM	类方法的内聚缺乏度（Lack of Cohesion of Methods）
STMTH	类中可得到的方法数
STNOC	直接子对象的数量
STNOP	直接父对象的数量
STRFC	类的响应（Response）
STWMC	每类加权方法数（Weighted Methods per Class）

STCBO　对象间的耦合

统计由一个类访问的其他类方法（成员函数）或对象的数量。这里只考虑那些不在继承层次内的类。对继承树之外的类的耦合，需要谨慎看待，因为它增加了类的依赖性、减弱了

类的重用性。这是关于面向对象度量的Chidamber & Kemerer[4]度量集之一。

STDIT　　继承的最深层次

该度量代表着从最远的基类派生到本类的派生数量。数量大预示本类要依靠积累的功能，导致对本类的理解会很困难。它是 Chidamber & Kermerer 定义的度量之一。

STLCM　　类方法的内聚缺乏度

类中的方法要划分到不同的集合中，它们分别访问独立的成员对象。该度量就是这种集合的数目统计。例如：

```
class cohesive
{
private:
    int i1;
    int i2;
    int i3;

public:
    void m1 (int a) { i1 = a + i2; }
    void m2 (int a) { i2 = a; }
    void m3 (int a) { if (a > i3) ++i3; }
};
```

上述代码的 STLCM 值为 2，因为方法 m1 和 m2 访问的是成员 i1 和 i2，而方法 m3 只访问了 i3，于是可以看做是独立于 m1 和 m2（m3 对其他两种方法来说没有内聚性）。这是 Chidamber & Kermerer 定义的度量之一。

STMTH　　类中可得到的方法数

即在类中声明的方法数量。它不包含声明在基类中的方法。含有许多方法的类难以理解。

STNOC　　直接子对象的数量

计算当本类作为直接基类时的子类的数量。该值越高，说明类的功能依赖越严重，以及类的改变带来的潜在影响越大。这是 Chidamber & Kermerer 定义的度量之一。

STNOP　　直接父对象的数量

计算类继承的数量。对根类来说，其值为 0；对只有一层继承关系的派生类，其值为 1。某些编程标准杜绝类的多重继承。

STRFC　　类的响应

衡量类对函数的直接调用。STRFC 定义为类中不同方法的数目加上在这些方法的定义

[4] ***Chidamber, S.R. and Kemerer, C.F. (1991). Towards a Metrics Suite for Object Oriented Design, Proceedings of OOPSLA, '91, Sigplan notices, vol.26, no.11, ACM Press.***

中调用到的其他函数的数目。这是 Chidamber & Kermerer 定义的度量之一。

STWMC　每类加权方法数

该度量是类中所有方法的圈复杂度的总和。圈复杂度衡量了每个函数的测试需求，与此类似，STWMC 衡量了每个类整体的测试需求。这是 Chidamber & Kermerer 定义的度量之一。

5 程序返回值

下表列出了多个可执行程序的返回码和匹配的 GUI 消息。在“说明”列中指示返回码所对应的程序。

分析状态	返回码	解释	说明
Completed	0	成功完成	(1)
Failed: parser hard error	1	发生硬错误	(2)
-	1-99	说明遇到了未被抑制的最高级别的警告消息	(3)
Failed: parser configuration	2	● #include 文件没有找到 ● 对文件的递归调用 ● 注释块没有结尾 ● 遇到了#error 指令	(2)
Failed: configuration	10	● 通用的配置问题 ● 没有 internat.rsc 文件	(1)
Failed: system error	18	分析器程序产生了异常	(1)
Failed: fatal error	19	内存分配错误，或异常未被捕获	(1)
Failed: licensing error	11	产品授权失败	(4)
Failed: GUI personality	-	没有找到工程定义的个性文件	(1)
Failed: process error	-	分析开始时 Windows 处理错误	(1)
Failed: Sec Anal Configuration	-	后续分析的配置错误	

说明：

(1) 应用于所有可执行程序（qacpp.exe、pal.exe、prjdsp.exe、errwrt.exe、errsum.exe）

(2) 只用于 qacpp.exe

(3) 只用于 errdsp.exe

(4) 用于 qacpp.exe 或 errdsp.exe

6 度量输出文件

在分析过程中，QA C++写进.met 文件的信息包括：

- 内存分配
- 头文件使用
- 函数调用
- 派生类
- 所有函数、变量、宏、标签、标记和类型定义的声明和定义
- 度量值
- #pragma 指令
- 异常的说明
- 外部引用

QA C++在处理翻译单元时，记录是顺序写入文件的。它们的格式描述如下。

内存分配的记录

<ALLOC> *object method file lineno columnno*

内存分配的三种方法：

- 使用 new 和 delete 对单一对象的分配。
- 使用 new[]和 delete[]为对象数组分配内存。
- 使用 C 风格的内存分配函数，如 malloc()、free()、realloc()或 calloc()。

对单一对象，如果使用了不一致的内存分配和释放函数，将导致未定义的行为。具有外部链接的对象可以被不同的转换单元访问，而且在理论上，对独立文件的分析无法检测到分配与释放的不一致性。QA C++将为每个分配与释放的方法做一个<ALLOC>记录，使得 CMA 分析可以检测到任何可能的不一致性。

关系的记录

<R>I *includingfile includedfile lineno flag*

无论何时遇到#include 语句都会产生上面的记录。*includingfile* 可以是源文件，也可以是另外的头文件。如果被包含的文件是系统头文件，该记录会在 *flag* 的位置标记为 S，否则标记为短划线（-）。

<R>C *callingfunction calledfunction lineno flag*

每当发生函数调用时都产生上述记录，不管函数是内部链接还是外部链接的。如果调用的是虚函数，则 *flag* 处标记为 V，否则标记为短划线（-）。

<R>X *callingfunction identifier lineno flag*

为每个外部标识符的引用产生上述记录，对数据的引用是读操作时，*flag* 为 R，写操作时 *flag* 为 W。

要说明的是，每个函数调用都会产生<R>C 记录，如果函数是外部的，会另外产生<R>X 记录。

<R>B *baseclass childclass lineno flag*

为每个派生类产生上述记录。如果一个类具有多个基类，则每个基类都会产生一个<R>B 记录。*flag* 为短划线（-）。

<R>V *class datatype lineno flag*

为类中每个由用户定义类型的数据成员产生上述记录。*flag* 为短划线（-）。

<R>R *class referencedclass lineno flag*

为每个作为类引用的数据成员产生上述记录。*flag* 为短划线（-）。

<R>P *class pointer lineno flag*

为每个指针类型的数据成员产生上述记录。*flag* 为短划线（-）。

关系类型的记录

<RDEF> I Includes

<RDEF> C Calls

<RDEF> X Refers-to

<RDEF> B Is-base-of

<RDEF> V Has-by-value

<RDEF> R Has-by-reference

<RDEF> P Has-by-pointer

这些记录的存在只是为了描述 QA C++菜单系统中相应的<R>关系类型（即“Includes”或“Is-base-of”）

每种类型的<RDEF>的最大数记录在.met 文件中，并且只有在一个或多个该种类型的<R>记录出现时，才会记录<RDEF>。

定义的记录

<DEFINE> *identifier filename lineno inc def space scope linkage type flag*

为每个标识符产生记录，包括函数、变量、宏、标签、标记和类型定义。

域	内容	描述
identifier		标识符名字。标识符名字带有限定符，可以包含域和函数块。
filename		声明所出现的文件名字
lineno		声明所在行
inc	I	被包含的文件
	N	不被包含的文件，即主程序文件
def	DF	定义
	DC	声明
	DR	重新声明
	DI	隐含定义
	DS	函数或类的特例化
space	TG	类、结构、联合或枚举的标记
	OT	普通标识符——类型定义
	OF	普通标识符——函数
	OE	普通标识符——枚举元
	OM	普通标识符——宏
	OV	普通标识符——变量
scope	F	文件范围
	B	块范围

	P	函数原型范围
	C	类范围
	T	模板
	S	名字空间
	-	不可用（针对宏）
linkage	I	内部
	X	外部
	N	无——局部或类型定义
	-	不可用（针对宏）
type	见下表	符号类型码，描述数据类型的简短记号
flag	F	函数宏
	O	对象宏
	I	内联声明
	V	虚声明
	P	纯虚声明
	S	静态存储期
	D	POD 类（C 风格的函数）
	A	聚合类
	1,2,3	分别为显式的公有、保护、私有访问
	5,7	分别为隐式的公有、私有访问
	R	函数参数
	-	没有进一步信息

符号类型码	**数据类型**
_c	char
nc	signed char
uc	unsigned char

wc	wchar_t
ns, ni, nl, nll	short, int, long, long long
us, ui, ul, ull	unsigned short, unsigned int, unsigned long, unsigned long long
_o	bool
fs, ff, fl	float, double, long double
P*X*	*X* 的指针，*X**
*{*T*}*X*	指向 *T* 的 *X* 成员，*X T*::*
&*X*	*X* 的引用，*X&*
(*Y, A1, …, An*)	参数为 *A1,…,An*，返回 *Y* 的函数
(:, *A1, …, An*)	参数为 *A1,…,An* 的构造函数
(~, *A1, …, An*)	参数为 *A1,…,An* 的析构函数
[N, X]	N 个元素的数组 X
p	void 指针，void*
.	如同“返回 void 的函数”
{cNAME}	类 NAME
{sNAME}	结构 NAME
{uNAME}	联合 NAME
{eNAME}	枚举 NAME
ne	枚举常量
nb,ub	有符号或无符号位域
:	如同“void f (…)”
=X	const X
^X	volatile X

示例：

源代码：

```
#define M(x)   ( (x) + 10 )
int a = 0;
```

```
extern int b (char *c);
extern int b (int *i);
typedef double D;
static int e (D f);
void (*signal (int, void(*) (int))) (int);
extern int g;
class K
{
public:
    virtual void f ();
private:
    D m_d;
};
static float h (int i)
{
    int j;
    j = i + g;
    return (j);
}
```

.met 文件中<DEFINE>记录如下：

```
<DEFINE> M doc.cxx 1 N DF OM - - - F
<DEFINE> ::a doc.cxx 2 N DC OV F X ni S
<DEFINE> ::a doc.cxx 2 N DF OV F X ni S
<DEFINE> ::b(ni,p_c) doc.cxx 3 N DC OF F X (ni, p_c) –
<DEFINE> ::b(ni,pni) doc.cxx 4 N DC OF F X (ni, pni) –
<DEFINE> ::D doc.cxx 5 N DC OT F N ff -
<DEFINE> ::e(ni,ff) doc.cxx 6 N DC OF F I (ni, ff) –
<DEFINE> ::signal(p(ni),ni,p(.,ni)) doc.cxx 7 N DC OF F X (p(ni),ni,p(.,ni)) -
<DEFINE> ::g doc.cxx 8 N DC OV F X ni S
<DEFINE> ::K doc.cxx 9 N DC TG F X {c::K} -
<DEFINE> ::K::m_d doc.cxx 14 N DC OV C X ff 3
<DEFINE> ::K::f(.) doc.cxx 12 N DC OF C X (.) V1
<DEFINE> ::K::@destructor() doc.cxx 9 N DI OF C X () I5
<DEFINE> ::K::@constructor() doc.cxx 9 N DI OF C X () I5
<DEFINE> ::K::@constructor(,&={c::K}) doc.cxx 9 N DI OF C X (,&={c::K}) I5
<DEFINE> ::K::=@(&{c::K}, &={c::K}) doc.cxx 9 N DI OF C X (&{c::K}, &={c::K})
I5
<DEFINE> ::K doc.cxx 9 N DF TG F X {c::K} -
<DEFINE> ::h(fs,ni) doc.cxx 16 N DC OF F I (fs, ni) -
<DEFINE> ::h(fs,ni) doc.cxx 16 N DF OF F I (fs, ni) -
```

<DEFINE> ::h(,ni)::j doc.cxx 18 N DC OV B N ni -
<DEFINE> ::h(,ni)::j doc.cxx 18 N DF OV B N ni -
<DEFINE> ::h(,ni)::i doc.cxx 16 N DC OV B N ni -
<DEFINE> ::h(,ni)::i doc.cxx 16 N DF OV B N ni -

控制流图的记录

示例：

<CTLG> *functionname (stype) 1*
<CTLG> 1 2
<CTLG> 2 3
<CTLG> 2 4
<CTLG> 3 4
<CTLGN> 35 35 36 38

<CTLG>记录是为每个函数产生的，用来描述其结构。目的只有一个，就是使函数可以显示在 **Function Structure Diagram** 中。

第一条记录包含函数的名称、函数的符号类型描述以及函数退出点的数目。

后续的记录体现了控制流图中成对节点之间的连接。

在完整的<CTLG>记录后有一个<CTLGN>记录，它顺序地列出了所有节点的行号。参考**“QA C++用户手册”第七章之“查看函数结构的源代码”**，得到行号信息。

度量的记录

<S> *MetricName MetricValues*

示例：

<S>STCYC 8
<S>STMIF 0
<S>STMTH 3

每个文件、函数或类在输出其度量之前都会产生<S>STNAM 记录，包含其名字。

每个文件、函数或类都会产生<S>STFIL 记录，包含被分析的文件名称。

注：一些类度量的计算只能通过 CMA 分析，这是因为可能会有类成员函数定义在不同的源文件中。

参见第 4 章中对各种度量的说明。

Pragma 的记录

<PRAGMA> *filename lineno pragma_name*

为所有#pragma 指令产生记录。

异常说明的记录

<THROWS> *fn E type-list*

为带有异常的函数 *fn* 的声明产生上述记录。*type-list* 中列出了异常说明中包含的所有数据类型的符号类型码。如果异常说明中没有任何数据类型，则此处为短划线（-），表明任何数据类型都可以包含在异常中。

<THROWS> *fn T type-list*

为函数 *fn* 中的每个异常抛出的表达式生成上述记录。如果异常抛出表达式是在函数中处理的，则不产生该记录。

<THROWS> *fn C called-fn type-list*

当函数 *fn* 调用另外一个可能抛出异常的函数 *called-fn* 时，产生上述记录。如果函数是在 try 块以外调用的，则 *type-list* 处为短划线(-)。如果函数是在 try 块以内调用的，那么 *type-list* 处就是被 try 处理函数捕获的异常的数据类型。

外部引用的记录

<EXTD> *external_identifier filename lineno*

每个外部对象定义的输出。

<EXTN> *external_identifier filename lineno*

每个外部对象声明的输出。

<EXTV>*identifier external_identifier lineno*

每个对象定义的输出，但该对象是由外部对象初始化的。

7 QA C++实用程序

QA C++随带了一些实用程序，在创建附加的检查或定制的报告时可以使用它们。

r_basename

类似于 Unix 的 basename，它移走所有无关的后缀。例如：

r_basename “C:\Program Files\PRQA\QAC++\file.cpp”

会打印出：

file

r_close

r_close 实用程序执行名称相近标识符的分析，这些标识符的差别只在一个字符。它从文件或标准输入中读取标识符名称列表，然后生成一个报告，给出所有可能混淆的标识符对。

以-report 参数执行 r_close 时，生成的报告是排列在中间的两列，每一行包含了可能混淆的一对名字。同时也产生一个头信息和汇总信息。

如果没有使用-report 选项，相近的名称对以标签<CLOSE>的形式记录。汇总信息为：

<CLOSEPAIR> 30 180

<CLOSENESS> 0.001862

<CLOSEPAIR>显示相近对的数量和名称的总数。<CLOSENESS>显示相近性度量，由相近对的数量被可能对的数量相除而得到。对于 n 个名称来说，可能对的数量为 n * (n – 1) / 2。

例如（close.cpp 文件的内容）：

```
void close();
void closed();
void closes();
int line0;
int line1;
```

使用下面的命令

type close.cpp.met | r_grep –p “<DEFINE>” | r_fields + 1 | r_uniq | r_close –report

会产生下面的输出：

The following identifiers might cause confusion because of their similarity:

::close(.) ::closed(.)
::close(.) ::closes(.)
::closed(.) ::closes(.)
::line0 ::line1

4 pairs of close names were found in 5 names. The “closeness” metric is 0.400000.

r_fields

使用 r_fields，可以在.met 文件中的每条记录上选取一部分域内容。使用+n 选项，则选取第 n 个域；使用-n 选项，则忽略第 n 个域。例如：

type close.cpp.met | r_grep –p “<DEFINE>” | r_fields +2 +3 +1 +6

会抽取文件名称、行号、标识符和名字空间，如下：

close.cpp	1	::close(.)	OF
close.cpp	2	::closed(.)	OF
close.cpp	3	::closes(.)	OF
close.cpp	4	::line0	OV
close.cpp	5	::line1	OV

同样的结果，也可以通过移除不想要的域来得到。如：

type close.cpp.met | r_grep –p “<DEFINE>” | r_fields -0 -4 -5 -7 -8 -9 -10 | r_uniq

会产生：

::close(.)	close.cpp	1	OF
::closed(.)	close.cpp	2	OF
::closes(.)	close.cpp	3	OF
::line0	close.cpp	4	OV
::line1	close.cpp	5	OV
…			

如果要得到和前面完全一样的结果，上面的记录可以通过增加下面的命令来得到：

| r_fields +1 +2 +0 +3

注：因为域是以空格来分隔的，内含空格的文件名称会覆盖多个域。

r_grep

r_grep 是一个可移植的应用，它类似于 Unix 的 grep，用作模式匹配。它从一个文件或标准输入中读取数据，然后输出.met 文件中包含了一个或多个指定模式的行。也可以使用这个程序打印出不包含任何模式的行。

使用-p 选项对模式进行搜索。因为 r_grep 不像 grep 那样能够识别正则表达式，所以模式必须是纯文本的。在这一点上，r_grep 更像是 fgrep。可以使用多个-p 选项指定多个模式。例如：

```
r_grep –p”<S>STNAM” –p”<S>STPTH” close.cpp.met
```

将显示文件 close.cpp.met 中的函数名称和路径统计度量，如下：

```
<S>STNAM   close.cpp
<S>STNAM   ::close(.)
<S>STPTH    1
<S>STNAM   ::closed(.)
<S>STPTH    1
```

第一项是出现在文件度量中的文件名称，这样其他工具可以区分文件度量和函数度量。

要排除那些不包含任何指定模式的记录，可以使用-v 选项。例如，在上例中要防止出现第一项，可以使用：

```
r_grep –p”<S>STNAM” –p”<S>STPTH” close.cpp.met | r_grep –v –p”close.cpp”
```

r_sort

这是 Unix 版本的 sort 实用程序，它读取标准输入，对行进行排序，然后把结果写进.met 文件。通过指定域的序号，你可以对指定的域进行排序。例如：

```
type data | r_sort +2 +1
```

对 data 的第三个域进行排序，后跟第二个域。域的序号是从 0 开始的。最多可以排序 100,000 个记录。

r_uniq

这是 Unix 版本的 uniq 实用程序，它把标准输入拷贝到标准输出，忽略复制行。输入必须是经过排序的。

8 代码抑制

本章讨论通过代码注释进行诊断性抑制的规范和语法。消息的抑制可以针对基本分析、后续分析和 CMA 分析。其实现是要把抑制指令和诊断的产生分离开，由浏览工具决定哪些诊断是被抑制的。

实现也有效地扩展了#pragma PRQA_MESSAGES_OFF 的功能。#pragma 形式也同时可用。

所有诊断（消息）具有如下的基本信息：消息号和位置。位置是描述消息所指向的代码上下文的最小信息。当前的位置信息是个数据三元组<文件，行，列>，这是在代码中识别任意位置的所有信息。

基于代码的注释

基于代码的注释用于定义位置和抑制。使用位置标签标记抑制的主语，抑制可以是在任何地方定义的。

所有注释都是 C 或 C++风格的，必须以字符串 PRQA 开头。

位置标签语法

位置标签的语法是：

// PRQA L: *tag_name* [*location_specifier*]

其中：

tag_name	特殊的标记，可以由其他注释使用来指向这个位置
location_specifier	指定一个可选的位置，作为从此开始的位置范围的边界

用例：

// PRQA L: tag_name

为当前行创建一个位置标签，其他位置标签或注释可以使用这个 tag_name。

// PRQA L: tag_name n

为当前行和后续的 n 行创建一个位置标签。

// PRQA L: tag_name tag_name_end

创建一个标签，它包含当前行到 tag_name_end 行之间的所有位置（含 tag_name_end 行）。基于代码的抑制不能指向这个位置范围，它只在将来的版本中定义了基于配置的抑制时使用。

其他的限制：

在所有情况下使用真实行号；含有续行的不作为单行处理。

指定位置范围时，tag_name_end 是在当前行标签后第一个被找到的位置，而且该位置标签必须出现在当前文件中，否则会产生一个注释错误。“PRQA L”注释行总是要给出注释的结尾位置，即使注释本身定义了一个范围。

当前位置取为包含字符串 PRQA 的行的第 0 列。

预定义的位置标签

存在一个预定义的位置标签，用以简化一般的抑制需求：

EOF　　当前文件的结尾

注：不能重新定义预定义的位置标签。

抑制语法

抑制的语法是：

// PRQA S[:tag_name] message_specifier [location_specifier]

其中，

tag_name	可选的位置标签，外部可用它指向这个抑制。
message_specifier	该抑制应用的消息集合
location_specifier	或者是代表抑制结束的位置标签，或者是物理行数，或者是连续的位置指定。

其他的限制：

没有标签的抑制注释会被安排一个一般的标签，该标签在文件内是唯一的。

没有 location_specifier 的抑制注释只指向当前行。

当 location_specifier 是一个位置标签时，抑制的作用范围是在当前行直到第一个位置标签出现为止。

持续抑制语法

可以应用一个特殊的语法直接替代现存的#pragma PRQA_MESSAGES_ON|OFF。持续抑制只需要使用一条抑制注释语句从当前行开始执行。

该语法是：

// PRQA S message_specifier [++|--]

其中，

message_specifier	消息集合
++	message_specifier 中指定的消息集合加入到当前被抑制的消息集合中
--	message_specifier 中指定的消息集合从当前被抑制的消息集合中移走

举例：

// PRQA S 100 ++

100 号消息加入到当前被抑制的消息集合中，从当前行开始对它持续抑制。

// PRQA S 200 –

200 号消息从当前被抑制的消息集合中删除，从当前行开始不对它抑制。

其他的限制：

持续抑制不改变其他抑制注释的效果。

在被包含文件的开始处加入的持续抑制可以由包含文件继承。

在#include 指令前后定义的持续抑制的消息集合不会改变当前源文件。持续抑制的特殊行为是，“继承”进头文件的消息可以从头文件内部移除，方法是使用语法：

// PRQA S message_specifier --

由于这种特殊行为，这里有一个重要限制。那就是，因为实现上的结构约束，对头文件第一行/列产生的诊断消息不能被抑制。

Use-Case 示例

本节中要举例给出不同情况下的消息抑制，包含代码注释和最终的结果。

单一实例的抑制

在当前行对某消息做简单抑制

```
int i; // PRQA S 100
```

效果：在对 i 的声明出现的任何消息中，抑制 100 号消息。

空行的抑制

```
// PRQA S 100
int i;
```

效果：出现在 i 声明之前的消息中，抑制 100 号消息。

无效的抑制：缺少消息指定

```
// PRQA S: S1
int i;
```

效果：对缺少消息指定的抑制语法错误，产生 4826 号消息。它不关心抑制是否包含类似于 S1 的标签。

使用位置标签的范围抑制

在当前行与标签之间抑制（例 1）

```
int i;
int j;
// PRQA S 100 L1
++i;
// PRQA L: L1
++j;
```

效果：在++i 语句上抑制 100 号消息。

在当前行与标签之间抑制（例 2）

```
int i;
int j;
// PRQA S 100 L1
++i;
// PRQA L: L1
++j;
```

```
// PRQA L: L1
```

效果：在++i 语句上抑制 100 号消息，但在++j 语句上不抑制。

在当前行与标签之间抑制（例 3）

```
int i; // PRQA S 100 L1
int j; // PRQA L: L1
int k;
```

效果：对 i 和 j 抑制 100 号消息。

抑制范围的始终点在相同的行上（错误）

```
/* PRQA S 100 L1 */ int i; /* PRQA L: L1 */       // ERROR
int j; // PRQA S 100       // OK
```

效果：对第一条抑制语句，因为其始终点是在同一行上，所以会产生 4811 号配置消息。第二条抑制语句是正确的等价用法。

通过头文件的抑制

```
// foo.h
int i;

// PRQA S 100 L1
#include “foo.h”
// PRQA L: L1
```

效果：对 i 抑制 100 号消息。

抑制范围的结尾点不在相同的文件中（错误）

```
// foo.h
// PRQA L : L2

// foo.cc
// PRQA S 200 L2          // ERROR – L2 not found in this file
#include “foo.h”

// PRQA S L1              // OK – L1 does appear in this file
// PRQA L : L1
```

效果：对第一条抑制语句产生 4810 号配置消息，因为其结束点 L2 不在本文件（foo.cc）中。第二条抑制语句是正确的。

使用行计数的范围抑制

包含后面 n 行（例 1）

```
// PRQA S 100 1
int i;
int j;
int k;
```

效果：对 i 抑制 100 号消息。

包含后面 n 行（例 2）

```
int i; // PRQA S 100 1
int j;
int k;
```

效果：对 i 和 j 抑制 100 号消息。

超过文件剩余行的抑制

```
// foo.h
// PRQA S 100 100
```

```
// foo.cc
#include “foo.h”
int i;
```

效果：对 i 不抑制 100 号消息。抑制不会从头文件回传给包含它的文件。

从当前物理行到文件结尾

```
int i;
int j;// PRQA S 100 EOF
int k;
```

效果：对 j 和 k 抑制 100 号消息。

试图重新定义预定义的为止标签

```
int i;
// PRQA L: EOF     // can not redefine tag ‘EOF’ (Msg 4828)
int j;
```

效果：产生 4828 号错误消息。

持续抑制

开/关持续抑制

```
// foo.cc
int i;
// PRQA S 100 ++
int j;
int k;
// PRQA S 100 --
int l;
```

效果：对 j 和 k 抑制 100 号消息。

带有头文件入口的开/关持续抑制

```
// foo.h
int j;
```

```
// foo.cc
int i;
// PRQA S 100 ++
#include “foo.h”
int k;
// PRQA S 100 --
int l;
```

效果：对 j 和 k 抑制 100 号消息。它说明在#include 周围的抑制对头文件内部也有效。

持续抑制中增加消息

```
// foo.cc
int i;
// PRQA S 100 ++
int j;
// PRQA S 200 ++
int k;
// PRQA S 100, 200 --
int l;
```

效果：对 k 抑制 100 和 200 消息，对 j 只抑制 100 号消息。

持续抑制中删除消息

```
// foo.cc
int i;
// PRQA S 100, 200 ++
int j;
// PRQA S 200 --
int k;
// PRQA S 100 --
int l;
```

效果：对 j 抑制 100 和 200 号消息，对 k 抑制 100 号消息。

持续抑制与明显抑制的非重迭组合

```
// foo.cc
int i;
// PRQA S 100 ++
int j;
// PRQA S 100 --
int k;      // PRQA S 100
int l;
```

效果：对 j 和 k 抑制 100 号消息。

持续抑制与明显抑制的重迭组合（例 1）

```
// foo.cc
int i;
// PRQA S 100 ++
int j;// PRQA S : S1 100
// PRQA S 100 --
int k;
```

效果：对 j 抑制 100 号消息。S1 的失效不会显示消息，因为持续抑制也在起作用。

持续抑制与明显抑制的重迭组合（例 2）

```
// foo.cc
int i;
// PRQA S 100 ++
int j;
// PRQA S : S1 100 L1
int k;
// PRQA S 100 --
int l;
```

```
// PRQA L: L1
int m;
```

效果：对 j、k 和 l 抑制 100 号消息。S1 的失效只改变对 j 和 k 的抑制，l 将不再被抑制。

持续抑制配置失效

```
// foo.cc
int i;
// PRQA S : S1 100 ++
int j;
```

效果：产生抑制配置消息 4812，因为标签 S1 无效。

头文件中的抑制

头文件尾部的范围抑制

```
// foo.h
// PRQA S 100 10
int j;
// EOF foo.h
```

```
// foo.cc
#include “foo.h”
int i;
```

效果：只对 j 抑制 100 号消息。

从头文件到源文件之中的无效的范围抑制

```
// bar.h
// PRQA S 100 L1
int i;
```

```
// foo.cc
#include “bar.h”
int j;
// PRQA L : L1
int k;
```

效果：在 bar.h 中想要做出的抑制不能维持到 foo.cc 文件，同时会给出一个抑制配置信息。修正方法是把该抑制移到源文件 foo.cc 中。

未终结的文件抑制

```
// foo.h
// PRQA S 200 ++
int j;

// foo.cc
int i;
// PRQA S 100 ++
#include “foo.h”
int k;
// PRQA S 100 --
int l;
```

效果：对 j 抑制 100 和 200 号消息。对 k 抑制 100 号消息。本例说明，在头文件中定义的抑制，尽管没有终结，在头文件的末尾之后也不会影响到源文件。

强制包含文件的抑制

同其他头文件不同，在 Force Include 文件中的声明和入口会被认为出现在源文件的第 0 行。

这使得基于代码的抑制可以存在于源文件的外部，而作用和它们在源文件内部一样。

这种特殊处理不会扩展到 Force Include 文件中的#include 文件。

Force Include 文件中的范围抑制（例 1）

```
// forceincl.h
// PRQA S 100 3

// foo.h
int i;

// foo.cc
#include “foo.h”
int j;
int k;
```

效果：对 i 和 j 抑制 100 号消息。

Force Include 文件中的范围抑制（例 2）

```
// bar.h
// PRQA S 100 10

// forceincl.h
#include “bar.h”
```

```
// foo.cc
int j;
int k;
```

效果：没有在任何声明上作出抑制。

Force Include 同未终结的持续抑制的组合

```
// forceinclude.h
int i;
// PRQA S 200 ++
int j;
```

```
// foo.cc
int k;
// PRQA S 100 ++
int l;
// PRQA S 100 --
int m;
```

效果：200 号消息的抑制对 i、j、k、l、m 都有效，在 l 上也抑制 100 号消息。这里比较奇怪的是 i 也有抑制消息，然而根据 Force Include 文件的处理方式，它们被当作出现在源文件的第 0 行第 0 列，那么在技术上，Force Include 的所有内容都被看作是处在同一行上。

Force Include 同位置抑制的组合

```
// forceinclude.h
int i;
// PRQA S 200 L1
int j;
```

```
// foo.cc
int k;
// PRQA L : L1
int l;
```

效果：200 号消息的抑制对 i、j、k 有效。

抑制输入故障

无效的/缺少注释类型

```
// PRQA L : L1          // Location annotation
// PRQA S 100           // Suppression annotation
```

```
// PRQA X              // ERROR: Unknown annotation type
// PRQA                // ERROR: No annotation specified
```

效果：在指定一个空的注释或无效的注释时，产生 4820 号消息。

缺少注释标签

```
// PRQA L : L1         // Location annotation
// PRQA L L1           // ERROR: Missing ':'

// PRQA S 100          // OK, no tag specified
// PRQA S : S1 100     // OK, tag specified
```

效果：对抑制注释来说，标签是可选的。对位置注释来说，标签是必需的。在遇到不正确的格式或缺少位置标签时，产生 4821 号消息。

无效的注释标签

```
// PRQA S : S1 100     // Annotation tag 'S1'
// PRQA L : L1         // Annotation tag 'L1'

// PRQA L : 123        // ERROR: improper format
// PRQA S : 123        // ERROR: improper format
```

效果：注释标签必须以字母开头。命名错误的标签会产生 4822 号消息。

一般的语法错误

存在两种类型的语法错误。位置注释的语法是：

// PRQA L : <tag-name> [<end-tag-opt>]

抑制注释的语法是：

// PRQA S [:<tag-name-opt>] <msg_spec> [<end-tag-opt>]

某些情况下无法确定故障的真正原因，因此会产生一般性的错误。

```
// PRQA S 100 -        // ERROR
```

效果：错误的抑制语法会产生 4823 号消息。

标签名字中的无效字符

注释标签含有的字符只能是字母、数字和下划线。在定义持续抑制时，特殊的“++”和“--”也是允许的。

```
// PRQA S 100 A1       // OK
// PRQA S 100 A_1      // OK
```

```
// PRQA S 100 ++                // OK

// PRQA L : A1
// PRQA L : A_1
```

在指定注释标签时，不能使用这些限定之外的字符。

```
// PRQA S 100 A&1               // ERROR in tag name

// PRQA S 100 A@1               // ERROR in tag name
```

效果：根据记号的顺序，标签名字中的无效字符会产生 4824 或 4825 号消息。

9 命名规范检查

介绍

名称检查器是一个后续分析的进程，能够强制执行命名规范。如果标识符同指定的正则表达式不匹配时，它会给出一个诊断信息。

通常，名称检查器是在 GUI 中配置运行的，但也可以运行在命令行方式。它一次分析一个文件。在进行命名规范检查之前，必须已经成功执行了基本分析。

配置基础

名称检查器需要下面两项输入：

- 源文件以及基本分析产生的.met 文件和.err 文件。
- 写有命名规则的配置文件——通常在 Message Personality 的 Secondary Analysis 的设置中指定。

对配置文件中声明的任何附加消息，也必须在用户消息文件中声明。在基本的消息文件中为这一目的保留了 4800 号消息，作为默认的配置入口。

配置文件

这里的配置文件，也称为命名规范文件，包含了为名称检查器指定的规则。每个规则最少应该指定那些应用到各种类型标识符（如宏、函数、类型定义、变量等）的模式（正则表达式）。通过在规则中增加附加项，可以限制模式匹配所应用的标识符种类。

在输入配置中，规则是顺序检查的；对相反的规则，不做任何检查，在一个特定标识符上可以应用多个规则。例如，一个公司内部的编程标准可以具有下面两项规则：

1) 标识符必须少于 31 个字符

2) 函数名称必须以大写字符开头

当然可能有的函数违背了上面的规则，因此在该标识符上会出现多条消息的情况。

规则格式（JSON 语法）

配置文件基于JSON语法——详细描述见 www.json.org。唯一的区别在于配置文件使用

单引号限定字符串，而不是使用双引号。

JSON 对象被定义为规则，是以“rule=”开始的，忽略空行和以“#”开始的行。每个规则必须写在单独的一行中，并且不能拆成两行。

除了下面的情况，所有的规则值都是字符串：

- 转换是布尔的（以 true 或 false 指定）
- 消息号是正数

规则名称和值要匹配.met 文件中的<DEFINE>记录——见“**度量输出文件**”一章。

允许使用空规则（rule={}），它为所有标识符产生 4800 号消息。相似地，规则 rule={‘space’:’OV”}对所有变量产生 4800 号消息。这样做是有用的，在继续添加正则表达式之前，它可以帮助检查该规则是针对指定标识符的。

规则名称

本节描述规则名称以及它们能取到的值。对一些只能用在 QA C 或 QA C++的配置选项，会在下面单独声明。

文件名称

名称：filename
值：字符串、正则表达式

该对象用于指定规则所应用的特定文件。它是一个正则表达式。只有名称匹配了正则表达式的文件才会应用这些规则。要注意的是，同正则表达式进行比较的是文件的全路径名称，因此很重要的一点是，要在正则表达式中包含“行尾标志”（$），以避免发生对文件片段名称的错误匹配。

被包含的文件

名称：inc
值：下表中的字符串

它指定规则是仅仅应用在主文件上，还是应用在那些通过预处理指令#include 而包含进主文件的文件上。

值	描述
I	**被包含的文件**
N	**不被包含的——即主文件**

省略时，规则会应用在主文件和被包含的文件上。

定义

名称：def
值：下表中的字符串

把规则应用在不同格式的声明/定义上。

值	描述
DF	**定义**
DC	**声明**
DI	**隐含定义**
DS	**类/函数的特化（只适于 C++）**

省略时，规则应用在所有的定义、声明和特化（只用于 QA C++）上。可以使用逗号或空格对这些值进行组合。

空间

名称：space
值：下表中的字符串

把规则应用在不同种类的标识符上。

值	描述
LB	**标签**
TG	**类（QA C++）、结构、联合或枚举的标记**
MB	**结构或联合的成员（QA C）**
OT	**类型定义**
OF	**函数**
OE	**枚举元（QA C++）**
OM	**宏**
OV	**变量**

省略时，规则应用在所有标识符上。可以使用逗号或空格对这些值进行组合。

作用范围

名称：scope
值：下表中的字符串

把规则应用在标识符的作用域上，要说明的是，作用域和链接经常容易混淆。

值	描述
N	**函数作用域——标签（QA C）**
F	**文件作用域**
B	**块作用域**
P	**函数原型**
C	**类所用域（QA C++）**

T	**模板作用域（QA C++）**
S	**名字空间（QA C++）**

省略时，规则应用在具有任何作用域的标识符上。可以使用逗号或空格对这些值进行组合。

链接

名称: linkage
值: 下表中的字符串

把规则应用在具有不同链接的标识符上。

值	描述
I	**内部链接**
X	**外部链接**
N	**无链接——局部的或类型定义**

省略时，规则应用在任意链接的标识符上。

类型

名称: type
值: 字符串、正则表达式

把规则应用在不同类型的标识符上。它是使用符号速记表示的正则表达式。这里给出几个例子：

值	描述
nc	**signed char**
uc	**unsigned char**
ni	**signed int**
g	**void 指针（QA C）**
p	**void 指针（QA C++）**
pX	**指向 X 的指针，如 pni 是指向 int 的指针**
{uc,ui}	**参数为 unsigned int、返回为 unsigned char 的函数**
{sNAME}	**结构 NAME**

省略时，规则应用在所有数据类型上。可以使用逗号或空格对这些值进行组合。

这里的类型是指标识符的真实类型，而非 typedef 指定的类型。于是，在下面的代码中：

```
typedef unsigned int UINT32;
UINT32 foo;
unsigned int bar;
```

foo 和 bar 都具有类型“ui”。

标志

名称：flag
值：下表中的字符串

规则针对标识符的额外信息，可以帮助缩减标识符的范围。省略时，规则应用在所有的类型/声明/访问上。

值	描述
F	**函数宏**
O	**对象宏**
S	**静态存储期**
R	**函数参数**
只针对 QA C++的额外标志：	
I	**内联声明**
V	**虚函数声明**
PV	**纯虚函数声明**
D	**POD 类——C++中的 C 风格结构**
A	**聚合类**
1,2,3	**显式的公有、保护、私有访问**
5,7	**隐式的公有、私有访问**

对 QA C++，define 记录中的 flag 域可以是上述值的组合。

模式

名称：pattern
值：字符串、正则表达式

指定匹配标识符的正则表达式。省略时，不会有任何标识符可以匹配——也就是说，命名检查失效。这可以用来检查规则的其他部分指向了所需要的标识符。

转换

名称：invert
值：Boolean

如果指定为真，则转换模式匹配的逻辑。通常地，可以创建一个正则表达式来执行这个否定，然而更容易的方法（也更容易维护）是指定哪些是不要匹配的。如果省略，匹配的结果不进行转换。

消息号

名称：message
值：数字

当标识符的模式匹配失败时，用它指定相应的消息号。省略时，则使用 QA C++内在的

消息号 4800。

重要的是，这里产生的消息不要同其他消息发生冲突，不管是来自基本分析（1-4999）还是来自后续分析（5000+）的。如果不使用默认的消息（4800），那么产生的消息必须出现在用户消息文件中，否则会产生 99 级的致命错误（level 99 fatal error）。

名字空间

名称：namespace
值：字符串、正则表达式

指定规则应用的名字空间。嵌套的名字空间内的标识符在其全名称中含有所有名字空间的名称。

它只用于 QA C++。

Perl 正则表达式

本节简单描述 Perl 正则表达式，并不是关于 Perl 的完全指南。您可以在许多书籍和在线资源中获得 Perl 正则表达式的完全知识。

在线资源：http://perldoc.perl.org/perlre.html。

匹配字符

命名规范中的大多数正则表达式会指定要匹配的字符或字符范围，以及要匹配多少次。

文本字符匹配自身。“.”匹配任何字符。字符范围在“[‘,’]”中指定。例如，[A-Z]将匹配任何大写字符。如果要在某些字符中选择（而不是字符范围），就需要把它们括在方括号中。如[AMz]将只匹配字符 A、M 和 z。

可以把它们连接使用以匹配字符序列，例如[A-Z][a-z]将匹配任意两个字符组合，其中第一个字符是大写的，第二个字符是小写的。

匹配多次

有许多方法指定匹配的次数。在没有指定匹配次数时，模式必须被准确地匹配一次。可以使用下面的方法指定匹配的次数：

- 星号“*”代表匹配 0 次或多次
- 加号“+”代表匹配一次或多次
- 问号“?”代表匹配 0 次或一次

- 花括号“{‘,’}”指定特定次数或次数范围

例如，“[A-Z][a-z]*”可以匹配任何以大写字符开始的字（包括单个字符），因此“Speed”和“V”都可以匹配，但“velocity”是不能匹配的。

“[AEIOU][a-z]+”可以匹配任意 2 个或多于 2 个的字符，这些字符以大写的元音开头，然后是小写的字母。因此“Add”和“Egg”是匹配的，但“A”和“open”不匹配。

在花括号中可以有一或两个数字和一个逗号：

- 一个数字表示匹配准确的次数，如“^[a-z]{6}$”将匹配 6 个小写字母组成的字符串。
- 两个数字可以指定匹配的次数范围，如“^[A-Z]{6,8}$”将匹配 6 个、7 个或 8 个大写字母组成的字符串。

要匹配特定长度的任意字符，可以使用句点“.”。如“^.{1,31}$”将匹配 1 到 31 个字母之间的任何字符串。

限定符

正则表达式很重要的一点是，它要匹配的是整个标识符。为了清晰说明，以下为例：

假设有个规则规定成员变量必须以“m_”开头，那么“m_.*”看上去是合适的，因为它能匹配“m_speed”和“m_size”。

但是，它也能够匹配“sim_speed”。实际上，模式“m_”的匹配可以从标识符的任意位置开始进行。

要确保正则表达式匹配了整个标识符，需要使用“^”符号和“$”符号分别限定标识符的开始和结尾。在当前的特定例子中，因为我们要匹配任意字符，所以结尾符号不是很重要。然而，如果我们想要所有的成员变量都以字母“e”结尾，那么修正过的表达式“^m_.*e”可以匹配“m_speed”，因为它是以“m_”开头的，同时在“e”之前有 0 个或多个字母。因此，正确的模式是“^m_.*e$”，从而保证在“m_”之后可以有任意数量的字符，但结尾必须是字母“e”。

轮流匹配

对某些特殊标识符来说，有时需要多个匹配模式。比如，函数中局部变量的名称可以遵循某种规范，但对循环变量如“i”就是个例外。与其使用异常机制，不如构建正则表达式，匹配可以被忽略的标识符。

管道符“|”用以分隔两个（或多个）正则表达式。这几个正则表达式轮流进行匹配，一旦某个匹配成功就停止检查。对上面的例子，正则表达式为：

“^[ij]$|^b_”——匹配变量 i 或 j，或者以 b_开头的变量。

避开特殊字符

有时需要匹配的字符在正则表达式中是有特殊含义的，这些字符是“^.[$()|*+?{\”。圆括号可能是 type 模式需要的，因为它指明了一个函数，或者左方括号指示类型是个数组。如果这些字符出现在正则表达式中，并试图要匹配它们，它们会被解释为具有特殊含义。通常，这代表着正则表达式不会被编译，或者不会发生我们想要的匹配。为了把这些解释为文本字符，需要以反斜线“\”开头。而反斜线也是需要避开的。因此，在正则表达式中要指定一个左圆括号，使用“\\(”；而指定一个反斜线，则使用“\\\”。

配置文件举例

下面给出一个简单的配置文件，每条规则具有相应的描述：

#1 internal functions words seperated by _ and begin with capitals
rule={‘space’:’OF’, ‘linkage’:’I’, ‘pattern’:’^([A-Z][a-z]*)(_[A-Z][a-z]*)*$’, ‘message’:4800}

#2 block scope variables (except i and j) must start b_
rule={‘space’:’OV’, ‘scope’:’B’, ‘linkage’:’N’, ‘pattern’:’[ij]$|^(b_.*)$’, ‘message’:4800}

#3 macros must be all capitals, underscores allowed
rule={‘space’:’OM’, ‘pattern’:’^[A-Z_]+$’, ‘message’:4800}

#4 all functions and variables must be 31 or less chars
rule={‘space’:’OF,OV’, ‘pattern’:’^.{1,31}$’, ‘message’:4800}

返回错误

一旦发生错误，它们就记录在文本文件“NameCheckErrors.txt”中，该文件位于输出目录下。如果工程有多个输出路径，该文件会出现在每个输出路径中（并且内容相同）。

配置文件

如果没有找到配置文件，日志文件会显示在 Message Personality 中指定的文件名称。检查指定的文件是正确的以及具有正确的访问权限。如果文件路径中包含空格，要确保文件的全路径名称是由引号括起的。

JSON 语法错误

所有的 JSON 错误消息以错误所发生的配置文件的行号开头。JSON 中的任何语法错误会记录成“Error: option parsing;”字样，后跟问题的解释和位置。

“invert”的值是布尔的，因此必须是文字 true 或 false；“message”的值是数字。对这

些值不能使用引号。

未知名字

规则名称非法时会发生这样的错误。要说明的是，它们必须是小写的。例如{‘Scope’:’I”}会产生下面的信息：

Line 1 Unknown object member name ‘Scope’ with string value I

{‘Scope’:’I’}

因此，把大写的 S 改成小写，就可以修正这个错误。

未知的值

如果某个对象的值不在允许的范围内，就会发生这样的错误。使用正则表达式作为取值的对象（filename、pattern、type）不会产生这个错误。比如，{‘linkage’:’E’}会生成下面的信息：

Line 2 Unknown value ‘E’

在这个例子中，想指定外部链接，其值应为“X”而不是“E”。

正则表达式

在处理配置文件时，会编译 filename、pattern、type 的正则表达式。如果它们存在错误，就被记录下来。例如：

Line 29 Pattern Regular expression ‘[[:ower:]]*’ compile fail: bad class at offset 3

{‘pattern’: ‘[[:ower:]]*’}

offset 指示正则表达式中发生问题的所在。一些情况下，问题（如不匹配的方括号）报告在字符串的结尾。

10 布局配置文件中的术语

控制代码布局的配置文件使用下面的术语描述代码的文法。为方便理解，这里对各术语没有进行翻译。

配置术语	简短描述
abstract-declaration	Declaration without an identifier
aggr-init	The outermost aggregate initialiser
aggr-init-clause	The initialiser in an aggregate initialisation
aggr-init-decl	Declaration with an aggregate initialiser
aggr-init-list	Aggregate initialisation containing a list of aggregate initialisers
aligned-name	Variable declaration in a bitfield
anonymous-class-def	Class definition without a class name
arith-expr	Arithmetic expression
array-abstract-declarator	Array declaration without an identifier (e.g. an array argument declared in a function pointer)
array-declarator	Declaration of an array variable
array-expr	Expression indexing into an array
asm-stmt	Embedded assembler code
assign-expr	Assignment expression
assign-init-decl	Declaration initialised with assignment syntax
base-clause	Base classes for a class definition
base-spec	Single base class in a base clause
base-spec-list	The list of base classes for a base clause
binary-oper-expr	Bitwise operator expression (&, ^, etc.) with two operands
block	Block of code nested inside a larger construct
break-stmt	Break statement, commonly used in switch statements

c-cast-expr	C-style case expression, the target type is in parentheses
cast-expr	C++ style cast expression such as *static_cast*<…>(…)
catch-list	The list of catch handlers after a try block
catch-stmt	Single catch handler
class-def	Class definition
class-member-node	Statement inside a class definition
cmpd-stmt	Compound statement
comma-expr	Expression with a comma
continue-stmt	Continue statement
conv-operator	Class conversion function (e.g. operator *bool* ())
cv-spec	Const-volatile specifier in a declaration
cv-spec-seq	Sequence of const-volatile specifiers
decl	Any type of declaration
decl-spec	Declaration specifier such as *int*, *firend*, *static*, etc.
decl-spec-seq	Sequence of declaration specifiers
decl-stmt	Declaration statement
declaration	Declaration with identifier
declaration-seq	Sequence of declarations
declarator	The identifier portion of a declaration
def-enum	Single enumerator definition with opertional initialiser
delete-expr	*delete* expression
dlay-expr-block	see block
dlay-fn-body-block	see block
dlay-stmt-block	see block
do-stmt	*do-while* statement
dtor-name	A destructor name
elab-type-spec	The type is either a *struct* or *enum* definition

enum-def-list	A list of enumerators that make up an enum definition
enum-list	The list of enum defs with an optional trailing comma
enum-spec	An enum definition
expr	Any expression
expr-list	Comma seperated list of expressions
expr-stmt	Statement that evaluates an expression
for-stmt	*for* loop
friend-spec	The *friend* keyword
func-defn	Function definition
function-decl	Function declaration
fncall-expr	Function call expression
fn-style-abstract-declarator	An abstract declarator with function style initialisation
fn-style-declarator	Declarator with function sytle initialisation
fn-init-decl	Declaration with funciton style initialisation
full-name	Fully qualified name
goto-stmt	*goto* statement
id-expr	An expression that names an object
idr	An identifier
if-stmt	*if* statement
label	Label
label-stmt	Label statement
literal-expr	Literal constant
memb-access-class	Access specifiers in a class
memb-func-defn	Function defined in a class definition
memb-qual-tmpl-name	Member qualified template name
memb-expr	A dot or arrow expression
member-name	A name of an object

member-template-name	The name of a member template
name	A reference to an identifier in some scope
new-expr	*new* expression
new-init-expr	Constructor arguments in a *new* expression
new-placement-expr	Pointer expression for placement *new*
not-expr	*not* expression
object-template-name	The name of an object template
overload-operator	Overloaded operator function
param-decl	Parameter declaration
paren-declarator	A declarator in parentheses
paren-expr	An expression in parentheses
postinc-expr	A postfix increment/decrement expression
ptr-memb-expr	A pointer to a member expression
ptr-oper	The pointer symbol with const volatile qualifications
ptr-oper-abstract-declarator	A pointer to an abstract declarator
ptr-oper-declarator	A pointer to a declarator
qualified-name	A qualified name
rel-expr	Relational expression
return-stmt	*return* statment
shift-expr	*shift* expression
sizeof-expr	*sizeof* expression
stmt	Any statement
stmt-seq	A sequence of statements
switch-label	*case* or *default* label
switch-label-stmt	A statement with a *case* or *default* label in it
template-arg	Template argument
template-arg-list	Template argument list

template-name	Name with template arguments
ternary-expr	Ternary expression
this-expr	Expression that uses *this* keyword
throw-expr	*throw* expression
throw-spec	Function throw list
tmpl-decl	Template declaration
tmpl-param-list	Template parameter list
truth-expr	Binary truth expressions
try-catch-stmt	*try-catch* statement
type-param	Template type parameter
typeid-expr	*typeid* expression
unary-expr	Unary expression
user-type-name	A user type name (e.g. a class name)
using-decl-stmt	*using* declaration
using-dirc-stmt	*using* namespace directive
while-stmt	*while* loop