Compilação Condicional em Cm

Sheila P. Maceira *Alexandre Prado Teles*
Rogério Drummond

Relatório Técnico IC 97-04

Maio de 1997

Compilação Condicional em Cm: Motivação e Análise de Requisitos[1]

Sheila P. Maceira
Alexandre Prado Teles
Rogério Drummond
{sheila, teles, rog}@ahand.unicamp.br

Laboratório A-HAND
Instituto de Computação
Unicamp

Novembro, 1996

RESUMO

Programas não-triviais escritos na linguagem de programação C exigem o uso de diretivas de pré-processamento –uma espécie de "linguagem dentro da linguagem". O pré-processador C (cpp) é essencial para a abstração de código e especialmente para a programação modular.

A linguagem Cm, derivada de C e com características para o suporte à orientação a objetos e distribuição, ainda não se utiliza do pré-processamento, em parte por suportar via construções próprias várias das funcionalidades providas por este recurso.

Este documento discute aplicações e problemas do cpp, para então analisar como introduzir em Cm facilidades de pré-processamento ainda não suportadas, buscando ainda restringir ao mínimo possível suas desvantagens. Uma proposta preliminar é incluída.

INTRODUÇÃO

A linguagem de programação C aceita [KR 88] uma ferramenta de compilação conhecida por C *preprocessor* (cpp) para executar uma tradução do código-fonte preliminar à compilação propriamente dita. Cpp é invocado automaticamente pelo compilador C como um passo separado e independente[2]. Como tal, define uma segunda linguagem cujas construções (dadas por diretivas) são analisadas e retiradas do código entregue ao compilador C. Elementos estranhos (isto é, código C) permanecem inalterados.

Um estudo mais detalhado com relação à implementação de um mecanismo de suporte ao pré-processamento inclui a análise de características e funcionalidades do cpp, restrições ao seu uso e a definição de requisitos desejáveis para a extensão pretendida.

Funcionalidade provida pelo cpp

As tarefas executadas pelo cpp são basicamente as seguintes:

[1] Este trabalho foi parcialmente financiado pelo PROTEM/CNPq, processo número 680089/94–2 e 380371/95-2 e pela FAPESP, processo número 94/5179-6.

[2] Essa separação é conceitual e não necessariamente se aplica a todos os compiladores C.

- *Inclusão de Arquivos*

  A inclusão de arquivos nada mais é do que a incorporação do texto de outros arquivos em meio ao código-fonte. Embora possa ser usada para particionar um programa em trechos gerenciáveis, essa técnica deve ser evitada em favor da compilação em separado. O melhor uso de inclusão é concentrar em arquivos de declaração ou cabeçalho construções que devem ser conhecidas por diversos módulos. Essa prática inclui a declaração das funções e tipos oferecidos pela biblioteca padrão (sub-rotinas e chamadas ao sistema) bem como bibliotecas e serviços adicionais ou *third-party*.

- *Definição de constantes textuais*

  A definição de constantes textuais consiste em associar a símbolos ou identificadores (seguindo a definição de C) uma *string* de texto. A partir dessa associação, ocorrências "maximais" do identificador fora de *strings* constantes e comentários são substituídos pela *string-valor*, num processo denominado *expansão de macro*. Algumas aplicações comuns incluem:

  - constantes numéricas, como DoisPI=(3.14159265358979323 8*2)
  - enumerações, como OK=1, Overflow=1, Underflow=2
  - "números mágicos", como OpCodeSave=0xff3c
  - limites do programa, como STACKSIZE=100
  - maquilagem da linguagem, como forever=for(;;)
  - abreviações, como va_dcl=char *va_alist;

  Símbolos definidos através deste mecanismo são desprovidos de tipo tal como definido em C. Caso sejam usados em expressões, o tipo dos mesmos é influenciado pelo contexto da expansão. Alguns símbolos jamais são definidos explicitamente, pois são previamente definidos pelo cpp. Podem servir como identificação do próprio sistema de compilação ou para obter informações sobre o código sendo compilado.

- *Definição de pseudo-funções*

  Pseudo-funções são trechos de código incluindo comandos a substituir literalmente em cada ocorrência (tipicamente parametrizável) do símbolo. Até [KR 88] as regras de resolução eram algo ambíguas, ocasionando problemas em casos especiais de recursão.

- *Inclusão Condicional*

  A inclusão condicional é uma incorporação de trechos de código decidida a partir de uma expressão calculável em tempo de compilação e usualmente envolvendo símbolos previamente definidos para cpp. Sua principal finalidade é auxiliar na manutenção de versões alternativas do mesmo programa usando o mesmo arquivo fonte, em especial para portabilidade entre diferentes arquiteturas e ambientes de execução. O trecho condicional pode conter diretivas cpp, incluindo outros condicionais.

- *Controle de Linhas de Código*

  Uma das atribuições do cpp é informar o compilador C a respeito da procedência de cada uma das linhas de código fonte constantes de um arquivo: de qual arquivo fonte vieram e em que linha estavam localizadas. Essa capacidade é usada principalmente em programas gerados automaticamente.

- *Modificação do comportamento do compilador C*

  Em certos aspectos é análogo a opções de linha de comando, mas podendo operar sobre trechos específicos do código-fonte.

- *Outras tarefas de alteração do código a compilar*

Tarefas como remoção de comentários, indicações sobre diagnósticos de erro, "expansão" de trígrafos, etc, são igualmente passíveis de execução via cpp.

A linguagem Cm [Drummond 88, Furuti 91, Teles 93], proposta e desenvolvida no âmbito do projeto A-HAND [Drummond 87] como linguagem básica de programação para projetos de grande porte, compreende as construções de C aliadas a características de orientação a objetos. Decidiu-se que Cm dispensaria algumas idiossincrasias de C que usualmente causam dificuldades aos programadores, tornando as linguagens de certa forma incompatíveis. Até a definição atual [Teles 93], Cm não prevê uma ferramenta análoga a cpp.

Restrições ao uso do cpp

Embora caracterize-se como uma ferramenta indispensável à programação na linguagem C, o uso de cpp é causa em potencial de uma série de dificuldades e armadilhas, particularmente com relação ao aspecto de substituição textual de símbolos, senão vejamos:

- Os símbolos criados pela cláusula define e o conteúdo de um arquivo incluído são tratados pelo cpp como expansão de texto. As cláusulas destinadas a cpp podem aparecer entre duas linhas de código quaisquer. Essa característica quase que obriga uma compilação em mais de um passo, com o compilador propriamente dito lendo o resultado do passo preliminar de processamento. A arquitetura do sistema de compilação torna-se mais complexa (mesmo tornando cada passo conceitualmente mais simples). Por outro lado, como o resultado da expansão de um símbolo pode ser um texto qualquer, apesar da flexibilidade oferecida algumas tarefas comuns em programação tornam-se inerentemente mais complexas:

  - criar uma lista ou índice das definições de funções, tipos e variáveis dos módulos de um programa;
  - viasualizar em cores diferentes os elementos (comentários, palavras reservadas, etc.) do programa;
  - analisar e conferir o "casamento" entre delimitadores ('{' e '}', '(' e ')', etc.);
  - navegar e editar um programa respeitando os elementos da linguagem, não apenas caracteres, palavras e linhas (*syntax-directed edition*);
  - verificar as dependências entre módulos.

  Os trechos sob compilação condicional são também fragmentos arbitrários (mesmo incompletos) em C, o que apenas agrava o problema. Benefícios de portabilidade podem incorrer em aumento da complexidade na compreensão e manutenção do projeto, consequência indesejável.

  Dessa forma, o uso de cpp pode complicar consideravelmente tanto a manutenção de código como a implementação de muitas das ferramentas de programação indispensáveis.

- A flexibilidade provida via substituição textual, particularmente em pseudo-funções, igualmente não está livre de armadilhas. A semântica da chamada de funções é violada e argumentos podem acabar sendo avaliados mais de uma vez com efeitos colaterais adversos. Uma outra questão é que erros em macros somente podem ser detectados na expansão.
- Como C não suporta a noção de tipos abstratos de dados, não são explícitas as dependências de um arquivo nem mesmo as dependências entre arquivos em relação a constantes de pré-processamento. Para se obter esse tipo de informação é preciso ler todo o arquivo, anotando todas as constantes de pré-processamento utilizadas e não definidas. E este processo deve ser estendido a todas os arquivos incluídos. De fato, ao programador cabe a tarefa de simular uma execução parcial do cpp, no que diz respeito a expansão dos comandos condicionais (#if e #ifdef) e inclusão de arquivos. Neste processo, deve-se ainda anotar as constantes de pré-compilação não definidas.

  Numa linguagem orientada a objetos é inadmissível que estas dependências sejam desconhecidas e mesmo que o custo de se obter esse tipo de informação seja alto e deixado a cargo do

programador. O pré-processamento de C estendido a C++ é inaceitável como base de um ambiente de desenvovimento para software de grande porte.

PRÉ-PROCESSAMENTO EM CM

No processo de determinação dos requisitos da extensão em estudo, vamos analisar questões relativas aos mecanismos já implementados em Cm que podem ser utilizados no sentido de obtermos funcionalidades equivalentes às providas pelo cpp para a linguagem C. Vejamos então os mecanismos para definição de constantes simples (símbolos), valoradas, e pseudo-funções.

Definição de Constantes

A definição de constantes simples em Cm é viabilizada pelas cláusulas const e enum. Conforme [Teles 93], a cada identificador Cm é associado um escopo de visibilidade, que é estático e corresponde ao trecho de código onde o identificador é visível. Caso o identificador seja declarado no nível léxico zero como exportável (export), seu escopo é estendido via seleção.

Itens definidos são de certa forma análogos a variáveis inalteráveis. Daí decorre que const não pode ser utilizado, tal como define, para abreviações ou maquilagens da linguagem; esta técnica é inclusive desaconselhada em Cm.

Note que símbolos constantes criados via define permanecem definidos e no escopo – independentemente das regras de escopo da linguagem C– até o final da unidade de tradução ou até que sejam indefinidos via diretiva undef.

Constantes podem ser de qualquer tipo e sempre devem ter um valor na declaração. A expressão que define o valor de uma constante é avaliada em tempo de compilação. Desse modo, podemos inferir que através da cláusula const não é possível definir um símbolo desprovido de um valor associado, pois constantes são basicamente valores: não têm alocação e não podem ser referenciadas.

A cláusula enum dispensa a técnica de definir constantes para representar um domínio de valores. enum permite definir um conjunto finito de nomes simbólicos que são tratados como constantes.

Pseudo-Funções

Pseudo-funções nos moldes do definido pelo cpp são atraentes basicamente por duas razões. Primeiro, tal funcionalidade é caracterizada pela expansão textual de código, imune a verificações tanto sintáticas quanto semânticas, o que pode ser bastante desejável em alguns casos –por exemplo, cada expansão de uma mesma macro pode envolver tipos diferentes, podendo inclusive conter construções C incompletas. Segundo, a expansão de código (substituição da chamada de funções por comandos equivalentes) evita a sobrecarga associada a chamadas de funções, com as respectivas passagem de parâmetros e cálculo e retorno de valores.

Para a caracterização de funções otimizáveis, Cm provê a cláusula inline. Sob alguns aspectos inline constitui um mecanismo mais restrito que define, exigindo a definição de uma função completa, com tipos relativos a parâmetros e retorno bem definidos. Porém, para os nossos propósitos de manutenibilidade e clareza de código, inline se mostra mais adequada, principalmente porque o mecanismo garante a preservação da semântica associada às funções e da verificação estática de tipos.

Abstrações em Cm

Para os casos em que a expansão de uma macro envolve tipos e outros itens impossíveis de parametrizar numa função convencional, Cm disponibiliza um mecanismo denominado classe parametrizável. Todo programa Cm é, por definição, uma meta-classe. Meta-classes são construtores de tipos usados para se especificar tipos denominados classe. Meta-classes podem ser parametrizadas, inclusive tendo especificações de tipos como parâmetros, podendo definir uma ampla variedade de tipos (uma para cada conjunto de possíveis valores de seus parâmetros).

Classes constituem o mecanismo padrão de abstração de tipos em Cm. Uma classe define uma representação concreta e específica de uma idéia ou conceito, englobando tanto a informação necessária para representar essa idéia como as operações que podem ser executadas sobre ela. O processamento de um projeto complexo –compilação e ligação– é gerenciado pelo ambiente de programação. Mecanismos da linguagem e do ambiente garantem, automaticamente, a exportação – quando necessária– de informações sobre uma classe para aquelas que dela dependem, sem que o programador precise interferir neste processo.

Requisitos

Diante do exposto, conclui-se que um novo mecanismo de definição seria necessário apenas para a determinação de compilação condicional. O compartilhamento de definições como constantes, macros e tipos, pode ser obtido através do mecanismo de herança. Basta construir uma classe (possivelmente sem métodos) com as definições que devem ser compartilhadas. Toda classe que desejar utilizar-se dessas definições deve herdar dessa classe. O que em C e C++ é realizado com inclusão textual de arquivos pode ser alcançado com herança, com vantagens de garantia de verificação de tipos e evitando a recompilação dos arquivos incluídos.

Portanto, concluímos que a compilação condicional caracteriza-se como a única facilidade de pré-processamento totalmente desprovida de suporte em Cm. Há necessidade então de se definir um mecanismo com funcionalidade equivalente e facilmente compreensível. Além disso, o esquema de decisão deve prover tipos para que seja efetivado o suporte à facilidade de classes polimórficas.

PROPOSTA

A extensão da linguagem ora proposta leva em consideração algumas preocupações básicas. A primeira delas no sentido de assegurar que todas as extensões se integrem à sintaxe da linguagem sem que seja necessário um passo intermediário de compilação; uma outra questão recai sobre o fato de que pretende-se criar um mecanismo o mais "inteligente" possível, no sentido de que seja capaz de resolver alguns conflitos simples (por meio do estabelecimento de regras de conduta), deixando a cargo do programador, quando for o caso, a resolução de questões mais elaboradas.

Escolhemos os *comandos* condicionais seguindo proximamente a sintaxe dos comandos em C. As palavras reservadas dos comandos condicionais são versões em maiúsculas dos similares em C.

Novas Construções e Palavras Reservadas

Considerando-se as questões levantadas nas seções anteriores, sugerimos a seguir algumas construções e palavras reservadas a serem introduzidas na linguagem. As palavras reservadas para comandos de compilação condicional são em geral similares aos comandos em Cm, mas escritas em letras maiúsculas.

Optamos pelo uso da palavra END ao invés do uso de chaves para delimitar sequências de comandos. Apesar de diferir da sintaxe usual de C caracteriza-se como um compromisso de clareza, evitando conflitos com as construções da própria linguagem.

Definição de Constantes de Compilação Condicional

DEFINE identificador **'='** valor **[','** identificador **'='** valor**]* ';'**

DEFINE associa a um símbolo identificador uma *string* de caracteres. O identificador é de uso exclusivo das cláusulas DEFINE, IF e SWITCH, não se confundindo com os identificadores usuais em Cm. Ocorrências do identificador fora dessas cláusulas não são substituídas pelo valor definido. O valor pode ser omitido juntamente com o sinal '='. Neste caso a constante é definida com valor não especificado. O seu valor em expressões condicionais é verdadeiro e em outros usos ela expande para uma *string* nula.

A *string* que define uma constante se estende até a primeira ocorrência dos caracteres **‘,’, ‘;’** ou fim de linha. Pode-se usar os delimitadores **‘<[’** e **‘]>’** para valores que contenham os caracteres **‘,’, ‘;’** ou que se estendam por várias linhas.

DEFINE é uma cláusula válida em qualquer ponto onde um comando ou declaração de variável são válidos, sendo efetiva do ponto onde ocorre até o final da classe sendo compilada. O escopo de uma constante é definido na seção *Escopo e Parametrização*.

Comando ECHO

ECHO mensagem

ECHO faz com que o pré-processador gere uma mensagem de diagnóstico que inclui a sequência de símbolos determinada por mensagem.

A *string* de caracteres mensagem pode conter símbolos de pré-processamento –especificados via DEFINE– que serão devidamente expandidos, e palavras especiais do pré-processador. Todo o texto não caracterizado como símbolo de pré-processamento ou palavra especial é simplesmente ecoado.

Comando IF

IF (expressãoC**)** declaração-ou-comando-1 **END**

IF (expressãoC**)** declaração-1 **ELSE** declaração-2 **END**

IF (expressãoC**)** comando-1 **ELSE** comando-2 **END**

A primeira declaração ou comando serão considerados caso a *expressãoC* seja calculada em tempo de compilação como verdadeira; em caso contrário (isto é, equivalente a uma *string* vazia, não-definida ou igual a “0”), será ignorada considerando-se em seu lugar a segunda declaração ou comando após ELSE, se houver. As sub-cláusulas 1 e 2 devem ser ambas comandos ou declarações Cm, sendo então a cláusula IF correta em lugar de um comando ou declaração, respectivamente.

A *expressãoC* deve ser uma constante calculável durante a compilação –como tal, não pode envolver chamadas ou endereços de funções, endereços e valores de variáveis. Pode, entretanto, conter descritores de tipos. Os operadores permitidos na expressão são apresentados na tabela a seguir:

Operador	**Funcionalidade**
==	para verificar igualdade entre *strings* (numéricas inclusive) ou efetuar comparação de compatibilidade de tipos
!=	para verificar desigualdade entre *strings* (numéricas inclusive) ou efetuar comparação de compatibilidade de tipos
&& || !	operadores lógicos
( )	parênteses para agrupamento
TYPEOF (X)	pseudo-função para obter o tipo da função ou variável X

Comando SWITCH

SWITCH (expressãoC**)**
    CASE (constante1**):** declaração-ou-comando-1
        [**BREAK**]
    . . .
    CASE (constante-n**):** declaração-ou-comando-n
        [**BREAK**]
    . . .
    [**DEFAULT:** declaração-ou-comando-default]
END

Caso a *expressãoC* tenha valor equivalente ao de uma constante, todo o código Cm contido entre a *constante* e a cláusula END ou BREAK serão considerados, sendo que o código restante é ignorado. Em caso contrário, considerar-se-á o trecho entre DEFAULT –se houver– e BREAK/END.

Todas as constantes (*constante-1* a *constante-n*) devem ser calculáveis durante a compilação, sendo certo ainda que devem possuir valores diferentes entre si. A *expressãoC* sofre as restrições da diretiva IF. As cláusulas DEFAULT e BREAK são opcionais.

Para uma mesma diretiva SWITCH, os itens entre as constantes e após o DEFAULT, ou são todos comandos ou são todos declarações.

Escopo e Parametrização

O escopo de uma constante é iniciado no ponto da sua declaração e se estende até o fim do arquivo. Uma constante só pode ser utilizada se for definida, caso contrário o programa está errado. Uma constante pode ser definida através do comando DEFINE ou como parâmetro da classe. Na declaração de uma classe,

class nome<lista-de-parâmetros>

a lista-de-parâmetros é estendida para suportar parâmetros de constantes de compilação condicional:

DEFINE nome [= valor]

O fragmento de código a seguir exemplifica o uso de constantes de compilação condicional:

```
class A<DEFINE DEBUG>
…
// código dentro de um método
IF (DEBUG)
cout << “passei por aqui”;
        END
```

O valor de uma constante é dependente de onde a constante é utilizada e é definido segundo as seguintes regras (que são excludentes e aplicadas nesta ordem):

- o valor do último DEFINE efetivo[3] anterior ao seu uso no arquivo
- o valor definido no escopo da classe externa a classe corrente[4]
- o valor *default* definido na lista de parâmetros da classe corrente

[3] Podem existir vários DEFINEs para uma dada constante sendo que essas definições ocorram dentro de comandos de compilação condicional de forma que não tenham sido expandidos. Por DEFINE *efetivo* entendem-se as ocorrências de fato incorporadas ao código, após o processamento dos comandos de compilação condicional.

[4] Para a classe mais externa numa compilação, o ambiente (*environment* do Unix) é considerado como seu escopo externo.

O uso de constantes que não se encaixam nestes casos é incorreto por definição.

Se uma classe A herda ou importa uma classe B, então A é dita classe *externa* a B. Dessa forma, as constantes de compilação condicional são passadas entre as classes segundo a hierarquia de dependências das mesmas.

Palavras Especiais

As ocorrências das seguintes palavras são automaticamente substituídas onde quer que ocorram fora das novas cláusulas, exceto em *strings* constantes e comentários. Não podem ser modificadas pelo programador. As novas palavras operam como reservadas. Os nomes são compatíveis com as convenções adotadas em [Teles 93].

Palavra Especial	**O que representa**	**Exemplo**
__FILE__	*string* constante com o nome do arquivo sendo compilado	"/home/ahand/usr/cmc/List/DList.cm"
__CLASS__	*string* constante com o nome da classe sendo compilada	"Stream"
__LINE__	constante int contendo a linha de código fonte sendo compilada	173
__COMPILER__	identificador único do compilador Cm	"A-HAND"
__VERSION__	*string* constante com a versão da linguagem suportada pelo compilador	"2.0"
__SYS__	*string* constante identificando o sistema operacional/ambiente de execução	"UNIX"
__DATE__(*String*)	constante identificando data e/ou hora da compilação. *String* é uma *string* de formatação no estilo de printf (escape sequences)	"22:10:35" "Jan 05, 1996"

REFERÊNCIAS

[Teles 93] Teles, A. *A Linguagem de Programação Cm*. Dissertação de Mestrado. DCC, Unicamp (dezembro 1993).

[Stroustrup 93] Ellis, Margareth A., Stroustrup B. *C++: Manual de Referência Comentado*. Editora Campus, Rio de Janeiro, 1993.

[Furuti 91] Furuti, C.A. *Um compilador para uma linguagem de Programação Orientada a Objetos*. Dissertação de Mestrado. DCC, Unicamp (julho 1991).

[Drummond 88] Drummond, R. e da Silva, F. *Linguagem C_m: Manual de Referência*. Anais da IV Reunião de Trabalho do Projeto ESTRA, SID Infomática (outubro 1988), pp. 175-210. Publicado também como Relatório Interno DCC, Unicamp (maio 1988).

[KR 88] Kernighan, Ritchie. The C Programming Language, 2nd, 1988.

[Drummond 87] Drummond, R. e Liesenberg, H. *A-HAND: Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados*. Anais da IV Encontro de Trabalhos do Projeto ETHOS, Petrópolis, RJ (abril 1987), pp. 313-322. Publicado também como Relatório Interno DCC, Unicamp (outubro 1987).