Departamento de Ciência da Computação
Instituto de Matemática, Estatística e Ciência da Computação
Universidade Estadual de Campinas

A linguagem de programação Cm

Alexandre Prado Teles

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Alexandre Prado Teles e aprovada pela comissão julgadora.

Prof. Dr. Rogério Drummond Burnier Pessôa de Mello Filho

Cidade Universitária Zeferino Vaz, Campinas, 10 de novembro de 1993

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, da Universidade Estadual de Campinas, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

A linguagem de programação Cm

Alexandre Prado Teles

DCC – IMECC – UNICAMP

A linguagem de programação Cm

Alexandre Prado Teles

DCC – IMECC – UNICAMP

A Tatyana e Gian

Agradecimentos

Os primeiros agradecimentos são para Tatyana Murer Cavalcante, minha companheira e principal responsável pela minha decisão de ingressar no programa de mestrado, que esteve presente com seu apoio e incentivo todas as vezes que precisei.

O meu orientador, Prof. Dr. Rogério Drummond, é o criador de Cm, e sem o seu apoio (não só acadêmico) e amizade, este trabalho provavelmente nunca seria concluído.

Todos os integrantes do Laboratório A_HAND, de uma forma ou de outra, contribuíram para esta tese. A gratidão é para todos em geral, e particularmente para Carlos Furuti, Cassius Di Cianni, Celso Gonçalves Jr., Marcelo Souza, Maurício França Jr. e Wanderley Ceschim.

Resumo

Cm é uma linguagem de programação em evolução. Ela surgiu em 1985, no DCC-Unicamp, e vem se desenvolvendo desde então. Atualmente, Cm apresenta características comparáveis com de C++ para programação orientada a objetos e caminha para além disso, incorporando facilidades para programação concorrente e distribuída.

Esta tese de mestrado descreve o estágio atual da linguagem de programação Cm e seu compilador, ressaltando as contribuições do autor em ambos. Inicialmente a tese incorporava o manual de referência da linguagem, mas, devido ao volume final, este foi removido e publicado em separado. Para uma referência completa de Cm, o leitor deve consultar também seu manual de referência (vide referências na própria tese).

Descrevendo brevemente, Cm é uma extensão de C com facilidades para programação modular e orientada a objetos. Ela preserva a flexibilidade e estrutura de comandos de C, adicionando uniformidade e verificação forte de tipos, encapsulamento de dados, polimorfismo paramétrico, herança múltipla, sobrecarga de operadores e funções e tratamento de exceções. As características de Cm são comparáveis às de C++, mas seus recursos inovadores foram introduzidos de forma mais clara e coerente, evitando ainda diversas inconveniências oriundas de C. Seu compilador analisa automaticamente as relações de dependência entre módulos necessárias ao processo de *make*, gerenciando projetos complexos sem necessidade de intervenção do programador.

Abstract

Since its first definition (1985, Computer Science Department of Unicamp) the Cm programming language has experienced continuous evolution.

Cm compares favorably with C++ about object-oriented features; it is also about to support concurrent and distributed programming.

This thesis describes the current state of the Cm programming language and its compiler emphasizing the author's contribution on both. For a complete language understanding please refer to the Cm Reference Manual, included in the first edition os this work but now published separately due to its bulk.

Cm is briefly described as an extension of C supporting modular and object-oriented programming. Cm presrves its ancestor's flexibility and command structure while adding uniformity and strong type checking, data encapsulation, parametric polymorphism, multiple inheritance, operator/function overloading and exception handling. Although somewhat similar to C++, Cm combines new features in a clearer and more coherent approach while avoiding several C inconsistencies. Also, the Cm compiler automatically analyzes modules in a comlpex project, so the programmer does not need to study dependency relations or write a makefile.

A linguagem de programação Cm

Alexandre Prado Teles

Dissertação de Tese de Mestrado
DCC - IMECC - UNICAMP

Projeto A_HAND
DCC - IMECC - UNICAMP

Laboratório de Pesquisas
Projeto A_HAND
Rua Angelo Vicentin, 226
13084-230 Campinas, SP
Brasil

Fone: +55 (192) 39-4685
Fax: +55 (192) 39-5693
e-mail: ahand-request@dcc.unicamp.br

Índice

Parte I
A linguagem Cm

Capítulo 1 — Introdução

Capítulo 2 — Visão geral da linguagem de programação Cm

Parte II
O compilador Cm

Capítulo 4 Funcionamento macroscópico do compilador Cm

Capítulo 5 Arquitetura do compilador

Capítulo 6 Geração de código

Parte III
Surveys e extensões futuras

Capítulo 7 Survey: Mecanismos de Tratamento de Exceções

Capítulo 8 Survey: Produção de Software

Capítulo 9 Extensões futuras (compilação e geração de código)

Capítulo 10 Extensões Futuras à Linguagem Cm

Parte IV
Apêndices

Lista de Tabelas

Lista de Figuras

I

A linguagem Cm

Capítulo 1

Introdução

A linguagem de programação Cm foi definida e desenvolvida pelo Projeto A_HAND com o objetivo de facilitar a produção de grandes programas por equipes de programadores relativamente autônomas, enfatizando o projeto modular e reaproveitamento de código. Ela é parte do ambiente A_HAND – Ambiente de desenvolvimento de *software* baseado em Hierarquias de Abstração em Níveis Diferenciados (**[AHand87]**, **[AHand87b]**).

1.1 Requisitos do ambiente A_HAND

A linguagem de programação adotada por um dado ambiente de programação exerce a maior influência na produtividade e eficiência do processo de desenvolvimento. O ambiente e a linguagem adotados devem apresentar características favoráveis como:

- **modularidade**: Deve ser possível uma organização hierárquica dos programas em módulos com inter-relacionamento reduzido e interfaces bem definidas, facilitando o desenvolvimento por equipes relativamente autônomas.
- **versatilidade**: O ambiente e linguagem devem ser genéricos o suficiente para permitir a construção de programas para diversas áreas de aplicação com rapidez e segurança .
- **flexibilidade**: O produto final deve se prestar facilmente a extensões e atualizações, e ser de fácil manutenção.
- **portabilidade**: Os programas devem ser executáveis em diferentes arquiteturas e sistemas operacionais, apenas com mudanças mínimas. As bibliotecas de suporte a execução e o compilador da linguagem também devem ser portáveis.
- **confiabilidade**: Os programas produzidos devem ser confiáveis, podendo se recuperar de eventuais erros na medida do possível.
- **eficiência**: O tempo de desenvolvimento deve ser minimizado, pois representa o maior custo no desenvolvimento de *software*. Ainda, o ganho na produtividade

não deve ser anulado por uma queda significativa no desempenho dos programas (em relação ao mesmo programa escrito em C, por exemplo).

- **prototipagem**: A geração de protótipos deve ser fácil, pois eles permitem o ganho de experiência no problema abordado e o refinamento da especificação do programa.

Ainda, certas características foram consideradas fundamentais para a linguagem adotada pelo A_HAND, tais como:

- **verificação de tipos**: a linguagem deve prover verificação forte de tipos, em tempo de compilação.
- **uniformidade de tipos**: todos os tipos (padrão ou definidos pelo usuário) devem possuir um conjunto mínimo de operações/propriedades comuns, como comparação e cópia.
- **reaproveitamento de código**: a linguagem deve prover mecanismos para utilização de tipos de dados polimórficos e herança para reaproveitamento de código.

Em torno de 1986, início do projeto A_HAND, as linguagens disponíveis para uso geral (C, C++, Modula-2 e Ada) não atendiam convenientemente estas exigências, motivando o desenvolvimento da linguagem Cm e de seu compilador. A versão atual de C++ (versão 3.0, dada em **[Cm91]**) poderia ser utilizada como linguagem do projeto.

1.2 O ambiente A_HAND

A_HAND é um ambiente de desenvolvimento de sistemas de *software* de grande porte, por equipes de programadores relativamente autônomas.

Dificilmente um sistema grande e complexo é monolítico. Em geral ele é composto por diversas partes, inter-relacionadas de alguma maneira. No A_HAND, o sistema e suas partes são representados de maneira homogênea como objetos do ambiente.

Objetos do ambiente podem ser *simples* – representando "entidades" como programas fonte, executáveis ou em linguagem de comando (*shell–scripts*) – ou *compostos* – representando conjuntos de outros objetos. Objetos compostos abstraem a forma como seus elementos se inter-relacionam, permitindo que o conjunto todo seja manipulado como um único objeto.

Dessa forma, um dado objeto pode ser composto por outros e ainda ser um componente de um terceiro, resultando em uma estrutura hierárquica, visualizada em diferentes níveis de abstração, com maior ou menor detalhamento.

Cm é a linguagem de programação básica do ambiente A_HAND, e enfatiza a modularidade e reutilização de código. Cada módulo Cm é chamado classe, e pode ser parametrizado polimórfico, ou ambos. Classes podem ser definidas em função de outras por derivação/especialização (herança) ou abstração. A estruturação das classes de um programa é semelhante à dos objetos de um sistema.

Além da linguagem de programação Cm, o ambiente provê duas linguagens de comandos, uma gráfica (LegoShell) e uma textual (CO_2). A LegoShell é a interface básica do usuário com o sistema.

1.3 A linguagem Cm

A linguagem Cm é derivada da linguagem C e em essência suporta um novo tipo: classe. Seu objetivo é facilitar a produção de grandes programas pelo projeto modular e reaproveitamento de código. Cm preserva a flexibilidade e a estrutura de comandos de C, adicionando verificação de tipos, encapsulamento de dados, polimorfismo paramétrico e herança múltipla, mantendo ligação estática.

Cm apresenta alta uniformidade de tipos: objetos de qualquer tipo compatível, mesmo estruturado, podem ser comparados, copiados, passados por parâmetro e retornados por funções. Tanto a passagem de parâmetros como o retorno de funções podem ser feitos por referência ou valor. Operadores podem sofrer sobrecarga (*overloading*), tornando ainda maior a uniformidade de tipos.

Expressões e parâmetros de funções sofrem verificação de tipos rigorosa. Enumerados são suportados como subtipos.

Classes introduzem na linguagem suporte a programação modular, abstração de dados e programação orientada a objetos. Cada classe define constantes, tipos, variáveis e funções, públicos ou privados. Cada objeto em Cm com tipo derivado de uma classe contém versões particulares dos dados declarados na classe e usa as funções para prover acesso a elas. Classes podem ser parametrizadas, e seus parâmetros podem ser tipos, permitindo a definição de classes polimórficas. Cada possível conjunto de valores para os parâmetros de classe definem um novo tipo, chamado de instância da classe polimórfica. Dessa forma, classes polimórficas definem, na realidade, um número ilimitado de novos tipos. Esse polimorfismo é comparável ao oferecido por Ada.

Uma classe pode herdar todas as características de outras, e alterar e/ou adicionar novas propriedades. Uma classe pode também apenas utilizar-se de outras classes, sem contudo herdar suas características. Programas em Cm constituem uma hierarquia complexa de classes.

Cm apresenta um mecanismo de tratamento de exceções semelhante ao de C++ (equivalente em poder, mas sintaticamente mais simples). Exceções podem ser sinalizadas (geradas) e tratadas pelo programador. Podem também carregar informações (um objeto de qualquer tipo), que permitem um tratamento mais seguro e elaborado. Exceções são propagadas automaticamente por vários níveis. Exceções não tratadas provocam interrupção da execução do programa, sinalizando o valor e ponto de ocorrência da exceção.

Ainda assim, Cm mantém grande semelhança com a linguagem C, apresentando todos os seus operadores, comandos e tipos padrão (pré-definidos), facilitando, assim, a migração de programadores e porte de programas de C para Cm.

1.3.1 Plataformas e ambiente de execução

O compilador Cm produz código C padrão (pré-ANSI, ou K&R), a ser processado por um compilador convencional. Isso torna Cm disponível numa ampla variedade de arquiteturas e sistemas operacionais (potencialmente, em qualquer uma que disponha de um compilador C). O compilador Cm verifica automaticamente dependências entre arquivos, reprocessando apenas os arquivos necessários. O compilador Cm também gera *makefiles* completos para manutenção dos programas C gerados, dispensando a criação e manutenção desses arquivos por parte do usuário.

A versão atual do compilador Cm foi desenvolvida em ambiente Unix (SunOS), porém é portável o suficiente para operar em outros sistemas Unix (Solaris, SCO) e em MS-DOS/PC-DOS (IBM-PC).

1.3.2 Evolução da linguagem

Os primórdios da linguagem Cm remontam a 1986, início do Projeto A_HAND. Nessa época Cm já apresentava polimorfismo paramétrico, herança múltipla e portas de comunicação de dados, sendo citada em **[AHand87]** e **[AHand87b]**. Porém a primeira definição formal da linguagem foi publicada somente em 1988 (**[Cm88]** e **[Cm88b]**).

O primeiro compilador Cm foi implementado em 1991, como parte da tese de mestrado de C. A. Furuti (**[Cm91]** e **[Cm91b]**). A definição da linguagem foi ligeiramente modificada, introduzindo algumas simplificações para torná-la mais independente do ambiente A_HAND.

Este trabalho apresenta uma nova versão da linguagem (2.0x). Esta terceira versão introduz recursos novos e resgata parte da definição inicial da linguagem na medida que bibliotecas de suporte de *run-time* estão sendo desenvolvidas no Projeto (vide **[linear92]**, **[OMNI92]**).

1.3.2.1 Versão 1986

A primeira definição da linguagem apresentava:

- tipos básicos, operadores, expressões e comandos de C, com mesma sintaxe e semântica;
- construtores de tipos de C (exceto por **class**), com mesma semântica que C, mas com sintaxe redefinida de forma a prover uma forma canônica de se especificar cada tipo e tornar as especificações mais claras e simples;
- suporte parcial a abstração de dados, permitindo estabelecimento de interfaces para classes, mascarando detalhes de implementação;
- mecanismos de herança simples e múltipla;
- polimorfismo paramétrico;
- verificação rigorosa de tipos;
- uniformidade de tipos, inclusive para tipos definidos pelo usuário (objetos de qualquer tipo podem ser comparados, atribuídos, passados por parâmetros de funções ou retornados por elas);
- declaração de funções estendida, incluindo tipos e, opcionalmente, valores *default* de seus parâmetros;

- definição de constantes;
- definição de portas de dados, para comunicação entre programas.

Essa definição requisitava um ambiente elaborado para configuração e execução de programas. Este ambiente permitiria a interligação de vários programas através de suas portas e execução de qualquer função exportável de cada programa.

1.3.2.2 Versão 1991 (Cm 1.0x)

Esta definição da linguagem sofreu poucas modificações, que visaram basicamente consolidar a semântica da linguagem e permitir sua utilização independentemente do ambiente A_HAND[1]. Essas modificações basicamente são dadas por:

- alterações sintáticas de menor importância;
- detalhamento mais completo da semântica da linguagem (principalmente com relação a pontos abertos ou obscuros em C, como a semântica dos operadores **sizeof** e '**%**');
- simplificação da interação dos programas gerados com o ambiente.

O suporte exigido do ambiente/*run-time* da linguagem é mínimo. A chamada a funções da classe foi restrita a uma única função, executada automaticamente pelo *run-time* (como a função *main* de C). Esta função é executada sempre que qualquer objeto do tipo da classe é criado, cumprindo o papel de um construtor não parametrizado. Também não foi provido suporte a portas de comunicação.

1.3.2.3 Versão 1993 (Cm 2.0x)

A versão corrente de Cm compreende as definições anteriores, provendo maior flexibilidade e segurança, bem como maior produtividade no uso da linguagem. O compilador Cm foi estendido para reconhecer a nova versão da linguagem. O resultado da compilação ainda é código fonte C (K&R).

As características introduzidas/estendidas na linguagem/compilador são:

- mecanismo de tratamentos de exceções;
- sobrecarga de operadores;
- sobrecarga de funções;
- suporte a programação modular (introdução de iniciadores e finalizadores de classe);
- suporte a tipos abstratos de dados (construtores e destrutores de objetos de uma classe);
- construtor de tipo referência, que pode ser usado para passagem e retorno de valores por referência;
- iniciadores de dados mais poderosos;
- sintaxe mais amigável e versátil para alguns comandos (**for** e **switch**);
- introdução de novos operadores, para completude ('**&&=**', '**||=**', etc) ou abreviação (short-cut) de comandos ('**->=**');

1. Evidentemente, a utilização de Cm fora do ambiente A_HAND restringe a utilização de alguns dos recursos da linguagem (como portas de comunicação).

- introdução de funções **inline** e constantes.

1.4 Trabalho realizado

O trabalho realizado no programa de mestrado pode ser dividido em três etapas principais. A primeira delas cobriu a análise e entendimento do compilador Cm existente em 1991; o estudo de outras linguagens (principalmente C++) e de vários mecanismos de tratamento de exceções (a primeira extensão prevista para ser realizada).

Em seguida, foram definidas as extensões a serem introduzidas na linguagem (descritas anteriormente na seção 1.3.2.3). Nesta estapa foi implementado um mecanismo de tratamento de exceções em C, inteiramente definido através de macros (pré-processamento), não exigindo nenhuma alteração do compilador C. Dessa forma, ele era bastante suscetível a erros de programação, mas de fácil implementação, permitindo a experimentação de algumas alternativas de implementação do mecanismo definitivo de Cm.

A fase final consistiu na implementação das extensões de Cm e da redação da presente tese de mestrado e do manual de referência comentado da linguagem Cm. A versão anterior do compilador compilador Cm foi bastante modificada, tanto para introduzir as extensões como para otimizar o código gerado.

1.5 Organização geral da tese

Esta tese é dividida em quatro partes:

1. Apresentação da linguagem Cm
2. O compilador Cm
3. *Surveys* e extensões futuras

O leitor casual interessado apenas em conhecer a linguagem Cm pode ler somente a primeira parte da tese. Em particular o capítulo 2, "Visão geral da linguagem de programação Cm", procura ser auto-contido, servindo como um artigo para apresentação da linguagem. Os demais capítulos descrevem melhor as características da linguagem através de exemplos.

A segunda parte trata do compilador Cm em si, descrevendo sua estrutura interna, suas bibliotecas de suporte à execução, e a forma de tradução do código Cm para C.

A parte final contém pequenos *surveys* sobre assuntos estudados; possíveis extensões futuras à linguagem, compilador e ambiente de execução e uma avaliação geral dos resultados obtidos (conclusões). Muitas das extensões evitadas já estão sendo realizadas em trabalhos paralelos e, em futuro próximo (final de 1994), permitirão o uso de Cm para desenvolvimento de sistemas distribuídos em Unix.

A tese contém alguns apêndices, que incluem exemplos de classes Cm e código gerado para elas, referências bibliográficas (parcialmente comentada) e *benchmarks* de Cm (comparando com C e C++).

Manual de referência

O manual de referência comentado de Cm foi escrito inicialmente como parte da dissertação de mestrado. Porém o volume final da tese forçou que o manual de referência fosse publicado separadamente da tese em si.

Uma avaliação completa da linguagem e do trabalho desenvolvido deve ser baseada nas duas publicações.

Capítulo 2

Visão geral da linguagem de programação Cm

Este capítulo descreve brevemente a linguagem de programação Cm através de exemplos e comparações com as linguagens C e C++. A apresentação formal da linguagem é feita nos capítulos subseqüentes.

2.1 Classes e meta-classes

Classes correspondem ao recurso mais complexo e importante de Cm. É através delas que a linguagem provê suporte a programação modular, abstração de dados e orientação a objetos, de uma maneira simples e segura.

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 classes (uma para cada conjunto de possíveis valores de seus parâmetros).

Meta-classes não parametrizadas definem um único tipo classe. Meta-classes que têm especificações de tipo por parâmetros são chamadas de *polimórficas*.

A definição de uma meta-classe ocupa um escopo global no "ambiente" Cm, compartilhado por todas as definições de meta-classes, podendo ser usada diretamente em outras meta-classes, bastando que seja importada ou herdada por elas. O construtor de tipos definido pela meta-classe é representado pelo seu nome, ou seja, os nomes das meta-classes são usados como operadores de construção dos respectivos tipos classe. Os nomes de todas as meta-classes ocupam um mesmo escopo no "ambiente Cm", e devem ser distintos entre si.

Uma meta-classe não parametrizada especifica um único tipo classe (obtido pela aplicação da meta-classe à lista de parâmetros vazia). Dessa forma, a distinção entre classes e meta-classes somente é evidente em classes parametrizadas. Por conveniência, o termo "classe" será usado significando "classe" ou "meta-classe",

indistintamente, a não ser que a distinção não seja evidente e/ou o texto possa levar a interpretações incorretas.

2.1.1 Tipos classe e Interface

Um *tipo* é uma representação concreta de uma idéia ou conceito, que engloba tanto a informação necessária para representar essa idéia como as operações que podem ser executadas sobre ela. Por exemplo, o tipo **int** representa o conceito matemático de números inteiros.

Tipos classe podem ser definidos para prover uma representação concreta e específica de uma idéia que não se enquadra propriamente em nenhum dos tipos padrão (pré-definidos) da linguagem. Ainda mais, classes podem ser definidas de forma que objetos de seu tipo sejam utilizados praticamente da mesma maneira que tipos padrão (a distinção somente é evidente na iniciação de objetos de tipo classe).

A informação representada e as operações possíveis sobre ela constituem a *interface* da classe, que deve ser conhecida publicamente. Já a implementação ou organização da classe é um mero detalhe, alheio ao conceito que ela representa, baseado em fatores como performance e memória utilizada, e que somente interessa a quem for implementá-la.

Em Cm a interface e implementação de uma classe são bem isoladas (apesar de estarem mescladas em um mesmo arquivo). Sua interface é representada por um subconjunto das declarações utilizadas em sua implementação. Mais precisamente, somente as declarações indicadas como exportadas pertencem à sua interface, e somente elas podem ser utilizadas por outras classes através do mecanismo de importação.

Assim, os detalhes de implementação ficam "encapsulados" na classe, tornando-a bastante independente das demais, e permitindo que seja até totalmente reimplementada sem afetar as demais classes que a importam, mantida a sua interface.

2.1.2 Hierarquia de importação e herança

Uma classe pode se utilizar de outras classes já existentes pelos mecanismos de importação e herança.

A *importação* de uma meta-classe torna seu nome – o construtor de tipos que ela define – conhecido na classe importadora, permitindo que ele seja utilizado para especificar tipos classe (da meta-classe importada) em declarações e expressões (da meta-classe importadora).

A *herança* de uma classe "incorpora" a classe herdada à classe herdeira, ou seja, a classe herdeira passa a ter todos os dados, tipos e funções da classe herdada, como se tivessem sido declarados diretamente nela. Este mecanismo permite a definição de classes a partir de outras, apenas acrescentando-se ou alterando-se partes da definição original. A herança de uma classe implica na importação da meta-classe correspondente, que pode então ser usada para especificar novos tipos classe.

As relações de importação e herança estabelecem uma organização hierárquica de classes, que pode ser utilizada pelo programador (o mecanismo de compatibilidade de tipos permite que funções escritas para classes bases podem atuar sobre classes derivadas, por exemplo, permitindo a escrita de funções "genéricas").

2.1.3 Arquivos de definição de meta-classes

Cada meta-classe é definida em um arquivo próprio, que contém todas as informações relativas a ela:

- hierarquias de importação e herança;
- declarações de tipos;
- declarações de dados (constantes e variáveis);
- declarações de funções (comuns, construtores, destrutores e operadores);
- rotina de validação da classe.

A definição de uma meta-classe é iniciada pela palavra reservada **class**, seguida de seu nome, um identificador.

Os parâmetros de uma classe são dados através de uma lista de declarações separadas por '**;**'. Essa lista é delimitada por '**<**' e '**>**', e pode ser vazia. Cada declaração pode introduzir parâmetros comuns ou parâmetros de tipos (**type**), como em:

```
class Empty <>
class Stack <type Element; int size>
class List <type NodeInfo>
```

Cada parâmetro pode ter um valor *default* (usado caso o correspondente valor não seja especificado na utilização da classe), como em

```
class Stack <type Element = int; int size = 1024>
```

Caso a classe não tenha parâmetros ou caso todos os parâmetros tenham valor *default*, então a lista de parâmetros pode ser omitida por completo na especificação de um tipo classe. Parâmetros do final da lista que tenham o valor *default* também podem ser omitidos. Assim, as seguintes especificações são equivalentes:

```
Stack <int, 1024>
Stack <int>
Stack <>
Stack
```

importação

A especificação de importações é opcional, e é dada pela palavra **import** seguida pela lista de meta-classes importadas (identificadores).

herança

A especificação de herança é opcional (para o caso de não se desejar herança), e é dada pela palavra reservada **inherit** seguida pela lista de meta-classes herdadas (especificações de classe).

A especificação do nome, parâmetros e das relações de importação e herança constituem o *preâmbulo* da classe.

declarações
As declarações introduzem as constantes, variáveis, funções e tipos utilizadas pela classe.

2.1.4 Compilação e execução

Toda classe Cm pode ser compilada e executada como um programa, exceto se contiver funções virtuais não definidas. A execução de uma classe se dá pela criação e destruição de um objeto estático do tipo a "ser executado", o que induz a execução dos construtores e destrutores associados à classe.

2.2 O exemplo clássico

Um dos exemplos mais clássicos na apresentação de linguagens de programação é um programa que escreve a mensagem "Hello, World!". Em Cm, esse programa poderia ser escrito assim:

```
class Hello<>
import Output;
static constructor
{
   Output out;
   out << "Hello, World!";
}
```

Mesmo este programa simples define uma meta-classe `Hello` que, por sua vez, define uma classe Hello<>, cuja função de iniciação escreve a mensagem. Esta classe tem pouca utilidade como um tipo uma vez que a mensagem é escrita na função de iniciação, que é executada uma única vez, na fase de iniciação do programa (ou seja, foge ao controle do programador); além disso, não exporta dados, funções ou tipos.

Uma versão mais útil deve prover meios para outras classes realizarem a árdua tarefa de escrever esta mensagem sem reescrever todos os seus comandos. Isso pode ser feito provendo uma função `print`, exportada, que realiza todo o serviço:

```
class Hello<>
import Output;

export void print()
{
   Output out;
   out << "Hello, World!";
}
```

Agora, uma outra classe qualquer pode importar a classe `Hello` e executar sua função `print`, como em

```
class UseHello<>
import Hello;
   ...
   Hello aHello;
   aHello.print ();
   ...
```

2.3 Expressões

Expressões são o mecanismo básico da linguagem para se especificar computações. É através delas que se efetuam os cálculos aritméticos e *booleanos* (comparações e testes), bem como acesso a dados e chamadas de funções.

Expressões são construídas pela aplicação de operadores da linguagem a outras expressões. Expressões básicas são identificadores e literais. Cm apresenta todos os operadores de C e praticamente todos os de C++[1], e acrescenta operadores lógicos com atribuição ('^^=', '&&=', '||='), ou-exclusivo lógico ('^^'), derreferênciação com seleção e atribuição ('->='), especificador de intervalos ('..'), construtor de valores ('@'), sinal positivo unário ('+') , operadores de escopo ('::') e operadores de alocação e liberação de memória (**new** e **release**)[2].

A tabela 2–1 sumariza os operadores de Cm, agrupados de acordo com sua funcionali-dade. Convém ressaltar que o operador de intervalo ('..') é utilizado somente na especificação de intervalos em cláusulas **case** do comando **switch**. Os operadores '+', '-', '*' e '&' podem ser unários ou binários e os operadores '++' e '--' podem ser prefixados ou posfixados. O operador correto é determinado pelo contexto em que é usado.

As expressões mais simples são dadas por valores literais e identificadores. Valores literais podem ser caracteres, inteiros, reais (possivelmente em notação científica) e *strings*, como em

```
'a'     '5'     '#'
123     1024    389956
3.1416  .975    .3526e-11
"O elefante tem quatro patas."
```

Identificadores podem representar variáveis, constantes, tipos ou funcões. Os identificadores **NIL**, **self** e **result** têm semântica especial.

NIL representa um apontador genérico de "valor nulo" (que não aponta para objeto algum).

self representa o objeto associado a funções dinâmicas durante a execução do programa.

1. Cm somente não implementa os operadores de *member selection* de C++ ('.*' e '->*').
2. C++ também possui os quatro últimos operadores (**release** é denominado **delete** em C++).

TABELA 2-1 Operadores de Cm

atribuição		= += -= *= /= %= &&= ^^= ||= &= ^= |= >>= <<= ->=
seqüenciamento		,
chamada de funções		()
relacionais		< <= > >= == !=
booleanos		&& ^^ || !
aritméticos	**operações algébricas**	+ - * / %
	incremento/decremento	++ --
manipulação de bits	**bit-a-bit**	& ^ | ~
	deslocamento	>> <<
acesso a membros	**seleção de escopo**	::
	indexação	[]
	seleção	-> .
endereçamento e memória dinâmica	**referência**	&
	derreferência	*
	alocação	new release
tipos	***casting* (converão)**	(type)
	tamanho	sizeof
construção de valores		@
condicional ou seleção ternária		?:
especificação de intervalos		..

result representa o valor que será retornado pela função, caso seu tipo de retorno não seja **void**. É particularmente útil caso o tipo de retorno seja estruturado.

A função `copy` do exemplo a seguir retorna uma cópia (**result**) do objeto a partir do qual foi ativada (**self**).

```
class X<>
int a,b,c;

X copy()
{
   result.a = self.a;
   result.b = self.b;
   result.c = self.c;
}
```

A sintaxe de expressões segue a linha de C e C++, como por exemplo em

```
out.format ("Media ponderada = %f\n")
     << ((a * 3 + b * 5 + c * 8) / 16);

d = Math :: sqrt (h * h + w * w);

while (*p)
   *q++ = *p++;

anywhere :: do_something ();
```

Os operadores novos também seguem a mesma linha, tornando seu uso bastante intuitivo para quem já está habituado com os operadores convencionais, como em

```
all_on = TRUE;
not_all_off = FALSE;
for (Node *node = head; node != NIL; node ->= next)
{
   all_on &&= node -> flag;
   not_all_off ||= node -> flag;
}
```

2.4 Comandos

Cm apresenta um conjunto pequeno de comandos, listados na tabela 2–2, que controlam o fluxo de execução do programa. Basicamente, comandos determinam a seqüência em que expressões são avaliadas durante a execução de um programa.

Os comandos em Cm são um superconjunto dos comandos de C, e são equivalentes a eles, exceto por extensões feitas aos comandos **for**, **return** e **switch** e pelos comandos de tratamento de exceções. A diferença com relação aos comandos de C++ é basicamente sintática (tratamento de exceções e comando **switch**).

Em Cm, o comando **switch** permite seqüências e intervalos nas cláusulas **case** e garante a inicialização de variáveis locais do comando.

```
switch (ch)
{
   case 'a'..'z', 'A'..'Z':
      return LETTER;
   case '0'..'9':
      return DIGIT;
   case ' ', '\n', '\t':
      return WHITE_SPACE;
   default:
      return OTHERS;
}
```

TABELA 2-2 Comandos de Cm

comando de avaliação de expressão	*expressão*;
comando composto	{ *declarações comandos* }
comando protegido	[*declarações comandos*]
comandos de decisão	**if** (*expressão*) *comando*
	if (*expressão*) *comando* **else** *comando*
	switch (*expressão*) *comando*
comandos de iteração	**while** (*expressão*) *comando*
	do *comando* **while** (*expressão*);
	for (*expressão*; *expressão*; *expressão*) *comando*
	for (*declaração*; *expressão*; *expressão*) *comando*
comandos de desvio	**break**;
	continue;
	goto identificador;
	return;
	raise *expressão*;
comandos rotulados	**case** *expressão*: *comando*
	default: *comando*
	identificador: *comando*
	when *tipo* identificador: *comando*
	when others: *comando*

Note que qualquer comando pode ser associado ao **switch**, mas é usual que sejam associados apenas comandos compostos ou protegidos. Não é necessário que os comandos com *label* **case** estejam no nível léxico mais externo do comando composto, como em

```
if (count)
{
   int n = (count + 7) / 8;
   switch (count % 8)
   {
      case 0:do
             {  *to++ = *from++;
      case 7:   *to++ = *from++;
      case 6:   *to++ = *from++;
      case 5:   *to++ = *from++;
      case 4:   *to++ = *from++;
      case 3:   *to++ = *from++;
      case 2:   *to++ = *from++;
      case 1:   *to++ = *from++;
```

```
            } while (--n > 0);
        }
    }
```

Este algoritmo, conhecido como "*Duff's device*", copia `count` objetos da região de memória apontada por `from` para a apontada por `to`. Apesar de eficiente, esse tipo de código é de difícil compreensão, e deve ser evitado, ou pelo menos muito bem documentado.

O comando **for** foi estendido, como em C++, para permitir a declaração de variáveis de controle (contadores, índices, temporários, etc) no próprio comando. Ao contrário de C++, o escopo dessas variáveis é restrito ao comando em que são declaradas.

```
for (int i = 0; i<5; i++)
  out << "Square of " << i << " is " << i*i << '\n';

for (Node *node = head, previous = NIL;
     node; previous = node, node ->= next)
  scranchize_nodes (previous, node);
```

As extensões ao comando **return** são basicamente semânticas. Caso o valor de **result** seja estabelecido, então o comando **return** sem valor explícito é entendido como retornando **result**. Ao final da função, o comando **return** implícito também retorna **result**. Por exemplo, as duas funções a seguir são semanticamente equivalentes:

```
int f1()
{
  int val = value ();
  if (test ())
    return val;
  val += adjust ();
  return val;
}

int f2()
{
  result = value ();
  if (test ())
    return;
  result += adjust ();
}
```

Os comandos de tratamento de exceção apresentam sintaxe mais amigável que em C++, ressaltando o escopo de atuação dos tratadores de exceção.

```
[
  try_to_do_it();
  when int errno:
    err << "Error number " << errno << "!\n";
```

```
    when char *errmsg:
      err << "Error " << errmsg << '\n';
    when others:
      err << "Unknown error!\n";
}
```

Devido à versatilidade das expressões da linguagem, o corpo de um comando iterativo pode ser vazio, ou seja, sem nenhuma tarefa a executar. Nesses casos utiliza-se o comando nulo.

```
while (*to++ = *from++) ;

for (int i = 0; i < 10; do_it (i++)) ;
```

Note que o comando **if** apresenta o problema clássico de ambigüidade (**else** pendente). A solução é a (também clássica) adotada por C e C++: o **else** sempre se refere ao último comando **if** ainda não completado (o mais interno). Algumas vezes é necessário se recorrer a comandos compostos ou nulos para alterar esse comportamento. Por exemplo, no comando

```
if (t1)
  if (t2)
    if_foo ();
else
  else_foo ();      // else relativo a if (t2)
```

o **else** se refere ao segundo **if**, apesar da formatação sugerir que ele se refere ao primeiro **if**. Os exemplos a seguir mostram alternativas de codificação que associam o **else** ao primeiro **if**:

```
if (t1)
  if (t2)
    if_foo();
  else
    ;               // else (nulo) relativo a (t2)
else
  else_foo ();      // else relativo a if (t1)

if (t1)
{
  if (t2)
    if_foo();       // if sem else, devido ao bloco
}
else
  else_foo ();      // else relativo a if (t1)
```

Os comandos composto e protegido introduzem um bloco (com novo nível léxico) no programa, e permitem declarações de dados e funções[1] desse bloco.

```
{
  int a, b; int* c;
```

```
    ...
  a = b;
  c[0] = 24;
  {
    int a = c[b - 4];
    c[b] = a * a;
  }
}
```

2.5 Declarações

A sintaxe geral de declarações de dados (variáveis e constantes) em Cm é a mesma que a de C e C++:

classe-de-armazenamento especificação-de-tipo lista-de-identificadores;

A classe de armazenamento afeta a forma com que uma variável é criada e destruída (alocação e tempo de vida), como são feitos os acessos a ela e sua visibilidade, se é referenciável e se é constante, e sua visibilidade. As classes de armazenamento de Cm são **auto**, **const**, **export**, **extern**, **inline**, **global**, **register**, **static** e **virtual**.

A especificação de tipo indica o tipo do dado, ou seja, o conjunto dos possíveis valores que ele pode conter/representar.

A lista de identificadores especifica os nomes das variáveis ou constantes sendo declaradas. Para cada identificador pode-se especificar o valor inicial da variável (ou seja, o valor que ela deve conter assim que for criada). O valor inicial é obrigatório no caso de constantes.

A sintaxe geral de declarações de funções também é a mesma que a de C e C++:

classe-de-armazenamento especificação-de-tipo-de-retorno
    (declarações-de-parâmetros)
comando-composto-ou-protegido

Tipos são declarados em cláusulas específicas (**type**), não seguindo a filosofia de C e C++ onde tipos são declarados como variáveis com classe de armazenamento **typedef**.

type identificador_1 = $\text{especificação-de-tipo}_1$,
...
identificador_n = $\text{especificação-de-tipo}_n$;

como, por exemplo, em

```
type RGB = enum { RED, GREEN, BLUE };
```

1. Somente são permitidas declarações de funções externas e globais, restringindo sua visibilidade. Não é permitido aninhamento de funções em Cm.

```
type name = char [50],
     address = char [100],
     cep = long,
     item = struct
            {
              name n; address a; cep c;
            };
```

Dessa forma, declarações de tipos, dados e funções se tornam mais distintas, facilitando a leitura e manutenção do código.

Todos os tipos, dados e funções declarados no nível léxico 0 de uma classe são chamados de componentes da classe.

2.5.1 escopo de visibilidade

A cada identificador Cm é associado um escopo de visibilidade, que é estático, e corresponde ao trecho do código onde o identificador é visível. Caso o identificador seja declarado no nível léxico 0 como exportável (**export**) seu escopo é estendido via seleção (operadores '.', '->' e '::').

O escopo de visibilidade de um identificador se inicia imediatamente após sua declaração e se estende até o final da classe (para declarações no nível léxico 0) ou até o final do bloco em que foi declarado (para declarações em outros níveis léxicos).

2.5.2 escopo de alocação

A cada identificador é também associado um escopo de alocação, dado pelas classes de armazenamento **auto**, **extern**, **global** e **static**, que determina como e onde o objeto associado ao identificador é alocado, bem como seu tempo de vida.

2.5.3 constantes

Constantes são declaradas como dados com classe de armazenamento **const**. 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.

```
const char TAB = '\t';
const int MIN_SIZE = 1024,
          MAX_SIZE = MIN_SIZE + (3 * 512);
const struct { int from, to; } RANGE =
                { MIN_SIZE, MAX_SIZE };
```

Constantes são basicamente valores: não têm alocação e não podem ser referenciadas. Todos os literais de um programa são exemplos de constantes.

2.5.4 variáveis

Variáveis podem ser alocadas estaticamente ou dinamicamente. A alocação estática se dá em tempo de carga do programa, e a existência (tempo de vida) das variáveis

alocadas dessa forma se prolonga durante toda a execução do mesmo. A alocação dinâmica ocorre em tempo de execução do programa, bem como sua destruição. A alocação e destruição dinâmica de variáveis podem ser realizadas automaticamente pelo compilador ou estar sob controle do programador (via operadores **new** e **release**).

Variáveis podem pertencer a cinco escopos de alocação: externo (especificado pela classe de armazenamento **extern**), global (**global**), de classe (**static**), de objeto (**auto**) e de bloco (**auto**). As variáveis de objeto e de bloco são dinâmicas e ditas automáticas. As variáveis dos demais escopos são estáticas.

Variáveis automáticas de bloco podem ser especificadas com classe de armazenamento **register**, que sugere ao compilador otimizar o tempo de acesso à variável.

Variáveis externas são usadas para *interface* com outras linguagens (por exemplo, com a biblioteca padrão de C). Variáveis globais são compartilhadas por todos os objetos de todas as classes que as declararam. Variáveis de classe são compartilhadas por todos os objetos de uma dada classe. Variáveis de objeto são particulares a um dado objeto de uma dada classe. Variáveis de bloco são particulares do bloco (comando composto ou protegido) em que são declaradas.

Por exemplo, dadas as seguintes classes:

```
class X<>                    class Y<>
extern int e;                extern int e;
global int g;                global int g;
static int s;                static int s;
int a;    // auto           int a;    // auto
void f() {                   void f() {
   int b; // auto              int b; // auto
}                            }
```

e as declarações

```
X<> x1, x2;
Y<> y1;
```

teremos, estaticamente, uma única "cópia" de `e` e `g`; duas de `s` (uma para cada classe, `X<>` e `Y<>`); três de `a` (uma para cada objeto, `x1`, `x2` e `y1`) e nenhuma de `b`. Dinamicamente, cada alocação de um objeto de tipo `X<>` ou `Y<>` criará uma nova cópia da variável `a` para o novo objeto.

Pode não haver nenhuma cópia de `b` (se nenhuma das funções `f` estiver em execução) ou uma cópia apenas (caso uma das funções esteja em execução).

2.5.5 funções

A alocação de funções é sempre estática (podendo ser externa ou global). Funções não podem ser aninhadas, podendo ser definidas somente no nível léxico 0; porém funções externas e globais podem ser declaradas em níveis léxicos maiores (em blocos), tendo visibilidade restrita.

Funções podem ser declaradas múltiplas vezes (declarações *forward*), e devem ser definidas uma única vez. A classe de armazenamento **virtual** é uma outra forma de declaração *forward*, relacionada a herança (vide seção 2.12.1, página 2-29).

As classes de armazenamento **auto** e **static** têm semântica diferenciada quando aplicadas a funções. Funções estáticas não dependem de objetos específicos para serem executadas, ou seja, não utilizam variáveis do escopo de objeto. Funções dinâmicas são associadas a um objeto em particular apenas quando são executadas. O objeto associado à função é representado por um parâmetro implícito denominado **self**, que tem tipo referência para a própria classe.

Funções podem ser declaradas com classe de armazenamento **inline**, sugerindo que sejam expandidas no momento da chamada, isto é, a chamada da função pode ser substituída por comandos equivalentes (determinado pelo corpo da função).

A classe de armazenamento **const** indica que a função não tem efeitos colaterais que alteram o "valor" do objeto **self** associado a ela, podendo ser executada mesmo para objetos constantes.

As classes de armazenamento **global** e **register** não se aplicam a funções.

2.5.6 especificação de tipos

Cm apresenta um pequeno conjunto de tipos padrão, pré-definidos, a partir dos quais pode-se especificar outros tipos mais complexos: enumerados[1], apontadores, referências, *arrays*, estruturas, uniões e classes.

A cada tipo é associado um *domínio* e um *tamanho*. O domínio representa o conjunto de todos os possíveis valores que um objeto com esse tipo pode conter. O tamanho representa a quantidade de memória usada para representar os objetos desse tipo. O tamanho é fixo e constante para cada tipo, e é medido em "unidades de memória" (*bytes*), que correspondem ao tamanho do tipo **char**. O tamanho de um tipo pode ser obtido através do operador **sizeof**.

A especificação de um tipo descreve como o tipo é construído ou estruturado a partir dos tipos padrão da linguagem.

Os construtores de tipo de Cm são: apontador (*), referência (&), *array* ([]), função (()), estruturas ou agregados (**struct**), uniões ou agregados justapostos (**union**), enumerado (**enum**) e classe (**class**). Os construtores de tipos de Cm são, exceto pelos construtores de classe e referência, os mesmos que os de C. Porém, sintaticamente, as duas linguagens são bastante diferentes: os construtores de tipos em Cm não se agregam a identificadores como em C e C++ (construtores de tipo apontador, *array* e função), resultando em uma sintaxe muito mais clara e menos propensa a erros.

1. Enumerados são um caso a parte. Eles funcionam como um subtipo de **int**, e podem até ser considerados tipos padrão.

A aplicação dos construtores é seqüencial, da direita para a esquerda. Construtores de *array* consecutivos são considerados um único construtor de *array* multidimensional. Cada dimensão de um construtor de *array* multidimensional é aplicada seqüencialmente da esquerda para a direita. Dessa forma os índices são escritos na mesma ordem tanto nas declarações como nas expressões em que são usados.

novos tipos e sinônimos

Especificações envolvendo construtores de tipo definem novos tipos que são anônimos e representados por sua estrutura. Especificações idênticas se referem ao mesmo tipo.

Assim, nas declarações

```
struct { int a; } x;
struct { int a; } y;
void f (struct { int a; } p) {...}
```

tanto as variáveis x e y como o parâmetro p têm o mesmo tipo.

Declarações **type** permitem associar identificadores a especificações de tipos, evitando a reescrita dessas especificações sempre que são utilizadas. Esses identificadores não caracterizam novos tipos; apenas são sinônimos para as especificações dadas. Por exemplo, as declarações

```
type code = int;
int i;
code x;
```

introduzem duas variáveis `i` e `x` com tipo **int**.

nomeação de novos tipos

Os tipos construídos são anônimos, identificados apenas por sua estrutura (especificação). Porém, tipos enumerados e agregados (estruturas e uniões) herdam o nome de seus sinônimos quando especificados em declarações de tipos, passando a constituir um novo tipo. Assim, nas declarações

```
type Struct1 = struct { int a; };
type Struct2 = struct { int a; };
type Struct3 = Struct1;
type Struct4 = struct { struct { int a; } b; };
struct { int a; } v0;
Struct1 v1;
Struct2 v2;
Struct3 v3;
Struct4 v4;
```

a variável `v0` e o membro `b` de `v4` têm um tipo (anônimo), `v1` e `v3` têm outro tipo (`Struct1`), `v2` tem outro tipo (`Struct2`) e `v4` tem outro tipo (`Struct4`).

2.5.7 exemplos de declarações

Declarações de variáveis simples (de tipo padrão) têm a mesma sintaxe de C e C++, como em

```
char achar = '%', anotherchar = '$';
int  oneK = 1024, twoK = 2 * oneK,
     threeK = oneK + twoK;

type byte = unsigned char;
byte b = 255, b1 = 0xAF;
```

O construtor de tipos **enum** permite definir um conjunto finito de nomes simbólicos, que são tratados como constantes. A sintaxe é idêntica à de C e C++, exceto por não se poder nomear o tipo **enum** (isso é feito através de declarações de tipos).

```
enum { VERDE, AMARELO, AZUL = 5, PRETO } cor = AZUL;

type Contrastes = { FRACO, MEDIO, FORTE };
Constrastes contraste = FRACO;
```

O tipo apontador (*pointer*) corresponde a um objeto que armazena a localização (endereço) de outro objeto. Apontadores são particularmente úteis na criação de estruturas de dados dinâmicas.

```
type ListNode = struct {
                    AType     info;
                    ListNode *next;
                };
ListNode *head = NIL;
```

O tipo referência corresponde, como um apontador, a um objeto que armazena a localização (endereço) de outro, mas é utilizado como se fosse este outro objeto, como um sinônimo (o compilador se encarrega de inserir os operadores '*' e '&' necessários).

```
int i = 10;
int &j = i;       // j é um sinônimo de i
out << j;         // imprime 10
j += 5;
out << i;         // imprime 15
```

O exemplo acima é semanticamente equivalente a

```
int i = 10;
int *j = &i;      // j é um sinônimo de i
out << (*j);      // imprime 10
(*j) += 5;
out << i;         // imprime 15
```

O tipo referência é particularmente útil na passagem e retorno de parâmetros por referência, sendo também usado na definição de sobrecarga de operadores. A função `succ` dada por

```
int & succ (int &i)
{
   i++;
   return i;
}
```

recebe um objeto inteiro por referência, incrementa-o e retorna este mesmo objeto por referência. Assim, a chamada

```
a = succ (b);
```

é semanticamente equivalente a

```
b++;
a = b;
```

Arrays consistem de uma coleção de objetos de mesmo tipo armazenados consecutivamente em memória. A cada objeto, chamado de elemento do *array*, é associado um índice no intervalo de zero (primeiro elemento) até o número de elementos menos um (último elemento).

```
type Mat10x20 = int [10][20];
Mat10x20 m2;
Mat10x20 [30] m3;

m3[4] = m2;
m2[i][j] = m3[6][i][j];
m2[3] = m2[7];
```

O tipo estrutura consiste em uma agregação de objetos de diferentes tipos, armazenados consecutivamente em memória[1]. A cada objeto, chamado de membro da estrutura, é associado um nome, distinto dos demais, usado para acesso ao objeto, via seleção. Pode-se especificar o tamanho em *bits* de membros de tipo integral, posfixando os respectivos identificadores com "**: expressão**".

```
type Book = struct
            {
               char [100]     titulo;
               char [100]     autor;
               int            ano;
               int            novo : 1,
                              importado : 1,
                              reservado : 1;
            };
```

1. Idealmente, a menos de *padding*.

O tipo união consiste em uma agregação de objetos de diferentes tipos, armazenados superpostos em memória, ou seja, todos os objetos são armazenados com início na mesma posição de memória[1]. Eles são declarados e utilizados como um estrutura, mas usando a palavra reservada **union** ao invés de **struct**.

O tipo classe é o mais complexo em Cm. Internamente a uma classe podem ser definidos constantes, variáveis, funções e tipos, que podem ou não ser visíveis externamente. A definição de uma classe deve ser feita em um arquivo próprio, e sua especificação é dada por seu nome e valores para seus parâmetros formais. Por exemplo, dada a classe

```
class Connector<type Label; int in_ports, out_ports>
   ...
```

podemos ter declarações como

```
type AndConnector = Conector<char *, 2, 1>;
type OrConnector = Conector<char *, 2, 1>;

AndConnector a1, a2, a3;
OrConnector o1, o2;
Connector<int, 2, 2> ff;
```

2.5.8 compatibilidade de tipos

Um tipo é dito compatível com outro se dados do primeiro tipo podem ser usados no lugar de dados do segundo tipo, como por exemplo em atribuições e passagem de parâmetros.

Em Cm, um tipo pode ser compatível com outro em três níveis, cada um deles especificando um certo grau de proximidade entre os tipos. Em ordem decrescente de proximidade, esse níveis são: *equivalência*, *generalização* e *conversão*.

A compatibilidade por equivalência se dá quando os dois tipos envolvidos representam exatamente o mesmo tipo, ou seja, quando são sinônimos ou têm especificações e nomes idênticos. A compatibilidade por equivalência é reflexiva, e os dois tipos são ditos equivalentes.

A compatibilidade por generalização se dá quando os tipos não são os mesmos, mas o primeiro pode ser promovido para o segundo, e o segundo tipo é considerado genérico pelo compilador Cm. A promoção é possível quando o domínio do primeiro tipo está contido no domínio do segundo tipo. Os tipos genéricos são: **int**, **double**, apontador para **void** e estruturas e uniões anônimas.

A compatibilidade por conversão se dá quando o primeiro tipo não é compatível nem por equivalência e nem por generalização com o segundo tipo, mas pode ser promovido ou projetado nele. A projeção é possível quando o domínio do primeiro

1. Idealmente, a menos de *padding*.

tipo contém o domínio do segundo tipo, sendo necessário se definir um mapeamento de um tipo no outro. Cm define projeções entre dois tipos numéricos quaisquer e de apontadores quaisquer para apontador para **void**.

Caso nenhum dos casos anteriores seja satisfeito, o primeiro tipo é dito *incompatível* com o segundo.

A verificação de tipo é *forte* em Cm, principalmente envolvendo tipos não padrão. A compatibilidade de apontadores exige equivalência dos tipos apontados. A compatibilidade de *arrays* exige equivalência dos tipos dos elementos e tamanhos dos dois *arrays*. A compatibilidade de estruturas e uniões exige a equivalência de tipos e nomes de seus membros, tomados dois a dois. O tipo referência não afeta a compatibilidade, que é determinada pela compatibilidade envolvendo os tipos referenciados. A compatibilidade de funções exige equivalência do tipo de retorno e de tipos e nomes dos parâmetros, tomados dois a dois, e também do parâmetro implícito **self**.

A compatibilidade de classes é a mais complexa, e baseia-se no mecanismo de herança. Duas classes são equivalentes se correspondem à mesma "instância" de uma dada meta-classe, ou seja, se seus nomes são idênticos e, no caso de classes parametrizadas, os valores de seus parâmetros são dois a dois equivalentes (tipos equivalentes, para parâmetros de tipo, ou valor idêntico, para parâmetros comuns).

Um tipo classe é compatível por conversão com outro se o segundo tipo é equivalente a uma das classes base do primeiro (em qualquer nível da hierarquia de herança).

2.6 Sobrecarga de operadores e funções

A *sobrecarga*[1] de operadores e funções consiste na associação de duas ou mais definições para um mesmo símbolo. A *resolução da sobrecarga* consiste na determinação da definição a ser efetivamente usada, com base no contexto em que o símbolo aparece.

Muitas linguagens procedurais, como C e Pascal, apresentam sobrecarga de operadores sobre tipos padrão, mas não provêem mecanismos para a extensão da sobrecarga para funções ou tipos definidos pelo programador. Considere, por exemplo, as seguintes expressões

```
1 + 2
3.14159 + 2.71828
```

o operador de soma corresponde a duas operações distintas: a soma de inteiros e a soma de reais. A escolha de uma delas depende do contexto, ou seja, de seus operandos.

1. Tradução livre do inglês *overloading*. Este termo será usado em todo o texto.

O processo de escolha da definição a ser utilizada é chamado de resolução de sobrecarga e se baseia nos tipos dos parâmetros ou operandos envolvidos na operação.

2.6.1 definição de operadores

Em Cm, a definição de operadores é feita através de funções com nomes especiais, dados pela palavra reservada **operator** seguida do símbolo do operador desejado (como mostrado na tabela 2–1); por exemplo:

```
class Complex<>
float re, im;

export Complex &operator += (Complex ropnd)
{
   self.re += ropnd.re;
   self.im += ropnd.im;
   return self;
}

export Complex operator + (Complex ropnd)
{
   result.re = self.re + ropnd.re;
   result.im = self.im + ropnd.im;
}
```

As funções que definem operadores devem respeitar a aridade[1] dos mesmos, ou seja, o número de parâmetros da função (incluindo o parâmetro **self**) deve corresponder a uma aridade válida para o operador. Alguns símbolos de operadores são usados para dois operadores. Nesses casos o número de parâmetros da função determina qual operador está sendo definido.

```
class X<>

void operator + (int i);
                        // correto: adição binária  X<> + int

void operator + ();
                        // correto: mais unário     + X<>

static void operator + (int i; X& x);
                        // correto: adição binária  int + X<>

static void operator + (X& x);
                        // correto: mais unário     X<>

void operator + (int i,j);
                        // incorreto: '+' ternário
```

1. Tradução livre do termo inglês *arity*, usada em todo o texto. A aridade de um operador corresponde ao número de operandos que ele admite.

```
static int operator + ();
                          // incorreto: aridade nula
```

As únicas exceções às regras são os operadores de incremento e decremento: o símbolo e número de parâmetros não são suficientes para especificar se são prefixados ou posfixados. Nestes casos pode-se prefixar ou posfixar o símbolo a um ponto decimal ('.') que representa o operando, como em:

```
void operator .++ (); // pós-incremento
void operator ++. (); // pré-incremento
```

Este ponto decimal auxiliar somente é usado na declaração e chamadas explícitas da função, pois em expressões comuns a posição do operando determina qual o operador desejado.

2.6.2 resolução da sobrecarga

A resolução da sobrecarga se baseia no número e tipo dos parâmetros ou argumentos envolvidos na operação, e basicamente determina qual definição tem os parâmetros formais mais "próximos" dos parâmetros/operandos reais, em termos de compatibilidade de tipos. Note que a aplicação de um operador sobrecarregado é convertida em uma chamada de função. Assim, a resolução de sobrecarga é baseada unicamente em funções, não importando se de fato essas funções representam operadores ou não.

A resolução pode ser impossível, se os parâmetros reais forem incompatíveis com todas as possíveis funções. Também pode ser ambígua, caso o mecanismo não seja capaz de determinar uma única função dentre as possíveis para o caso em questão.

Por exemplo, considere as seguintes definições de funções e declarações de variáveis:

```
void f (char c);
void f (int i);

void g (int i);
void g (long l);
void g (double f);
void g (char* s);

void h (X x);
void h (char* s);
void h (void* p);

void l (long l);
void l (double* p);
void l (char* s);

X       aX;
```

Funções cujos parâmetros tenham tipos equivalentes aos da chamada têm precedência sobre as demais. Este é o que ocorre nas seguintes chamadas:

`f('a')`	corresponde a `f(char)`
`f(10)`	corresponde a `f(int)`
`g(35L)`	corresponde a `g(long)`
`g(3.14)`	corresponde a `g(double)`

`h("str")`	corresponde a `h(char*)`
`h(NIL)`	corresponde a `h(void*)`
`h(aX)`	corresponde a `h(X)`

Quando os tipos dos parâmetros reais não têm equivalência total com os tipos dos parâmetros formais, mas não chegam a ser incompatíveis, Cm determina a função com compatibilidade "mais próxima". Nessa operação, é dada preferência para promoções de tipos "menores" que **int** para **int**, de **float** para **double**, de qualquer apontador para **void** * e de um tipo nomeado para o equivalente não nomeado, como em[1]

`f(3H)`	corresponde a `f(int)`
`g('a')`	corresponde a `g(int)`
`g(3.14f)`	corresponde a `g(double)`
`g(NIL)`	corresponde a `g(char*)`

E também são possíveis conversões entre tipos numéricos, de **void** * para outros apontadores e entre apontadores e integrais,[2] como em:

`l(5U)`	corresponde a `l(long)`
`l(3.14)`	corresponde a `l(long)`
`g(NIL)`	corresponde a `g(char*)`

As chamadas a seguir são ambíguas, pois não aspresentam um "melhor caso":

`f(123L)`	`f(char)` ou `f(long)`?
`l(NIL)`	`l(double*)` ou `l(char*)`?

e as chamadas a seguir não podem ser resolvidas, uma vez que os parâmetros usados são incompatíveis com todas as possíveis definições:

```
f(aX)       g(aX)       l(aX)
h('a')      h(10)       h(3.14)
```

2.7 Tratamento de exceções

Exceção é qualquer evento ou situação anormal que ocorra durante a execução de uma dada função. Ela deve ser informada (sinalizada) aos níveis anteriores da seqüência dinâmica de chamadas de funções, sendo propagada até atingir o nível onde ações corretoras possam ser adequadamente efetuadas. Se uma exceção não puder ser tratada em nenhum nível, então a execução do programa é interrompida, detalhando a natureza e ponto de ocorrência da exceção.

1. O sufixo `f` (ou `F`) especifica literais de tipo **float**.

2. Desde que o integral tenha "tamanho" suficiente para armazenar o "valor decimal de um endereço".

Note que uma exceção não corresponde necessariamente a um erro. O programador está livre para utilizar o mecanismo para obter um melhor controle sobre o fluxo de execução do programa. No entanto não se deve abusar do mecanismo, uma vez que o processo de sinalização de uma exceção e busca do respectivo tratador pode ser computacionalmente dispendioso.

Em Cm exceções são representadas por um valor qualquer e são identificadas pelo tipo desse valor. É vantajoso o uso de exceções de tipo classe, pois:

- nomes de tipo classe ocupam um escopo global em Cm, evitando problemas de conflitos de nome;
- rotinas para auxílio no tratamento podem ser introduzidas na classe que define a exceção, provendo maneiras padronizadas de tratamento e evitando replicação de código;
- o mecanismo de herança estabelece uma organização hierárquica de exceções.

A sinalização de exceções é feita através do comando **raise**, como em:

```
raise 10344;
raise "bad code";
raise @MathError<OVERFLOW>;
raise @OutOfRange<v, i>;
```

Internamente a um tratador, a expressão associada ao comando raise pode ser omitida, reiniciando a propagação da mesma exceção para tratadores mais externos.

Tratadores são introduzidos via comando protegido, como em:

```
[
   [
        tarefa ();
      when int:
        out << "tratador interno: int\n";
      when others:
        out << "tratador interno: não é int";
        raise;
   ]
   when int:
      out << "nunca vai escrever isso!";
   when char *msg:
      out << "tratador externo: char* (" <<
             msg << ")\n";
   when others:
      out << "tratador externo: também não é char*\n";
]
```

Neste caso, se a rotina `tarefa` sinalizar uma exceção de tipo **int**, o resultado será:

```
tratador interno: int
```

Mas se o tipo da exceção for **char***, com valor "mensagem", o resultado será:

```
tratador interno: não é int
tratador externo: char* (mensagem)
```

Finalmente, para qualquer outro tipo de exceção, o resultado será:

```
tratador interno: não é int
tratador externo: também não é char*
```

Note que o comando protegido interno impede que exceções tipo **int** sejam propagadas para fora dele, inutilizando o tratador **when int** do comando protegido externo (nunca será executado).

2.8 Classes padrão

Classes padrão são meramente arquivos fonte Cm distribuídos juntamente com o compilador, que padronizam algumas características deliberadamente não definidas na linguagem, como por exemplo entrada e saída.

Basicamente toda a biblioteca C está disponível a um programador Cm, bastando para isso a definição de classes a serem utilizadas em substituição aos arquivos de *header* de C. Há várias estratégias de implementação dessas classes:

1. criar uma única classe que sirva de interface para toda a biblioteca C;
2. criar uma classe para cada arquivo de inclusão de C, tais como `stdio` e `math`;
3. criar classes vinculadas a conceitos presentes na biblioteca, que encapsulem as informações e funções relativas a cada conceito, tais como FILE e SOCKET.

A melhor – e mais trabalhosa – dentre essas opções é a terceira. Uma projeto cuidadoso dessas classes permite organizá-las hierarquicamente, aproveitando melhor os conceitos de Cm e facilitando extensões. Por exemplo, um arquivo indexado (INDEXEDFILE) pode ser definido por derivação da classe que define arquivos simples (FILE).

Cm ainda não definiu formalmente um conjunto de classes padrão, mas apresenta algumas classes para facilitar operações de entrada e saída (classes Input e Output), bem como tornar essas operações mais seguras. Cm também apresenta uma classe para facilitar o tratamento da linha de comando (Arg). Segue-se uma descrição informal dessas classes.

2.8.1 classe Output

A classe Output implementa operações de escrita de dados através do operador '<<'. O operador recebe como operando esquerdo uma referência a um objeto de tipo Output(), e como operando direito um tipo padrão ou apontador qualquer. Ele efetua a operação de escrita do operando direito no arquivo definido pelo operando esquerdo, e retorna a mesma referência usada como operando esquerdo. Dessa forma pode-se encadear operadores '<<' para escritas sucessivas, como por exemplo:

```
Output out;
out << "string" << 10 << 'e' << NIL;
```

Pode-se especificar um formato como o de `printf` para os próximos dados, através da função `format` , como em:

```
out.format (">>> %5.5s: %d%d\n") << "BULE";
out << 10 << 2;
```

Comandos esdrúxulos podem surgir, tais como

```
(out << "[ ").format("%8.4f") << PI << "]";
```

Ao ser criado, um objeto da classe Output está associado ao *standard output*[1] do processo sendo executado. Pode-se alterar o descritor de arquivo associado através das funções:

```
out.set_fd (int fd);
out.open (char *file_name);
out.reopen (char *file_name);
out.close ();
```

2.8.2 classe Input

A classe Input é análoga à classe Output, provendo leitura através do operador '`>>`', que tem como operando direito uma referência a um objeto de qualquer tipo básico, como por exemplo:

```
Input in;
char c; int i; float f; char *s;
in >> c >> i >> f >> s;
```

Os valores são lidos, inicialmente, do *standard input*[2], e são delimitados por caracteres brancos (ou seja, espaço, tabulação, CR ou LF), e devem ter tipos válidos para os objetos dados. Os valores devem seguir as convenções léxicas do compilador Cm. A classe Input verifica se os tipos dos valores lidos são válidos. Se o programador quiser ler também caracteres brancos, deve usar a função `getc`. O fim de arquivo é sinalizado pela função `eof`. O arquivo de entrada pode ser mudado por funções análogas às da classe Output.

```
in.getc (char &c);
in.eof ();
in.set_fd (int fd);
in.open (char *file_name);
in.reopen (char *file_name);
in.close ();
```

1. Em ambiente Unix, o descritor de arquivos 1, associado ao terminal – a menos que haja redireção – e referenciado por `stdout` na biblioteca padrão de C.

2. Em ambiente Unix, o descritor de arquivos 0, associado ao terminal (teclado) – a menos que haja redireção – e referenciado por `stdin` na biblioteca padrão de C.

2.8.3 classe Arg

A classe Arg permite acesso simples aos parâmetros da linha de comando, exportando a variável `argc` e função `argv`, que recebe um parâmetro inteiro e retorna o parâmetro correspondente da linha de comando, dado por uma *string*. Por exemplo:

```
Arg a;
for (int i = 0; i < a.argc; i++)
   out << "Arg[" << i << "] =\"" << a.argv(i) << "\"";
```

2.9 Classes como módulos

O paradigma de programação modular prega que procedimentos (funções) inter-relacionados e os dados que eles manipulam devem ser agrupados em um módulo e separados do resto do programa.

Cada módulo especifica qual parcela de seus dados e funções são visíveis externamente, ou seja, sua *interface* com o resto do programa. Esta *interface* deve, preferencialmente, conter somente funções.

A classe Cm suporta diretamente este paradigma de programação sendo cada módulo definido em uma nova classe.

Por exemplo, uma tabela de símbolos poderia ser definida em uma classe como

```
class SymbolTable <>
     ...
export void insert (char *symbol, void *info) {...}
export void * search (char * symbol) {...}
init {...}
```

e usada em uma classe compilador, através de uma variável:

```
class Compiler <>
import SymbolTable;
SymbolTable symtab;
     ...
{
   symtab.insert (ident, desc);
   desc = symtab.search (ident);
     ...
```

Se mais de uma classe vai utilizar um módulo, então a variável deve ser global:

```
     ...
global SymbolTable symtab;
     ...
```

Ainda, se não se deseja utilizar nenhuma variável do módulo alocada no escopo de objeto e nem funções dinâmicas da classe que define o módulo, então não é preciso nem mesmo criar um objeto desse tipo, bastando acessar seus membros via seleção em escopo, como em:

```
class Compiler <>
import SymbolTable;
     ...
{
   :: SymbolTable :: insert (ident, desc);
   desc = :: SymbolTable :: search (ident);
     ...
```

2.10 Classes como Tipos Abstratos de Dados

Um módulo que define um tipo e um conjunto completo de operações sobre este tipo é chamado "tipo abstrato de dados". Por exemplo, podemos definir uma pilha de inteiros com a classe

```
class Stack <>
const ST_SIZE=100;
int [ST_SIZE] st;
int top = 0;
export void push (int element)
{
   st [top++] = element;
}
export int pop ()
{
   return st [--top];
}
```

Outras classes podem agora criar pilhas de inteiros e executar operações de `push` e `pop` sobre elas, além das operações de cópia (atribuição), comparação, alocação e liberação dinâmicas, que são automaticamente definidas por Cm (através dos operadores '`=`' e '`==`', **new** e **release**).

```
Stack s,t,u;
Stack *p;

s.push (10);
s.push (20);
t = s;
u.push (t.pop ());
p = new (Stack());
p -> push (10);
```

2.11 Polimorfismo

A classe `Stack` mostrada anteriormente implementa uma pilha de elementos de tipo inteiro com capacidade de até 100 elementos. Pilhas com capacidade ou tipo de elemento diferentes têm implementação praticamente idêntica. Esta "simetria" é facilmente aproveitada, bastando implementar a classe com base em parâmetros (no caso, tipo dos elementos e capacidade da pilha). Dessa forma, definimos classes genéricas (polimórficas).

A versão polimórfica da classe `Stack`, dada por

```
class Stack <type Element; int size>
Element [size] st;
int top = 0;
export void push (Element element)
{
   st [top++] = element;
}
export Element pop ()
{
   return st [--top];
}
```

permite definirmos pilhas de qualquer capacidade e com elementos de qualquer tipo (inclusive pilhas), como em

```
Stack<float, 10> sf;
Stack<Stack<float, 10>, 20> ssf;

sf.push (3.1416);
ssf.push (sf);
(ssf.pop ()).pop ();     // resulta 3.1416
```

2.12 Herança

O mecanismo de *herança* permite estender uma classe para introduzir novas propriedades ou redefinir parcialmente as já existentes. Por exemplo, podemos definir uma pilha de inteiros com capacidade de "auto–impressão" (operação `print`) a partir da classe `Stack`:

```
class PStack <int size>
import Output;
inherit Stack<int, size>;
export void print()
{
   Output out;
   for (i=0; i<top; i++)
      out << st[i];
}
```

A capacidade de se definir novos "conceitos" a partir dos já existentes apenas expressando-se as diferenças entre eles é o princípio básico da programação orientada a objetos. Além da herança, Cm também suporta orientação a objetos através de funções virtuais.

2.12.1 funções virtuais

A definição de uma classe genérica com capacidade de se imprimir esbarra em alguns problemas, pois a implementação adotada pode restringir os tipos que podem ser utilizados. No caso anterior (`PStack`) o operador '`<<`' foi aplicado sobre operandos de tipo `Output<>` e **int**. Caso a implementação seja generalizada, parametrizando-se o tipo dos elementos, essa restrição continua. Dessa forma somente poderão ser utilizados tipos sobre os quais o operador '`<<`' puder operar (tendo como primeiro argumento um objeto de tipo `Output<>`).

O problema é agravado caso a saída seja formatada por um comando como

```
out.format ("Elemento: %5d") << st[i];
```

Os mecanismos de parametrização de classe e herança nem sempre são suficientes para definirmos uma classe genérica sem recorrermos a artifícios. O problema ocorre porque a operação de impressão é bastante diferente para cada tipo.

O mecanismo de *funções virtuais* permite a solução desse problema de maneira simples: uma classe genérica é implementada isolando-se as operações que variam de tipo para tipo em funções ditas virtuais. A implementação dessas funções (virtuais) fica em aberto. Novas classes são então derivadas da classe genérica para definir o tipo a ser usado e a implementação das funções virtuais para esses tipos.

A implementação genérica da classe `PStack`, usando-se uma função virtual `print_element` torna-se:

```
class PStack <type Element; int size>
inherit Stack<Element, size>;
virtual void print_element (Element element);

export void print()
{
   for (i=0; i<top; i++)
      print_element (st[i]);
}
```

e dela pode-se derivar classes para elementos de tipo inteiro e *string:*

```
class IntPStack <int size>
import Output;
inherit PStack<int, size>;
Output out;

void print_element (int element)
{
```

```
    out.format ("Elemento: %5d") << element;
}


class StringPStack <int size>
import Output;
inherit PStack<int, size>;
Output out;

void print_element (char *element)
{
    out.format ("Elemento: %20.20d") << element;
}
```

Um outro caso em que funções virtuais são desejáveis é a implementação de algoritmos de ordenação: a implementação do algoritmo *QuickSort*, por exemplo, para dois tipos distintos é a mesma, exceto pela operação de comparação de dois elementos. Basta então definirmos uma classe genérica com uma função virtual para comparação de dois elementos, que teremos o algoritmo de ordenação disponível para qualquer tipo, bastando para isso implementar a função de comparação adequada.

Capítulo 3

Classes

Este capítulo apresenta exemplos completos de classes. Cada classe é apresentada gradualmente, introduzindo características diversas da linguagem.

3.1 Uma classe para datas

Cm não provê um tipo padrão que implemente o conceito de datas. Porém pode-se facilmente definir uma classe `date` para tal. Essa classe será apresentada por partes, exemplificando vários pontos da linguagem.

3.1.1 preâmbulo

A classe `date` representa um conceito bastante simples, auto-contido, e não necessita de parametrização nem herança. A priori também não precisa utilizar nenhuma outra classe, ou seja, pode ser implementada sem o mecanismo de importação. Porém, para manter homogeneidade com as bibliotecas padrão de entrada e saída (*streams*) da linguagem, operações de entrada e saída serão definidas com base nas classes `Input` e `Output`, que devem então ser importadas.

Assim, o preâmbulo da classe date é dado por

```
class date<>
import Input, Output;
```

3.1.2 representação da data

dados em escopo de objeto

Existem várias representações possíveis para uma data. Por simplicidade, serão usados valores inteiros para representar o dia, mês e ano, que inicialmente corresponderão ao dia 1 de janeiro do ano 1. Esses valores devem ser declarados no escopo de objeto, pois são específicos para cada objeto tipo `date` a ser criado.

```
int _day = 1, _month = 1, _year = 1;
```

Os membros para o dia, mês e ano correpondem a uma implementação particular da classe `date`, e não devem ser exportados. O acesso a eles é feito única e exclusivamente por métodos apropriados da *interface* da classe.

dados em escopo de classe

Os membros dia, mês e ano são suficientes para representar uma data, porém algumas informações adicionais podem ser desejáveis para a implementação da classe, tais como mapeamento dos valores numéricos do mês e ano em *strings*, ou mapeamento do mês no seu número de dias (provável).

Como essas informações são idênticas e constantes para todos os objetos da classe `date`, os respectivos dados podem ser declaradas como constantes, tornando-se membros de escopo de classe e sendo compartilhados por todos os objetos tipo `date` que porventura sejam criados.

```
const char[] NO_OF_DAYS =
     { 0, 31,28,31, 30,31,30, 31,31,30, 31,30,31 };

const char * [] DAY_NAME =
     { "domingo", "segunda", "terça", "quarta",
       "quinta", "sexta", "sabado"
     };

const char * [] MONTH_NAME =
     { "janeiro", "fevereiro", "março", "abril",
       "maio", "junho", "julho", "agosto",
       "setembro", "outubro", "novembro", "dezembro"
     };
```

3.1.3 *interface*

Os métodos e dados de uma classe declarados como exportados (**export**) constituem sua *interface*. Somente eles podem ser vistos fora da classe (via seleção). Como os dados da classe `date` não são exportados, todo acesso a eles é feito via métodos de sua *interface*, encapsulando sua organização/implementação.

3.1.3.1 métodos de acesso

Métodos para armazenar (`set`) e recuperar (`get`) o valor de uma data são bastante convenientes, se não necessários:

```
export void set (int day, month, year)
{
   self._day = day;
   self._month = month;
   self._year = year;
}

export void get (int & day, month, year)
{
```

```
    day = self._day;
    month = self._month;
    year = self._year;
}
```

funções *inline*

Para maior flexibilidade na utilização da classe, são fornecidos métodos para consultar o valor do dia, mês e ano em separado. Essas operações são bastante simples e declaradas como **inline**, permitindo ao compilador substituir chamadas do método por comandos equivalentes (corpo).

```
export inline int day () { return _day; }
export inline int month () { return _month; }
export inline int year () { return _year; }
```

Note que as funções foram declaradas **export**, sendo incluidas na *interface*, e **inline**,visando geração de código mais eficiente.

Essas funções são suficientes para que possamos criar e utilizar objetos tipo `date`, porém de forma pouco flexível ou intuitiva. A interface será gradualmente estendida, incorporando novas características, até obtermos uma classe `date` bastante flexível e poderosa.

3.1.3.2 construtores e destrutores

Da maneira que está definida, todos os objetos `date` são criados com o valor inicial `1/1/1`. Construtores permitem declarar objetos e iniciá-los com um valor qualquer, como por exemplo:

```
date d1;
date d2 = d1;
date d3 = @( 12, 7, 93 );
date d4 @( "21/10/73" );
```

Como a classe ainda não tem construtores, as única declarações válidas são as de `d1` e `d2`, que utilizam recursos pré-definidos da linguagem.

construtor *default*

Um construtor que não exige parâmetros é chamado de *default* (usa um valor inicial *default*). Toda classe deve ter um construtor *default*. Caso não seja explicitamente declarado e caso todos os membros tenham um valor inicial em suas respectivas declarações, o compilador gera um construtor *default* pré-definido para a classe.

`d1` é inicializado pelo construtor *default* pré-definido, que atribui aos seus membros os valores dados nas respectivas declarações, ou seja, seu valor inicial é `1/1/1`. O uso desse construtor somente é válido porque todos os membros da classe `date` foram inicializados.

Este construtor poderia ser explicitamente declarado da seguinte forma:

```
constructor ()
{
```

```
    _day = _month = _year = 1;
}
```

construtor de cópia

Um construtor que exige um único parâmetro constante cujo tipo é sua própria classe é chamado de construtor de cópia (copia o valor de um objeto em outro). Caso não seja explicitamente declarado, o compilador gera um construtor de cópia pré-definido, que efetua uma cópia "*byte-a-byte*" de um objeto no outro.

d2 é criado usando o construtor de cópia pré-definido para a classe, que corresponde a uma cópia "*byte-a-byte*" do valor dado (d1) para o objeto criado (d2). Assim, o valor de d2 é o mesmo que o de d1, ou seja, 1/1/1.

Este construtor poderia ser explicitamente definido da seguinte forma:

```
constructor (const date d)
{
   self = d;    // '=' corresponde a cópia byte-a-byte
}
```

outros construtores

Outras formas de inicialização de dados não têm construtores pré-definidos associados, assim d3 e d4 somente podem ser declarados caso os construtores adequados sejam explicitamente declarados. Os construtores para permitir essas declarações acima podem ser:

```
constructor (int day, month, year)
{
   set (day, month, year);
}

constructor (char *s)// 's' na forma "dd/mm/yy"
{
   const int DELTA = '0' * 11;
   set (s[0]*10+s[1]-DELTA,
        s[3]*10+s[4]-DELTA,
        s[6]*10+s[7]-DELTA + 1900);
}
```

destrutores

Os vários construtores de uma classe determinam várias formas de se inicializar objetos daquela classe, quando são criados. O destrutor de uma classe serve para "desconstruir" objetos daquela classe, ou seja, realizam a tarefa inversa do construtor.

Destrutores somente são necessários quando os objetos de uma classe alocam recursos tais como memória (*heap*) e descritores de arquivos.

Cada classe pode ter somente um único destrutor, apesar de poder ter vários construtores. Uma vez inicializados, os objetos da classe são representados de forma homogênea (dados da classe) sobre a qual opera o destrutor.

Como objetos tipo `date` são bastante simples, não exigem destrutores. Porém pode-se definir um destrutor "vazio" para ela, da seguinte forma:

```
destructor {}
```

3.1.3.3 operações aritméticas com datas

O maior ganho de flexibilidade da classe `date` vem com a adição de operações sobre datas. Essas operações englobam, por exemplo:

- determinar quantos dias existem entre duas datas;
- avançar ou retroceder uma data um determinado número de dias, meses ou anos;
- determinar que dia da semana corresponde a uma determinada data;
- escrever/ler datas.

Essas operações poderiam ser definidas com funções comuns, sem prejuízo da flexibilidade que proporcionam. Porém, é mais conveniente a (re)definição de operadores sobre datas, pois seu uso se torna mais intuitivo e os programas ficam mais simples e legíveis. Algumas operações interessantes são:

subtração ('-'): entre duas datas, retorna o número de dias entre elas; entre uma data e um inteiro, retrocede a data o número de dias dado pelo inteiro;

adição ('+'): entre uma data e um inteiro, calcula uma nova data avançando a data corrente o número de dias dado pelo inteiro.

Essas operações são definidas com funções especiais para sobrecarga de operadores:

```
export int operator- (const date d);
export date & operator- (const int i);
export date & operator+ (const int i);
```

A implementação dessas funções fica bastante simples caso tenhamos uma maneira de mapear datas em números seqüenciais e vice-versa. Uma possibilidade é relacionar cada data com sua "distância" em dias do dia `1/1/1`. Assim, teremos:

```
long get_l ()
{
   // calcula quantos dias se passaram entre 1/1/1 e
   // a data dada por self
}

void set_l (long l)
{
   // determina o l-ésimo dia a partir de 1/1/1 e
   // atribui esta data para self
}
```

```
export int operator- (const date d)
{
  return (self.get_l() - d.get_l());
}

export date & operator- (int i)
{
  result.set_l (self.get_l() - i);
}

export date & operator+ (int i)
{
  result.set_l (self.get_l() + i);
}
```

3.1.3.4 entrada e saída

As operações de entrada e saída podem ser definidas com funções como `write` e `read`:

```
export void write(Output out)
{
  out  << self._day << '/' << self._month << '/'
       << self._year ;
}

export void read(Input in)
{
  char dummy; // para pular as '/' da entrada
  in >> self._day >> dummy >> self._month >> dummy
     >> self._year;
}
```

Note que `write` e `read` podem operar em conjunto, ou seja, a saída de `write` é entendida por `read`. Essas funções permitem as operações de entrada e saída de dados de tipo `date`, mas não permitem o uso homogêneo das classes `Input` e `Output` para tal, ou seja, expressões da forma

```
out << "Período: de " << d1 << " a " << d2 ".\n";
in >> d3;
```

não podem ser usadas. Para permitir essas expressões, é preciso definir os operadores '**>>**' e '**<<**' operando sobre dados tipo `date`:

```
export static
Output & operator<< (Output & out; const date d)
{
  out << d._day << '/' << d._month << '/' << d._year;
}

export static
```

```
Input & operator>> (Input & in; date &d)
{
   char dummy; // para pular as '/' da entrada
   in >> d._day >> dummy >> d._month >> dummy
      >> d._year;
}
```

3.1.3.5 conversões

Operações de conversão permitem o mapeamento de datas em outros tipos. Conversões entre datas e tipos numéricos já foram definidas na classe, para uso interno (`get_l` e `set_l`), e podem ser exportadas via métodos *inline*, por exemplo.

```
export inline long to_l () { return get_l(); }

export inline void from_l (long l) { set_l (l) };
```

Conversões para *strings*, da data inteira ou de parte dela, também são interessantes:

```
export char *to_s ()
{
   char[20] buffer;
   sprintf (buffer, "%d/%s/%d",
            _day, MONTH_NAME[_month], _year);
   return buffer;
}

export inline char *day_name ()
{
   return DAY_NAME[_day];
}

export inline char *month_name ()
{
   return MONTH_NAME[_month];
}
```

Analogamente, poderiam ser definidas operações para conversão de *strings* em datas:

```
export void from_s (char *s)
{
   ...
}
```

3.2 *Arrays* associativos: usando o tipo referência

Arrays associativos correspondem a tabelas que associam o valor de uma das colunas aos valores das demais colunas na mesma linha. Por exemplo, podemos ter uma tabela contendo nomes de pessoas e seus telefones.

TABELA 3-1 Array associativo de nomes e telefones

Nome	Telefone
Dudi	54-1809
A_HAND	39-4685
MSouza, Bolha	54-5136
Wandeco	55-3417
André	41-0274
Lesmão	54-5264
Teles	39-5812

Dado um nome, a tabela pode ser consultada para determinar seu telefone. Esta tabela pode ser vista como um *array* indexado por nomes e que armazena telefones. *Arrays* associativos são indexados por valores quaisquer, sem uma seqüência definida. Na realidade, eles estabelecem uma relação entre alguns elementos, usados como "índices", e alguma informação relativa a esse valor.

Porém os *arrays* convencionais da linguagem são indexados por números inteiros consecutivos a partir de 0 e não representam diretamente arrays associativos. Este exemplo implementa uma classe `AssocArr` para arrays associativos.

definição

A sintaxe utilizada para a definição do *array* associativo é intencionalmente a mesma que a de *arrays* comuns, tornando seu uso bastante intuitivo. Por exemplo, um *array* associativo 'v' de inteiros indexado por *strings* permite as seguintes operações:

```
v["uma"] = 10;
v["outra"] = 24;
v["ambas"] = v["uma"] + v["outra"]++;
```

E um programa que lê uma série de palavras e conta a ocorrência de cada uma poderia ser escrito da seguinte forma:

```
char *s;
AssocArr v;

in >> s;
while (s != NIL)
{
   v[s] ++;
   in >> s;
}
```

Este programa supõe que o primeiro uso de um índice provoca a criação da entrada correspondente no *array*, que permanece até que este seja destruído.

Implementação

A implementação em Cm do *array* associativo mostrado anteriormente é bastante simples utilizando-se o tipo referência e o operador de indexação. A idéia básica é definir o operador de indexação de forma que ele receba uma *string* como argumento, busque o respectivo valor inteiro e retorne uma referência para ele. A associação dos índices aos valores pode ser feita por meio de dois *arrays* comuns (um para os "índices" *string* e outro para os valores inteiros).

Como descrito, a primeira utilização de um índice (*string*) no *array* provoca sua inserção na tabela, com valor associado 0. Implementações mais sofisticadas podem restringir quando um elemento deve ser inserido na tabela do *array* associativo.

```
class AssocArr<>

const int SIZE = 100;
int [SIZE] values;
char * [SIZE] indices;
int last_used = -1;

int search ()
{
   // busca a string 'index' no array 'indices'
   // e retorna seu índice. Caso não
   // encontre, retorna -1.
}

export int &operator[] (char* index)
{
   extern char* strdup (const char* s);
   int place = search (index);
   if (place < 0)
   {
      place = ++last_used;
      indices [place] = strdup (index);
      values [place] = 0;
   }
   return values [place];
}
```

3.3 Revendo construtores e destrutores

Esta seção descreve a implementação de uma tabela de símbolos simplificada, baseada na tabela de símbolos do próprio compilador Cm. Esta implementação utiliza uma tabela de *hashing* para tornar a busca de um símbolo mais eficiente.

Construtores e destrutores de classe (estáticos) são utilizados para inicializar e destruir a tabela de *hashing*. Construtores e destrutores de objeto (dinâmico) são utilizados para criar, inserir, remover e destruir símbolos na tabela. Nesta versão

simplificada, os descritores dos símbolos em si são dados por parametrização, devendo ser criados e destruídos externamente.

Símbolos
Esta classe em última instância implementa o tipo Símbolo. As operações sobre a tabela de símbolos em si são feitas ou através de símbolos e, possivelmente, através de seleção no escopo da classe.

As informações que caracterizam cada símbolo são seu identificador – um apontador de caracteres – e seu descritor – de tipo parametrizado. Informações auxiliares são introduzidas para implementação da tabela de *hashing*.

A tabela de *hashing*
A tabela de *hashing* consiste basicamente de um vetor de listas de símbolos. Uma função de *hashing* é aplicada ao identificador de cada símbolo para determinar o índice da lista na qual este deve ser inserido.

A tabela é alocada dinamicamente, mas é compartilhada por todos os símbolos, devendo ser declarada como um membro estático. Dessa forma, ela será declarada como um apontador estático (`hashTab`) para um vetor de listas de símbolos, cujo tamanho é dependente da função de *hashing*. A única imposição dada pelo mecanismo usado nessa implementação é que o tamanho deve ser dado por um número primo, caso se deseje um uso eficiente da tabela.

A lista é ligada por um membro (`next`) introduzido no escopo do objeto símbolo. O índice determinado pela função de *hashing* também é introduzido no escopo do objeto para maior eficiência (evita que este valor tenha que ser recalculado).

A figura 3-1 descreve graficamente a estrutura da tabela de símbolos após a inserção de alguns símbolos.

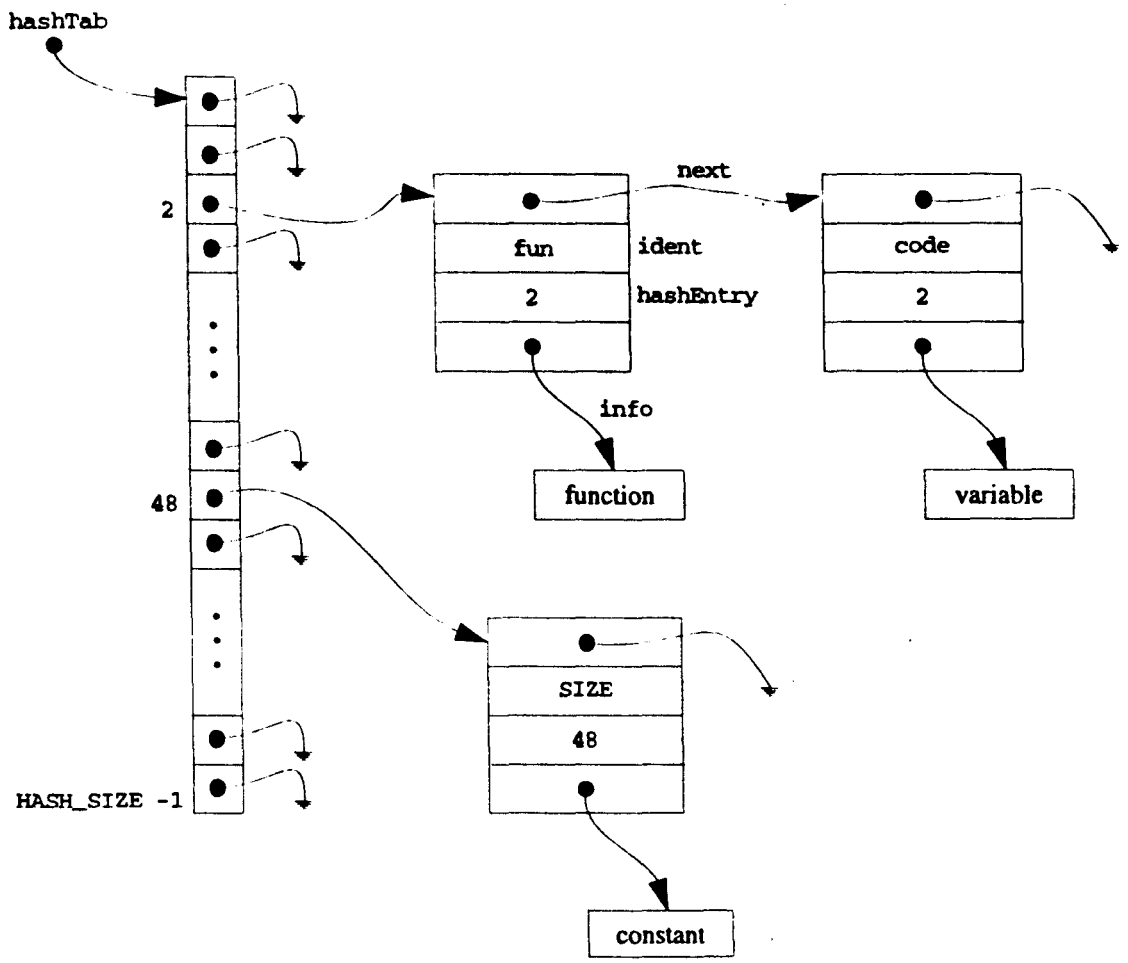
3.3.1 Declarações

O uso da tabela de *hashing* é mera decisão de implementação, e não deve ser visível externamente à classe. Apenas para simplificar o exemplo, os membros relativos ao objeto serão exportados (evitando a necessidade de funções de acesso explícitas).

O preâmbulo e declarações de dados da classe `Symbol` descrita anteriormente é dada por

```
class Symbol <type Info>

const int HASH_SIZE = 251;

static Symbol** hashTab;

export char* ident;
export Info* info;
Symbol* next;
int hashEntry;
```

FIGURA 3-1 Estrutura da tabela de símbolos



3.3.2 A função de *hashing*

A função de *hashing* depende somente do identificador utilizado, podendo ser declarada como estática para maior eficiência (evitando a passagem do parâmetro implícito **self**). Não cabe no momento entrar em detalhes do funcionamento da função, dada por

```
static int hash (char *identifier)
{
   register unsigned long int h = 0, g;
   while (*identifier)
   {
      h = (h << 4) + *identifier++;
      if (g = h & 0xf0000000L)
      {
```

```
            h = h ^ (g >> 24);
            h = h ^ g;
          }
        }
        return (int) (h % HASH_SIZE);
    }
```

3.3.3 Construtores e destrutores estáticos

A alocação e destruição da tabela de *hashing* é dada pelos construtores e destrutores estáticos. O construtor deve alocar a memória necessária ao vetor de listas de símbolos e inicializá-las como vazias:

```
static constructor
{
  hashTab = new (Symbol*, HASH_SIZE);
  for (int i = 0; i < HASH_SIZE; i++)
    hashTab[i] = NIL;
}
```

O destrutor deve destruir eventuais símbolos restantes na tabela e liberar a memória utilizada pelo vetor. Note que objetos criados pelo compilador são sempre destruídos convenientemente antes do destrutor estático ser ativado. Assim, todos os símbolos restantes na tabela foram alocados diretamente pelo programador, através do operador **new**, e podem ser liberados com **release**.

```
static destructor
{
  for (int i = 0; i < HASH_SIZE; i++)
    for (Symbol *p = hashTab[i]; p != NIL;)
    {
      Symbol *next = p -> next;
      release (p);
      p = next;
    }
  release (hashTab);
}
```

3.3.4 Construtores e destrutores de objeto

O construtor dinâmico cria um símbolo e o insere na tabela. O destrutor dinâmico remove um símbolo da tabela e o destrói. A operação de busca de um símbolo é sempre seqüencial, a partir do início da lista correspondente, e o símbolo é sempre inserido no início da lista, fazendo com que símbolos definidos mais recentemente tenham precedência sobre símbolos inseridos anteriormente.

O construtor calcula o índice adequado para o símbolo, inicializa seus membros e o insere na lista correspondente:

```
constructor (char *pident; Info *pinfo)
{
```

```
    extern char *strdup (const char *s);
    ident = strdup (pident);
    hashEntry = hash (pident);
    info = pinfo;
    next = hashTab [hashEntry];
    hashTab [hashEntry] = &self;
}
```

Já o destrutor busca o símbolo na lista correspondente, remove-o da lista, e destrói o objeto (no caso, libera a memória utilizada por `ident`):

```
destructor
{
    Symbol **prev = &hashTab [hashEntry];
    while (*prev != &self)
        prev = &((*prev) -> next);
    *prev = self.next;

    release (ident);
}
```

3.4 Agrupamento de dados

Classes como pilhas, conjuntos, vetores e listas implementam formas de se agrupar dados. O tipo do dado utilizado é praticamente irrelevante, pois essas classes dificilmente manipulam diretamente seus dados, exceto possivelmente para comparações e cópia de seus valores. Dessa forma, a implementação dessas classes para tipos de objetos diferentes pode ser estruturalmente idêntica, diferindo apenas no tipo dos objetos armazenados. Em Cm essa similaridade das implementações pode ser aproveitada via polimorfismo paramétrico e herança, conforme será visto nos exemplos subseqüentes.

3.4.1 Uma classe polimórfica para pilhas

Pilhas são estruturas de dados bastante utilizadas em programação, sendo fortes candidatas a integrar uma biblioteca de classes de uso geral. Porém, pilhas diferem entre si com relação ao tamanho (número de elementos que podem armazenar) e tipo dos elementos que armazenam. Em Cm isso não se caracteriza um problema, pois a definição de uma classe pilha pode ser parametrizada, incluindo especificações de tipo.

As operações básicas sobre uma pilha são inserção (`push`), retirada (`pop`) e consulta ao último elemento (`top`). A listagem a seguir apresenta uma classe polimórfica que oferece essas operações, onde a pilha é representada por um *array* de elementos com o tamanho e tipo dados por parâmetros.

O preâmbulo da classe determina seus parâmetros, `Element` e `size`, utilizados nas declarações de sua estrutura interna e métodos.

```
class Stack <type Element; int size>
```

```
Element[size] st;
int st_top = -1;

export inline void push (const Element e)
{
  st [++st_top] = e;
}

export inline Element pop ()
{
  return st [st_top--];
}

export inline Element top ()
{
  return st [st_top];
}
```

Esta versão da classe apresenta os requisitos mínimos necessários para se criar e operar uma pilha. Porém algumas melhorias ainda podem ser feitas.

parâmetros *default*

Uma operação desejável é a consulta a elementos que não o topo da pilha. Isso pode ser feito estendendo-se a função `top`, que passa a receber o índice, em relação ao topo da pilha, do elemento desejado:

```
export inline Element top (int i = 0)
{
  return st [st_top - i];
}
```

O novo argumento tem valor *default* 0, de forma que a função pode ser usada em substituição à anterior sem que seja necessário alterar nenhum dos programas que porventura já a utilizem.

A consulta a um elemento da pilha também pode ser oferecida de maneira mais elegante via operador de indexação:

```
export Element operator[] (int i)
{
  return st [st_top - i];
}
```

Assim, é possível escrevermos código como

```
Stack <int, 100> s1;

s1.push (10);
s1.push (20);
s1.push (30);
```

```
out << s1.top ();      // 30
out << s1[1];          // 20 (segundo elemento)
out << s1.pop ();
```

3.4.2 Usando pilhas de pilhas

A definição de uma classe permite seu uso em qualquer momento que uma especificação de tipos é aceita, inclusive como parâmetros de outras classes (ou dela mesma). Dessa forma são possíveis pilhas de pilhas, pilhas de pilhas de pilhas, e assim por diante.

```
Stack <Stack <int,10>, 10> sst;
Stack <int, 10> st;

st.push (10);
st.push (20);
sst.push (st);

st.push (30);
sst.push (st);

st.pop ();             // 30
st.pop ();             // 20

sst[1].top ();         // 20 (topo do segundo
                       // elemento da pilha sst)

st = sst.pop ();
st.pop ();             // 30

(sst.top ()).top ();   // 20
```

3.4.3 Uma classe para conjuntos

Conjuntos de objetos também podem ser implementados de forma genérica, como pilhas. As operações básicas sobre um conjunto (`set`) são inserção (`insert`), remoção (`remove`) e teste de pertinência ao conjunto (`is_member`).

A classe `set` será implementada utilizando um vetor para armazenar os objetos. Por simplicidade, este vetor será alocado estaticamente (limitando o número máximo de objetos que o conjunto pode conter[1]) e não é realizada nenhuma verificação de consistência na execução das operações. As operações podem ser implementadas de maneira independente do tipo do objeto (via polimorfismo).

```
class set<type T>
const MAX_SIZE = 50;
T[MAX_SIZE] members;
```

1. Na realidade sempre haverá um limite, dado pelas limitações físicas da plataforma de execução.

```
int last = -1;

export int is_member (const T m)
{
   for (int i = 0; i <= last; i++)
      if (members[i] == m)
         return 1;
   return 0;
}

export void insert (const T m)
{
   if (! is_member (m))
      members [++last] = member;
}

export void remove (const T m)
{
   for (int i = 0; i <= last; i++)
      if (members[i] == m)
      {
         if (i < last)
            members[i] = members[last];
         last--;
         break;
      }
}
```

Note que esta implementação faz algumas suposições sobre o tipo T usado como parâmetro da classe:

1. o tipo T deve suportar a operação de cópia para ser passado por parâmetro;
2. o tipo T deve suportar a operação de atribuição para ser armazenado e retirado do vetor;
3. o tipo T deve suportar a operação de comparação para o teste de pertinência no conjunto.

Essas suposições são aceitáveis, uma vez que essas operações são definidas para qualquer tipo (explicitamente pelo programador ou implicitamente pelo compilador). Contudo, em alguns casos a definição implícita não tem a semântica desejável para o teste de pertinência. O caso mais evidente disso aparece para conjuntos de apontadores de caracteres (*strings*). A operação de comparação verifica se dois apontadores têm o mesmo valor, e não se refenciam *strings* idênticos, possivelmente em regiões diferentes de memória. Soluções alternativas são possíveis através dos mecanismos de herança.

3.5 Herança

Herança é um mecanismo de compartilhamento de código/comportamento. As relações de herança estabelecem uma *hierarquia*, um mecanismo geral de compar-

tilhamento de propriedades de ancestrais por descendentes, análogo ao aninhamento de blocos. Da mesma forma que a estrutura de blocos permite que dados de um bloco (ancestral) sejam compartilhados pelos blocos internos a ele (descendentes), a hierarquia de herança permite que o código/comportamento de uma classe (base, ou superclasse) seja utilizado pelas classes que a herdam (derivadas, ou subclasses).**[OOP90]**

O mecanismo de herança permite a definição de uma classe a partir de outras, estendendo ou introduzindo novas características. A nova classe é dita *derivada* das classes envolvidas na herança, que, por sua vez, constituem as *classes base* da nova classe. Caso haja somente uma classe base, a herança é *simples*. Nos demais casos a herança é *múltipla*.

A capacidade de se definir novos "conceitos" a partir de "conceitos" já existentes apenas expressando-se as diferenças entre eles é o princípio básico da programação orientada a objetos. O suporte à orientação a objetos é dado pelos mecanismos de herança, complementado pelos mecanismos de polimorfismo e funções virtuais.

As relações de herança estabelecem uma hierarquia de classes. Classes base estão localizadas em um nível mais "baixo" na hierarquia, agrupando funções mais genéricas. Classes derivadas então em um nível mais "alto" da hierarquia, fornecendo funções mais específicas. O grau de especialização aumenta a cada nível que se sobe na hierarquia.

O uso do mecanismo de herança se dá de várias formas. Os usos mais comuns são

- especialização de uma classe já existente;
- extensão de uma classe já existente;
- fatoração de código de diversas classes;
- definição de uma interface comum para um conjunto de classes.

3.5.1 Especialização de uma classe já existente

A especialização de uma classe consiste em se estabelecer um comportamento mais específico para uma classe, sem necessariamente adicionar propriedades novas. É o que ocorre quando definimos uma classe Retângulo a partir de uma classe Paralelogramo (retângulos são casos particulares de paralelogramos).

Assim, se a classe paralelogramo constrói um objeto a partir dos tamanhos de seus lados (`l1` e `l2`) e do ângulo interno entre seus lados (`a1`) e externo entre o primeiro lado e um eixo fixo qualquer (`a0`), como em

```
class paralelogramo<>
  ...
export void set (int a0, l1, a1, l2)
{
  ...
}

constructor (int a0, l1, a1, l2)
{
```

```
    set (a0, l1, a1, l2);
}
```

A classe retângulo pode ser implementada apenas "convertendo" as operações sobre retângulo para as correspondentes operações sobre paralelogramos. Lembrando que retângulos são paralelogramos com ângulo interno de 90 graus, teremos

```
class retângulo<>
inherit paralelogramo;
  ...
export void set (int a0, l1, l2)
{
  paralelogramo::set (a0, l1, 90, l2);
}

constructor (int a0, l1, l2)
{
  set (a0, l1, l2);
}
```

3.5.2 Extensão de uma classe já existente

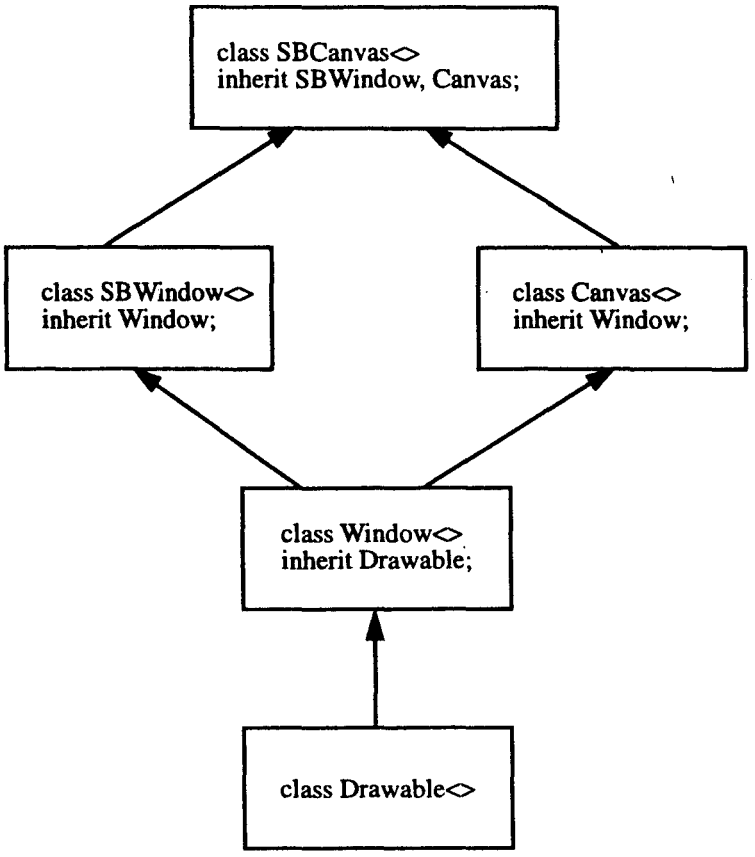
A extensão de uma classe consiste em se adicionar novas características a ela, sem necessariamente alterar o comportamento já definido. É o que ocorre, por exemplo quando adicionamos um controle (por exemplo, `ScrollBar`) a uma janela (`Window`). Supondo que existam classes que implementem objetos de tipo `Window` e `ScrollBar`, como em

```
class ScrollBar<>
  ...
constructor (int x, y, min, max) { ... }


class Window<>
  ...
constructor (int x, y, width, height) { ... }
```

Podemos implementar uma classe `SBWindow` como uma extensão das duas outras, como em

```
class SBWindow<>
inherit Window, ScrollBar;
  ...
constructor (int x, y, width, height, min, max)
{
  ::Window::set (x, y, width, height);
  ::ScrollBar::set (x+width, y, min, man);
  ...
}
```

FIGURA 3-2 Exemplo de hierarquia de herança (Windows)



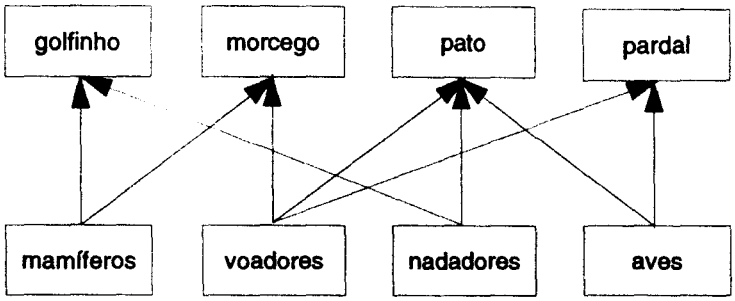
O mecanismo de herança é bastante utilizado na criação de interfaces gráficas. Essas interfaces são constituídas de vários objetos como janelas e botões, definidos a partir de elementos gráficos básicos. Novas características são introduzidas em cada objeto através de herança, que organiza hierarquicamente as clases de objetos gráficos. A figura 3-2 mostra uma possível relação de herança para alguns elemntos de uma interface gráfica.

3.5.3 Fatoração de código

A fatoração de código através de herança é conveniente quando determinados conjuntos de características são compartilhados por diversas classes, porém de forma que poucas sejam um subconjunto próprio das demais.

Considere, por exemplo, um conjunto de classes para representar alguns animais como morcego, golfinho, pardal e pato. Pardal e pato são aves e voam (algumas aves não voam), mas somente o pato nada. O golfinho é mamífero aquático. O morcego é um mamífero que voa. Nesse caso é possível fatoras as características

FIGURA 3-3 Hierarquia de animais, fatorando caracteríticas através de herança



relativas a mamíferos, aves, peixes, "voadores" e "nadadores" em classes que são então herdadas convenientemente por cada animal. Características comuns a todos os animais podem ser dadas em uma classe ainda mais genérica. A figura 3-3 mostra esquematicamente essas relações.

3.5.4 Interface comum através de herança (uso de funções virtuais)

Considere a implementação de um módulo de gerenciamento de fitas magnéticas. Os objetos que ele manipula, ou seja, as fitas, têm um comportamento bastante homogêneo, ou seja, pode-se abrir (`open`), fechar (`close`), rebobinar (`rewind`), ler (`read`) e escrever (`write`) em uma fita. Apesar desse comportamento homogêneo, cada operação é realizada de forma bastante diferente, dependendo do tipo da fita (*streamer*, DAT, etc).

Uma possível abordagem seria a definição de uma única classe `Tape`, com um membro `kind` indicando o tipo da fita. As operações realizadas deveriam, então, fazer um teste (**switch**) para agir adequadamente de acordo com o tipo dado. Essa abordagem, porém, apresenta alguns inconvenientes: a inserção de uma fita nova exige alteração do código relativo às fitas anteriores (extensão do **switch**), e o compilador não consegue garantir que todas as operações sejam devidamente estendidas.

Uma solução muito melhor pode ser obtida através de herança. Pode-se definir uma classse Tape que determina a interface de acesso a uma fita através de *funções virtuais*, que são então definidas em classes derivadas. Cada tipo de fita é criado derivando-se uma classe nova a partir de Tape. Dessa forma, cada novo tipo de fita é implementado de maneira independente dos demais, mantendo uma interface comum. Além disso, o compilador é capaz de verificar se todas as funções da *interface* foram devidamente definidas.

A classe `Tape` é chamada de *classe abstrata*. Não se pode criar um objeto de tipo `Tape`, devido às funções virtuais, mas a manipulação das fitas pode ser feita através de apontadores ou referências a objetos desse tipo.

A classe Tape pode ser definida por:

```
class Tape<>
export virtual int open();
export virtual int close();
export virtual int rewind ();
export virtual int read (int &size; char *buffer);
export virtual int write (int size; char *buffer);
```

E algumas classes derivadas dela podem ser:

```
class DAT_Tape<>
inherit Tape;
export int open() { ... }
export int close() { ... }
export int rewind () { ... }
export int read (int &size; char *buffer) { ... }
export int write (int size; char *buffer);{ ... }


class Streamer_Tape<>
inherit Tape;
...
```

A inserção de um novo tipo de fita, por exemplo uma fita simulada em arquivos (para depuração do sistema), é feita derivando-se uma nova classe. Note que o uso desses tipos derivados somente é necessário para a criação de fitas, e pode ser concentrado em uma única função, como create_tape, listada a seguir. As demais funções devem manipular fitas apenas através de sua *interface* comum, como em copy_tape:

```
class UsesTape<>
import Tape, DAT_Tape, Streamer_Tape, Simulated_Tape;
  ...
static Tape* create_tape (int kind)
{
   switch (kind)
   {
      case DAT_TAPE: return new (DAT_Tape);
      case STR_TAPE: return new (Streamer_Tape);
      case SIM_TAPE: return new (Simulated_Tape);
      default: raise ("Unknown tape kind.");
   }
}

static void copy_tape (Tape* to, from)
{
   char[1024] buffer;
   int size = 1024;

   to->open ();
```

```
    from->open ();
    while (from->read (size, buffer))
      to->write (size, buffer);
    from->close ();
    to->close ();
}

...
Tape* dat = create_tape (DAT_Tape);
Simulated_Tape sim;
copy_tape (&sim, dat);
...
```

II

O compilador Cm

Capítulo 4

Funcionamento macroscópico do compilador Cm

O compilador Cm gera código em linguagem C, e invoca o compilador C para produzir o código objeto ou executável. Esta operação, no entanto, é transparente ao usuário. Este capítulo descreve o funcionamento geral do compilador, sua configuração (variáveis de ambiente) e execução.

4.1 Tradução de código

Há muitas vantagens e desvantagens em se gerar código numa linguagem de alto nível. As principais desvantagens são:

- Maior tempo de compilação: o código C gerado deve ser lido e compilado por outro compilador. Uma operação direta seria mais rápida.
- Menor controle sobre o código gerado: fica-se restrito aos recursos oferecidos pela linguagem utilizada para geração de código, sendo necessário, às vezes, recorrer a artifícios complicados para se obter o efeito desejado. Um código gerado diretamente pode ser mais otimizado.
- Dificuldade de depuração: a inserção de um outro compilador difculta a depuração do programa, pois as informações de depuração presentes no código gerado se referem ao código C gerado, e não ao código Cm original.

As vantagens, porém, são maiores, a curto e médio prazo:

- A geração de código é bastante facilitada usando-se C como linguagem alvo, uma vez que C é bastante flexível e bastante próximo a Cm a nível de expressões, comandos e estruturas de dados.
- A verificação do código gerado é facilitada, uma vez que quanto maior o "nível" de uma linguagem, mais fácil é seu entendimento.
- A geração de código cruzado é bastante fácil, bastando ter um compilador C adequado na máquina alvo.

4.1.1 Por que C?

O uso de C com base para a geração de código (e também como ponto de partida da definição de Cm) é ditado pelas características de Cm e pelo ambiente de desenvolvimento desejado:

- desenvolvimento em Unix: em ambientes Unix o uso de C é bastante vantajoso, pois C é a "linguagem oficial" em Unix, possuindo ferramentas de desenvolvimento e depuração difíceis de se encontrar para outras linguagens;[1]
- flexibilidade: C não apresenta limitações inerentes que impeçam seu uso em quase qualquer área de aplicação, com quase qualquer técnica de programação;
- eficiência: a semântica de C é em muitos aspectos próxima ao "baixo nível", ou seja, muitas de suas construções espelham diretamente aspectos de computadores tradicionais, permitindo facilmente um uso eficiente dos recursos da máquina;
- disponibilidade: C é provavelmente a linguagem mais difundida no mundo, sendo provável que se encontre pelo menos um compilador C razoável (que suporte pelo menos a versão padrão da linguagem e de suas bibliotecas), para qualquer máquina desejada;
- portabilidade: programas C são bastante dependentes das bibliotecas utilizadas, e o porte de um programa C de uma plataforma para outra não é automático nem necessariamente fácil. Porém é possível, com um mínimo de disciplina e organização, a escrita de programas facilmente portáveis.

4.2 Esquema de tradução de Cm

O usuário encara a geração do código executável para uma determinada classe como uma operação atômica. De maneira geral, ele somente manipula arquivos fonte Cm e arquivos de código executável, não utilizando ferramentas usuais, como `make` e `ld`, na manutenção de seus programas.

Porém, na realidade o compilador Cm gera arquivos de código fonte C e os processa com um compilador C convencional, gerando arquivos de código objeto que são então "ligados" com as bibliotecas Cm para produzir o código executável. Toda a gerência de recompilação de arquivos (`make`) é feita automaticamente pelo compilador Cm.

4.2.1 Arquivos gerados

Cada arquivo fonte Cm pode dar origem a um número indeterminado de arquivos fonte C, caso a classe Cm seja parametrizada. Cada um dos arquivos C corresponde a uma instância diferente da classe, dada pelos parâmetros utilizados em sua geração.

As informações relativas a cada instância, tais como o nome e localização do arquivo que a gerou, bem como os parâmetros utilizados na sua geração, são armazenadas em uma base de dados.

1. C++ é o sucessor natural de C e, atualmente, já está "tomando" o seu lugar em Unix.

A implementação atual dessa base de dados é muito simples e pouco eficiente: para cada instância de classe é gerado um arquivo próprio. A ineficiência provém do fato do compilador precisar consultar uma série de arquivos para poder localizar uma classe específica.

As instâncias de uma determinada classe são rotuladas com número inteiros distintos entre si, usados para identificá-las e nomear seus arquivos. A numeração é independente para cada classe.

Para cada instância são gerados um arquivo de *header* (`.h`) contendo as declarações necessárias àquela classe, um com sua implementação (`.c`) e um com sua descrição (`.dat`).

organização dos arquivos

O número de arquivos gerados pelo compilador é bastante grande. Além disso, alguns sistemas limitam o tamanho do nome dos arquivos. Dessa forma optou-se por utilizar uma hierarquia de diretórios, tanto para facilitar a organização/manutenção dos arquivos (pelo compilador) como para evitar problemas com tamanho do nome e com o número de arquivos gerados.

Cm utiliza basicamente dois diretórios: um para armazenar suas bibliotecas e os arquivos gerados, e outro para armazenar a base de dados de classes. Cada um desses diretórios contém um sub-diretório para cada classe, que contém os arquivos de cada instância.

Os arquivos de cada instância são nomeados `pNN.EXT`, onde `NN` corresponde ao número de seqüência da instância e `EXT` corresponde ao tipo (extensão) do arquivo.

variáveis de ambiente

Os diretórios de dados e de bibliotecas são dados pelas variáveis de ambiente `CMDATA` e `CMC`, respectivamente. A utilização dessas variáveis depende do sisteme e programa sendo usado. No caso de *makefiles*, e Unix em geral, o valor de uma variável é expresso precedendo-se seu nome com '`$`'.

Assim, a quinta instância de uma classe `Stack` é dada pelos seguintes arquivos:[1]

```
Stack.cm                arquivo fonte Cm
$CMC/Stack/p05.h        arquivo de header (gerado)
$CMC/Stack/p05.c        arquivo fonte C (gerado)
$CMDATA/Stack/p05.dat descrição da instância (gerado)
```

arquivos objeto

Os arquivos objeto gerados dependem do sistema operacional utilizado. No caso de Unix, é gerado um arquivo objeto com extensão `.o` junto ao respectivo fonte C:

```
$CMC/Stack/p05.o
```

1. No caso de IBM-DOS, as barras são invertidas ('\').

No caso de IBM-DOS podem ser gerados até seis arquivos, dependendo dos modelos de compilação utilizados:

```
$CMC\Stack\p05t.obj   tiny model
$CMC\Stack\p05c.obj   compact model
$CMC\Stack\p05s.obj   small model
$CMC\Stack\p05m.obj   medium model
$CMC\Stack\p05l.obj   large model
$CMC\Stack\p05h.obj   huge model
```

arquivos temporários
O processo de compilação de uma classe gera, além dos arquivos de cada instância descritos acima, arquivos temporários para *makefiles* e programa principal. Normalmente estes arquivos são removidos após a compilação, porém podem ser mantidos a pedido do usuário (via opção da linha de comando). Para o exemplo citado anteriormente, os arquivos temporários seriam:

```
$CMC/main--.c          programa principal C
Stack.umk              makefile (para Unix)
Stack.dmk              makefile (para IBM-DOS)
```

arquivo executável
O arquivo executável final é gerado no diretório corrente, com o mesmo nome da classe. Assim, para o exemplo anterior, este arquivo seria

```
Stack
```

4.3 Variáveis de ambiente

O compilador Cm se utiliza de três variáveis de ambiente principais e outras auxiliares. As principais descrevem os diretórios utilizados diretamente pelo compilador, e são:

CMDATA	diretório raiz da base de dados utilizada pelo compilador;
CMC	diretório raiz da biblioteca Cm (inclui as bibliotecas padrão e classes geradas pelos usuários);
CMSRC	lista de diretórios (*pathnames*) separados por '`:`' que devem ser utilizados na busca dos arquivos fonte a serem compilados.

Caso uma ou mais dessas variáveis não sejam definidas (ou sejam definidas com valor vazio), o diretório corrente é utilizado como *default*.

As demais variáveis de ambiente reconhecidas pelo compilador não são usadas diretamente por ele, e sim repassadas ao compilador C e/ou *linker* utilizados, permitindo ao usuário interagir (parcialmente) na forma de compilação e "ligação" do código. Essas variáveis são:

CMCFLAGS	*flags* a serem passadas para o compilador C;
CMLDFLAGS	*flags* a serem passadas para o *linker*;
CMCLIBS	bibliotecas e arquivos objetos a serem incluídos no código.

4.4 Linha de comando

A sintaxe da linha de comando do compilador Cm é dada por

```
cmc [options] classname...
```

Pode-se compilar várias classes com um único comando, bastando especificar os seus nomes (sem a extensão .cm).

As opções da linha de comando permitem alterar alguns parâmetros do compilador, bem como requisitar listagens e mensagens de depuração (do próprio compilador). As opções disponíveis atualmente são descritas a seguir. Algumas, como as de depuração, não são essenciais e somente estão disponíveis dependendo da forma como o próprio compilador Cm foi gerado.

4.4.1 Help

-h (help) Esta opção faz o compilador exibir uma breve descrição de como utilizá-lo, incluindo a sintaxe da linha de comando e as opções efetivamente disponíveis.

4.4.2 Opções de compilação

Afetam parâmetros de compilação e listagens.

-e (echo) Lista os arquivos sendo compilados (inclusive os importados) na saída padrão. Cada linha lida é prefixada pelo seu número seqüencial no arquivo e pelo nível de inclusão (importação) do arquivo na compilação corrente.

-as # Especifica o tamanho máximo em caracteres que uma *string* literal pode ter. O valor *default* é 200 caracteres.

-al # Especifica o tamanho máximo em caracteres que uma linha do arquivo de entrada pode ter. O valor *default* é 200 caracteres.

-b (build) Recompila todas as classes envolvidas na compilação corrente, mesmo que o mecanismo de *make* automático indique não ser necessário.

-ga (ansi) Gera código em ANSI-C (não implementado ainda).

4.4.3 Opções de controle do make

Afetam a forma como os arquivos de make são criados e tratados.

-mt *Default* em IBM-DOS; gera *makefiles* com sintaxe compatível com o "Borland's Turbo Make" (com extensão .dmk).

-mu *Default* em Unix; gera *makefiles* com sintaxe compatível com o *make* padrão de Unix (com extensão .umk).

-mM Executa o *makefile* após a geração dos arquivos C e o remove após a sua execução (*default*).

-mm Não executa nem remove o *makefile* após a geração dos arquivos C.

-mp Não remove o arquivo de *makefile*, mesmo que seja executado.

-ms Caso o *make* seja executado, sua execução é feita em modo silencioso (não mostra os comandos sendo executados). Por *default*, o *make* é executado em modo verboso (mostra os comandos).

4.4.4 Opções de depuração

Permitem depuração do compilador Cm. Somente são disponíveis caso o compilador seja gerado com facilidades para auto-depuração (compilação condicional).

-y Habilita depuração do *parser* (`yacc/bison`). Somente disponível caso o *parser* (`y.tab.h`) tenha sido gerado com facilidades de depuração.

-x # Estabelece o nível de depuração a ser usado. Quanto maior o nível, menos informação é mostrada. O intervalo válido é -9 a 9. O *default* é 7. Níveis menores que zero são excessivamente detalhados e tendem a gerar listagens enormes.

-X # Habilita depuração seletiva: cada *bit* no argumento, um valor inteiro decimal, ativa a depuração de uma função específica do compilador:

- 0 (1) mensagens gerais (modo "verboso");
- 1 (2) analisador léxico;
- 2 (4) arquivos gerados[1];
- 3 (8) tabela de símbolos;
- 4 (16) *swap* da tabela de símbolos[2];
- 5 (32) declarações;
- 6 (64) expressões;
- 7 (128) inicialização de dados;
- 8 (256) pilha de tipos da geração de expressões;
- 9 (512) comandos;
- 10 (1024) tipos;
- 11 (2048) funções (definição, sobrecarga e chamadas);
- 12 (4096) geração de código.

Essa opção é cumulativa, por exemplo, "`-X 9 -X 32`" solicita a emissão de mensagens gerais e nos módulos de tabelas de símbolos e declarações.

1. Escreve comentários no código gerado quando o fluxo de execução atinge pontos pré-determinados do compilador.
2. Mudança de contexto que ocorre quando da compilação de classes importadas.

Capítulo 5

Arquitetura do compilador

5.1 Módulos do compilador

5.1.1 Definições globais e módulo principal

Os arquivos de definições globais são os únicos necessariamente utilizados por todos os módulos do compilador.

O arquivo `global.h` contém definições genéricas, independentes do projeto em si, baseado na metodologia do projeto A_HAND. Essas definições incluem macros para portabilidade com relação ao compilador C usado (`K&R`, `ANSI`, `POSIX`, etc) e definição de macros simples de uso geral (`assert`, `max`, `min`, `NULL`, etc).

O arquivo `cm.e` contém definição das principais estruturas de dados utilizadas no compilador e que devem ser conhecidas por muitos de seus módulos, como estruturas de símbolos, tipos e expressões.

O arquivo `cm.c` contém a rotina principal do compilador. É responsável pelo *parsing* da linha de comandos e ativação dos demais módulos do compilador.

arquivos

cm.c	Programa principal do compilador
cm.e	Declarações globais (específicas do compilador Cm)
global.h	Declarações globais (de uso geral)

5.1.2 *Parser*

O *parser* é composto de dois módulos: o analisador léxico e o analisador sintático. O analisador sintático é gerado pelo `bison` ou `yacc` **[yacc78]** a partir da gramática do Cm. Esta gramática é bastante complexa e foi necessário um analisa-

dor léxico mais sofisticado para evitar conflitos e/ou um analisador sintático (autômato) muito grande para reconhecê-la.

A complexidade do analisador léxico decorre da necessidade de tratar alguns *tokens* de maneira dependente de contexto (do *parser*) e da conveniência de agrupar *tokens* com mesmo efeito sintático (classes de armazenamento, literais, etc). Dessa forma, foi implementado diretamente em C, e não através de geradores como `lex`.

arquivos

analex.e	declarações públicas do analisador léxico
analex.c	analisador léxico
bison.sim	arquivo auxiliar do `bison` (estendido para tratamento de erros)
gram.e	declarações públicas do analisador sintático
gram.h	declarações locais do analisador sintático
gram.y	gramática (entrada para bison ou yacc)
gramact.c	ações semânticas gerais
gramstmt.c	ações semânticas para comandos
gramact.h	declarações das ações semânticas
y.tab.h	definição de tokens da gramática gerada pelo bison/yacc
y.tab.c	parser gerado pelo bison/yacc

Implementação

Alguns dos requisitos especiais do analisador léxico envolvem:

- classificação de identificadores já conhecidos em categorias específicas, bem como diferenciação de identificadores sucedidos por '`(`' ou '`<`', podendo resultar em:

`IDENTIFIER`	identificador novo ou não reconhecido
`PIDENTIFIER`	identificador sucedido por '`(`'
`CLASS_ID`	identificador de classe
`TYPE_ID`	identificador de tipos
`FUN_ID`	identificador de funções

- reconhecimento e construção de descritores apropriados para literais, retornando um mesmo *token* (`LITERAL`) para qualquer literal encontrado;
- reconhecimento de delimitadores '`<`' e '`>`' correspondentes a uma lista de parâmetros de classe (evitando conflitos no uso de '`>`' em expressões);
- inserção automática de *tokens*, por exemplo '`<`' e '`>`' após identificadores de classe, caso omitidos (a gramática exige os *tokens*, sendo mais simples e evitando conflitos);
- reconhecimento de seqüências que nomeiam operadores (**operator** *tokens*) e mapeamento da seqüência em um identificador (`OPERATOR_ID`);

- tratamento especial de classes importadas, incluindo error de programação/ especificação.

Os conflitos restantes na gramática são inerentes (como o caso do clássico conflito do comando **if – if (...) if (...) ... else ...**) ou referentes a cláusulas de recuperação de erros sintáticos, e não representam problema algum.

5.1.3 Declarações

Este módulo é reponsável pela gerência da tabela de símbolos do compilador. As rotinas básicas permitem a declaração (inserção) e busca de símbolos. O módulo utiliza uma tabela de *hashing* para tornar essas operações mais eficientes. Também são oferecidas rotinas para declaração de símbolos de categorias específicas, como dados (constantes, variáveis), funções, parâmetros, tipos, *labels*, etc.

arquivos

decl.c	rotinas de declarações de dados/funções
decl.e	declarações públicas do módulo de declarações
symtab.c	gerenciador da tabela de símbolos
symtab.e	declarações públicas do módulo de tabela de símbolos

5.1.4 Expressões

Este módulo é responsável pela geração das estruturas de dados que representam expressões. Oferece rotinas para geração de expressões básicas (literais, identificadores) e de expressões montadas a partir de outras (operadores). Essas rotinas são chamadas pelo *parser*, que vai montando expressões complexas aos poucos. O módulo verifica as restrições e transformações impostas pelos operadores, tais como compatibilidade de tipos, conversão de operandos (promoções e projeções), endereçabilidade.

arquivos

expr.c	rotinas de geração de expressões
expr.e	declarações públicas do módulo de geração de expressões

5.1.5 Comandos

A versão corrente do compilador não gera estruturas para representar comandos, pois o código é gerado comando-a-comando. Assim, as poucas rotinas necessárias para gerenciar a manipulação de comandos foram incorporadas às ações semânticas de comandos (arquivo `gramstmt.c`).

5.1.6 Tipos

Este módulo é responsável pelo controle de descritores de tipos, provendo operações para alocação e liberação de descritores, geração de assinaturas de tipo (*strings* para representação de tipos em tempo de execução), verificação de compatibilidade de tipos, entre outras.

arquivos

type.c	rotinas de gerenciamento de tipos
type.e	declarações públicas do módulo de tipos
typechk.c	rotinas de verificação de tipos
typesig.c	rotinas de geração de "assinaturas" de tipos

5.1.7 Geração de código

Este módulo é responsável pela geração de código do compilador. Ele é dividido em quatro submódulos razoavelmente independentes, um para cada "tipo" de arquivo gerado (*makefile*, *headers* e código C para classes e para rotina principal).

***headers* (codehgen.c):**
Responsável pela geração da interface C (arquivo de *header*) da classe.

código C de classes (codecgen.c):
Responsável pela geração do código C da classe. As funções de geração de código são bastante fragmentadas, pois esta é realizada *on-the-fly*, paralelamente ao *parsing* da classe Cm.

código C da rotina principal (codemain.c):
Responsável pela geração da rotina principal correspondente à compilação.

***makefiles* (make.c):**
Responsável pela geração do *makefile* da classe.

Estes módulos, exceto pelo segundo, oferecem apenas uma função.

arquivos
Cada submódulo possui um arquivo próprio, que contém o código específico dele. O código compartilhado pelos módulos é dado em outros arquivos, como `codecdcl.c`, para geração de declarações de variáveis e funções, `codeexpr.c`, para geração de expressões, e `codemisc.c`, com rotinas gerais.

code.e	declarações públicas do gerador de código
code.h	declarações privadas do gerador de código
codecdcl.c	rotinas de geração de código para declarações
codecgen.c	rotinas de geração de código para comandos
codeexpr.c	rotinas de geração de código para expressões
codehgen.c	rotinas de geração de arquivos de *header* (`.h`)
codemain.c	rotinas de geração do programa principal (`main`)
codemisc.c	rotinas auxiliares do gerador de código
make.c	rotinas de geração de *makefiles*
make.e	declarações públicas do m,ódulo gerador de *makefiles*

5.1.8 Geração de mensagens de erro

O módulo de geração de mensagens de erro é responsável pela emissão de todas as mensagens de erro geradas pelo compilador, inclusive as decorrentes de erros sintáticos, geradas pelo *parser*.

As mensagens de erro sintático são bem mais amigáveis que as mensagens normalmente geradas pelo `yacc/bison`. O analisador sintático foi ligeiramente modificado para indicar todos os *tokens* esperados no ponto em que o erro foi detectado. O módulo de geração de erros mapeia estes *tokens* em *strings*, possivelmente agrupando-os conforme sua categoria (por exemplo literais e classes-de-armazenamento), gerando mensagens mais compactas e claras.

O mapeamento de *tokens* em *strings* (arquivo `tknames.h`) é gerado automaticamente (via make) a partir do arquivo da gramática (`gram.y`)

arquivos

error.c	gerador de mensagens de erros
error.e	declarações públicas correspondentes
tknames.c	mapeamento de *tokens* em *strings*
tknames.e	declarações públicas do módulo de mapeamento de *tokens*
tknames.h	definições das *strings* de correspondentes a cada *token*

5.1.9 Depuração

O módulo de depuração permite acompanhar o fluxo de execução do compilador Cm e visualizar estruturas de dados geradas. Pode ser totalmente removido do programa final através de compilação condicional.

arquivos

debug.c	rotinas de depuração
debug.e	declarações públicas do módulo de depuração

5.1.10 Módulos de uso geral

Os demais arquivos do compilador não caracterizam módulos propriamente ditos, apenas concentram rotinas de uso geral, auxiliares para os demais módulos.

comunicação com o ambiente de execução (sistema operacional)

environ.c	rotinas de interação com o ambiente
environ.e	protótipos das rotinas de interação com o ambiente

gerência de arquivos

expfn.c	rotinas para expansão canônica de nomes de arquivos
file.c	rotinas de entrada/saída
file.e	protótipos das rotinas de gerenciamento de arquivos

filer.c	gerenciador de arquivos indexados
filer.e	protótipos das rotinas de gerenciamento de arquivos indexados
wildcard.c	rotinas de busca de arquivos via *wildcards*
wildcard.e	protótipos das rotinas de busca de arquivos

alocação de memória

gmem.c	rotinas de alocação de memória dinâmica
gmem.e	protótipos das rotinas de alocação de memória dinâmica

listas-ligadas genéricas

llist.c	gerenciador de listas ligadas genéricas
llist.e	declarações públicas do módulo gerenciador de listas

rotinas de uso geral

utils.c	rotinas de uso geral
utils.e	protótipos das rotinas de uso geral

5.2 Arquivos de *make*

O compilador Cm pode ser gerado, a partir de seus fontes, inteiramente através de *makefiles*. Os arquivos de *make* permitem gerar o compilador inteiro (incluindo as bibliotecas de *run-time*) com ou sem depuração.

Atualmente o Cm dispõe de *makefiles* para MS-DOS/PC-DOS (`makefile.dos`) e Unix: SCO-ODT (`makefile.sco`) e SunOS/Solaris (`makefile.sun`).

5.3 Bibliotecas de *run-time*

A biblioteca de *run-time* Cm (`libcm.a`, em Unix) é incorporada pelo compilador aos programas executáveis gerados. É bastante simples, sendo escrita inteiramente em C; sua interface é dada pelo arquivo `libcm.h` (incluído por todos os arquivos gerados pelo compilador).

arquivos fonte da biblioteca

Os arquivos utilizados para gerar a biblioteca de *run-time* Cm são

global.h	definições globais (vide "Definições globais e módulo principal", página 5-1)
cm.h	definições globais da biblioteca Cm
makelib.dos	*makefile* para geração da biblioteca em IBM-DOS
makelib.sco	*makefile* para geração da biblioteca em SCO-Unix
makelib.sun	*makefile* para geração da biblioteca em SunOs/Solaris

cmlib*.c arquivos de implementação da biblioteca, descritos a seguir.

5.3.1 Tratamento de exceções (cmlibmte.c)

Gerenciamento do mecanismo de tratamento de exceções, descrito em detalhes em "Tratamento de exceções", página 6-32.

5.3.2 Operadores (cmlibop.c)

Rotinas que implementam os novos operadores ('^^', '^^=', '&&=' e '||=') sobre tipos padrão. São rotinas bastante simples, que recebem os argumentos e retornam um valor *boolean* (0 para falso e 1 para verdadeiro). Para os operadores com atribuição, o primeiro elemento é passado por "referência" (via apontador) sendo alterado pela própria função.

5.3.3 Gerência de memória dinâmica (cmlibmem.c)

Rotinas de gerenciamento de memória dinâmica. Essas rotinas são usadas para execução dos operadores **new** e **release**. A implementação desses operadores deve considerar o número e tamanho dos objetos a serem alocados, bem como se são de tipo classe ou se são organizados em *arrays* (**[C++93]** e **[C++93a]** descrevem a implementação desses operadores em C++, que é bastante parecida).

As rotinas de alocação e destruição de memória podem ser utilizadas para executar os construtores e destrutores dos objetos manipulados.

alocação de memória

```
void *__Cmalloc (int n, int size);
void *__Cmalloc2 (int n, int size, void (*init)());
void *__Cmalloc3 (int n, int size, void (*init)(),
                  void (*deinit)());
```

Estas rotinas alocam um bloco de memória para armazenar `n` objetos (consecutivos) de tamanho `size`. A alocação é feita através da rotina `calloc`, garantindo que a memória alocada seja inteiramente preenchida com zeros (assim todos os *bits* de *padding* ficam garantidamente com o mesmo valor, evitando problemas com a comparação de objetos via memcmp).

O parâmetro `init` corresponde à função de construção dos objetos sendo criados (construtor *default*, no caso de classes). Ela é executada para cada um dos elementos alocados. Seu valor pode ser NULL no caso de `__Cmalloc3`.

O parâmetro `deinit` corresponde à função de destruição que deve ser executada quando a rotina `__Cmfree` relativa a este comando for executada. A biblioteca de *run-time* mantém em tempo de execução uma "tabela" dinâmica com as informações necessárias para se destruir corretamente um objeto na execução do operador **release**. Esta tabela inclui o endereço do primeiro objeto (valor retornado por `calloc`), o destrutor a ser utilizado, o número de objetos e o tamanho de cada um deles.

liberação de memória

```
void __Cmfree (void *ptr);
void __Cmfree2 (void *ptr);
```

Libera o bloco de memória apontado por `ptr`. __Cmfree2 consulta a "tabela" de destrutores e determina como o(s) objeto(s) deve(m) ser destruído(s). Caso `ptr` seja NULL, a função simplesmente retorna, sem fazer nada.

comparação de memória

```
int __Cmbcmp (void *s1, void *s2, int size);
```

Esta rotina compara os `size` bytes apontados por `s1` e `s2` (através de memcmp), retornando 0 caso essas regiões de memória sejam diferentes em pelo menos um deles.

cópia de memória

```
void *__Cmbcopy (void *to, void *from, int size);
```

Esta rotina copia `size` bytes da região de memória referenciada por `from` para a região de memória referenciada por `to` (através de memmove). A cópia é feita corretamente mesmo que ocorra *overlapping* dessas regiões. O valor de retorno é o endereço dado por `to`.

5.3.4 Gerência de portas de comunicação (cmlibprt.c)

Rotinas de suporte a portas de comunicação rudimentares, com uma funcionalidade mínima. Uma versão bastante poderosa e flexível desta biblioteca está sendo implementada como parte do trabalho de mestrado de Celso Gonçalves Júnior, e proverá suporte a portas tipadas e a várias formas de controle do fluxo de comunicação (a nível de *bytes*, de blocos, sem fronteiras, etc).

5.3.5 Teste da biblioteca (libtest.c)

Programa para teste da biblioteca de *run-time*. Basicamente ele executa as rotinas da biblioteca com valores pré-determinados e verifica se os resultados foram os esperados. Apesar de abrangentes, os testes não são exaustivos.

5.4 Biblioteca de classes padrão

As bibliotecas de classe padrão foram escritas puramente em Cm. A funcionalidade dessas bibliotecas já foi descrita e a implementação efetivamente dada a elas não interessa no momento (trata-se apenas de um "exercício" de programação em Cm). Por completude e a título de exemplo, a implementação atual dessas bibliotecas foi incluída como apêndice da tese.

A distribuição corrente do Cm inclui como bibliotecas padrão as classes `Arg.cm`, `Input.cm`, `Output.cm` e `Port.cm`.

As bibliotecas padrão de entrada e saída foram implementadas baseando-se em uma classe auxiliar `Cstdio`. Esta classe é puramente uma *interface* com a biblioteca padrão C, exportando tipos (como `FILE`), variáveis globais (`errno`) e funções de entrada e saída (`printf` e `fopen`). Ela não é uma biblioteca padrão Cm, apenas foi utilizada para implementá-las. Tampouco procura ser uma interface completa para a biblioteca padrão C, implementando apenas um superconjunto dos requisitos necessários para as classes `Input` e `Output`. Há várias versões dessa biblioteca, uma para cada sistema em que o Cm já foi utilizado (MS-DOS/PC-DOS, SunOS, Solaris e SCO-ODT).

Capítulo 6

Geração de código

6.1 Mapeamento de identificadores

Muitos dos identificadores utilizados em uma classe Cm têm um escopo local na classe, porém têm um escopo global no código C gerado. Para se evitar conflitos de identificadores entre classes é preciso haver um mecanismo de mapeamento de identificadores (exceto para identificadores com escopo estritamente local, que podem não ser mapeados).

Todo o mapeamento de identificadores em Cm é concentrado em uma única função, que gera um identificador com base em:

- nome original do identificador (no código Cm);
- "categoria" do elemento referenciado pelo identificador (tipo, variável, etc);
- nome e "instância" da classe na qual o identificador foi definido;
- número de seqüência do identificador (seqüencial para cada classe, principalmente para geração de temporários).

mapeamento

Os identificadores gerados seguem um padrão fixo, dado por

```
__<T><NNN><II><XXXX>
```

onde <T> indica a "categoria" do identificador, <NNN> sua seqüência, <II> a instância da classe e <XXXX> o nome da classe em que foi declarado.

Este mapeamento não é muito bom porque depende da ordem em que as declarações aparecem em uma classe. A recompilação de uma classe que foi alterada pode exigir recompilação das classes que a utilizam, o que poderia ser evitado com um mapeamento melhor. De qualquer forma, a recompilação dessas outras classes sempre será necessária se a *interface* ou estrutura da classe for alterada. Este mapeamento foi adotado por não exigir identificadores longos. Caso

seja permitido o uso de identificadores arbitrariamente longos, o mapeamento pode ser melhorado substituindo-se <NNN> por uma "assinatura" gerada com base na declaração (nos tipos e identificadores envolvidos). **[C++89f]** e **[C++90]** apresentam um mapeamento de identificadores para C++ nessa linha, mas que não se enquadra a Cm (nem mesmo a C++, quando envolvendo templates). A melhor opção é uma mescla dos dois mecanismos.

categoria de identificadores

A "categoria" do identificador separa os nomes gerados em grandes grupos, facilitando sua identificação no código gerado. Essas "categorias" são:

A	tipos classe
T	tipos definidos pelo usuário
t	variáveis temporárias
c	variáveis que representam valores constantes (usadas quando o tipo da constante é estruturado, o que em muitos casos impede o uso direto do valor)
v	valores (variáveis estáticas representando valores iniciais de variáveis automáticas/dinâmicas)
s	membros estáticos
i	membros de objetos correspondentes a classes herdadas
f	funções
a	construtores (dinâmicos)
b	construtores estáticos
D	construtor *default*
d	destrutores (dinâmicos)
e	destrutores estáticos
C	biblioteca Cm
H	símbolos utilizados para compilação condicional
_	outros

geração dos números de seqüência

O número de seqüência do identificador, <NNN>, é mapeado em três caracteres correspondendo a um número na base 52 com dígitos 'A', ..., 'Z', 'a', ..., 'z'.

O número de seqüência da instância, <II>, é mapeado em dois caracteres correspondendo a um número decimal (até 99) ou em um número na base 52 (a partir de 100).

NOTA:
Para maior clareza, os exemplos dados nas seções seguintes não apresentam muitos dos mapeamentos que ocorrem na geração de código, caso estes mapeamentos não sejam relevantes ao tópico discutido. Em alguns casos a ocorrência de mapeamento é apenas indicada prefixando-se o identificador com '__'.

6.2 Estruturas de dados

As estruturas de dados geradas são um simples mapeamento da estrutura Cm para a respectiva estrutura C. Os únicos tipos Cm que não têm contrapartida direta em C são os tipos referência e classe.

6.2.1 tipo referência

O tipo referência é convertido para um apontador para o tipo referenciado e todos os acessos a objetos de tipo referência são ajustados de forma coerente. Esses ajustes basicamente consistem na inserção de uma operação de derreferência ('*') para obter o objeto a partir do apontador, quando é utilizado, e na inserção de uma operação de referência ('&') para obter o valor inicial do apontador, quando é declarado. Deve-se tomar cuidado para evitar aplicações alternadas e consecutivas de referências e derreferências no código final, e para evitar inserção da referência/ derreferência em situações envolvendo dois tipos referência (declarações, passagem de parâmetros, etc).

Por exemplo, o seguinte trecho de código Cm

```
int i;
int& x = i;
int& y = x;
i = 10; x = y + 1; y = 30 - i;
```

é traduzido para

```
int i;
int *x = &i;
int *y = x;
i = 10; (*x) = (*y) + 1; (*y) = 30 - i;
```

6.2.2 tipo classe

Classes são mapeadas em uma estrutura (**struct**), que contém todos os dados do escopo de objeto da classe dada. Dados de escopo de classe, global ou externo são mapeados em variáveis globais.

Assim, o código gerado para a seguinte classe

```
class x<>
int a, b;
static int c, d;
global int e, f;
extern int g, h;
```

é algo como

```
struct __A00x
{
   int a, b;
```

```
};

int __sAAA00x, __sAAB00x;     /* c, d */
extern int e, f;
extern int g, h;
```

A diferença básica entre variáveis globais e externas é que as primeiras são definidas no arquivo principal (`main`) gerado pelo compilador, enquanto que as últimas devem estar definidas em alguma biblioteca ou arquivo objeto incluído no código final.

6.2.2.1 herança

Para cada classe herdada é gerado um membro com seu tipo na estrutura da classe herdeira. Os acessos aos membros herdados são traduzidos de forma a utilizar os membros dessas estruturas.

Assim, o código gerado para a seguinte classe (onde `x` corresponde à classe citada anteriormente)

```
class y<>
inherit x;
char* b;
void f() { a = 1; b = 2; c = 3; e = 4; g = 5; }
```

é algo como

```
struct __A00y
{
   struct __A00x __i00x;
   int b;
}

void __fAAA00y (struct __A00y *self)
{
   self->__i00x.a = 1; self->b = "ab"; __sAAB00x = 3;
   e = 4; g = 5;
}
```

6.2.2.2 funções virtuais

Para cada função virtual de uma dada classe são introduzidos dois apontadores na sua estrutura. Um dos apontadores armazena o endereço da função a ser efetivamente chamada e o outro o endereço do objeto usado como **self** dessa função quando ela for chamada.

Dadas as seguintes declarações de classes

```
class Base<>
virtual void f();

class Deriv<>
inherit Base;
```

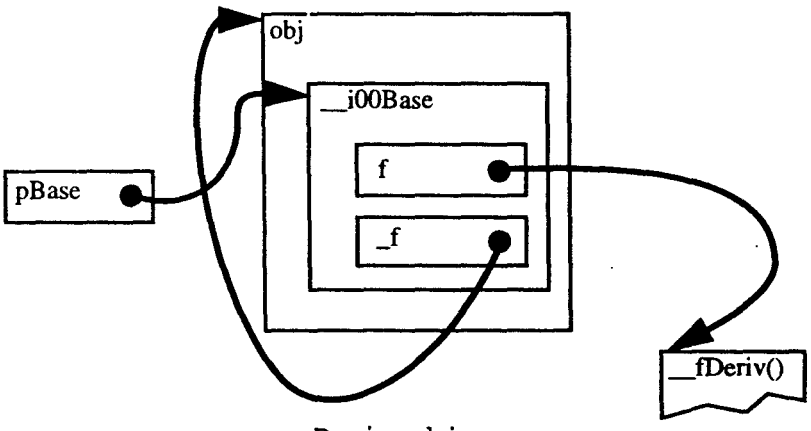
```
void f () { ... }
```

representando uma relação de herança simples com uma função virtual `f`, teremos as seguintes estruturas no código C gerado

```
struct __A00Base                    /* class Base */
{
  void (*f)();                     /* ptr para função f */
  void *_f;                        /* ptr para self de f */
};

struct __A00Deriv                   /* classe Deriv */
{
  struct __A00Base __i00Base;/* membro herdado */
};

void __fDeriv() { ... }             /* função f de Deriv */
```

E quando um objeto qualquer da classe `Deriv` for criado, (`obj`, por exemplo), os membros relativos à função `f` (`obj.__i00Base.f` e `obj.__i00Base._f`) deste objeto são iniciados com a função `f` da classe `Deriv` e o endereço do próprio objeto (`__fDeriv` e `&obj`, respectivamente). A figura 6-1 mostra a estrutura final obtida pela criação do objeto obj (classe `Deriv`) e um apontador para ele (classe `Base`).

FIGURA 6-1 Implementação de funções virtuais



6.3 Inicialização e execução do programa

Os construtores e destrutores de uma classe são responsáveis pela correta inicialização e finalização tanto da classe em si como de seus objetos. O compilador gera

funções construtoras e destrutoras para realizar as operações mínimas requeridas para a classe. Esses construtores e destrutores são chamados de *implícitos*. O programador pode estender estas operações definindo outros construtores e destrutores para a classe, chamados de *explícitos*.

Essas operações podem ocorrer em três níveis: global (função `main`), de classe ou de objeto, de acordo com o escopo de alocação das variáveis envolvidas.

Esta seção apenas descreve a estrutura geral de construtores e destrutores, bem como os mecanismos de controle de execução e a forma como são invocados. A tradução de declarações e funções em si será descrita mais detalhadamente em seções posteriores.

6.3.1 A função principal (*run-time*)

A execução do programa começa pela função principal, ou `main`, definida na biblioteca Cm. Esta função é responsável por inicializar o *run-time* e executar o programa do usuário. Ela executa as seguintes tarefas (na ordem dada):

1. reconhece os parâmetros da linha de comando referentes ao *run-time* do Cm;
2. estabelece o valor das variáveis globais `__Cmargc`, `__Cmargv` e `__Cmenvp` do Cm, usadas para "repassar" os parâmetros da linha de comando não referentes à biblioteca de *run-time* para o programa (equivalentes a `argc`, `argv` e envp de C);
3. inicializa a biblioteca de *run-time*;
4. executa a função principal do programa do usuário (`____main`);
5. finaliza a biblioteca de *run-time*;
6. encerra a execução do programa.

6.3.2 A função principal (usuário)

A função principal do programa do usuário, denominada ____main, é gerada pelo compilador Cm especificamente para a classe sendo compilada. Ela executa as inicializações necessárias, cria um objeto da classe em questão e executa seus construtores e destrutores. Seu arquivo, para uma classe XXX qualquer, segue o seguinte modelo:

```
/* declarações globais */

void ____main()
{
   struct __A00XXX instance;

   /* inicializações globais */

   __b00XXX ();            /* construtor de classe */
   __a00XXX (&instance);   /* construtor de objeto */
   __d00XXX (&instance);   /* destrutor de objeto*/
   __e00XXX ();            /* destrutor de classe */
```

```
    /* finalizações globais */
}
```

6.3.3 Construtores e destrutores de classe (estáticos)

Os construtores e destrutores estáticos de uma classe são responsáveis pela inicialização e finalização das classes envolvidas em suas hierarquias de herança e importação e das variáveis alocadas no seu escopo de classe.

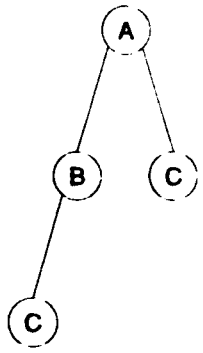
O construtor estático implícito de uma classe executa o construtor estático implícito de todas as classes herdadas ou importadas diretamente por ela, garantindo suas inicializações. Prossegue com a inicialização de sua própria classe, ou seja, das variáveis com alocação no escopo de classe. Ao final, executa o construtor de classe explícito (caso tenha sido definido pelo programador).

O destrutor estático implícito dessa classe executa as funções inversas, isto é, inicialmente executa o destrutor estático explícito da classe (caso tenha sido definido). Prossegue com a destruição das variáveis alocadas no escopo da classe (na ordem inversa da inicialização). Ao final, invoca os destrutores estáticos implícitos de todas as classes importadas ou herdadas, na ordem inversa em que os respectivos construtores foram invocados.

ordem de criação e destruição

Como uma dada classe pode aparecer mais de uma vez na hierarquia de herança, cuidados adicionais devem ser tomados para garantir que cada classe seja inicializada e finalizada uma única vez, e que a ordem de finalização seja a inversa da inicialização. A figura 6-2 mostra uma hierarquia de classe que pode gerar problemas caso estes cuidados não sejam tomados (note que somente evitar a reexecução não é suficiente para garantir uma execução correta).

FIGURA 6-2 Seqüência de execução de construtores e destrutores de classe



Controle	Ordem de construção	Ordem de destruição
nenhum	C B C A	A C B C
evitando reexecução	C B A	A C B
correto	C B A	A B C

O controle necessário é bastante simples, bastando um contador estático (compartilhado pelo construtor e destrutor) incrementado a cada execução do construtor e decrementado a cada execução do destrutor. O construtor só é efetivamente exe-

cutado quando o contador tem seu valor inicial (0) e o destrutor só é executado quando o contador volta a este valor.

Assim, a forma geral do código dos construtores e destrutores de classe explícitos é dada por:

```
static int __tSTCount = 0;

void __b00XXX ()
{
   if (__tSTCount++) return;
   /* chamadas a construtores estáticos implícitos */
   /* inicialização das variáveis com escopo de classe */
   /* chamada ao construtor estático explícito */
}

void __e00XXX ()
{
   if (--__tSTCount) return;
   /* chamada ao destrutor estático explícito */
   /* finalização das variáveis com escopo de classe */
   /* chamadas a destrutores estáticos implícitos */
}
```

Os construtores e destrutores explícitos são traduzidos para funções sem parâmetros e sem retorno, codificadas da mesma forma que uma função estática de Cm.

6.3.4 Construtores e destrutores de objeto (dinâmicos)

Os construtores e destrutores de objetos são responsáveis pela inicialização e finalização das variáveis alocadas no escopo de objeto da sua classe.

O construtor de objeto implícito de uma classe inicializa as variáveis com alocação no escopo de objeto da classe que tenham valor inicial determinado na respectiva declaração. Como o programador pode fornecer vários construtores de objeto explícitos, a execução destes é feita externamente ao construtor de objeto implícito, imediatamente em seguida à execução deste.

O destrutor de objeto implícito da classe executa as funções inversas: inicialmente executa o destrutor estático explícito da classe (caso tenha sido definido), e prossegue com a destruição das variáveis alocadas no escopo de objeto (na ordem inversa da inicialização).

código gerado

Os construtores e destrutores de objeto recebem o endereço do objeto a ser manipulado como primeiro parâmetro; parâmetros adicionais dos construtores explícitos são passados após esse endereço. Os destrutores não têm retorno e os construtores retornam este mesmo endereço. Quando um construtor de objetos explícito é utilizado, seu primeiro argumento é uma chamada ao construtor de objetos implícito.

Assim, a forma geral dos construtores e destrutores de objeto implícitos é dada por:

```
struct __A00XXX *__a00XXX (self)
   struct __A00XXX *self;
{
   /* inicializações das variáveis com escopo de objeto */
   return self;
}

void __d00XXX (self)
   struct __A00XXX *self;
{
   /* chamada ao destrutor de objetos explícito */
   /* destruição das variáveis com escopo de objeto */
}
```

Os construtores e destrutores explícitos são traduzidos como funções dinâmicas (com self), com os parâmetros e retorno descritos acima.

Invocação do construtor de objetos

As declarações de objetos tipo classe são decompostas em duas etapas, a declaração em si e comandos de inicialização, que são chamadas ao construtor de objetos implícito e ao construtor de objetos explícito correspondente. Assim, a seguinte declaração

```
XXX x @ (123, "abóbora", 3.14);
```

é traduzida para algo como

```
struct __A00XXX x;
__fNNN00XXX (__a00XXX (&x), 123, "abóbora", 3.14));
```

6.3.5 O construtor *default*

O código gerado pelo compilador Cm considera que o construtor *default* é implementado por uma função sem argumentos (além do **self**). Isso facilita a geração do código, como proposto em **[C++93]**.

Porém, o construtor *default* definido pelo programador pode ter argumentos se estes tiverem valor *default*. Nestes casos o compilador define uma função para ser usada como construtor *default* que apenas invoca o construtor definido pelo programador com os argumentos dados.

O construtor default é sempre usado de maneira homogênea pelo identificador __D<II><XXX>, definido ou como uma macro (caso o construtor *default* definido pelo programador não tenha parâmetros adicionais) ou como uma função. Assim, no primeiro caso teremos algo como

```
#define __D00XXX(self) __fAAAXXX(__a00XXX((self)))
```

e no segundo caso algo como

```
struct __A00XXX *__D00XXX(self)
```

```
    struct __A00XXX *self;
{
    return __fAAAXXX(__a00XXX(self),
        /* valores default dos parâmetros adicionais */
    );
}
```

onde `__fAAAXXX` corresponde à função gerada para o construtor *default* definido pelo usuário.

6.4 Inicialização de dados (estáticos e automáticos)

A inicialização de uma determinada variável é feita na própria declaração, sempre que possível, ou seja, sempre que seu valor seja uma constante em tempo de compilação. A inicialização de variáveis cujos valores somente sejam definidos em tempo de execução é feita por comandos, ou seja, as "definições" dessas variáveis são desmembradas em "declarações" e "comandos de inicialização". Esses comandos dependem do tipo da variável envolvidas. Cada caso é discutido a seguir.

6.4.1 variáveis de tipo padrão e referência

A incialização de variáveis de tipo padrão é bastante simples, sendo executada por um comando de atribuição. O tipo referência é traduzido para um apontador e uma referência (operador '&') é introduzida no seu valor inicial, sendo então tratado como um tipo padrão.

Assim, as declarações

```
int i = 10;
double d = i;
double *pd = &d;
int &pi = i;
```

podem ser traduzidas para algo como[1]

```
int i = 10;
double d;
double *pd;
int *pi;

d = i;
pd = &d;
pi = &i;
```

1. A tradução efetivamente realizada depende do escopo de alocação dessas variáveis, como será visto em seções posteriores.

6.4.2 variáveis tipo classe

Variáveis de tipo classe podem ser inicializadas de três formas distinta, dependendo de sua declaração:

1. inicialização por construção de valor: ocorre caso um construtor seja indicado explicitamente pelo operador '@'. Neste caso a variável é iniciada invocando-se esse construtor.
2. inicialização por atribuição: ocorre caso o valor inicial seja dado pelo operador '='. Neste caso a inicialização é realizada pela primeira operação possível entre as seguintes:
   I. utilizar o construtor de cópia da classe em questão;
   II. utilizar um operador de atribuição da classe em questão;
   III. realizar uma cópia *byte-a-byte* do valor inicial.
3. inicialização *default*: ocorre caso o valor inicial não seja explicitado. Neste caso apenas invoca-se o construtor *default* da classe.

Uma possível tradução para cada um desses casos, por exemplo, a partir de

```
Date d1 @ (31,12,1945);
Date d2i = d1;
Date d2ii = 3457;
Date d2iii = d1;
Date d3;
```

se assemelha a

```
Date d1, d2i, d2ii, d2iii, d3;
__a00Date (&d1, 31, 12, 1945);
__fCopy (&d2i, &d1);
__fAtrib (&d2ii, 3547);
__Cmbcopy (&d2iii, &d1, sizeof (d2iii));
__D00Date (&d3);
```

Onde `__fCopy` representa o construtor de cópia da classe `Date` e `__fAtrib` representa um operador de atribuição de um **long** para um `Date`, por exemplo.

6.4.3 estruturas e uniões

Estruturas e uniões não iniciadas estaticamente requerem ou uma cópia *byte-a-byte* de seu valor (quando iniciadas a partir de outra estrutura do mesmo tipo) ou uma expansão do código para iniciar seus membros um a um. Por exemplo, a declaração

```
struct { int a; Date d1 } xx = { 10, @Date(1,1,1) };
```

precisa ser expandida, uma vez que o valor do membro `d1` deve ser inicializado executando-se um construtor. O código gerado é semelhante a:

```
struct { int a; Date d1; } xx;

xx.a = 10;
```

```
__Date (&xx.d1, 1, 1, 1);
```

Note que, caso a variável tenha alocação estática, o valor de `xx.a` pode ser incluído na própria declaração, evitando a primeira atribuição.

6.4.4 vetores

Vetores não inicializados estaticamente requerem a execução de um comando iterativo para iniciar cada um de seus membros. O comando executado depende do tipo dos elementos do vetor.

Por exemplo, a seguinte declaração

```
Date[100] da;
```

é traduzida para algo como

```
Date da[100];
int __tmp0;

for (__tmp0 = 0; __tmp0 < 100; __tmp0++)
  __D00Date (&da[__tmp0]);
```

6.5 Gerência de memória (dados dinâmicos)

O gerenciamento de memória é feito através de funções da biblioteca Cm, descritas na seção 5.3.3, página 5-7. Basicamente estas rotinas implementam os operadores **new** (baseado na rotina `malloc` da biblioteca padrão C), **release** (baseado na rotina `free` da biblioteca padrão C), de igualdade e de atribuição (com semântica de cópia e comparação *byte-a-byte*).

Os operadores **new** e **release** são mapeados nas rotinas `__Cmalloc` e `__Cmfree`. A rotina `__Cmalloc` recebe o número de objetos a ser alocado e o tamanho de cada um deles, e também o construtor a ser invocado para cada objeto criado (no caso de alocação de objetos de tipo classe). No caso da alocação de *arrays* ou matrizes, o número de objetos a ser alocado corresponde ao número total de objetos do *array* ou matriz. Nos demais casos o número é 1 (um único objeto).

As operações de comparação e cópia são mapeadas nas rotinas `__Cmbcmp` e `__Cmbcopy`.

Assim, o seguinte trecho

```
struct { int a, b; } s1, s2;
Date* d = new (Date, 50);

if (s1 != s2)
  s1 = s2;
```

é traduzido para

```
struct { int a, b; } s1, s2;
__A00Date *d;

d = __Cmalloc (50, sizeof (__A00Date), __D00Date);
if (! __Cmbcmp (&s1, &s2, sizeof (s1)))
   __Cmbcopy (&s1, &s2, sizeof (s1));
```

6.6 Declarações de dados

A tradução sintática de declarações é bastante simples, exceto para tipos classe. A maior dificuldade na tradução de declarações é resultante das diferenças semânticas das classes de armazenamento de Cm e de C, além das formas estendidas de inicialização de dados (principalmente classes) de Cm.

A inicialização de variáveis tipo classe pode requisitar a execução do construtor apropriado, praticamente impondo que as definições de dados sejam separadas em declarações puras (sem inicialização) e comandos de inicialização (avaliação e atribuição dos valores iniciais, chamadas a construtores e chamadas a operadores de cópia ou de atribuição).

Os comandos de inicialização devem ser executados de forma a manter a integridade semântica das declarações originais. Para isso, o compilador Cm introduz comandos a nível global (função main), de classe (construtores/destrutores estáticos) e de objeto (construtores/destrutores dinâmicos), responsáveis por realizar as operações necessárias em cada nível. Essas operações são descritas nas sub-seções seguintes.

6.6.1 Constantes

A princípio, constantes não são declaradas no código gerado, pois o compilador pode substituir a ocorrência de uma constante por seu valor. Porém esta substituição pode ser complexa, ou mesmo impossível, quando o tipo da constante é complexo. Note ainda que a determinação do valor de constantes tipo classe provavelmente envolve a execução de um construtor. Nesses casos o compilador introduz uma variável global que armazena o valor da constante, substituindo suas ocorrências por essa variável.

Para padronizar e simplificar a geração de código, o compilador Cm sempre substitui ocorrências de constantes de tipo padrão ou apontador por seu valor, e sempre gera variáveis para representar constantes de tipos estruturados. Os comandos de inicialização e destruição necessários são introduzidos no programa principal (função `main` gerada).

Por exemplo, o seguinte trecho de uma classe `X<>` qualquer

```
const int SIZE = 10;
const int[SIZE] KEYS = { 123, 423, 367, 88, 47, *:1 };
  ...
for (i = 0; i < SIZE; i++)
   key = g (key, KEYS[i]);
```

é traduzido para

```
static int __cAAA00X[10] =
   { 123, 423, 367, 88, 47, 1, 1, 1, 1, 1 };
   ...
for (i = 0; i < 10; i++)
   key = g (key, __cAAA00X[i]);
```

6.6.2 Variáveis externas

Variáveis exterrnas são apenas declaradas como tal, e nunca são alocadas nem inicializadas automaticamente pelo compilador Cm.

6.6.3 Variáveis globais

Variáveis globais são traduzidas como variáveis externas nos arquivos ou blocos em que são declaradas, sendo tratadas da mesma forma que variáveis externas. Além disso, variáveis globais são introduzidas como variáveis globais no arquivo do programa principal gerado pelo compilador. Seu valor inicial é introduzido na própria declaração, se possível. Caso sejam necessários, comandos para inicialização e destruição são introduzidos no programa principal (função `main`).

6.6.4 Variáveis estáticas

Variáveis estáticas de Cm são traduzidas como variáveis globais estáticas ou como membros da estrutura da classe, de acordo com seu nível léxico. Essa distinção se deve ao fato que variáveis estáticas de bloco são estáticas do ponto de vista da função em que são declaradas, mas podem ser automáticas do ponto de vista dos objetos da classe (mantém seu valor em relação ao objeto usado para ativar a função, mas não entre qualquer ativação da função).

Os vários casos de variáveis estáticas serão discutidos com base no seguinte exemplo:

```
class x<>
import Date;

static int st_s = 10;
static Date st_c @ (6,4,66);
static struct { int a,b; } st_e = { 1,2 };

static void sf()
{
   static int st_sf_s = 20;
   static Date st_sf_c @ (21,12,70);
   static struct { int a,b; } st_sf_e = { 3,4 };
}

void f()
{
```

```
  static int st_f_s = 30;
  static Date st_f_c @ (23,7,68);
  static struct { int a,b; } st_f_e = { 5,6 };
}
```

6.6.4.1 alocação no escopo de classe (estática)

Variáveis estáticas com alocação no escopo de classe (nível léxico 0 ou pertencentes a blocos de funções estáticas) são declaradas como variáveis globais estáticas no arquivo da classe a que pertencem. Os valores iniciais são introduzidos na própria declaração, quando possível. Comandos necessários para inicialização e destruição são introduzidos nos construtores e destrutor estáticos da classe.

No exemplo acima, as variáveis da classe `x` e da função `sf` são tratadas dessa forma e alocadas como variáveis globais:

```
int __st_s = 10;
struct Date __st_c;
struct { int a,b; } __st_e = { 1,2 };
int __st_sf_s = 30;
struct Date __st_sf_c;
struct { int a,b; } __st_sf_e = { 3,4 };
```

As variáveis podem ser inicializadas na própria declaração, exceto pelas variáveis tipo classe (`Date`), que são tratadas nos construtores/destrutor estáticos:

```
/* No construtor estático: */
  __aDate (&__st_c, 6, 4, 66);
  __aDate (&__st_sf_c, 21, 12, 70);

/* No destrutor estático: */
  __dDate (&__st_sf_c);
  __dDate (&__st_c);
```

6.6.4.2 alocação no escopo de objeto

Variáveis estáticas com alocação no escopo de objeto (nível léxico maior que 0 e pertencentes a blocos de funções automáticas) são declaradas como variáveis globais no arquivo da classe a que pertencem. Comandos para inicialização e destruição são introduzidos nos construtores e destrutor de objetos da classe. Valores iniciais de tipo estruturado são declarados como variáveis estáticas (nomeadas com "categoria" 'v', no formato __v<NNN><II><XXX>), e copiados para os respectivos objetos pelos comandos de inicialização.

No exemplo acima, somente as variáveis da função `f` são tratadas dessa forma. Elas são inseridas como membro da estrutura do objeto:

```
struct x {
  int st_f_s;
  struct Date st_f_c;
  struct { int a,b; } st_f_e;
};
```

E inicializadas/destruídas nos construtores/destrutores de objeto (dinâmicos). O valor inicial da variável `st_f_e` é armazenado em uma variável global[1]. Somente a variável `st_f_c` precisar ser tratada no destrutor.

```
static struct { int a,b; } __vst_f_e = { 5,6 };

/* No construtor de objeto: */
   st_f_s = 30;
   __aDate (&st_f_c, 23, 7, 68);
   __Cmbcopy (&st_f_e, &__vst_f_e, sizeof(__vst_f_e));

/* No destrutor de objeto: */
   __dDate (&st_f_c);
```

6.6.5 Variáveis automáticas

Variáveis automáticas de Cm pertencem ou ao escopo de objeto (nível léxico 0) ou ao escopo de bloco (nível léxico maior que 0).

Elas sempre são iniciadas por comandos introduzidos no nível adequado. Valores iniciais de tipo estruturado são declarados como variáveis estáticas (nomeadas com "categoria" 'v', no formato `__v<NNN><II><XXX>`), e copiados para os respectivos objetos pelos comandos de inicialização. Comandos para ativação de construtores/destrutores podem ser inseridos.

A classe a seguir apresenta os vários casos de variáveis automáticas discutidos a seguir.

```
class x<>
import Date;

int v1 = 10;
Date v2 @ (6,4,66);
struct { int a,b; } v3 = { 1,2 };

void f()
{
   int v1_f = 30;
   Date v2_f @ (23,7,68);
   struct { int a,b; } v3_f = { 5,6 };
}
```

6.6.5.1 alocação em escopo de objeto

Variáveis automáticas de nível léxico 0 pertencem ao escopo de objeto, e são traduzidas como membros da estrutura da classe a que pertencem. Elas são iniciadas e destruídas (quando necessário) por comandos introduzidos nos construtores e

1. Este valor poderia ser armazenado como uma variável local estática, porém sendo uma variável global é mais fácil para o compilador fazer otimizações, como por exemplo criar uma única variável para todas as declarações com o mesmo valor inicial.

destrutor de objetos de sua classe. O tratamento é basicamente o mesmo que o dado a variáveis estáticas com alocação a nível de objeto.

No exemplo acima, somente as variáveis da classe x são tratadas dessa forma. Elas são inseridas como membro da estrutura do objeto:

```
static struct { int a,b; } __vv3 = { 1,2 };

/* No construtor de objeto: */
   v1 = 10;
   __aDate (&v2, 6, 4, 66);
   __Cmbcopy (&v3, &__vv3, sizeof(__vv3));

/* No destrutor de objeto: */
   __dDate (&v2);
```

6.6.5.2 alocação em escopo de bloco (automática)

Variáveis automáticas de nível léxico maior que 0 pertencem ao escopo de bloco, e são traduzidas como variáveis locais de blocos. Elas são iniciadas e destruídas (quando necessário) por comandos introduzidos no bloco a que pertencem.

O C padrão K&R não permite inicialização de variáveis estruturadas automáticas. Estes valores são então armazenados em variáveis globais e copiados para as variáveis locais.

No exemplo acima, somente as variáveis da função f são tratadas dessa forma. O código gerado para esta função é dado por:

```
struct { int a,b; } __vv3_f = { 5,6 };

void __f_x ()
{
   int v1_f = 30;
   Date v2_f;
   struct { int a,b; } v3_f;

/* inicialização */
   __aDate (&v2_f, 23, 7, 68);
   __Cmbcopy (&v3_f, &__vv3_f, sizeof (__vv3_f));

/* destruição */
   __dDate (&v2_f);
}
```

Os comandos do corpo da função, caso existissem, seriam inseridos entre as seções de inicialização e destruição das variáveis locais.

6.7 Funções

Funções Cm são traduzidas para funções C (K&R). Esta tradução é em grande parte sintática, exceto possivelmente pelos comandos que compõe o corpo da função, envolvendo:

- tradução das construções Cm para construções análogas em C;
- geração de novo nome para a função, caso necessário;
- introdução de parâmetros implícitos, como **self** e **result**, caso necessário;
- simulação de passagem e retorno de objetos de tipo estruturado[1].

As maiores dificuldades com relação a funções surgem no momento em que são utilizadas (chamadas de funções), o que pode exigir uma reestruturação da expressão original.

A seguir são mostrados vários aspectos da tradução de declarações e chamadas a funções. Para tornar a explicação mais clara, cada tópico é apresentado isoladamente dos demais, porém convém ressaltar que uma dada função pode recair em vários dos casos (o que em geral ocorre).

Note que muitos identificadores são mapeados para evitar conflitos. Para maior simplicidade e clareza, este mapeamento será indicado apenas prefixando-se os identificadores envolvidos com "__", exceto se for conveniente explicitar o mapeamento utilizado.

Note ainda que esta seção está preocupada mais em expor a tradução de funções em si (declarações e chamadas), relegando a tradução dos comandos de seu corpo a segundo plano.

6.7.1 Os casos mais simples

Funções externas ou globais, com parâmetros e tipo de retorno não-estruturados são as que apresentam tradução mais direta, pois não envolvem parâmetros implícitos nem artifícios na passagem dos parâmetros.

Assim, a função `succ` da classe `x`

```
class x<>

global int succ (int i)
{
   return ++i;
}
```

é traduzida diretamente para

1. Note que o código gerado segue padrão K&R, que não permite parâmetros de tipos estruturados.

```
int succ (i)
  int i;
{
  return ++i;
}
```

6.7.2 Mapeamento do identificador da função

Caso a função não seja externa nem global, pelo menos seu identificador é alterado, para evitar conflitos entre funções de mesmo nome entre classes distintas ou entre funções sobrecarregadas em uma mesma classe. Note que as funções sempre deverão ter um escopo global em C, pois elas podem ser usadas por outras classes (mesmo as não exportadas podem ser usadas a partir de classes derivadas).

Assim, caso a função `succ` mostrada anteriormente tivesse classe de armazenamento **static**, ao invés de **global**, seria convertida em

```
int __fAAA00x (i)
  int i;
{
  return ++i;
}
```

6.7.3 Parâmetro implícito self

O objeto **self** definido em funções automáticas (de escopo de objeto) é implementado como um parâmetro adicional de tipo apontador para a própria classe. Este parâmetro é adicionado antes dos parâmetros explícitos da função (somente não é o primeiro caso o tipo de retorno seja estruturado – vide "Retorno de tipo estruturado", página 6-21). Assim, caso a função `succ` fosse declarada como **auto** (ou sem classe de armazenamento explícita), seria traduzida para algo como

```
int __fAAA00x (self, i)
  struct __A00x *self;
  int i;
{
  return ++i;
}
```

Onde `__A00x` corresponde à estrutura de um objeto da classe `x`. O endereço do objeto utilizado para fazer a seleção da função é passado como valor para o parâmetro **self** no momento da chamada da função, e todas as variáveis de escopo de objeto mencionadas na função são usadas através de **self**. Assim, cada execução da função opera sobre o objeto utilizado na respectiva chamada.

Por exemplo, o seguinte trecho

```
class y<>
int a;
void f (int i)
{
```

```
    a=i;
}
  ...
y<> y1, y2;
y1.f(10);
y2.f(20);
```

é traduzido para

```
struct __y {
  int a;
};

void __f (self, i)
  struct __y *self;
  int i;
{
  (self -> a) = i;
}
  ...
struct __y y1, y2;
__f (&y1, 10);
__f (&y2, 20);
```

6.7.4 Parâmetros de tipo estruturado

C padrão K&R não permite a passagem de parâmetros de tipo estruturado. Porém Cm oferece essa facilidade, simulando-a através de apontadores. Assim, a passagem de cada parâmetro de tipo estruturado é feita nos seguintes passos:

1. Ao invés da estrutura em si, é passado seu endereço (isto é, um apontador para ela);
2. A função declara uma variável local com o tipo da estrutura e copia o valor dado pelo apontador para esta variável (através da função de biblioteca __Cmbcopy, por exemplo);
3. Os acessos ao parâmetro são substituídos por acessos à respectiva variável local.

Note que a criação de uma variável local e da cópia são necessárias para manter a semântica de passagem por valor.

Assim, a seguinte classe

```
class x<>
type Y = struct { int a, b; }
static void fpar (Y y)
{
  fpar (y);
}
```

é traduzida para

```
typedef struct { int a, b; } __Y;
void __fpar (_y)
   __Y *_y;
{
   __Y y;
   __Cmbcopy (&y, _y, sizeof (__Y));

   __fpar (&y);
}
```

Esta função recursiva é completamente inútil, a não ser como exemplo.

6.7.5 Retorno de tipo estruturado

O retorno de objetos de tipo estrutura apresenta um problema equivalente ao da passagem de parâmetros de tipo estruturado. A solução adotada é semelhante – a função retorna o endereço do objeto, e não o objeto em si – mas difere em um ponto importante: o objeto que contém o valor de retorno não é alocado pela função em si, mas sim por quem a está executando. Dessa forma, é necessário um parâmetro adicional para passar o endereço deste objeto para a função. Este parâmetro, quando utilizado, é sempre inserido como o primeiro parâmetro da função.

Essa solução tem a vantagem de não exigir alocação dinâmica (via `malloc` ou equivalente), evitando probelams de fragmentação e liberação da memória alocada. Note que a alocação do valor de retorno fora da função permite uma tradução bem mais eficiente de recursão e chamadas encadeadas.

Assim, a seguinte classe

```
class x<>
type Y = struct { int a, b; }
static Y fret (int i)
{
   Y value;
   value.a = value.b = i;
   return value;
}
```

é traduzida para

```
typedef struct { int a, b; } __Y;
__Y *__fret (__tret, i)
   __Y *tret;
   int i;
{
   __Y value;
   value.a = value.b = i;
   __Cmbcopy (__tret, &value, sizeof (value));
   return __tret;
}
```

chamadas a funções com retorno de tipo estruturado
Os temporários utilizados para armazenar valores estruturados retornados por funções são alocados de forma automática (variáveis locais de blocos) no nível do menor comando que engloba a chamada da função. Se necessário, é introduzido um novo bloco para permitir a declaração do temporário.

Por exemplo, o seguinte comando

```
if (ycmp (fret (i), fret (j)))
  yprint (fret (i));
```

é traduzido para

```
{
  __Y __tmp1, __tmp2;
  if (__ycmp (__fret(&__tmp1,i), __fret(&__tmp2,j)))
  {
    __Y __tmp3;
    __yprint (__fret (&__tmp3, i));
  }
}
```

otimização na criação de temporários
O compilador evita a criação de temporários para armazenar o valor retornado pela função caso estes sejam usados apenas para fazer uma cópia deste valor para outro objeto. Quando possível, o compilador passa o endereço do objeto diretamente no lugar do temporário. Esta situação ocorre mais freqüentemente em declarações, atribuições e passagens de parâmetros. Por exemplo, o comando

```
Y y = fret (5);
```

que normalmente seria traduzido para

```
__Y y;
{
  __Y __tmp1;
  __Cmbcopy (&y, __fret (&__tmp1, 5), sizeof (__Y));
}
```

é gerado na forma otimizada:

```
__Y y;
__fret (&y, 5);
```

6.7.6 Pseudo-variável result

A pseudo-variável **result** é implementada de forma diferente dependendo do tipo de retorno da função, visando sempre uma melhor eficiência. Obviamente, se o tipo de retorno for **void**, **result** não é utilizada.

Caso o tipo de retorno seja não-estruturado, **result** é implementada como uma variável local de bloco (`__tret`) com este tipo, declarada no nível léxico do corpo da função (nível léxico 1), e acessos (explícitos ou implícitos) a **result** são traduzidos como acessos a `__tret`. Caso **result** não seja usado de forma alguma, `__tret` não precisa ser declarada. Como exemplo, a função

```
static int fres1 ()
{
   result = 10;
   return;
}
```

é traduzida para

```
int __fres1 ()
{
   int __tret;
   __tret = 10;
   return __tret;
}
```

Caso o tipo de retorno seja estruturado, acessos a **result** são traduzidos diretamente como acessos ao temporário usado para implementar o retorno de valores estruturados, dado pelo parâmetro implícito `__tret`. Assim, a função

```
static Y fres2 ()
{
   result.a = 10; result.b = 20;
}
```

é traduzida para

```
__Y *__fres2 (__tret)
   __Y *__tret;
{
   (*__tret).a = 10; (*__tret).b = 20;
   return __tret;
}
```

otimizações obtidas com result

O artifício implementado pela pseudo-variável **result** permite a escrita de código bem mais compacto e eficiente, apesar de não introduzir nenhuma novidade semântica na linguagem. As funções `get1` e `get2` dadas a seguir são semanticamente equivalentes

```
static Y get1 ()
{
   Y y; y.a = 1; y.b = 2;
   return y;
}
```

```
static Y get2 ()
{
   result.a = 1; result.b = 2;
}
```

porém sua implementação final é bastante diferente

```
__Y *__get1 (__tret)
   __Y *__tret;
{
   __Y y; y.a = 1; y.b = 2;
   __Cmbcopy (__tret, &y, sizeof (__Y));
   return __tret;
}

__Y *__get2 (__tret)
   __Y *__tret;
{
   (*__tret).a = 1; (*__tret).b = 2;
   return __tret;
}
```

6.7.7 Otimização de parâmetros constantes

A princípio, a qualificação **const** apenas indica que o parâmetro não é alterado, restringindo seu uso sem afetar o código gerado. Porém pode-se evitar a cópia de parâmetros estruturados constantes, usado-os diretamente através do endereço passado pelo parâmetro (tratando-os como se tivessem sido passados por referência). Note que isto não fere a passagem por valor do parâmetro porque os acessos feitos a ele somente podem ser de leitura (justamente por ser constante).

Assim, a função

```
static void fcte (const Y y)
{
   int i = y.a;
}
```

que normalmente seria traduzida para

```
void __fcte (_y)
   __Y *_y;
{
   __Y y;
   int i;
   __Cmbcopy (&y, _y, sizeof (__Y));
   i = y.a;
}
```

é traduzida de forma otimizada como

```
void __fcte (y)
  __Y *y;
{
  int i;
  i = (*y).a;
}
```

6.7.8 *Ellipsis*

O uso de parâmetros livres (*ellipsis*) não afeta a chamada da função em si, mas acarreta várias modificações em sua declaração/definição. Basicamente o compilador deve inserir o código para utilizar parâmetros livres, dados pelas macros de `varargs.h` ou `stdarg.h` (ANSI-C) da bilioteca padrão de C.

Essas macros são usadas para:

1. declarar um parâmetro auxiliar para obtenção do endereço da lista (`va_alist` e `va_dcl`[1]);
2. declarar um apontador para o parâmetro corrente (`va_list`);
3. inicializar este apontador e a biblioteca de *run-time* (`va_start`);
4. manipular este apontador para obter um dado parâmetro (`va_arg`);
5. finalizar a biblioteca de run-time (`va_end`).

O compilador Cm insere as macros nos locais adequados, mapeando a utilização da pseudo-função `getarg` na macro `va_arg` ou no apontador do parâmetro corrente. Devido à macro `va_end`, comandos **return** são mapeados em comandos **goto** para o final da função, usando `__tret` (vide seção 6.7.5 e seção 6.7.6) para indicar o valor de retorno.

Assim, a função Cm dada por

```
static int err_printf (char *file; int line; ...)
{
  extern int vfprintf (const char *fmt; ...);
  result = vfprintf ("%s [%d]:\t", file, line);
  if (result >= 0 && getarg)
  {
    char *format = getarg (char*);
    return printf (format, getarg);
  }
  return -1;
}
```

é traduzida para

```
int __err_printf (va_alist)
  va_dcl
```

1. No caso de varargs.

```
{
   char *file; int line;
   va_list __targs;
   int __tret;
   extern int printf ();
   va_start (__targs);
   file = va_arg (__targs, char*);
   line = va_arg (__targs, line);

   __tret = vfprintf ("%s [%d]\t", file, line);
   if (__tret >= 0 && __targs)
   {
      char *format = va_arg (__targs, char*);
      __tret = printf (format, __targs);
      goto ____return;
   }
   __tret = -1;
   goto ____return;

____return:
   va_end (__targs);
   return __tret;
}
```

6.7.9 Chamadas de funções

Chamadas de funções têm tradução bastante complexa, afetada pelos diferentes recursos utilizados, podendo envolver

- inserção de parâmetro implícito **self**;
- inserção de parâmetro `__tret` para simular retorno de valor estruturado (e alocação do temporário para armazenar este valor);
- simulação de passagem de parâmetros estruturados (passagem do endereço do parâmetro, ao invés do parâmetro em si);
- inserção de temporários (em chamadas de funções virtuais);
- inserção de *casting* nos parâmetros da chamada;
- inserção de referências ('**&**', devido a passagem por referência);
- projeção de argumentos e/ou promoção do retorno (em funções herdadas).

Os três primeiros itens já foram descritos anteriormente. Os demais serão descritos a seguir.

6.7.9.1 funções virtuais

Chamadas a funções virtuais são feitas de forma indireta, através dos apontadores inseridos no objeto (vide "funções virtuais", página 6-4). Num caso simples, a tradução é imediata, como na expressão

```
obj.fvirtual (...)
```

traduzida para

```
(*obj.fvirtual) (&obj._fvirtual, ...)
```

Note que a expressão usada para selecionar a função é utilizada duas vezes no código gerado: uma para determinar a função a ser chamada – `(*obj).fvirtual` – e outra para determinar o valor a ser usado para **self** – `obj._fvirtual`. Em outros casos essa expressão pode ter efeitos colaterais ou ser muito complexa, impedindo ou tornando indesejável essa duplicação, como em

```
int i = 0;
Obj[10] obj;
  ...
(obj[i++]).fvirtual (...);
```

A tradução simples deste comando utilizaria os membros `fvirtual` e `_fvirtual` de objetos diferentes, além de incrementar `i` duas vezes. Assim, é necessária a introdução de um dado temporário, evitando a reavaliação da expressão. Este temporário deve armazenar o endereço do objeto obtido pela expressão, calculando-o somente uma vez, antes da chamada ser realizada (para evitar problemas com ordem de avaliação). A tradução da expressão acima fica

```
{
  __Obj *__tmp1;
  (__tmp1 = &(obj[i++]),
    (*__tmp1).fvirtual ((*__tmp1)._fvirtual, ...));
}
```

6.7.9.2 parâmetros tipo referência

Em passagens por referência (parâmetro formal com tipo referência) é inserido um operador '`&`' no parâmetro real, associando seu endereço ao parâmetro formal correspondente (um apontador, como mostrado na seção 6.2.1). Assim, o seguinte trecho

```
static void swap (int& i, j)
{
  int t=i; i=j; j=t;
}
  ...
int a,b;
swap (a,b);
```

é traduzido para

```
void __swap (i,j)
  int *i, *j;
{
  int t=(*i); (*i)=(*j); (*j)=t;
}
  ...
int a,b;
__swap (&a,&b);
```

6.7.9.3 *casting*

Como o código gerado segue padrão K&R, os tipos dos argumentos formais não são considerados pelo compilador C para gerar o código correspondente à chamada. Para que estas chamadas fiquem corretas é necessária a conversão (*casting* explícito) de parâmetros reais para os tipos apropriados, nos casos em que o tratamento normal dado pelo padrão K&R não seja conveniente.

Assim,

```
extern int check (int i,j,k);
  ...
check (10, 'a', 3.1416);
```

é traduzido para

```
extern int check ();
  ...
check (10, 'a', (int)3.1416);
```

6.7.9.4 funções herdadas

Suponha as seguintes declarações:

```
class Base<>
export Base& getself ()
{
  return self;
}

class Deriv<>
inherit Base;
```

A função `getself` pertence à classe `Base`, sendo traduzida para

```
__Base *__getself (__Base *self)
{
  return self;
}
```

porém pode ser usada a partir de objetos da classe `Deriv`

```
Deriv obj;
obj.getself ();
```

Uma opção para suportar esta chamada seria a recodificação automática da função `getself` para a classe `Deriv`. Esta opção é aceitável, e até mesmo vantajosa, no caso de funções simples como `getself`. No caso geral, porém, a replicação de código decorrente das relações de herança torna essa opção inconveniente ou até mesmo inaceitável.

A solução adotada por Cm é não replicar o código da função da classe base, convertendo os parâmetros e o valor retornado para os tipos apropriados, quando necessário.

projeção (derivada para base)
A projeção de um objeto de uma classe derivada para uma de suas classe base é feita inserindo-se uma seleção do membro apropriado (membro que representa a classe base na estrutura da classe derivada).

```
base:       obj
derivado:   (obj).__i00Base
```

promoção (base para derivada)
A promoção de um objeto de uma dada classe (base) para uma de suas classes derivadas é mais complicada. A idéia da conversão é determinar o endereço de um objeto da classe derivada tal que sua promoção para a classe base resulte no objeto inicial, ou seja, para que seu membro herdado correspondente à classe base tenha o mesmo endereço que o objeto original.

A "distância" em *bytes* desses dois endereços pode ser expressa a partir de um objeto qualquer da classe derivada, subtraindo-se os endereços do objeto e do membro herdado correspondente à classe base desse objeto:

```
DELTA == ((char*)&(obj.__i00Base) - (char*)&obj)
```

Porém, no momento em este cálculo é necessário, pode acontecer de não haver um objeto da classe derivada disponível. Isso não chega a ser problema pois como o objeto em si não é utilizado, o cáculo pode ser feito em cima de qualquer endereço válido (ADDR), inserindo-se operações de `cast` adequadas:

```
ADDR == 0[1]
OBJ == (*(__Base*)ADDR)
DELTA == ((char*)&(ADDR.__i00Base) - (char*)&ADDR)
```

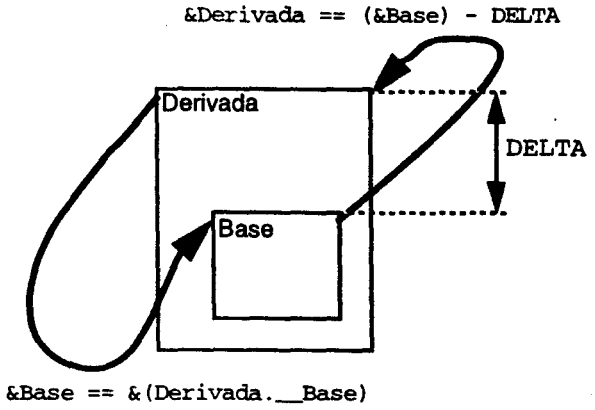
Tendo este valor em mãos, a promoção fica simples, bastando "deslocar" o endereço disponível do número de *bytes* dados por este valor. Assim, dado um obeto `obj` da classe base, o correspondente objeto da classe derivada é dado por

```
*((__Deriv*)(((char*)(&obj))-DELTA))
```

A figura 6-3 mostra esquematicamente essas conversões.

1. Alguns sistemas podem não permitir o uso do endereço 0 (NULL), apesar dele não ser derreferenciado efetivamente. Porém é fácil determinar um endereço válido que possa ser usado, com base nas restrições de alinhamento de dados da máquina em questão.

FIGURA 6-3 Conversões entre classes base e derivadas



6.8 Comandos

A geração de comandos Cm é bastante simples, uma vez que os comandos Cm e C são bastante parecidos. As maiores complicações ocorrem nos comandos **for**, **switch** e **return**, cuja semântica foi estendida com relação a C. A seguir são discutidas as alterações que podem ser introduzidas para cada comando.

Alguns comandos não sofrem alteração alguma, exceto as possivelmente introduzidas por suas expressões ou comandos internos. É o caso, por exemplo, de comandos de avaliação de expressões e de comandos compostos, **if**, **while** e **do–while**.

A tradução de comandos protegidos é descrita posteriormente (seção 6.9).

6.8.1 comando for

O comando **for** pode ter uma declaração no lugar de sua primeira expressão. Nesse caso é criado um bloco contendo a declaração (sem valor inicial) e o comando **for**, convertendo-se a declaração em uma expressão. Note que a declaração pode introduzir mais de um identificador e que o bloco externo é necessário para delimitar o escopo das variáveis correspondentes.

Assim, o comando

```
for (Node* p = head, q = NIL; p ; p ->= next)
  q = work_on_node (p, q);
```

é traduzido para

```
{
```

```
    Node *p, *q;
    for (p = head, q = NIL; p; p = p -> next)
      q = work_on_node (p, q);
}
```

6.8.2 comando switch

O comando **switch** apresenta dois problemas na tradução: a extensão sintática dos *labels* **case**, e a possibilidade de declarações dentro do bloco.

O primeiro problema, mostrado em

```
switch (c)
{
  case '0'..'9':
    digit ();
    break;
  case ' ', '\t', '\n':
    white ();
    break;
}
```

tem solução trivial, bastando uma tradução sintática:

```
switch (c)
{
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
    digit ();
    break;
  case ' ': case '\t': case '\n':
    white ();
    break;
}
```

Já o segundo problema é mais complicado, uma vez que as declarações inseridas no case podem exigir a execução de comandos na tradução (para inicializar dados de tipo classe, por exemplo), como no exemplo a seguir:

```
switch (n++)
{
  Date d @ (23,4,1923);
  int i = n;

  case 'i': i = 4; break;
  case 'd': d ++;
}
```

Nesse caso é preciso "tirar" as declarações do **switch** para permitir que os dados sejam inicializados, inserindo-se um bloco envolvendo o comando e passando as declarações e inicializações correspondentes para o bloco. Note que, asssim, a

avaliação das expressões da declarações e do seletor do **switch** ficam invertidas, podendo gerar problemas (a variável `i` é inicializada erradamente). Para evitar estes problemas, a expressão do **switch** é avaliada antes das declarações e seu valor armazenado em um dado temporário:

```
{
   long __tsw;
   Date d;
   int i;
   __tsw = n++;
   __D00Date (&d);
   i = n;
   switch (__tsw)
   {
      case 'i': i = 4; break;
      case 'd': __finc (&d);
   }
}
```

6.9 Tratamento de exceções

O mecanismo de tratamento de exceções (MTE) é baseado nas pseudo-funções (macros) `setjmp` e `longjmp` da biblioteca padrão C.[1]

`setjmp` armazena o contexto do processador em um *buffer* (de tipo `jmp_buf`) que permite a `longjmp` forçar a retomada da execução a partir daquele ponto; esta função retorna 0 no caso do retorno "normal" ou um valor diferente de 0 (dado por `longjmp`) no caso do retorno forçado por `longjmp`.

`longjmp` recebe um *buffer* de contexto do processador e força a retomada do processamento no ponto dado pelo contexto; também recebe um valor inteiro que é usado como valor de retorno de `setjmp`.

6.9.1 Visão geral do MTE

De forma geral, o controle do MTE é bastante simples: o compilador mantém uma pilha de contextos ativos (`jmp_buf`). A ativação/desativação de um tratador corresponde a um "empilhamento/desempilhamento" do correspondente contexto. O levantamento de uma exceção corresponde a um desvio ao contexto indicado pelo topo desta pilha.

representação de tipos em tempo de execução

Como exceções são identificadas por seu tipo é necessário um mecanismo para representar um tipo em tempo de execução. Esta representação é feita por uma

1. Futuramente, com a inserção de concorrência e portas de comunicação, será necessário o uso de `sigsetjmp` e `siglongjmp`, que são mais dispendiosas mas operam corretamente em conjunto com sinais (`signal`, `kill`, etc).

string, denominada *assinatura do tipo*, que permite identificar univocamente o tipo correspondente.

A assinatura de um tipo basicamente descreve a seqüência em que operadores de tipos devem ser aplicados para produzi-lo. Estes operadores e os tipos padrão são representados por caracteres alfabéticos, resultando em uma representação bastante genérica. Por motivos de compatibilidade de tipos, a assinatura também contém os nomes dos tipos envolvidos, quando necessário.

comando protegido
Cada comando protegido é mapeado em um comando condicional (**if**) sobre o retorno de uma chamada a `setjmp` (que já "empilha" o contexto corrente). Um dos ramos do comando condicional corresponde ao processamento normal (bloco protegido) e o outro aos tratadores de exceção.

O contexto é "desempilhado" ao final do comando protegido (correspondendo a uma execução bem sucedida do comando) ou nos tratadores (correspondendo à ocorrência de uma exceção).

Esquematicamente, a tradução de

```
[
   bloco-protegido
   tratadores
]
```

corresponde a

```
if (! setjmp (context[top++]))
{
   tradução-do-bloco-protegido
   top--;
}
else
{
   tradução dos tratadores
}
```

tratadores
A seqüência de tratadores é mapeada em um comando **if** encadeado. Cada condição testa a aplicabilidade do corresponde tratador à exceção corrente, e o **else** final da seqüência corresponde ao tratador *default*. Caso o tratador *default* não seja fornecido, é introduzido um tratador que apenas re-sinaliza a exceção ocorrida, forçando a sua propagação para níveis mais externos de tratadores, caso existam.

comando raise
Os comandos **raise** são mapeados em chamadas à função `__CmMTE_raise` da biblioteca Cm, que recebe o valor e assinatura da exceção, e seu ponto de ocorrência (número da linha e nome do arquivo). Assim, um comando como

```
raise 10;
```

é traduzido para uma chamada à biblioteca do MTE

```
__CmMTE_raise ("i", 10, "XXX.cm", 12);
```

A implementação de `__CmMTE_raise` armazena os valores passados em variáveis globais do MTE, e executa `longjmp` tendo como parâmetro o topo da pilha de contextos (que é desempilhado)[1]:

```
__CmMTE_sig = "i";
  ...
longjmp (context[--top], 1);
```

exceções não tratadas
Caso uma exceção seja sinalizada (ou re-sinalizada) e não haja nenhum tratador estabelecido, então o MTE emite uma mensagem de erro de execução detalhando a exceção ocorrida e o ponto em que ocorreu, e a execução do programa é interrompida.

6.9.2 Implementação

O código gerado para o MTE introduz muitos comandos e altera a estrutura do programa, sendo de difícil compreensão se comparado ao código original. Para reduzir este problema, e também para simplificar a geração do código, o compilador Cm gera código baseado em macros pré-definidas.

6.9.2.1 variáveis globais

O MTE utiliza algumas variáveis globais, que armazenam a pilha de contextos, seu topo e tamanho corrente, e informações relativas à exceção corrente, caso exista. Elas são únicas e seus identificadores são prefixados por "`__CmMTE`", não causando conflitos.

```
int __CmMTE_maxNest = 0;
```

Tamanho máximo da pilha. Pode ser alterado dinâmicamente em tempo de execução, se necessário. Inicialmente nula.

```
int __CmMTE_lvl = -1;
```

Topo corrente da pilha, corresponde ao índice da última posição utilizada (-1 corresponde a pilha vazia).

```
jmp_buf *__CmMTE_ctx = NULL;
```

Pilha de contextos em si, dada por apontadores para *buffers* de contexto.

```
void *__CmMTE_exc = NULL;
char *__CmMTE_sig = NULL;
```

1. O segundo parâmetro de `longjmp` não é usado, mas por convenção do MTE é sempre 1.

Valor e assinatura da exceção corrente.

```
char *__CmMTE_fname = NULL;
int __CmMTE_lno = 0;
```

Ponto de ocorrência da exceção (nome do arquivo e número da linha).

```
int __CmMTE_trc = 0;
```

Trace flag. Se diferente de zero, põe o MTE em modo verboso. Utilizada somente para depuração.

6.9.2.2 funções de biblioteca

O controle do MTE é feito por diversas funções. A maioria é bastante simples, e algumas são utilizadas apenas para acompanhamento (*tracing*) do mecanismo.Ainda, algumas delas executam a mesma função, sendo diferenciadas apenas pelas mensagens de depuração geradas (permitindo melhor acompanhamento por parte do usuário).[1]

```
void __CmMTE_startup (int maxNest);
```

Inicia a biblioteca do MTE, preparando-a para operação. O parâmetro determina o tamanho inicial desejado para a pilha de contextos.

```
void __CmMTE_enterTryBlock ();
```

Rotina executada para "criar" um novo contexto. Basicamente verifica se há espaço na pilha de contextos.

```
void __CmMTE_leaveTryBlock ();
```

Rotina para encerrar ("desempilhar") um contexto, quando a execução do comando protegido encerrou-se normalmente.

```
void __CmMTE_breakTryBlock (int n);
```

Rotina para encerrar um conjunto de contextos, utilizada quando há um desvio "irregular" do fluxo de execução (por exemplo, devido a comandos **break** ou **return**).

```
void __CmMTE_leaveToHandler ();
```

Rotina executada quando o controle é desviado para os tratadores de um comando condicional (quando ocorre uma exceção).

```
void __CmMTE_raise (char *code, void *instance,
                    char *fname, int lineno);
```

1. Todo o mecanismo de *tracing* pode ser removido por compilação condicional, removendo as duplicações e funções exclusivas de *tracing*.

Rotina para sinalização de uma exceção.

```
void __CmMTE_reraise ();
```

Rotina para re-sinalização da exceção corrente.

```
int __CmMTE_matchSignature (char *signature);
```

Rotina para verificação se um dado tratador pode "tratar" a exceção corrente.

6.9.2.3 macros utilizadas

As definições das macros utilizadas na geração de código são dadas a seguir. Note que os caracteres '\' necessários ao final de cada linha (sintaxe de C) foram removidos para obter melhor legibilidade.

```
#define TRY_BEGIN
     {
        __CmMTE_enterTryBlock ();
        if (setjmp (__CmMTE_ctx [__CmMTE_lvl]) == 0)
          {

#define TRY_EMPTY_END
            __CmMTE_leaveTryBlock ();
          }
        else
          {
            __CmMTE_leaveToHandler ();
            __CmMTE_reraise ();
          }
        __CmMTE_sig = (char *)0;
     }

#define TRY_DEFAULT_HANDLER
            __CmMTE_leaveTryBlock ();
          }
        else
          {
            __CmMTE_leaveToHandler ();
            {

#define TRY_HANDLER(s)
            __CmMTE_leaveTryBlock ();
          }
        else
          {
            __CmMTE_leaveToHandler ();
            if (__CmMTE_matchSignature (s))
            {
```

```
#define TRY_WHEN(s)
            }
          else
            if (__CmMTE_matchSignature (s))
              {

#define TRY_DEFAULT
              }
            else
              {

#define TRY_END
              }
          }
        __CmMTE_sig = (char *)0;
      }

#define TRY_RERAISE_END
              }
            else
              __CmMTE_reraise ();
          }
        __CmMTE_sig = (char *)0;
      }
```

6.9.2.4 Exemplos

A utilização das macros torna a tradução bastante simples, e o código traduzido bastante claro. A seguir são mostrados exemplos de como devem ser utilizadas as macros. Note que um programa C qualquer pode se utilizar do MTE como uma biblioteca, incorporando facilidades de tratamento de exceção sem muito esforço.

Quando são utilizados tratadores comuns (específicos para um dado tipo) e o tratador *default*, temos o caso mais completo ou geral do comando. A omissão de um desses "tipos" de tratadores força a adaptação do esquema geral (através do uso de macros diferentes que "englobam" as tarefas de duas ou mais macros do caso geral). A seguir são mostrados exemplos de cada um dos possíveis casos.

caso 1 (geral)

O caso geral envolve tanto tratadores comuns como o tratador *default*, como em

```
[
  doit ();
when char ch:
  handle_char (ch);
when int:
  handle_int (exc);
when char *msg:
  handle_char_ptr (msg);
when default:
  handle_all ();
]
```

No caso geral, o comando é delimitado pelas macros TRY_BEGIN e TRY_END. O primeiro tratador é dado por TRY_HANDLER, o *default* por TRY_DEFAULT e os demais por TRY_WHEN. Assim, o código produzido é dado por:

```
TRY_BEGIN
   doit ();
TRY_HANDLER("c")
   char ch = (char)__CmMTE_exc;
   handle_char (ch);
TRY_WHEN("i")
   int exc = (int)__CmMTE_exc;
   handle_int (exc);
TRY_WHEN("Pc")
   char *msg = (char*)__CmMTE_exc;
   handle_char_ptr (msg);
TRY_DEFAULT
   handle_all ();
TRY_END
```

caso 2
Nenhum tratador é utilizado. O comando é finalizado com a macro TRY_EMPTY_-END, que introduz um tratador *default* para re-sinalizar qualquer exceção ocorrida. Assim,

```
[
   doit ();
]
```

é traduzido para

```
TRY_BEGIN
   doit ();
TRY_EMPTY_END
```

caso 3
Não é usado um tratador *default*. O comando é finalizado com a macro TRY_RE-RAISE_END, que introduz um tratador *default* para re-sinalizar qualquer exceção não tratada pelos tratadores fornecidos. Assim,

```
[
   doit();
when int:
   handle_int (exc);
]
```

é traduzido para

```
TRY_BEGIN
   doit();
TRY_HANDLER("i")
```

```
    int exc = (int)__CmMTE_exc;
    handle_int (exc);
  TRY_RERAISE_END
```

caso 4

Somente o tratador *default* é utilizado. O tratador *default* deve ser indicado com a macro TRY_DEFAULT_HANDLER por ser o primeiro tratador.

```
  [
    doit ();
  when default:
    handle_all ();
  ]
```

é traduzido para

```
  TRY_BEGIN
    doit ();
  TRY_DEFAULT_HANDLER
    handle_all ();
  TRY_END
```

III

Surveys e extensões futuras

Capítulo 7

Survey: Mecanismos de Tratamento de Exceções

O propósito deste *survey* é apresentar brevemente a bibliografia estudada ressaltando seus pontos mais interessantes, sem entrar em muitos detalhes. Os interessados em mais detalhes podem consultar **[MTE92]** ou se referir diretamente às referências.

7.1 Introdução

Um mecanismo de tratamento de exceções (MTE) é um mecanismo de controle de erros que permite ao programa se manter em execução mesmo diante de eventuais falhas ou exceções. Ele estabelece e controla a maneira como "erros" são representados e tratados em um programa. Não há um consenso geral sobre o que são erros ou exceções e nem como devem ser tratados (**[Fail90b]**, **[Fail90]**, **[Fail90c]**), o que dificulta, e enfatiza a necessidade, de uma formalização de um MTE.

Exceção é qualquer falha ou situação anormal que ocorra durante a execução de um programa. Ela pode ter origem tanto no *hardware* sendo utilizado (falha de leitura do disco rígido) como no próprio programa (tentativa de retirar um elemento de uma pilha vazia).

Quando uma exceção ocorre, o sistema deve tomar "precauções" para tentar corrigir os efeitos negativos ou destrutivos que ela possa causar. Essas "precauções" correspondem ao tratamento de exceções. Por exemplo, se ocorre uma falha na leitura de um registro de um arquivo, o programa sendo executado não deve processar o registro lido (pois seu conteúdo é provavelmente incorreto), e sim tentar refazer a leitura.

Erros e exceções

Uma exceção não deve ser encarada como um erro, e sim como um evento que exige um comportamento "fora do comum" do programa. Um erro corresponde a uma implementação incorreta de um algoritmo, enquanto que uma exceção corre-

sponde a um funcionamento incorreto de um algoritmo codificado corretamente ou a uma falha no *hardware* (que, a princípio, está correto).

Note, porém, que erros podem gerar exceções e que exceções não tratadas devem ser encaradas como erros. Exceções não esperadas são forte indício de erro(s) na codificação do algoritmo ou até do próprio compilador, sistema ou equipamento utilizado.

Tolerância a falhas
Os termos tolerância a falhas e tratamento de exceções são muitas vezes utilizados como sinônimos, mas na realidade representam conceitos bem distintos. O primeiro é muito mais "forte", seguro e caro, pois implica em replicação/redundância de equipamentos e programas, e se propõe a manter o sistema em funcionamento (correto) mesmo que haja pane permanente de alguns de seus componentes. Já o segundo supõe que o sistema como um todo está funcionando corretamente, e tenta se recuperar de falhas esporádicas que eventualmente possam ocorrer.

O termo "recuperação de erros" é bastante utilizado significando tratamento de exceções, mas pode levar a confusões como a citada.

7.2 Tratamento de exceções sem MTE

Um programador não pode se ver livre de lidar com exceções pois, por melhor que seja, não conseguirá que falhas de equipamento ocorram ou que seus programas sejam utilizados de forma incorreta.

Porém nen todas linguagens proveêm mecanismos explícitos de tratamento de exceções. A linguagem C (**[C-K&R78]**, **[C-ANSI88]**), por exemplo, é uma das mais difundidas no mundo e até hoje não definiu um mecanismo padrão de tratamento de exceções. Na verdade nem definiu uma maneira homogênea de comportamento perante uma situação excepcional. Algumas das alternativas utilizadas para contornar essas situações em uma chamada de função, por exemplo, são:

- ignorar a possibilidade da ocorrência de uma exceção e supor que ela não ocorrerá ou não terá nenhum efeito danoso ou indesejável na execução do programa, caso ocorra;
- terminar a execução do programa;
- retornar um valor que indique o erro ocorrido;
- indicar o erro por meio de uma variável global e retornar um valor qualquer que, apesar de não ser o correto, não causará "estragos" até que a variável possa ser consultada;
- executar uma função que tente se recuperar da situação;
- utilizar um MTE definido em cima da linguagem.

As duas primeiras opções são extremas. A primeira em particular é muito ingênua, podendo levar a resultados catastróficos. A segunda opção é mais correta (os resultados produzidos são, a princípio, corretos), mas nem sempre é desejável. Por exemplo, em sistema interativos o usuário poderia "indicar o caminho", evitando que o trabalho realizado até o momento seja desperdiçado.

As três opções seguintes são mais versáteis, e freqüentemente utilizadas. Elas impõem um compromisso com o programador (testar um resultado/variável ou estabelecer rotinas de tratamento) que nem sempre é cumprido (por descaso ou descuido). O compilador não é capaz de evitar ou detectar uma "quebra" do compromisso, ou seja, podemos recair no primeiro caso inadvertidamente. Por outro lado, quando o programador cumpre fielmente suas obrigações o código se tranforma em uma colcha de retalhos, intercalando o código de verificação de erros e o código original (do algoritmo) praticamente a nível de comandos ou expressões.

A última opção pode ser a melhor escolha. Tudo depende dos recursos que a linguagem oferece para a implementação de um MTE razoável (em termos de confiabilidade e facilidade de utilização). A linguagem C, por exemplo, permite a implementação de um MTE bastante razoável, em cima de macros e bibliotecas padrão. Esses mecanismos podem ser vistos em **[Exc83]** e **[MTE92]**.

7.3 Motivação

Um bom programador pode escrever programas completos e com verificação/tratamento de erros sem se utilizar de nenhum mecanismo de tratamento de exceções. Porém, um mecanismo desses reduz a sobrecarga sobre o programador, trazendo vantagens muito importantes, tais como:

- estabelece uma maneira uniforme de tratamento de situações excepcionais;
- provê maior separação entre o código de tratamento de erros e o código do algoritmo em si, ou seja, provê maior clareza e legibilidade do código fonte;
- desobriga o programador de ficar verificando códigos de retorno ou variáveis globais indicando situações de erro.

Note que um MTE pode ser flexível a ponto de permitir que o programador ignore a possibilidade da ocorrência de exceções em alguns casos, enquanto garante que, caso ocorram, não passarão desapercebidas, podendo ser tratadas em níveis mais "altos" do programa. No pior dos casos, que ocorre quando uma exceção não pode ser tratada de forma alguma, a execução do programa é encerrada (em geral, produzindo uma mensagem de erro indicando a natureza e ponto de ocorrência da exceção).

7.4 Mecanismos de tratamento de exceções

Um mecanismo de tratamento de exceções deve considerar os seguintes aspectos:

1. como representar e identificar exceções;
2. como sinalizar (produzir) exceções;
3. como estabelecer tratadores de exceções;
4. como uma exceção é propagada e associada a um tratador;
5. como prosseguir após o tratamento da exceção.

7.4.1 Representação de exceções

Uma exceção é basicamente uma indicação da ocorrência de um evento, não sendo necessário mais que um identificador ou um valor simples (um inteiro ou uma *string*) para indentificá-la.

Ada, Mesa, CLU e PL/I representam exceções dessa forma. Em Ada ([**ADA79**], [**ADA**]), exceções são representadas por identificadores declarados com tipo "*exception*". Em CLU ([**CLU77**], [**CLU79**], [**Exc84**]) e Mesa ([**Exc84**]), exceções são identificadores declarados em *headers* de procedimentos. Em PL/I ([**PL/I77**], [**Exc84**]) são identificadores pré-definidos.

Porém, somente o "nome" ou "categoria" de uma exceção não permite muita versatilidade no tratamento da mesma. Uma única exceção deve, neste caso, ser dividida em duas ou mais para permitir um tratamento diferenciado. Como cada uma das novas exceções é tratada de maneira independente, o aproveitamento das características comuns dessas exceções fica prejudicado. Ainda, a explosão do número de exceções torna o código fonte muito mais complexo e confuso, dificultando a manutenção e facilitando a ocorrência de erros. Para se evitar esta explosão de exceções deve haver um mecanismo para passagem de parâmetros para os tratadores.

Em PL/I, a primeira linguagem a incorporar um MTE, e em ADA, é possível se passar valores para os tratadores através de variáveis globais, o que apenas ameniza os problemas discutidos acima.

Em CLU as exceções podem ser parametrizadas, tornando-se fácil passar valores do sinalizador da exceção para o respectivo tratador. Pequenas variações no tratamento de uma exceção podem ser indicadas por esses parâmetros, evitando a explosão de exceções discutida anteriormente. A parametrização de exceções adiciona bastante poder ao mecanismo, sendo adotada em diversos deles (como, por exemplo, nos de Goodenough [**Exc75**] e Yemini [**Exc85**]).

Em se tratando de uma linguagem orientada a objetos, como é o caso de Cm, muitos autores propõem que exceções sejam representadas por objetos (de tipo classe) e organizados hierarquicamente (via relações de herança). É o caso de Lore ([**Lore88**]), Smaltalk ([**Stalk90**]), C++ ([**C++90**], [**Exc90**]) e do mecanismo de Borgida ([**Borg85**]). Essa representação torna o mecanismo conceitualmente mais simples (não é necessária parametrização, pois valores adicionais podem ser introduzidos no objeto que representa a exceção, caso necessário) e poderoso (principalmente se as relações de herança forem "entendidas" pelo MTE).

7.4.2 Escopo e associação de tratadores

Cada tratador é associado a um determinado "tipo" de exceção. Seu escopo indica a "região" do código de um programa na qual a ocorrência de uma exceção desse tipo provoca sua ativação. Escopos podem ser "encaixados", permitindo que em um determinado momento da execução de um programa existam vários tratadores ativos, responsáveis pelo tratamento de exceções distintas. O mecanismo pode permitir mais de um tratador ativo para uma determinada exceção, estabelecendo regras que determinam qual desses tratadores deve ser utilizado em cada caso.

Em PL/I, o conceito de escopo não é bem definido. Os tratadores são associados diretamentea exceções, de forma dinâmica, através de comandos ON, como em

```
ON <exceção> <tratador>
```

Somente é permitido um tratador para cada exceção, que pode, no entanto, ser substituído dinamicamente através de novos comandos ON.

Nos demais mecanismos citados anteriormente o escopo de um tratador é determinado estaticamente, porém sua ativação é "estendida" dinamicamente por chamadas a funções/procedimentos. Tratadores podem ser associados a expressões, comandos, procedimentos/funções, blocos ou classes (no caso de linguagens orientadas a objetos). Uma vez ativado, o tratador permanece ativo até que o fluxo de controle de execução "saia" de seu escopo estático. Assim, o tratador pode ser ativado devido a exceções ocorridas em pontos não pertencentes ao seu escopo estático, mas que foi atingido a partir dele. Dessa forma a determinação do tratador a ser executado é mais complexa, pois depende da sequência de ativação dos procedimentos/funções do programa.

7.4.3 Sinalização de exceções

Exceções podem ser geradas por *hardware* ou *software*. A princípio, não há diferença conceitual entre elas, porém alguns MTEs tratam essas exceções de maneiras distintas. Alguns MTEs, inclusive, nem permitem a definição de exceções por parte do usuário. Exceções ocorridas em bibliotecas ou detectadas pelo usuário são essencialmente exceções de *software*, assim, um MTE poderoso deve definir mecanismos que permitam o programador (comum ou de bibliotecas) sinalizar uma exceção. Isso é feito, em geral, via um comando ou uma função de biblioteca.

O comando ou função que sinaliza exceções é normalmente batizado de raise (Cm, Ada), signal (PL/I, Mesa, CLU), error (Mesa), cause ou throw (C++).

Caso um tratador conclua que não é capaz de resolver o problema corrente (indicado pela exceção), ele pode sinalizar uma nova exceção. Alguns MTEs permitem que o tratados sinalize a mesma exceção que recebeu através de comandos como reraise, raise (Cm, Ada), resignal (CLU, Mesa), throw (C++).

7.4.4 Propagação de exceções

Quando uma exceção é detectada, o MTE ganha controle sobre o fluxo do programa, que é desviado para o tratador associado ao ponto em que ela ocorreu. Caso o tratador trate a referida exceção, a execução do programa prossegue normalmente (de acordo com o MTE utilizado). Porém, caso o tratador não saiba tratar a exceção, ou caso ocorra outra exceção dentro do tratador, o MTE ganha novamente o controle sobre o fluxo do programa e pode sinalizar uma exceção (possivelmente a mesma) para os níveis mais externos de tratadores ou encerrar a execução do programa.

Os mecanismos de CLU, Goodenough e Yemini apresentam mecanismos com propagação em um único nível, ou seja, a exceção sinalizada deve ser tratada por tratadores estabelecidos no nível imediatamente anterior da cadeia dinâmica de

chamadas de funções, ou seja, uma determinada função deve ser capaz de tratar todas as possíveis exceções produzidas em seu corpo e todas as exceções que sinalizar deverão ser tratadas pela função que a chamou. Estes mecanismos são bem restritivos, o que os torna formalmente mais "belos", mas na prática não é incomum a proliferação de tratadores que recebem uma exceção e a ressinalizam para o nível seguinte, implementando a propagação por vários níveis de maneira explícita.

Os mecanismos de Ada, C++ e Cm permitem a propagação automática em vários níveis, ou seja, se um tratador não reconhece uma exceção, ela é repassada automaticamente para os tratadores dos níveis mais externos (estabelecidos anteriormente).

7.4.5 Modelos de finalização de um tratador

Existem basicamente três modos para se terminar a execução de um tratador: terminação (*termination*), retomada (*resumption*) e substituição (*replacement*).

Modelo de terminação

O modelo de terminação determina que os escopos (níveis de ativação de funções) que são percorridos na busca do tratador são "terminados", e a execução prossegue no comando subseqüente ao tratador que foi executado. É o caso de Cm e C++.

O modelo pode permitir repetições (*retry*), indicando que a execução do bloco protegido (escopo estático do tratador) pode ser repetida um determinado número de vezes até que sua execução seja feita com sucesso.

Modelo de retomada

O modelo de retomada determina que a execução é retomada no ponto em que foi interrompida, como se nada tivesse acontecido (exceto os efeitos causados pela execução do tratador). O MTE deve determinar em que ponto ocorre a retomada da execução. Em geral, a execução é retomada no início ou logo após o menor comando (ou expressão) que engloba o ponto de ocorrência da exceção. Uma alternativa é a retomada no início do escopo estático do tratador associado (*retry*).

Modelo de substituição

Este modelo foi proposto por Yemini, e requer que o tratador seja associado a expressões. É um esquema que mescla terminação e retomada. O tratador é encerrado retornando um valor a ser usado em substituição ou do valor da expressão associada ao tratador (terminação) ou do valor da menor expressão que envolve o ponto em que ocorreu a exceção (retomada).

Capítulo 8

Survey: Produção de Software

Este capítulo não chega a ser um survey. Ele é melhor caracterizado como uma apresentação informal do assunto, baseado apenas na experiência do autor e nos trabalhos de Stroustrup e Booch ([C++91] e [OOD91]).

Inicialmente são descritos brevemente os paradigmas de programação que podem ser utilizados com a linguagem Cm. O melhor, pelo menos para grandes sistemas e a médio/longo prazo, é o de orientação a objetos. Pode-se imaginar os paradigmas apresentados numa escala de evolução partindo do paradigma procedural e chegando no paradigma de orientação a objetos.

Em seguida é apresentado um modelo de desenvolvimento de programas mais adequado para o desenvolvimento em Cm que o tradicional modelo em cascata.

8.1 Paradigmas de Programação

Um paradigma de programação basicamente descreve uma maneira ou estilo de se produzir programas, englobando:

- metodologias de especificação de programas;
- técnicas de programação;
- suporte em tempo de compilação;
- suporte em tempo de execução.

Uma dada linguagem pode prover suporte para mais de um paradigma, porém em geral elas focalizam um determinado paradigma e proveêm um suporte melhor a este paradigma. Note que permitir o uso de um paradigma não significa prover suporte a ele. Por exemplo, pode-se utilizar o paradigma de orientação a objetos ainda que programando em C, porém é necessário muita disciplina e organização por parte do programador.

Mesmo que a linguagem ofereça suporte a um paradigma, o programador deve se disciplinar para usar os recursos disponíveis da melhor maneira, buscando o "estilo de boa programação" no paradigma.

8.1.1 Suporte a paradigmas

Uma linguagem somente suporta um paradigma de programação caso apresente facilidades que a tornem "conveniente" para o estilo de programação que o paradigma propõe, ou seja, se provê mecanismos que tornem fácil, segura e eficiente a programação e execução de programas seguindo o paradigma.

Estes mecanismos podem aparecer de várias formas, como facilidades a nível de linguagem e verificações em tempo de compilação ou execução do programa. Preferencialmente, estes mecanismos devem:

- verificar possíveis "violações" ao paradigma o mais cedo possível;
- ser simples e genéricos, integrados de maneira clara e "elegante" na linguagem;
- ser independentes e ortogonais entre si, permitindo ao programador utilizar (ou mesmo conhecer) somente o subconjunto dos mecanismos que lhe interessa;
- não produzir *overhead* significativo (tanto na programação como na execução do programa), principalmente se o mecanismo em particular não for utilizado no programa em questão.

8.1.2 Programação procedural

O paradigma de programação procedural é centrado em procedimentos ou funções. Ele focaliza o processamento e algoritimos necessários para realizar a tarefa desejada.

A "boa programação" consiste na definição de funções que realizem "computações" bem definidas e delimitadas, e sejam independentes das demais funções (ou seja, dependam somente de seus parâmetros).

técnica

1. dividir o problema em procedimentos ou funções;
2. implementar cada função usando o algoritmo mais adequado que for encontrado.

suporte

O suporte a este paradigma basicamente consiste em mecanismos para se definir procedimentos e funções e para passagem e retorno de valores para eles.

linguagens

Fortran é a primeira linguagem a suportar este paradigma, que foi seguido posteriormente por outras linguagens, como C e Pascal.

8.1.2.1 Programação estruturada

Programação estruturada é um avanço sobre programação procedural, definindo estruturas de controle mais sofisticadas, porém não adiciona aletrações significati-

vas ao paradigma: se enquadra muito mais como uma técnica sofisticada de se programar no mesmo paradigma do que como um paradigma novo.

8.1.3 Programação modular

O paradigma de programação modular é uma evolução do de programação procedural. Ele focaliza a organização dos dados em um programa, que passa a ser dividido em módulos. Cada módulo contém um conjunto de dados e as funções que os manipulam. Módulos podem "encapsular" seus dados estabelecendo uma maneira padronizada de manipulá-los (sua *interface*).

Cada módulo isoladamente é implementado segundo o paradigma procedural. Uma vez que seus dados são encapsulados, o módulo deve prover funções para inicializá-los.

técnica

1. dividir o problema em módulos;
2. definir a interface de cada módulo;
3. implementar cada módulo de maneira independente dos demais.

suporte

O suporte ao paradigma consiste basicamente em

- impedir que a interface de um módulo seja violada, ou seja, impedir que dados encapsulados sejam manipulados de maneira incorreta;
- garantir que as rotinas de inicialização do módulo sejam executadas antes do módulo ser utilizado;
- suportar programação procedural internamente a cada módulo.

linguagens

C suporta uma versão mais simplista de módulos (unidades independentes de compilação), que deixa grande parte da responsabilidade de uso correto do paradigma na mão do programador. C++ suporta o conceito de módulos com suas classes, porém não provê mecanismos de controle de inicialização das classes em si (somente para seus objetos). Ada, Modula-2 e Cm suportam diretamente este paradigma. Todas essas linguagens, evidentemente, suportam programação procedural como decorrência.

8.1.4 Abstração de dados

A abstração de dados, ou programação de tipos abstratos de dados (TAD), focaliza os tipos utilizados em um programa. Um tipo abstrato de dados é basicamente um módulo que gerencia todos os tipos de dados de um determinado tipo (definido pelo usuário), ou seja, é basicamente uma extensão de módulos (a diferença principal reside no fato que um módulo simples representa uma única "entidade" enquanto que um módulo de TAD representa um conjunto de "entidades" de um mesmo tipo).

técnica

1. determinar os tipos de "entidades" manipuladas pelo programa;

2. definir as informações e o conjunto completo de operações relacionadas a cada tipo;
3. implementar cada um desses tipos como um TAD (módulo gerenciador de tipos);
4. implementar o programa usando os TADs (esta implementação deve ser simples, caso sejam escolhidos os tipos adequados).

suporte
O suporte a programação de TAD consiste basicamente no suporte a módulos que implementam tipos. Quanto melhor este suporte, mais homogêneo é o tratamento dado a tipos pré-definidos (*built-in*) e tipos definidos pelo usuário. Este suporte engloba a definição de operações sobre os tipos (via funções comuns ou sobrecarga de operadores) e formas de declaração e inicialização dos objetos do tipo dado (construtores e destrutores).

linguagens
Ada, CLU, Cm e C++ permitem a definição de tipos abstratos de dados com tratamento bastante próximo ao dado a tipos pré-definidos. Cm e C++ suportam TADs em decorrência do suporte a orientação a objetos, pois TADs são um caso particular de objetos, como definidos no paradigma de orientação a objetos.

8.1.5 Orientação a objetos

O paradigma de orientação a objetos focaliza a atenção na organização dos dados e operações em "entidades" (da mesma forma que programação modular o faz em módulos ou abstração de dados o faz em tipos), enfatizando o aproveitamento das características em comum que essas "entidades" (ou qualquer subconjunto delas) possam ter. Essas "entidades" são chamadas de classes, e representam módulos ou TADs.

O paradigma de orientação a objetos visa reduzir o esforço de programação, permitindo a definição de classes novas em função de classes já existentes; a fatoração de partes comuns de um conjunto de classes; e a definição de módulos genéricos que podem ser "configurados" para atuar de maneiras variadas e/ou sobre tipos variados.

técnica
1. dividir o programa em classes;
2. definir as informações e o conjunto completo de operações relacionadas a cada classe;
3. agrupar classes semelhantes em uma única classe polimórfica ou parametrizada, quando possível;
4. fatorar semelhanças significativas entre classes em uma única classe (base), redefinindo as demais (derivadas) a partir dessa, quando possível;
5. implementar cada uma das classes;
6. implementar o programa usando as classes (esta implementação deve ser simples, caso sejam escolhidas as classes adequadas).

suporte
Uma linguagem somente suporta orientação a objetos se permite que as propriedades em comum de um conjunto de classes sejam aproveitadas de maneira simples. Os mecanismos mais comuns para isso são polimorfismo e heranca. Note que compatibilidade entre classes base e derivadas também é muito importante (é necessário pelo menos que objetos de classes derivadas possam ser usadas de maneira mais genérica, como se fossem objetos de classe derivada.

O suporte a orientação a objetos é bastante variado, dando margem a subdivisões como linguagens baseadas em objetos e liguagem orientada a objetos. Essa distinção é devida ao conceito de herança, oferecido por linguagens orientadas a objetos (Smalltalk e C++, por exemplo) e ausente nas linguagens baseadas em objetos (Ada, por exemplo).

linguagens
Atualmente a programação orientada a objetos está "na moda", e existem várias linguagens que suportam o paradigma. As mais difundidas são Smalltalk e C++. Cm também suporta este paradigma.

8.2 Projeto de programas orientados a objetos

Um projeto é dito orientado a objetos se utiliza uma metodologia que utilize técnicas capazes de aproveitar de maneira fácil e produtiva as facilidades de linguagens orientadas a objetos (tais como abstração de dados, herança e polimorfismo) na produção de programas. Por produção de programas entende-se toda e qualquer atividade realizada para se chegar a um produto (programa) acabado.

Não é possível, para o caso geral, estabelecer uma "receita" que ensine a produzir bons programas. A experiência e bom senso dos projetistas/programadores é fundamental. Por melhor que seja, qualquer técnica só é bem utilizada na medida em que estas pessoas se tornem experientes e familiarizadas com ela. Porém, uma boa técnica facilita bastante na medida que estabelece um padrão (mesmo que em linhas gerais) a ser seguido por todas as pessoas envolvidas. Este "padrão" envolve formas de documentação, terminologia, notação, etc, que facilitam a "comunicação" entre os envolvidos.

8.2.1 metodologias de desenvolvimento

A metodologia de desenvolvimento dá as linhas gerais de atuação em cada "etapa" da produção de programas. Essas etapas são basicamente análise, projeto e implementação. A análise define o escopo do problema (o que deve ser feito), o projeto determina a estrutura do sistema (como fazer), e a implementação corresponde à programação em si. Atividades como experimentação, documentação, testes e validações são realizadas em todas essas etapas.

As etapas do desenvolvimento muitas vezes são feitas por pessoas diferentes. A metodologia deve prover formalismos, bem documentados, para facilitar a comunicação entre essas pessoas, ajudando a formar uma cultura compartilhada por todos.

divisão e conquista
O maior problema na produção de programas é a complexidade envolvida, que é exponencial: caso o tamanho do programa seja dobrado, a complexidade total é muito maior que o dobro da complexidade inicial. Por outro lado, caso o tamanho do programa seja reduzido à metade, a complexidade cai para muito menos que a metade. Isso sugere a utilização de técnicas de divisão e conquista, procurando "quebrar" o problema em partes relativamente independentes que possam ser "trabalhadas" mais facilmente. Este é o ponto em que a experiência e bom senso dos projetistas é mais necessária.

desenvolvimento de programas
O projeto de um produto em particular pode ter começo e final, porém o projeto de produtos extrapola as fronteiras de cada produto. Idealmente esse processo não tem nem início nem fim: cada produto parte de "algo" já existente, como bibliotecas, produtos parecidos e experiência adquirida pela equipe, e faz esse "algo" crescer, como uma bola de neve rolando pela encosta de uma montanha, que cresce a cada volta.

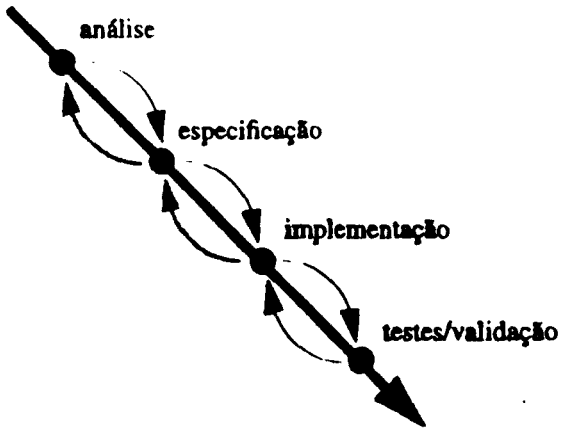
Dessa forma, o desenvolvimento de um programa deve considerar não só suas próprias fronteiras, devendo sempre ter em conta que os resultados obtidos (bibliotecas, programas, técnicas, experiência, etc) serão utilizados nos projetos futuros.

Assim, o desenvolvimento de programas é um processo iterativo e incremental, que está constantemente refinando e aumentando a "bola de neve" na qual se baseia. No início, esta "bola" é pequena e gira lentamente. Os primeiros produtos desenvolvidos dessa forma provavelmente demorarão mais que se desenvolvidos de outras maneiras, mas, se bem gerenciada, a "bola" ganha velocidade com o tempo e a médio/longo prazo a reutilização chega a um nível que compensa múltiplas vezes as perdas iniciais.

componentes
A reutilização se dá basicamente em cima de classes. Às vezes uma classe é bastante relacionada a outras, principalmente quando estão na mesma hierarquia de herança. Esse conjunto de classes inter-relacionadas constituem, em geral, uma "unidade" a nível de documentação e utilização. Essas "unidades" são denominadas componentes.

8.2.1.1 modelo em cascata
As metodologias mais tradicionais são baseadas no modelo em cascata de desenvolvimento de programas. Este modelo "serializa" as etapas da produção de programas partindo da análise e chegando à validação final do produto (figura 8-1). Esse direcionamento dificulta o *feedback* inerente à produção de programas: uma vez passada uma etapa, há forte resistência em se modificar as etapas anteriores. Esse retorno é encarado como "atraso" ou "incompetência", mas é conveniente muitas vezes, quer seja para melhorar a etapa anterior (com base na experiência adquirida) quer seja para corrigir erros detectados. O que ocorre muitas vezes é uma adaptação de etapas seguintes para encobrir ou reduzir problemas de etapas anteriores, resultando em um produto com implementação "tortuosa", ineficiente e/ou de difícil manutenção (extensão/porte).

FIGURA 8-1 Modelo cascata para produção de programas



8.2.1.2 modelo cíclico

O modelo em cascata não é conveniente para metodologias orientadas a objetos pois não representa propriamente a "circularidade" (bola de neve) que ocorre nesses casos.

O desenvolvimento de um produto, de um ponto de vista mais geral, tem fases seqüenciais, como no modelo cascata. De início é feita uma análise do problema para verificar as características necessárias ou desejáveis para o sistema, passa-se então para o projeto (especificação) do produto, e, na fase final, implementação e teste do produto.

A circularidade ocorre como parte do projeto do sistema, se desdobrando em várias seqüências de especificação, adaptação/implementação e testes de componentes do sistema. É nessa fase que ocorre o reaproveitamento de componentes já existentes e a geração de novas componentes. A figura 8-1 esquematiza este processo.

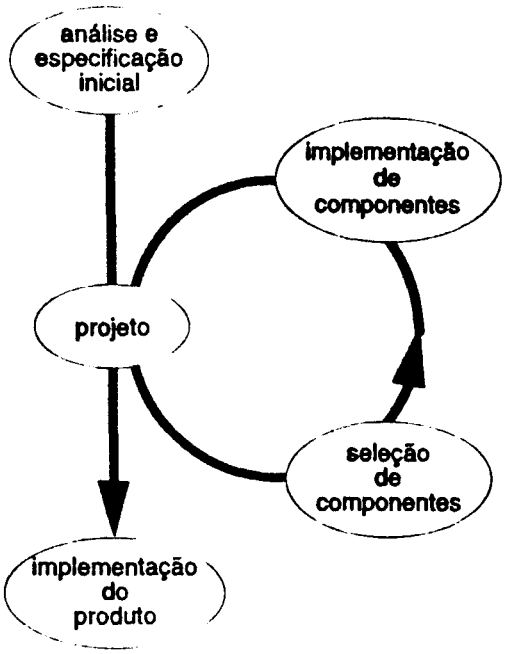
seleção de componentes

O reaproveitamento de componentes pode se dar em vários níveis, desde a utilização direta de um componente já disponível até a especificação e implementação de um componente totalmente novo. Entre esses dois extremos se encontram a configuração ou refinamento (adaptação, reestruturação) de componentes já existentes.

Assim, cada componente deve ser analisada para verificar quais podem, e de que forma, ser reutilizadas. A reutilização pode envolver

- customização, que consiste na alteração de parâmetros ou configurações de componentes já disponíveis;
- refinamento: que consiste na alteração de um componente para torná-lo mais genérico e ser utilizável no produto sendo desenvolvido (sem, contudo, deixar de ser utilizável nos produtos anteriores);
- reestruturação: que corresponde a reorganizações maiores em um ou mais componentes, de forma que se tornem mais homogêneos ou adequados para serem utilizados nos projetos correntes e futuros.

FIGURA 8-2 Modelo cíclico para produção de programas



Após a busca de componentes já existentes deve-se projetar e implementar novos componentes, que podem ser derivados de componentes disponíveis ou totalmente novos. A escolha e implementação desses novos componentes devem ser a mais genérica possível, sempre considerando que o componente é para ser usado para este e muitos outros projetos. Note que a ênfase na reutilização complica o processo de produção do componente, e somente é vantajosa para componentes que sejam realmente reutilizáveis.

Dessa forma, o conjunto de componentes disponíveis vai sendo melhorado e aumentado a cada produto. Note que essas melhorias podem tornar a nova "versão" de um componente inadequada para os usos dados à versão anterior. Dessa forma é preciso ter um bom mecanismo de controle de versões.

Capítulo 9

Extensões futuras (compilação e geração de código)

A linguagem Cm está em constante evolução desde 1986. O estágio atual da linguagem já é bastante satisfatório, sendo poucas as extensões previstas a nível de linguagem (apenas as descritas no capítulo seguinte, e as referentes a programação distribuída[1]).

Os principais trabalhos previstos são a nível de implementação e ferramentas de suporte à programação em Cm, descritos nas seções seguintes.

9.1 Front-end e back-end

A versão atual do compilador é em um único passo, que é mais eficiente em termos de tempo de compilação e demanda de memória (a estrutura de dados gerada somente contém informações relativas aos "símbolos" utilizados e valores de algumas expressões). O código C vai sendo na medida que o fonte Cm vai sendo lido.

Porém uma implementação separando o *parsing* (*front-end*) da geração de código (*back-end*) – o primeiro criando uma estrutura de programa completa, e o último gerando o código a partir dela – facilita a implementação de ferramentas de suporte ao desenvolvimento. Essas ferramentas podem ser implementadas apenas como um *back-end* para o compilador (segundo passo), se utilizando de todo o mecanismo de *parsing* e análise semântica realizado pelo primeiro passo. Como exemplos de ferramentas implementáveis dessa forma temos formatadores de programa (*pretty-printers*), tradutores para outras linguagens (*back-ends* para geração de código nativo em outras linguagens) e analisadores de código (geradores de referências cruzadas, otimizadores a nível de linguagem, etc).

1. As extensões referentes a programação distribuída estão sendo definidas e implementadas por Celso Gonçalves Júnior, em paralelo com as descritas nessa tese.

Note que essa separação não impõe uma compilação em dois passos no sentido do *parsing* ter duas fases.

9.1.1 Geração de código RTL (Gnu)

O principal objetivo da implementação em dois passos é permitir a implementação de um gerador de código RTL para utilização do *back-end* da Gnu Software, aumentando a "portabilidade" do Cm, que poderia ser utilizado nas mesmas plataformas em que o Gnu C[1], porém as principais vantagens são a possibilidade de geração de código mais eficiente, maior facilidade de depuração de código (uma vez que se tem maior controle sobre os arquivos objetos gerados) e maior velocidade de compilação (pelo menos a geração e *parsing* de arquivos C não serão mais necessárias).

9.2 Porte para C++

Prevê-se um porte do compilador Cm para C++, que oferece melhores recursos que C. A implementação em Cm é interessante, mas contraproducente, pois gera problemas de porte para outras plataformas (principalmente no caso de Cm passar a gerar código de máquina, e não fontes C).

9.3 Depuração de programas Cm

Atualmente a depuração de programas Cm é bastante trabalhosa, principalmente devido à geração de código C (mapeamentos de identificadores, alteração da estrutura do programa, etc). Depuradores tradicionais operam a partir do código C gerado, e não do código Cm original.

Para suprir essa deficiência, ou pelo menos reduzir o problema, foram levantadas algumas opções, mas sem nenhuma conclusão definitiva até o momento. As principais linhas consideradas são descritas a seguir.

adaptação de um depurador
A solução que a curto prazo parece mais viável é a adaptação de um depurador existente, tornando-o capaz de entender as tabelas de símbolos e hierarquia de arquivos geradas pelo compilador Cm. Dessa forma o depurador poderia mapear os comandos dados pelo usuário com relação ao programa Cm para comandos equivalentes com relação aos arquivos C gerados. Possivelmente o compilador Cm precisará gerar diretivas "`#line`" ou tabelas de mapeamento da numeração de linhas entre os fontes Cm e C para correta visualização dos arquivos.

inserção de comandos para acompanhamento da execução
O compilador Cm pode ser adaptado de modo a inserir comandos para acompanhamento da execução do programa gerado (*tracing*). Essa inserção seria controlada por argumentos da linha de comando, permitindo *tracing* seletivo de funções ou

1. Uma vez que Cm té escrito em C e gera código fonte C, isso não chega a ser realmente uma vantagem.

classes. Este mecanismo é similar ao implementado pelo programa `ctrace` disponível em Unix para a linguagem C. Entretanto, a informação semântica detida pelo compilador Cm possibilita *tracing* ao mesmo tempo mais abrangente e econômico.

9.4 Tradução de arquivos C

O uso de bibliotecas C é dificultado pela incompatibilidade sintática de Cm e C, porém podendo ser facilitado por um tradutor de arquivos de *header* C. A idéia básica é gerar uma classe Cm que sirva de interface para a biblioteca C desejada. Uma versão preliminar do tradutor já foi definida e implementada.**[Cm92]**

9.5 Edição de programas fonte Cm

Editores orientados à sintaxe estão entre as ferramentas de suporte à programação previstas para o ambiente A_HAND. Um modo de edição Cm para o GNU-Emacs já foi implementado **[EdCm89]**. Os trabalhos futuros englobam porte desse modo de edição para a nova sintaxe da linguagem, implementação de de um modo de operação específico para Cm no HT (um sistema de hipertextos desenvolvido no A_HAND), e a implementação de um editor orientado à sintaxe integrado ao ambiente (facilitando o processo de edição, compilação e depuração de programas).

9.6 Extensão de comentários

O artigo **[CAP89]** disserta sobre a importância de comentários em programas, propondo que estes devem fazer parte da gramática da linguagem, tendo a semântica tradicional de "espaço branco" e semânticas especiais de *pragmas* (diretivas de compilação) e *asserções* (pré e pós-condições, invariantes de malha e condições genéricas). O uso de comentários como parte integrante da gramática permite um tratamento mais elaborado dos programas por ferramentas de suporte como *pretty-printers*, editores, verificadores, etc. Estas ferramentas podem, inclusive, compartilhar o mesmo *parser*.

Pragmas são diretivas para o compilador, e são comentários no sentido de não afetarem a semântica do programa. As classes de armazenamento **register** e **inline** são exemplos de *pragmas* incorporados à linguagem. Uma possível extensão é a inclusão de recursos para controle de compilação condicional.

Asserções são testes que, se realizados, devem produzir um resultado "verdadeiro". Estes testes, porém, não precisam ser realizados, e sua inclusão ou não no código final não deve alterar a semântica do programa. Desse ponto de vista, asserções também podem ser consideradas comentários. Uma possível extensão é a inclusão de uma pseudo-função `assert`, que aceita um argumento numérico qualquer e não tem retorno ("retorna" **void**). Essa função verifica se seu argumento, e caso seu valor seja "verdadeiro" a execução é interrompida ou uma exceção é gerada. O compilador pode controlar a inserção ou não desses testes através de opções da linha de comando, e deve impedir o uso de expressões com efeitos colaterais,[1] evitando situações indesejáveis como as que ocorrem em C, onde `asserts` são

removidos pelo pré-processador e podem fazer um programa "correto" deixar de funcionar.

Outro uso interessante para comentários é a especificação de palavras-chaves e descrições sucintas (de uma linha, por exemplo), para uso por ferramentas de busca e classificação de classes (*browsers*).

9.7 Representação de funções virtuais

A implementação atual de funções virtuais pode ser melhorada em termos da memória utilizada, ao custo de maior tempo de execução.

No escopo do objeto de uma classe é alocado um apontador para cada função virtual dessa classe. Porém, todos os objetos de uma mesma classe derivada têm os mesmos valores para esses apontadores. Assim pode-se economizar memória armazenando esses apontadores em uma tabela e alocando apenas um apontador para essa tabela em cada objeto (figura 9-1). Note que, assim, o acesso à funções virtuais exige uma indireção a mais. A melhor opção depende de vários fatores, como o número de objetos de cada classe criados, o número de funções virtuais, o número de chamadas a funções virtuais, o tempo de resposta esperado e a memória disponível. Idealmente deve ser possível o uso de qualquer dos dois mecanismos (selecionado via opções da linha de comando).

1. Evidentemente essa verificação seria bastante conservadora, para ser factível.

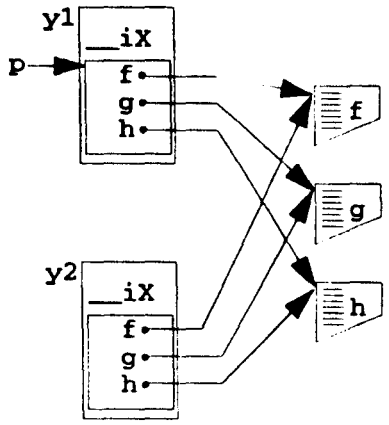
FIGURA 9-1 Possíveis representações de funções virtuais

```
class X<>
// dataX
virtual void f();
virtual void g();
virtual void h();
```

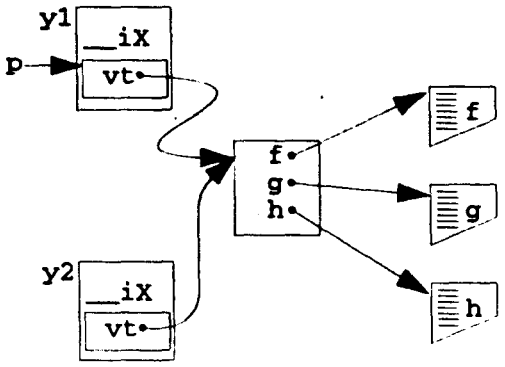
```
class Y<>
inherit X;
// dataY
...
```

```
Y y1, y2;
X *p = &y1;

p -> f ();
```

Esquema atual:



Esqema proposto:



Acesso:
`(*p).f (...)`

Acesso:
`((*p)->vt).f (...)`

Capítulo 10

Extensões Futuras à Linguagem Cm

10.1 Inicialização de membros tipo classe/referência

A inicialização de campos do escopo de objeto somente pode ser feita nas respectivas declarações ou por comandos em construtores. A inicialização de objetos tipo classe e tipo referência fica pouco flexível, pois seu valor inicial não pode ser atribuído por comandos, uma vez que a utilização de objetos tipo referência em comandos sempre se refere ao objeto referenciado e chamadas explícitas a construtores não são permitidas. Assim, a inicialização de objetos tipo classe e tipo referência deve ser feita nas declarações, não podendo ser feita de forma diferente de acordo com o construtor utilizado.

Para permitir que essas inicializações sejam feitas em construtores é preciso estender a linguagem. Uma forma bastante simples e concisa é permitir o uso do operador '@' de construção de valor para determinar o valor inicial de objetos. A seguir é dado um exemplo utilizando essa proposta:

```
class x<>
import Date;
Date dt;
int &ir;
int i;

constructor (int d,m,y; int &ref)
{
   dt @ (d,m,y);
   ir @ ref;
   i @ d+m+y;
   // declarações do construtor
   // comandos do construtor
}
```

Note que essas inicializações aparecem imediatamente no início do corpo do construtor. Isso ressalta a ordem em que ocorrem (essas inicializações devem ser feitas logo no início da execução do construtor).

10.1.1 Uso de '@' em expressões

Outra possível extensão é a permissão do uso do operador '@' para reinicializar uma referência ou classe livremente em expressões. Porém é preciso tomar cuidado com as implicações semânticas disso. É preciso estudar os benefícios e problemas que isso pode trazer para verficar se realmente é interessante permitir tal coisa. Numa análise superficial, o tipo referência parece não trazer problemas, e reinicializações de tipo classe teriam que ser tratadas de forma especial (a aplicação do operador '@' pode implicar em ativar o destrutor e um construtor, ou em criar um temporário e ativar o operador de atribuição e destruir o temporário, por exemplo), porém aparentemente esse recurso não traz poderio ou flexibilidade à linguagem.

10.2 Inicialização de classes herdadas

Classes herdadas, da maneira como estão definidas, são inicializadas pelo seu construtor *default*. É interessante, contudo, que elas possam ser inicializadas por outros construtores, inclusive de maneira dependente de parâmetros de construtores.

Isso pode ser obtido de maneira muito homogênea com a solução proposta na seção anterior, através do operador '@', que poderia ser usado na especificação de herança ou em especificações de valores inicias em construtores (operando sobre o tipo da classe herdada). Especificações de membros herdados devem preceder especificicações de membros não herdados.

A sintaxe proposta é dada através dos seguintes exemplos:

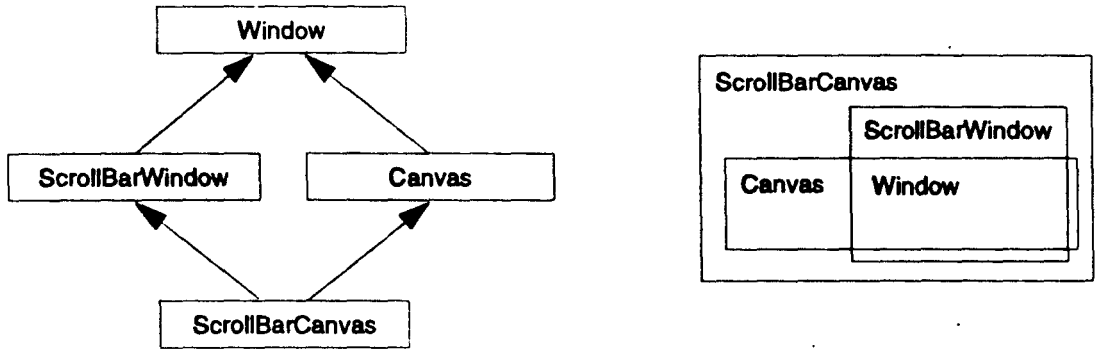
```
class x<>
inherit Date @ (31,12,1999);
constructor () { ... }

class y<>
inherit Date;
constructor (int d,m,y)
{
   Date @ (d,m,y);
   ...
}
```

10.3 Herança Por Referência

Os mecanismos atuais da linguagem Cm não permitem que uma classe herdada mais de uma vez (em diferentes ramos da hierarquia) possa ser "compartilhada" pelos dois ramos. Assim, dadas as seguintes classes

```
class Window<> ...
```

FIGURA 10-1 Hierarquia de classes compartilhando classe base




```
class Canvas<>
inherit Window; ...

class ScrollBarWindow<>
inherit Window; ...

class ScrollBarCanvas<>
inherit Canvas, ScrollBarWindow;
```

temos duas "ocorrências" da classe `Window` na hierarquia de `ScrollBarCanvas`, dadas por

```
::ScrollBarCanvas::ScrollBarWindow::Window
::ScrollBarCanvas::Canvas::Window
```

Idealmente esta hierarquia deveria corresponder à estrutura dada pela figura 10-1, mas em Cm isso só é possível utilizar a estrutura dada pela figura 10-2.

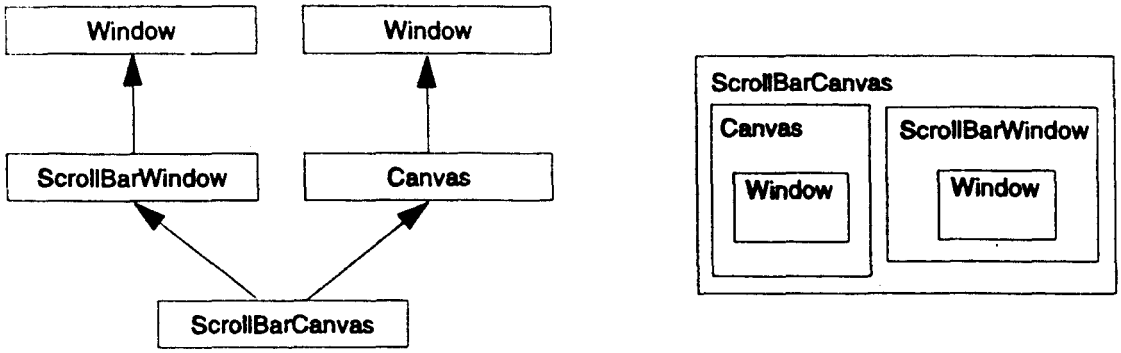
10.3.1 Implementação alternativa

O efeito desejado pode ser conseguido, mas sem utilizar herança, declarando-se uma referência ou apontador para a classe `Window` nas classes `Canvas` e `ScrollBarWindow`, que são iniciados referenciando o mesmo objeto.

Essa implementação pode ser dada (esquematicamente) por

```
class Window<> ...


class Canvas<>
import Window;
export Window *win;
```

FIGURA 10-2 Hierarquia de classes não compartilhando classe base




```
constructor (Window *w)
{
   win = w;
   ...
}


class ScrollBarWindow<>
// análoga a Canvas


class ScrollBarCanvas<>
import ScrollBarWindow, Canvas;
ScrollBarWindow sbw;
Canvas cw;

constructor (...)
{
   sbw.win = new (Window);
   cw.win = sbw.win;
   ...
}
```

Esta implementação, porém, apresenta diversas desvantagens por não utilizar a herança, tais como:

- a hierarquização das classes não fica explícita;
- as classes não são compatíveis (por exemplo, Canvas não é compatível com Window);
- o programador fica responsável por controlar explicitamente a alocação dos objetos;

- todos os métodos têm que ser redeclarados nas classes (deixam de ser herdados);
- acessos a dados têm que ser feitos por seleção explícita, através dos nomes dados aos objetos (a "localização" de cada dado não fica explícita);
- as classes que seriam derivadas (`Canvas`, por exemplo), não podem acessar dados privados de `Window`.

Essas desvantagens justificam a introdução de um mecanismo que permita o compartilhamento de classes herdadas.

10.3.2 Solução em C++

Em C++ uma classe base pode ser declarada como virtual, indicando que deve ser compartilhada na hierarquia de herança de classes derivadas. A implementação do mecanismo é feita de forma semelhante à descrita na seção anterior, usando apontadores para representar a classe base compartilhada.

Em C++ a hierarquia desejada seria dada por algo como

```
class Window { … };

class ScrollBarWindow : public virtual Window { … };

class Canvas : public virtual Window { … };

class ScrollBarCanvas
     : public Canvas, ScrollBarWindow { … };
```

Este esquema elimina as desvantagens de não se utilizar a herança, porém é bastante restritivo com relação à forma com que as classes bases são compartilhadas: todas as ocorrências de classes base virtuais são compartilhadas e na hierarquia de um mesmo objeto, e não é possível compartilhar classes bases entre objetos distintos.

O mecanismo proposto para Cm procura ser mais flexível, permitindo essas variações.

10.3.3 Proposta de solução em Cm

A classe de armazenamento **virtual** não é semanticamente muito apropriada para indicar o compartilhamento de classes herdadas. Sintaticamente ela é muito conveniente em C++, pois gramaticalmente já são aceitas classes de armazenamento para classes herdadas (**public**, **protected** e **private**). Acrescentar **virtual** a essa lista é bastante fácil. A escolha de **virtual** provavelmente foi baseada nessa conveniência e na semelhança de implementação de funções virtuais e classes base compartilhadas.

O conceito de objetos tipo referência é bem mais apropriado para indicar compartilhamento de classes, pois é semanticamente mais aproximado e indica mais precisamente a forma de implementação do mecanismo (se é que isso chega a ser vantagem...)

O mecanismo proposto para Cm indica classes que devem ser compartilhadas através de "herança por referência":

```
inherit &SharedBaseClass;
```

Assim, a implementação da classe ScrollBarCanvas seria dada, em linhas gerais, por:

```
class Window<> ...

class Canvas<>
inherit &Window;

class ScrollBarWindow<>
inherit &Window;

class ScrollBarCanvas<>
inherit Canvas, ScrollBarWindow;
```

Assim, a classe base `Window` é compartilhada na hierarquia de cada objeto da classe `ScrollBarCanvas` (mas não entre dois deles).

IV

Apêndices

Apêndice A

Exemplo de Cm e código gerado

Este apêndice mostra uma classe Cm típica, que se utiliza das classes padrão da linguagem e implementa uma calculadora simples. Os arquivos gerados também foram incluídos.

A.1 Arquivo fonte Cm (deskcalc.cm)

```
class deskcalc<>
import Input, Output, Arg;

const int TBLSZ = 23;

type Token_value = enum {
        NAME,           NUMBER,         END,
        PLUS='+',       MINUS='-',      MUL='*',        DIV='/',
        PRINT=';',      ASSIGN='=',     LP='(',         RP=')',
        };

type Name = struct {
        char *  string;
        Name *  next;
        double  value;
        };

Arg     a;
Input   in;
Output  out;

Token_value     curr_tok;
double          number_value;
char[256]       name_string;
Name*[TBLSZ]    table;


extern int strlen (char *s);
extern int strcmp (char *s1, s2);
extern char *strcpy (char *s1, s2);

static int isspace (char ch)
{
  return (ch == ' ' || ch == '\t');
}

static int isalnum (char ch)
{
  return ((ch >= 'a' && ch <= 'z') ||
          (ch >= 'A' && ch <= 'Z') ||
          (ch >= '0' && ch <= '9'));
```

```
}

static int isalpha (char ch)
{
  return ((ch >= 'a' && ch <= 'z') ||
          (ch >= 'A' && ch <= 'Z'));
}

Name* look (char* id; int ins = 0)
{
  int ii = 0;
  char *pp = id;
  Name *n, nn;

  while (*pp)
    ii = ii << 1 ^ *pp++;
  if (ii < 0)
    ii = -ii;
  ii %= TBLSZ;

  for (n = table[ii]; n; n=n->next)
    if (strcmp (id, n->string) == 0)
      return n;

  if (ins == 0)
    raise "name not found";

  nn = new (Name);
  nn -> string = new (char, strlen(id)+1);
  strcpy (nn->string, id);
  nn -> value = 1;
  nn -> next = table[ii];
  table[ii] = nn;
  return nn;
}

Name* insert (char *id)
{
  return look (id, 1);
}

Token_value get_token()
{
  char ch;

  do {
    if (! in.get (ch))
      return curr_tok = END;
  } while (ch != '\n' && isspace (ch));

  switch (ch)
  {
    case ';', '\n', ',':
      // in >> ws;
      return curr_tok=PRINT;
    case '*', '/', '+', '-', '(', ')', '=':
      return curr_tok=(Token_value)ch;
    case '0'..'9', '.':
      in.putback (ch);
      in >> number_value;
      return curr_tok=NUMBER;
    default:
      if (isalpha (ch))
      {
        char *p = name_string;
        *p++ = ch;
        while (in.get(ch) && isalnum (ch))
          *p++ = ch;
        *p = '\0';
        in.putback (ch);
        return curr_tok=NAME;
      }
      raise "bad token";
  }
}

double expr (); // forward declaration

double prim ()
{
  switch (curr_tok) {
    case NUMBER:
      get_token ();
      return number_value;
      break;
    case NAME:
      if (get_token () == ASSIGN)
```

```
      {
        Name *n = insert (name_string);
        get_token ();
        n -> value = expr ();
        return n -> value;
      }
      return look (name_string) -> value;
    case MINUS:
      get_token ();
      return - prim ();
    case LP:
      get_token ();
      {
        double e = expr ();
        if (curr_tok != RP)
          raise ") expected";
        get_token ();
        return e;
      }
    case END:
      return 1;
    default:
      raise "primary expected";
  }
}

double term ()
{
  double left = prim();
  for (;;)
    switch (curr_tok) {
      case MUL:
        get_token ();
        left *= prim ();
        break;
      case DIV:
        get_token ();
        {
          double d = prim ();
          if (d == 0)
            raise "Division by 0";
          left /= d;
        }
        break;
      default:
        return left;
    }
}

double expr ()
{
  double left = term();
  for (;;)
    switch (curr_tok) {
      case PLUS:
        get_token ();
        left += term ();
        break;
      case MINUS:
        get_token ();
        left -= term ();
        break;
      default:
        return left;
    }
}

export void start ()
{
  insert ("pi") -> value = 3.1415926535897932385;
  insert ("e") -> value = 2.7182818284590452354;
}

export void eval ()
{
  [
    for (;;)
    {
      get_token();
        if (curr_tok == END)
          break;
        if (curr_tok == PRINT)
          continue;
        out.format ("Value: %f\n") << expr ();
    }
    when char *:
      out.format("Error: %s\n") << exc;
```

```
      in.skip_line ();
  ]
}

constructor ()
{
  start ();
  if (a.argc == 1)
  {
    out << "Desk-calculator v1.0\n";
    for (;;)
    {
      out << "> ";
      if (in.eof ())
        break;
      eval ();
    }
  }
  else
  {
    char *s;
    for (a.setarg(1); a.getarg(s); )
    {
      in.setstr (s);
      eval ();
    }
  }
}
```

A.2 Makefile gerado (deskcalc.umk)

```
# Makefile for Cm project 00deskcalc (Unix version)

# This should be invoked by "make" from the same
#   directory whence the Cm compiler was invoked
CDIR = /proj/cm/cm/c
PROG = deskcalc
CFLAGS = -g -I$(CDIR) $$CMCFLAGS
LDFLAGS = -L$(CDIR)
CMLIB = -lcm $$CMCLIBS
MAIN = main--
CC = cc
OBJS = $(CDIR)/$(MAIN).o\
        $(CDIR)/Arg/p00.o\
        $(CDIR)/Cstdio/p00.o\
        $(CDIR)/Input/p00.o\
        $(CDIR)/deskcalc/p00.o\
        $(CDIR)/Output/p00.o

$(PROG): $(OBJS)
        $(CC) $(LDFLAGS) $$CMLDFLAGS -Fe$(PROG) $(OBJS) $(CMLIB)

clear:
        rm $(OBJS)


.c.o:
        $(CC) $(CFLAGS) -Fo$*.o -c $<

$(CDIR)/$(MAIN).o: $(CDIR)/$(MAIN).c $(CDIR)/cm.h
        $(CC) $(CFLAGS) -Fo$(CDIR)/$(MAIN).o -c $(CDIR)/$(MAIN).c
$(CDIR)/Arg/p00.o: $(CDIR)/Arg/p00.c $(CDIR)/cm.h
$(CDIR)/Cstdio/p00.o: $(CDIR)/Cstdio/p00.c $(CDIR)/cm.h
$(CDIR)/Input/p00.o: $(CDIR)/Input/p00.c $(CDIR)/cm.h
$(CDIR)/deskcalc/p00.o: $(CDIR)/deskcalc/p00.c $(CDIR)/cm.h
$(CDIR)/Output/p00.o: $(CDIR)/Output/p00.c $(CDIR)/cm.h
```

A.3 Arquivo principal (main--.c)

```
/* Cm main file for top-level compilation of
   class deskcalc[00] */

#include "cm.h"
#include "/proj/cm/cm/c/deskcalc/p00.h"

struct __A00deskcalc instance;

int __Cmargc;
```

```
char **__Cmargv, *__Cmarge;

__CmPortInfo __CmPort[] = {
        { "dummy", ___NIL___, -1 },
};
int __CmMaxPort = (sizeof(__CmPort)/sizeof(__CmPort[0]));
_Void_ main (argc, argv, arge)
int argc; char *argv[], *arge;
{
  __CmSetupArgs (argc, argv, arge);
  __CmMTE_startup (20);

  __b00deskcalc ();
  __fAAt00deskcalc (__a00deskcalc (&instance));
  __d00deskcalc (&instance);
  __e00deskcalc ();
}
```

A.4 Arquivo de Interface C (p00.h)

```
/* Cm translation for class 00deskcalc */
/*    C header file for instance */

#ifndef __H_00deskcalc__
#define __H_00deskcalc__

#include "Output/p00.h"
#include "Input/p00.h"
#include "Arg/p00.h"

struct  __TAAA00deskcalc {
          char (*string);
                struct  __TAAA00deskcalc (*next);
          double value;
    } ;
typedef struct __A00deskcalc
        {
          struct __A00Arg a;
          struct __A00Input in;
          struct __A00Output out;
          int curr_tok;
          double number_value;
          char name_string [256];
                  struct  __TAAA00deskcalc (*table [23]);
        } __A00deskcalc;

extern struct __A00deskcalc *__a00deskcalc();
extern _Void_ __b00deskcalc();
extern _Void_ __d00deskcalc();
extern _Void_ __e00deskcalc();

extern int strlen ();
extern int strcmp ();
extern char * strcpy ();
extern int __fAAW00deskcalc ();
extern int __fAAX00deskcalc ();
extern int __fAAY00deskcalc ();
extern struct __TAAA00deskcalc *__fAAZ00deskcalc ();
extern struct __TAAA00deskcalc *__fAAe00deskcalc ();
extern int __fAAf00deskcalc ();
extern double __fAAi00deskcalc ();
extern double __fAAj00deskcalc ();
extern double __fAAm00deskcalc ();
extern _Void_ __fAAq00deskcalc ();
extern _Void_ __fAAr00deskcalc ();
extern struct __A00deskcalc*__fAAt00deskcalc ();


#endif
```

A.5 Arquivo de código C (p00.c)

```
/* Cm translation of class 00deskcalc */

#include "cm.h"
#include "/proj/cm/cm/c/deskcalc/p00.h"

extern int strlen ();
```

```
extern int strcmp ();
extern char * strcpy ();
/* ---------------------------------------------------------------------------
** Function "isspace(...)"
*/
int __fAAW00deskcalc (ch)
        char ch;
{
  int ___Cmrpar;
return ((ch== ' ')|| (ch== '\t'));
}


/* ---------------------------------------------------------------------------
** Function "isalnum(...)"
*/
int __fAAX00deskcalc (ch)
        char ch;
{
  int ___Cmrpar;
return ((((ch>= 'a')&& (ch<= 'z'))|| ((ch>= 'A')&& (ch<= 'Z')))
                || ((ch>= '0')&& (ch<= '9')));
}


/* ---------------------------------------------------------------------------
** Function "isalpha(...)"
*/
int __fAAY00deskcalc (ch)
        char ch;
{
  int ___Cmrpar;
return (((ch>= 'a')&& (ch<= 'z'))|| ((ch>= 'A')&& (ch<= 'Z')));
}


/* ---------------------------------------------------------------------------
** Function "look(...)"
*/
struct __TAAA00deskcalc *__fAAZ00deskcalc (self,
        id,
        ins)
        struct __A00deskcalc *self;
        char (*id);
        int ins;
{
    struct  __TAAA00deskcalc (*___Cmrpar);
  int ii;
  char (*pp);
    struct  __TAAA00deskcalc (*n);
    struct  __TAAA00deskcalc (*nn);
ii = 0;
pp = id;
while ((*pp))
    (ii=((ii<< 1)^(*(pp)++)));
  if ((ii<0))
    (ii= -ii);
    (ii%=23);
for ((n=self->table[ii]);n;(n=(*n).next))
  if ((strcmp((char (*))id,
        (char (*))(*n).string)== 0))
return n;
  if ((ins== 0))
  __CmMTE_raise (**c", (void *) "name not found", "/proj/cm/tst/Ok/deskcalc.cm", 67);
    (nn=__Cmalloc (1,sizeof(struct  __TAAA00deskcalc ),(int(*)())0));
    ((*nn).string=__Cmalloc ((strlen((char (*))id)+1),sizeof(char ),(int(*)())0));
    strcpy((char (*))(*nn).string,
        (char (*))id);
    ((*nn).value=1);
    ((*nn).next=self->table[ii]);
    (self->table[ii]=nn);
return nn;
}


/* ---------------------------------------------------------------------------
** Function "insert(...)"
*/
struct __TAAA00deskcalc *__fAAs00deskcalc (self,
        id)
        struct __A00deskcalc *self;
        char (*id);
{
    struct  __TAAA00deskcalc (*___Cmrpar);
return __fAAZ00deskcalc(self,
        (char (*))id,
        (int)1);
```

```
}

/* ------------------------------------------------------------------------
** Function "get_token(...)"
*/
int __fAAf00deskcalc (self)
        struct __A00deskcalc *self;
{
  int ___Cmrpar;
  char ch;
  __CmDW0:
    {/*C*/
          if (!__fAAX00Input((&self->in),
        (&ch)))
return (self->curr_tok=2);
    }/*C*/
if (((ch!= '\n')&& __fAAW00deskcalc((int)ch))) goto __CmDW0;
  {/*T*/
  long int __CmSwitch = ch;
    {/*C*/
    switch (__CmSwitch)
    {/*Csw*/

      case ';':
      case '\n':
      case ' ':
return (self->curr_tok=(';'+0));

      case '*':
      case '/':
      case '+':
      case '-':
      case '(':
      case ')':
      case '=':
return (self->curr_tok=((int )ch));

      case 48: /* '0' */
      case 49: /* '1' */
      case 50: /* '2' */
      case 51: /* '3' */
      case 52: /* '4' */
      case 53: /* '5' */
      case 54: /* '6' */
      case 55: /* '7' */
      case 56: /* '8' */
      case 57: /* '9' */
      case '.':
        __fAAU00Input((&self->in),
        (int)ch);
        __fAAj00Input((&self->in),
        (&self->number_value));
return (self->curr_tok=1);

      default:
      if (__fAAY00deskcalc((int)ch))
        {/*C*/
          char (*p);
p = self->name_string;
                    ((*(p)++)=ch);
while ((__fAAX00Input((&self->in),
        (&ch))&& __fAAX00deskcalc((int)ch)))
            ((*(p)++)=ch);
            ((*p)='\000');
            __fAAU00Input((&self->in),
        (int)ch);
return (self->curr_tok=0);
        }/*C*/
      __CmRTE_raise ("c", (void *) "bad token", "/proj/cm/tst/Ok/deskcalc.cm", 114);
    }/*C*/
}/*Csw*/
  }/*T*/
  return ___Cmrpar;
}


/* ------------------------------------------------------------------------
** Function "prim(...)"
*/
double __fAAj00deskcalc (self)
        struct __A00deskcalc *self;
{
  double ___Cmrpar;
  {/*T*/
  long int __CmSwitch = self->curr_tok;
    {/*C*/
```

```
    switch (__CmSwitch)
    {/*Csw*/

      case 1:
        __fAAf00deskcalc(self);
return self->number_value;
break;

      case 0:
      if ((__fAAf00deskcalc(self)== ('='+0)))
        {/*C*/
                        struct  __TAAA00deskcalc (*n);
n = __fAAe00deskcalc(self,
        (char (*))self->name_string);
                        __fAAf00deskcalc(self);
            ((*n).value=__fAAi00deskcalc(self));
return (*n).value;
        }/*C*/
return (*__fAAZ00deskcalc(self,
        (char (*))self->name_string,
        (int)0)).value;

      case ('-'+0):
        __fAAf00deskcalc(self);
return  -__fAAj00deskcalc(self);

      case ('('+0):
        __fAAf00deskcalc(self);
        {/*C*/
          double e;
e = __fAAi00deskcalc(self);
                    if ((self->curr_tok!= (')'+0)))
          __CmMTE_raise (**c", (void *) ") expected",
                                                  "/proj/cm/tst/Ok/deskcalc.cm", 144);
            __fAAf00deskcalc(self);
return e;
        }/*C*/

      case 2:
return 1;

      default:
      __CmMTE_raise (**c", (void *) "primary expected",
                                                  "/proj/cm/tst/Ok/deskcalc.cm", 151);
    }/*C*/
}/*Csw*/
  }/*T*/
  return ___Cmrpar;
}


/* ----------------------------------------------------------------------------
** Function "term(...)"
*/
double __fAAm00deskcalc (self)
        struct __A00deskcalc *self;
{
  double ___Cmrpar;
  double left;
left = __fAAj00deskcalc(self);
for (;;)
  {/*T*/
  long int __CmSwitch = self->curr_tok;
    {/*C*/
    switch (__CmSwitch)
    {/*Csw*/

      case ('*'+0):
        __fAAf00deskcalc(self);
        (left*=__fAAj00deskcalc(self));
break;

      case ('/'+0):
        __fAAf00deskcalc(self);
        {/*C*/
          double d;
d = __fAAj00deskcalc(self);
                    if ((d== 0))
            __CmMTE_raise (**c", (void *) "Division by 0",
                                                  "/proj/cm/tst/Ok/deskcalc.cm", 169);
              (left/=d);
        }/*C*/
break;

      default:
return left;
    }/*C*/
```

```
)/*Csw*/
  }/*T*/
  return ___Cmrpar;
}


/* ---------------------------------------------------------------------------
** Function "expr(...)"
*/
double __fAAi00deskcalc (self)
        struct __A00deskcalc *self;
{
  double ___Cmrpar;
  double left;
left = __fAAm00deskcalc(self);
for (;;)
  {/*T*/
  long int __CmSwitch = self->curr_tok;
    {/*C*/
    switch (__CmSwitch)
    {/*Csw*/

      case ('+'+0):
        __fAAf00deskcalc(self);
        (left+=__fAAm00deskcalc(self));
break;

      case ('-'+0):
        __fAAf00deskcalc(self);
        (left-=__fAAm00deskcalc(self));
break;

      default:
return left;
    }/*C*/
)/*Csw*/
  }/*T*/
  return ___Cmrpar;
}


/* ---------------------------------------------------------------------------
** Function "start(...)"
*/
_Void_ __fAAq00deskcalc (self)
        struct __A00deskcalc *self;
{
    ((*__fAAe00deskcalc(self,
        (char (*))"pi")).value=3.1415926535897932385);
    ((*__fAAe00deskcalc(self,
        (char (*))"e")).value=2.7182818284590452354);
}


/* ---------------------------------------------------------------------------
** Function "eval(...)"
*/
_Void_ __fAAr00deskcalc (self)
        struct __A00deskcalc *self;
{
    {/*C*/
    TRY_BEGIN
for (;;)
        {/*C*/
                    __fAAf00deskcalc(self);
          if ((self->curr_tok== 2))
break;
          if ((self->curr_tok== (';'+0)))
          continue;
            __fAAX00Output(__fAAs00Output((&self->out),
        (char (*))"Value: %f\n"),
        (double)__fAAi00deskcalc(self));
        }/*C*/
    TRY_HANDLER("*c")
        char (*exc) = (char (*)) __CmMTE_exc;
            __fAAY00Output(__fAAs00Output((&self->out),
        (char (*))"Error: %s\n"),
        (char (*))exc);
            __fAAM00Input((&self->in));
    TRY_RERAISE_END
    }/*C*/
}


/* ---------------------------------------------------------------------------
** Function "__CO__(...)"
*/
```

```
struct __A00deskcalc*__fAAt00deskcalc (self)
        struct __A00deskcalc *self;
{
  struct __A00deskcalc (*___Cmrpar);
    __fAAq00deskcalc(self);
  if ((self->a.argc== 1))
    {/*C*/
          __fAAY00Output((&self->out),
        (char (*))"Desk-calculator v1.0\n");
for (;;)
        {/*C*/
                  __fAAY00Output((&self->out),
        (char (*))"> ");
          if (__fAAR00Input((&self->in)))
break;
          __fAAr00deskcalc(self);
        }/*C*/
    }/*C*/
  else    {/*C*/
      char (*s);
    for (__fAAG00Arg((&self->a),
        (int)1);__fAAH00Arg((&self->a),
        (&s));)
        {/*C*/
                  __fAAT00Input((&self->in),
        (char (*))s);
          __fAAr00deskcalc(self);
        }/*C*/
    }/*C*/
  return self;
}


/* ----------------------------------------------------------------------------
** Class automatic data initializer:
*/
struct __A00deskcalc *__a00deskcalc (self)
        struct __A00deskcalc *self;
{
  __fAAI00Arg (__a00Arg (&self->a));
  __fAAm00Input (__a00Input (&self->in));
  __fAAf00Output (__a00Output (&self->out));
  return (self);
}

/* ----------------------------------------------------------------------------
** Class destructor:
*/
_Void_ __d00deskcalc (self)
        struct __A00deskcalc *self;
{
  __d00Output (&self->out);
  __d00Input (&self->in);
  __d00Arg (&self->a);
}
/* ----------------------------------------------------------------------------
** Class static constructor (init):
*/
_Void_ __b00deskcalc ()
{
  static char done = 0;
  if (done)
    return;
  done = 1;
  __b00Output ();
  __b00Input ();
  __b00Arg ();
}
/* ----------------------------------------------------------------------------
** Class static destructor:
*/
_Void_ __e00deskcalc ()
{
  static char done = 0;
  if (done)
    return;
  done = 1;
}
```

Database	=A1

Apêndice B

Referências Bibliográficas

Este apêndice apresenta as referência bibliográficas levantadas durante a elaboração da tese de mestrado. Inicialmente são apresentadas notas gerais sobre as referências, que são apresentadas em seguida, agrupadas pelo assunto principal que abordam, e com breves comentários sobre seu conteúdo e qualidade.

B.1 Notas gerais sobre as referências bibliográficas

As referências marcadas com '*', por exemplo **[*C++90b]**, não foram lidas, mas as menções a elas foram suficientes para inseri-las aqui.

B.1.1 A linguagem C

Os livros clássicos sobre a linguagem C são de Kernighan e Ritchie: **[C-K&R78]**, para o "padrão" `K&R`, e **[C-ANSI88]**, para o "padrão" `ANSI`. Estes livros descrevem a linguagem e seu uso. O manual de referência é dado como apêndice.

B.1.2 A linguagem C++

O livro mais completo sobre C++ é de Stroustrup **[C++91]**; descreve os principais recursos e características da linguagem, apresenta seu manual de referência (completo), e chega inclusive a discutir aspectos relacionados a projeto e desenvolvimento de sistemas e bibliotecas.

O livro de Lippman **[C++91b]** também é bastante abrangente, mas não tão completo. Tem como vantagem ser mais acessível.

As demais referências são principalmente artigos ou coletâneas de artigos apresentando tuoriais, pontos específicos da linguagem e/ou possíveis técnicas de implementação. Estas referências são dadas nas coletâneas **[C++84]** e **[C++89]**, e nos artigos **[C++90]** (tratamento de exceções), **[C++93]** e **[C++93a]** (gerenciamento de memória).

A referência **[*C++90b]** é provavelmente uma versão anterior de **[C++90]**.

o outro lado
Os artigos **[C++Crit88]** e **[C++Crit91]** apresentam críticas à linguagem C++. A argumentação básica é que a linguagem não pode ser considerada orientada a objetos, no sentido mais amplo do paradigma.

B.1.3 A linguagem Cm

As primeiras referências à linguagem de programação Cm aparecem em **[AHand87]** e **[AHand87b]**. Apesar da linguagem estar praticamente definida desde 1986, sua primeira especificação formal publicada é **[Cm88]**. **[Cm88b]** apresenta de maneira menos formal aspectos interessantes da linguagem com alguns exemplos. A versão seguinte da linguagem, redefinida e implementada por Carlos Furuti, é descrita em **[Cm91]** e **[Cm91b]**. A versão atual é descrita nesta tese.

sistema de run-time
Os artigos **[linear92]**, **[OMNI92]** e **[RTS92]** descrevem bibliotecas C usadas (ou a serem usadas) pela biblioteca de *run-time* de Cm para implementação de portas de comunicação.

suporte à programação
O artigo **[Cm92]** apresenta um tradutor de arquivos de *header* C para Cm. O artigo **[EdCm89]** descreve a implementação de um modo de edição Cm para o editor emacs.

B.2 Publicações do Projeto A_HAND

1. **[AHand87]** **A-HAND: Ambiente de desenvolvimento de *software* baseado em Hierarquias de Abstração em Níveis Diferenciados**
Drummond, R. e Liesenberg, H.
IV Encontro de Trabalhos do Projeto ETHOS, Petrópolis, RJ, abril 1987.
Revisto e reimpresso como relatório técnico, Projeto A_HAND, outubro 1987.

2. **[AHand87b]** **Requisitos para um ambiente de desenvolvimento de PROGRAMAS**
Rogério Drummond e Hans Liesenberg
I Encontro IBM de Ciência e Tecnologia em Informática, Rio de Janeiro, RJ, novembro 1987.

3. **[Cm88]** **Manual de referência da Linguagem Cm (versão preliminar)**
Rogério Drummond e F. Q. Bueno da Silva
Projeto A_HAND, DCC-IMECC-UNICAMP, março 1988.

4. **[Cm88b]** **Programação em Cm**
F. Q. B. da Silva, Hans K. E. Liesenberg e Rogério Drummond
Anais do XV Semish, pp 101–102, Rio de Janeiro, RJ, julho 1988.

5. **[Cm91]** **Introdução a Cm**
Carlos Alberto Furuti
Relatório técnico, Projeto A_HAND, DCC–IMECC–UNICAMP, setembro 1991.

6. **[Cm91b]** **Um compilador para uma linguagem de programação orientada a objetos**
Carlos Alberto Furuti
Tese de mestrado, DCC-IMECC-UNICAMP, julho 1991.

7. **[Cm92]** **Suporte à Programação em Cm**
Maria Claudia Borges Barros
Relatório técnico, Projeto A_HAND, DCC-IMECC-UNICAMP, setembro 1992.

8. **[linear92]** **Linear – linearizador de estruturas complexas**
Rogério Drummond e Marcelo Augusto Holloway de Souza
Relatório técnico, Projeto A_HAND, DCC–IMECC–UNICAMP, 1992.

9. **[OMNI92]** **OMNI – Sistema de suporte a aplicações distribuídas**
Rogério Drummond e Cassius Di Cianni
Anais do VI Simpósio Brasileiro de Engenharia de Software, pp 309–324, novembro 1992.

Descreve o Sitema OMNI de suporte a aplicações distribuídas, destacando o servidor de nomes e os mecanismos de gerenciamento de processos e portas de comunicação do OMNI.

10. **[MTE92]** **Mecanismos de Tratamento de Exceções**
Alexandre Prado Teles
Relatório interno (curso de verão), Projeto A_HAND, DCC–IMECC–UNICAMP, fevereiro 1992.

Apresenta os modelos de mecanismos de tratamento de exceções, e sua implementação em diversas linguagens. Descreve brevemente o mecanismo proposto para Cm e dá uma possível implementação em C de um mecanismo semelhante (a "base" do mecanismo de Cm).

11. **[RTS92]** **Aspectos da Implementação de Objetos Distribuídos**
Rogério Drummond, Celso Gonçalves Júnior e Alexandre P. Teles
Anais do 11° Simpósio Brasileiro de Redes de Computadores, pp 139–152, maio 1993.

Apresenta a solução proposta pelo Projeto A_HAND para a implementação de Objetos Distribuídos numa rede de computadores heterogêneos tendo Unix como sistema operacional. Entre outros assuntos, discute o suporte provido pela linguagem Cm a nível de linguagem e de *run–time*.

12. **[EdCm89]** **Edição de Classes Cm com GNU Emacs**
Christina von Flach
Relatório interno, Projeto A_HAND, DCC–IMECC–UNICAMP, agosto 1989.

Descreve a implementação de um editor de classes Cm e a criação de um modo de edição específico para Cm utilizando o GNU Emacs.

B.3 Teoria de linguagem e compiladores

13. **[Comp86]** **Compilers Principles, Techniques, and Tools**
Alfred V. Aho, Ravi Sethi, e Jeffrey D. Ullman
Addison–Wesley Publ. Co, 1986

14. **[CK90]** **On open arrays and variable numbers of parameters**
C. S. R. Carvalho e T. Kowaltowski
Relatório técnico, IMECC, Unicamp, 1990.

15. **[lang92]** **The Euclidean Definition of the Functions div and mod**
Raymond T. Boute
ACM Transactions on Programming Languages and Systems, 14 (2), pp 127–144, abril 1992.

Propõe uma definição (E-definition) para os operadores `div` e `mod` baseadas no teorema de Euclides. Compara esta definição com as definições tradicionais baseadas em divisão com truncamento (T-definition) e divisão com arredondamento/*flooring* (F-division).

16. **[CAP89]** **Comments, Assertions, and Pragmas**
Peter Grogono
ACM Sigplan Notices 24 (3): 79–84, march 1989.

Disserta sobre a importância de comentários em programas, propondo que estes devem fazer parte da gramática da linguagem, tendo a semântica tradicional de "espaço branco" e semânticas especiais de *pragmas* (diretivas de compilação) e *asserções* (pré e pós-condições, invariantes de malha e condições genéricas).

B.4 Sobre orientação a objetos

17. **[OOD91]** **Object Oriented Design with applications**
Grady Booch
The Benjamin/Cummings Publishing Company, Inc., 1991.

O livro explica em detalhes os conceitos envolvidos em projeto de programas, e apresenta o método com notação gráfica voltado para projeto orientado a objetos. Também apresenta a aplicação do método utilizando várias linguagens (Smaltalk, Object Pascal, C++, Common Lisp e Ada).

18. **[OOP90]** **Concepts and Paradigms of Object-Oriented Programming**
Peter Wegner
OOPS Messenger, 1 (1)7–87, agosto 1990.

Este artigo ocupa a revista inteira, e é bastante interessante, apresentando diversos conceitos relativos a paradigmas de programação e orientação a objetos, inclusive mostrando as diferentes interpretações de um mesmo conceito em linguagens/paradigmas diferentes.

19. **[Booch90]** **The Design of the C++ Booch Components**
Graddy Booch e Michael Vilot
ACM Sigplan Notices ECOOP/OOPSLA'90 Proceedings, pp 1–11, outubro 1990.

B.5 Ferramentas de desenvolvimento

20. **[yacc78]** **Yacc: yet another compiler-compiler**
S. C. Johnson
Relatório técnico, Bell Laboratories, 1978.

Guia (mistura de tutorial e referência) para utilização do gerador de parsers *yacc*.

B.6 Linguagens de programação

21. **[HOPL93]** **Preprints of The Second ACM Sigplan History of Programming Languages Conference**
ACM Sigplan Notices 28 (3), march 1993.

Coletânea de artigos bastante interessantes sobre a origem e evolução de diversas linguagens, incluindo Concurrent Pascal (Brinch Hansen), Prolog, Icon, SmallTalk, Algol68, CLU (B. Liskov), Forth, C (Ritchie), FORMAC, Lisp, C++ (Stroustrup), Ada e Pascal (Wirth).

22. **[HOPL93a]** **A History of C++: 1979–1991**
Bjarne Stroustrup
HOPL-II, ACM Sigplan Notices 28(3):271–297, march 1993 (**[HOPL93]**).

Mostra a origem e evolução de C++, mostrando as alternativas consideradas para vários recursos da linguagem e justificando as decisões e restrições adotadas em C++. É um artigo bastante interessante, mas pouco adequado para o aprendizado da linguagem em si.

23. **[C-K&R78]** **The C Programming language**
Brian W. Kernighan e Dennis M. Ritchie
Prentice–Hall Inc., 1978.

Descrição formal oficial da linguagem C padrão, ou pré-ANSI.

24. **[C-ANSI88]** **The C Programming language**
Brian W. Kernighan e Dennis M. Ritchie
2ª edição, Prentice–Hall Inc., 1988.

Descrição formal oficial da linguagem C padrão ANSI.

25. **[C++93]** **Implementing new and delete**
Stephen D. Clamage
C++ Report, 5 (4), pp. 30-34, maio 1993.

26. **[C++93a]** **New operators for array new and delete ... and more**
Josée Lajoie
C++ Report, 5 (4), pp. 53-56, maio 1993.

27. **[C++91]** **The C++ Programming Language (2nd ed.)**
Bjarne Stroustrup
Addison–Wesley Publishing Company, 1991

28. [C++91b] **C++ primer (2nd ed.)**
Stanley B. Lippman
Addison–Wesley Publishing Company, 1991.

29. [C++90] **Exception handling for C++**
Andrew Koenig, Bjarne Stroustrup
Journal of object oriented programming, pp 16–33, julho/agosto 1990.

Este artigo é bastante interessante, desrevendo detalhadamente o mecanismo de tratamento de exceções de C++. Também mostra em linhas gerais alguns pontos interessantes da geração de código relacionados a este mecanismo, tais como tratamento de construtores e destrutores, e representação de tipos em tempo de execução.

30. [*C++90b] **Exception Handling for C++**
Bjarne Stroustrup
C++ conference, USENIX Associations, San Francisco, CA, abril 1990.

31. [C++89] **AT&T C++ Language System Selected Readings**
Sun Microsystems
PartNo: 800-3434-10
Revision A, 20 October 1989

32. [C++89a] **Evolution of C++: 1985 to 1989**
Bjarne Stroustrup
Reprinted in **[C++89]**.

33. [C++89b] **An Introduction to C++**
Keith Gorlen
Reprinted in **[C++89]**.

34. [C++89c] **An Overview of C++**
Bjarne Stroustrup
ACM Sigplan Notices, October 1986, pp. 7-18.
Reprinted in **[C++89]**.

35. [C++89d] **What is Object-Oriented Programming**
Bjarne Stroustrup
IEEE Software Magazine, May 1988, pp 10-20.
Reprinted in **[C++89]**.

36. [C++89e] **Multiple Inheritance for C++**
Bjarne Stroustrup
Proceedings of the EUUG Spring Conference, May 87.
Revised and reprinted in **[C++89]**.

37. [C++89f] **Type-safe Linkage for C++**
Bjarne Stroustrup
Computing Systems, Volume VI, no. 4, Fall 1988, pp. 371-404.
Reprinted in **[C++89]**.

38. **[C++89g]** **Access Rules for C++**
Phil Brown
Reprinted in **[C++89]**.

39. **[C++84]** **C++ Release E Documentation**
AT&T Bell Laboratories
November 1984

Coletânea de artigos sobre C++

40. **[C++84a]** **A C++ Tutorial**
Bjarne Stroustrup
AT&T Bell Laboratories
Reprinted in **[C++84]**.

41. **[C++84b]** **Data Abstraction in C++**
Bjarne Stroustrup
AT&T Bell Laboratories
Reprinted in **[C++84]**.

42. **[C++84c]** **Operator Overloading in C++**
Bjarne Stroustrup
AT&T Bell Laboratories
Reprinted in **[C++84]**.

43. **[C++84d]** **Operator Overloading in C++**
Bjarne Stroustrup
AT&T Bell Laboratories
Reprinted in **[C++84]**.

44. **[C++84e]** **Complex Arithmetic in C++**
Leonie V. Rose, Bjarne Stroustrup
AT&T Bell Laboratories
Reprinted in **[C++84]**.

45. **[C++84f]** **A Set of C++ Classes for Co-routine Style Programming**
Bjarne Stroustrup
AT&T Bell Laboratories
Reprinted in **[C++84]**.

46. **[C++84g]** **The C++ Programming Language - Reference Manual**
Bjarne Stroustrup
AT&T Bell Laboratories
Reprinted in **[C++84]**.

47. [C++Crit88] **On the darker side of C++**
Markku Sakkinen
Proceedings of European conference on object–oriented programming
Springer–Verlag, Lecture Notes in Computer Science 322, pp 162–176.

O artigo apresenta vários problemas da primeira versão de C++ (1.0) que foram, em sua maioria, solucionados em versões subsequentes da linguagem. Ainda assim o artigo é bastante interessante, por sua visão mais crítica.

48. [C++Crit91] **On the object orientedness of C++**
Chung–Shyan Liu
ACM Sigplan Notices, 26 (3), pp 63–77, março 1991.

O artigo apresenta e compara as definições de programação orientada a objetos como "uma sintaxe de envio de mensagens com herança" (SmallTalk) e como "um tipo abstrato de dados com herança" (C++). Cada uma dessas definições têm suas vantagens e desvantagens, e a principal desvantagem de C++ é "permitir" que um objeto altere a área de dados privada de outro objeto da mesma classe.

49. [ADA79] **Rationale for the design of the Ada programming language**
J. D. Ichbiah, J. G. Barnes, J. C. Heliard, B. Krieg-Bruckner, O. Roubine, B. A. Wichmann
ACM Sigplan Notices, 14 (6), junho 1979.

50. [ADA] **Programming in Ada (2nd ed)**
J. G. P. Barnes
International Computer Science Series
Addison-Wesley Publishing Company, pp 133–144

B.7 Tratamento de exceções

51. [ExcC92] **Exceptional C or C with Exceptions**
N. H. Gehani
Software Practice & Experience, 22 (10), pp 827–848, outubro 1992.

Exceptional C é um superconjunto de C que provê facilidades de tratamento de exceções. O mecanismo proposto trabalha tanto com o modelo de terminação (implementado com setjmp/longjmp) como com o de retomada (implementado com sinais Unix).

52. [Exc75] **Exception handling: issues and a proposed notation**
John B. Goodenough
Communications of the ACM, 18 (2), pp 683–696, dezembro 1975.

53. [*Exc82] **Exception handling and software fault tolerance**
F. Cristian
IEEE Trans. on Computers, C–31 (6), pp 531–540, junho 1982.

54. [Exc83] **Exception handling in C programs**
P. A. Lee
Software Practice & Experience, 13 (5), pp 389–405, maio 1983.

55. [Exc84] **Exception Handling**
Ellis Horowitz
Fundamental of programming languages (2nd ed.), capítulo 9, pp 265–285,
Springer–Verlag, NY, 1984.

Conceitua tratamento de exceções e descreve brevemente os mecanismos de tratamento de exceções das linguagens PL/I, CLU, Mesa e Ada.

56. [Exc85] **A modular verifiable exception–handling mechanism**
S. Yemini, D. M. Berry
ACM TOPLS, 7 (2), pp 214–243, abril 1985.

57. [Exc86] **Exceptions in object–oriented languages**
Alexander Borgida
ACM Sigplan Notices, 21 (10), pp 107–119, outubro 1986.

58. [*Exc88] **Exception handling without language extensions**
W. M. Miller
C++ Conference, USENIX Association, Denver CO, outubro 1988.

59. [Exc90] **An exceptional ideal**
Andrew Koenig
Journal of object oriented programming, pp 52–59, julho/agosto 1990.

Apresenta o mecanismo de tratamento de exceções de C++, justificando porque foi introduzido na linguagem, e avaliando os impactos que pode causar em programadores C++.

60. [Mach89] **The Mach exception handling facility**
David L. Black, David B. Gloub, Karl Hauth, Avadis Tevanian e Richard Sanzi
Proceedings of workshop on parallel and distributed debugging
ACM Sigplan Notices, 24 (1), pp 45-56, janeiro 1989.

61. [uSys92] **Synchronous and Asynchronous Handling of Abnormal Events in the μSystem**
Peter A. Buhr, Hamish I. MacDonald, C. Robert Zarnke
Software Parctice & Experience, 22 (9), pp 735–776, setembro, 1992.

O artigo apresenta o modelo de controle de eventos excepcionais do μSystem (uma biblioteca C que provê concorrência por processos light–weight em Unix). Há duas formas diferentes de tratamento: exceções, com semântica de terminação, e intervenções, com semântica de retomada.

62. [Borg85] **Language features for flexible handling of exceptions in information systems**
Alexander Borgida
ACM Transactions on Database Systems, 10 (4), pp 565–603, dezembro 1985.

Apresenta um mecanismo de controle de exceções propício para sistemas de informação baseados em banco de dados. Basicamente este mecanismo formaliza uma maneira de se manter informações "excepcionais" (que violam as restrições normais de integridade) e permitir que essas informações sejam manipuladas. Não é um mecanismo de uso geral aplicável a uma linguagem de programação procedural convencional.

63. [CHILL] CCITT Rec. Z200
Fascículo X.6, pp 120–121.

64. [Lore88] **An object–oriented exception handling system for an object-oriented language**
Christophe Dony
Proceedings of European conference on object–oriented programming
Lecture Notes in Computer Science 322, pp 146–161, Spring-Verlag, 1988.

65. [Stalk90] **Exception handling and object-oriented programming: towards a synthesis**
Christophe Dony
ACM Sigplan Notices ECOOP/OOPSLA'90 Proceedings, pp 322–330, outubro 1990.

66. [Horo] **Programming languages: A grand tour (3rd ed.)**
Ellis Horowitz

67. [CLU77] **Abstraction Mechanism in CLU**
B. H. Liskov
Communications of the ACM, 20 (8), pp 564–576, agosto 1977.

68. [CLU79] **Exception Handling in CLU**
B. H. Liskov e A. Snyder
IEEE Transactions on Software Engineering, SE–5 (6), pp 546–558, novembro 1979.

69. [PL/I77] **Exception handling in PL/I**
M. Donald MacLaren
ACM Sigplan Notices, 12 (3), pp 101–104, 1977.

70. [MSR77] **Software reliability: the role of programmed exception handling**
P. M. Melliar-Smith, B. Randell
ACM Sigplan Notices, 12 (3), pp 101–104, 1977.

71. [Eiffel89] **Eiffell: The language (version 2.2)**
Bertrand Meyer
Interactive Software Egineering Inc., TR–EI–17/RM, agosto 1989.

72. [Mylo&al80] **A language facility for designing database–intensive applications**
John Mylopoulos, Philip A. Bernstein, Harry K. T. Wong
ACM TODS, 5 (2), pp 185–207, junho 1980.

73. [Pur&al90] **Message pattern specifications: a new technique for handling errors in parallel object oriented systems**
Jan A. Purchase, Russel L. Winder
Proceedings of ECOOP/OOPSLA'90, ACM Sigplan Notices, pp 116–125, outubro 1990.

74. [Fail90] **Treating failure as a value**
Limsoon Wong, B. C. Ooi
ACM Sigplan Notices, 25 (1), pp 29–32, janeiro 1990.

75. **[Fail90b]** **Failure is not just one value**
Brian Meek
ACM Sigplan Notices, 25 (8), pp 80–83, agosto 1990.

76. **[Fail90c]** **Treating failure as state**
Limsoon Wong, B. C. Ooi
ACM Sigplan Notices, 25 (8), pp 24–26, agosto 1990.