Apresentação

A disciplina de *Estruturas de Dados* (ED) está sendo ministrada em sua nova versão desde o segundo semestre de 1998. Trata-se da segunda disciplina de informática oferecida no curso de Engenharia da PUC-Rio. Na primeira disciplina, *Introdução à Ciência da Computação* (ICC), são apresentados os conceitos fundamentais de programação. ICC, em sua versão mais nova, utiliza a linguagem *Scheme*, de fácil aprendizado, o que permite a discussão de diversos conceitos de programação num curso introdutório. Isso acontece porque Scheme, como a linguagem LISP da qual descende, é uma linguagem funcional, baseada em conceitos familiares aos alunos, como a definição de funções e sua aplicação em expressões que devem ser avaliadas.

O enfoque do curso de Estruturas de Dados é diferente. Discutem-se técnicas de programação e estruturação de dados para o desenvolvimento de programas eficientes. Adota-se a linguagem de programação C. Apesar de reconhecermos as dificuldades na aprendizagem da linguagem C, optamos por sua utilização neste curso simplesmente porque C é a linguagem básica da programação do UNIX, da Internet, do Windows, do Linux. Além de C, usam-se nestes sistemas e em aplicações desenvolvidas para eles linguagens derivadas de C, como C++ e Java. Um ponto adicional a favor da escolha de C é que o estudo de várias disciplinas posteriores a ED será facilitado se os alunos já puderem programar com desenvoltura nessa linguagem.

Este curso foi idealizado e montado pelo Prof. José Lucas Rangel. Neste semestre, estamos reformulando alguns tópicos, criando outros e alterando a ordem de apresentação. Esta apostila foi reescrita tendo como base a apostila do Prof. Rangel, utilizada nos semestres anteriores.

O curso está dividido em três partes. A Parte I apresenta os conceitos fundamentais da linguagem C e discute formas simples de estruturação de dados; a Parte II discute as estruturas de listas e árvores, e suas aplicações; e a Parte III discute algoritmos e estruturas de dados para ordenação e busca.

A apostila apresenta todos os tópicos que serão discutidos em sala de aula, mas recomendamos fortemente que outras fontes (livros, notas de aula, etc.) sejam consultadas.

Rio de Janeiro, 19 de fevereiro de 2002
Waldemar Celes

1. Conceitos fundamentais

W. Celes e J. L. Rangel

1.1. Introdução

O curso de *Estruturas de Dados* discute diversas técnicas de programação, apresentando as estruturas de dados básicas utilizadas no desenvolvimento de *software*. O curso também introduz os conceitos básicos da linguagem de programação C, que é utilizada para a implementação das estruturas de dados apresentadas. A linguagem de programação C tem sido amplamente utilizada na elaboração de programas e sistemas nas diversas áreas em que a informática atua, e seu aprendizado tornou-se indispensável tanto para programadores profissionais como para programadores que atuam na área de pesquisa.

O conhecimento de linguagens de programação por si só não capacita programadores – é necessário saber usá-las de maneira eficiente. O projeto de um programa engloba a fase de identificação das propriedades dos dados e características funcionais. Uma representação adequada dos dados, tendo em vista as funcionalidades que devem ser atendidas, constitui uma etapa fundamental para a obtenção de programas eficientes e confiáveis.

A linguagem C, assim como as linguagens Fortran e Pascal, são ditas linguagens "convencionais", projetadas a partir dos elementos fundamentais da arquitetura de von Neuman, que serve como base para praticamente todos os computadores em uso. Para programar em uma linguagem convencional, precisamos de alguma maneira especificar as áreas de memória em que os dados com que queremos trabalhar estão armazenados e, freqüentemente, considerar os endereços de memória em que os dados se situam, o que faz com que o processo de programação envolva detalhes adicionais, que podem ser ignorados quando se programa em uma linguagem como Scheme. Em compensação, temos um maior controle da máquina quando utilizamos uma linguagem convencional, e podemos fazer programas melhores, ou seja, menores e mais rápidos.

A linguagem C provê as construções fundamentais de fluxo de controle necessárias para programas bem estruturados: agrupamentos de comandos; tomadas de decisão (*if-else*); laços com testes de encerramento no início (*while*, *for*) ou no fim (*do-while*); e seleção de um dentre um conjunto de possíveis casos (*switch*). C oferece ainda acesso a apontadores e a habilidade de fazer aritmética com endereços. Por outro lado, a linguagem C não provê operações para manipular diretamente objetos compostos, tais como cadeias de caracteres, nem facilidades de entrada e saída: não há comandos READ e WRITE. Todos esses mecanismos devem ser fornecidos por funções explicitamente chamadas. Embora a falta de algumas dessas facilidades possa parecer uma deficiência grave (deve-se, por exemplo, chamar uma função para comparar duas cadeias de caracteres), a manutenção da linguagem em termos modestos tem trazido benefícios reais. C é uma linguagem relativamente pequena e, no entanto, tornou-se altamente poderosa e eficiente.

1.2. Modelo de um computador

Existem diversos tipos de computadores. Embora não seja nosso objetivo estudar *hardware*, identificamos, nesta seção, os elementos essenciais de um computador. O

conhecimento da existência destes elementos nos ajudará a compreender como um programa de computador funciona.

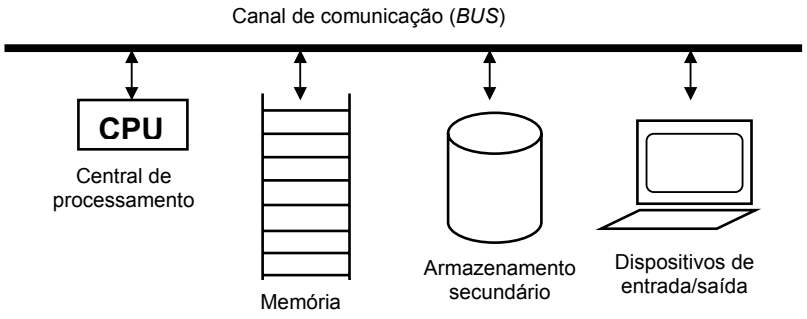

Figura 1.1: Elementos básicos de um computador típico.

A Figura 1.1 identifica os elementos básicos de um computador típico. O canal de comunicação (conhecido como *BUS*) representa o meio para a transferência de dados entre os diversos componentes. Na memória principal são armazenados os programas e os dados no computador. Ela tem acesso randômico, o que significa que podemos endereçar (isto é, acessar) diretamente qualquer posição da memória. Esta memória não é permanente e, para um programa, os dados são armazenados enquanto o programa está sendo executado. Em geral, após o término do programa, a área ocupada na memória fica disponível para ser usada por outras aplicações. A área de armazenamento secundário é, em geral, representada por um disco (disco rígido, disquete, etc.). Esta memória secundária tem a vantagem de ser permanente. Os dados armazenados em disco permanecem válidos após o término dos programas. Esta memória tem um custo mais baixo do que a memória principal, porém o acesso aos dados é bem mais lento. Por fim, encontram-se os dispositivos de entrada e saída. Os dispositivos de entrada (por exemplo, teclado, *mouse*) permitem passarmos dados para um programa, enquanto os dispositivos de saída permitem que um programa exporte seus resultados, por exemplo em forma textual ou gráfica usando monitores ou impressoras.

Armazenamento de dados e programas na memória

A memória do computador é dividida em unidades de armazenamento chamadas *bytes*. Cada byte é composto por 8 *bits*, que podem armazenar os valores zero ou um. Nada além de zeros e uns pode ser armazenado na memória do computador. Por esta razão, todas as informações (programas, textos, imagens, etc.) são armazenadas usando uma codificação numérica na forma binária. Na representação binária, os números são representados por uma seqüência de zeros e uns (no nosso dia a dia, usamos a representação decimal, uma vez que trabalhamos com 10 algarismos). Por exemplo, o número decimal 5 é representado por 101, pois $1*2^2 + 0*2^1 + 1*2^0$ é igual a 5 (da mesma forma que, na base decimal, $456=4*10^2 + 5*10^1 + 6*10^0$). Cada posição da memória (byte) tem um endereço único. Não é possível endereçar diretamente um bit.

Se só podemos armazenar números na memória do computador, como fazemos para armazenar um texto (um documento ou uma mensagem)? Para ser possível armazenar uma seqüência de caracteres, que representa o texto, atribui-se a cada caractere um código

numérico (por exemplo, pode-se associar ao caractere 'A' o código 65, ao caractere 'B' o código 66, e assim por diante). Se todos os caracteres tiverem códigos associados (inclusive os caracteres de pontuação e de formatação), podemos armazenar um texto na memória do computador como uma seqüência de códigos numéricos.

Um computador só pode executar programas em linguagens de máquina. Cada programa executável é uma seqüência de instruções que o processador central interpreta, executando as operações correspondentes. Esta seqüência de instruções também é representada como uma seqüência de códigos numéricos. Os programas ficam armazenados em disco e, para serem executados pelo computador, devem ser carregados (transferidos) para a memória principal. Uma vez na memória, o computador executa a seqüência de operações correspondente.

1.3. Interpretação versus Compilação

Uma diferença importante entre as linguagens C e Scheme é que, via de regra, elas são implementadas de forma bastante diferente. Normalmente, Scheme é interpretada e C é compilada. Para entender a diferença entre essas duas formas de implementação, é necessário lembrar que os computadores só executam realmente programas em sua linguagem de máquina, que é específica para cada modelo (ou família de modelos) de computador. Ou seja, em qualquer computador, programas em C ou em Scheme não podem ser executados em sua forma original; apenas programas na linguagem de máquina (à qual vamos nos referir como M) podem ser efetivamente executados.

No caso da interpretação de Scheme, um programa interpretador (I_M), escrito em M, lê o programa P_S escrito em Scheme e simula cada uma de suas instruções, modificando os dados do programa da forma apropriada. No caso da compilação da linguagem C, um programa compilador (C_M), escrito em M, lê o programa P_C, escrito em C, e traduz cada uma de suas instruções para M, escrevendo um programa P_M cujo efeito é o desejado. Como conseqüência deste processo, P_M, por ser um programa escrito em M, pode ser executado em qualquer máquina com a mesma linguagem de máquina M, mesmo que esta máquina não possua um compilador.

Na prática, o programa fonte e o programa objeto são armazenados em arquivos em disco, aos quais nos referimos como arquivo fonte e arquivo objeto. As duas figuras a seguir esquematizam as duas formas básicas de implementação de linguagens de programação.

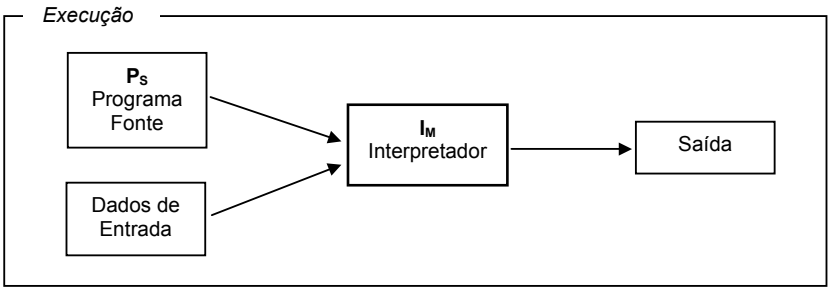

Figura 1.2: Execução de programas com linguagem interpretada.

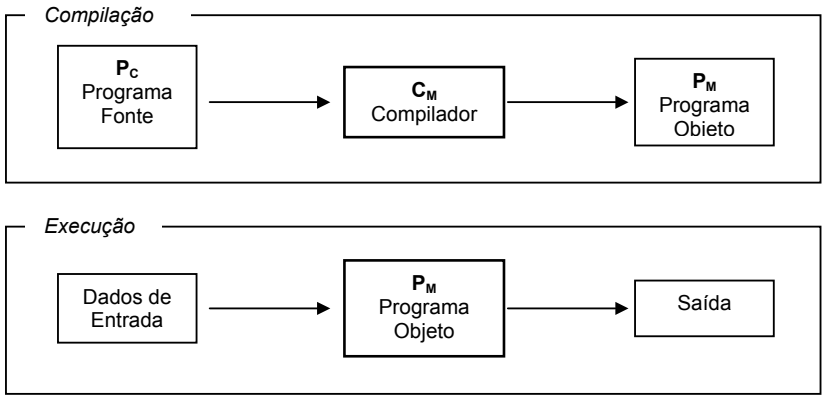

Figura 1.3: Execução de programas com linguagem compilada.

Devemos notar que, na Figura 1.2, o programa fonte é um dado de entrada a mais para o interpretador. No caso da compilação, Figura 1.3, identificamos duas fases: na primeira, o programa objeto é a saída do programa compilador e, na segunda, o programa objeto é executado, recebendo os dados de entrada e gerando a saída correspondente.

Observamos que, embora seja comum termos linguagens funcionais implementadas por interpretação e linguagens convencionais por compilação, há exceções, não existindo nenhum impedimento conceitual para implementar qualquer linguagem por qualquer dos dois métodos, ou até por uma combinação de ambos. O termo "máquina" usado acima é intencionalmente vago. Por exemplo, computadores idênticos com sistemas operacionais diferentes devem ser considerados "máquinas", ou "plataformas", diferentes. Assim, um programa em C, que foi compilado em um PC com Windows, não deverá ser executado em um PC com Linux, e vice-versa.

1.4. Exemplo de código em C

Para exemplificar códigos escritos em C, consideremos um programa que tem a finalidade de converter valores de temperatura dados em Celsius para Fahrenheit. Este programa define uma função principal que captura um valor de temperatura em Celsius, fornecido via teclado pelo usuário, e exibe como saída a temperatura correspondente em Fahrenheit. Para fazer a conversão, é utilizada uma função auxiliar. O código C deste programa exemplo é mostrado abaixo.

```
/* Programa para conversão de temperatura */

#include <stdio.h>

float converte (float c)
{
   float f;
   f = 1.8*c + 32;
   return f;
}
```

```
int main (void)
{
   float t1;
   float t2;

   /* mostra mensagem para usuario */
   printf("Digite a temperatura em Celsius: ");

   /* captura valor entrado via teclado */
   scanf("%f",&t1);

   /* faz a conversao */
   t2 = converte(t1);

   /* exibe resultado */
   printf("A temperatura em Fahrenheit é: %f\n", t2);

   return 0;
}
```

Um programa em C, em geral, é constituído de diversas pequenas funções, que são independentes entre si – não podemos, por exemplo, definir uma função dentro de outra. Dois tipos de ambientes são caracterizados em um código C. O ambiente global, externo às funções, e os ambientes locais, definidos pelas diversas funções (lembrando que os ambientes locais são independentes entre si). Podem-se inserir comentários no código fonte, iniciados com `/*` e finalizados com `*/`, conforme ilustrado acima. Devemos notar também que comandos e declarações em C são terminados pelo caractere ponto-e-vírgula (`;`).

Um programa em C tem que, obrigatoriamente, conter a função principal (`main`). A execução de um programa começa pela função principal (a função `main` é automaticamente chamada quando o programa é carregado para a memória). As funções auxiliares são chamadas, direta ou indiretamente, a partir da função principal.

Em C, como nas demais linguagens "convencionais", devemos reservar área de memória para armazenar cada dado. Isto é feito através da declaração de variáveis, na qual informamos o tipo do dado que iremos armazenar naquela posição de memória. Assim, a declaração `float t1;`, do código mostrado, reserva um espaço de memória para armazenarmos um valor real (ponto flutuante – `float`). Este espaço de memória é referenciado através do símbolo `t1`.

Uma característica fundamental da linguagem C diz respeito ao tempo de vida e à visibilidade das variáveis. Uma variável (local) declarada dentro de uma função "vive" enquanto esta função está sendo executada, e nenhuma outra função tem acesso direto a esta variável. Outra característica das variáveis locais é que devem sempre ser explicitamente inicializadas antes de seu uso, caso contrário conterão "lixo", isto é, valores indefinidos.

Como alternativa, é possível definir variáveis que sejam externas às funções, isto é, variáveis globais, que podem ser acessadas pelo nome por qualquer função subseqüente

(são "visíveis" em todas as funções que se seguem à sua definição). Além do mais, devido às variáveis externas (ou globais) existirem permanentemente (pelo menos enquanto o programa estiver sendo executado), elas retêm seus valores mesmo quando as funções que as acessam deixam de existir. Embora seja possível definir variáveis globais em qualquer parte do ambiente global (entre quaisquer funções), é prática comum defini-las no início do arquivo-fonte.

Como regra geral, por razões de clareza e estruturação adequada do código, devemos evitar o uso indisciplinado de variáveis globais e resolver os problemas fazendo uso de variáveis locais sempre que possível. No próximo capítulo, discutiremos variáveis com mais detalhe.

1.5. Compilação de programas em C

Para desenvolvermos programas em uma linguagem como C, precisamos de, no mínimo, um editor e um compilador. Estes programas têm finalidades bem definidas: com o editor de textos, escrevemos os programas fontes, que são salvos em arquivos[1]; com o compilador, transformamos os programas fontes em programas objetos, em linguagem de máquina, para poderem ser executados. Os programas fontes são, em geral, armazenados em arquivos cujo nome tem a extensão "`.c`". Os programas executáveis possuem extensões que variam com o sistema operacional: no Windows, têm extensão "`.exe`"; no Unix (Linux), em geral, não têm extensão.

Para exemplificar o ciclo de desenvolvimento de um programa simples, consideremos que o código apresentado na seção anterior tenha sido salvo num arquivo com o nome `prog.c`. Devemos então compilar o programa para gerarmos um executável. Para ilustrar este processo, usaremos o compilador `gcc`. Na linha de comando do sistema operacional, fazemos:

```
> gcc –o prog prog.c
```

Se não houver erro de compilação no nosso código, este comando gera o executável com o nome `prog` (`prog.exe`, no Windows). Podemos então executar o programa:

```
> prog
Digite a temperatura em Celsius: 10
A temperatura em Fahrenheit vale: 50.000000
>
```

Em itálico, representamos as mensagens do programa e, em negrito, exemplificamos um dado fornecido pelo usuário via teclado.

Programas com vários arquivos fontes

Os programas reais são, naturalmente, maiores. Nestes casos, subdividimos o fonte do programa em vários arquivos. Para exemplificar a criação de um programa com dois arquivos, vamos considerar que o programa para conversão de unidades de temperatura

[1] Podemos utilizar qualquer editor de texto para escrever os programas fontes, exceto editores que incluem caracteres de formatação (como o Word do Windows, por exemplo).

apresentado anteriormente seja dividido em dois fontes: o arquivo `converte.c` e o arquivo `principal.c`. Teríamos que criar dois arquivos, como ilustrado abaixo:

Arquivo `converte.c`:

```
/* Implementação do módulo de conversão */

float converte (float c)
{
   float f;
   f = 1.8*c + 32;
   return f;
}
```

Arquivo `principal.c`:

```
/* Programa para conversão de temperatura */

#include <stdio.h>

float converte (float c);

int main (void)
{
   float t1;
   float t2;

   /* mostra mensagem para usuario */
   printf("Entre com temperatura em Celsius: ");

   /* captura valor entrado via teclado */
   scanf("%f",&t1);

   /* faz a conversao */
   t2 = converte(t1);

   /* exibe resultado */
   printf("A temperatura em Fahrenheit vale: %f\n", t2);

   return 0;
}
```

Embora o entendimento completo desta organização de código não fique claro agora, interessa-nos apenas mostrar como geramos um executável de um programa com vários arquivos fontes. Uma alternativa é compilar tudo junto e gerar o executável como anteriormente:

```
> gcc –o prog converte.c principal.c
```

No entanto, esta não é a melhor estratégia, pois se alterarmos a implementação de um determinado módulo não precisaríamos re-compilar os outros. Uma forma mais eficiente é compilarmos os módulos separadamente e depois ligar os diversos módulos objetos gerados para criar um executável.

```
> gcc –c converte.c
> gcc –c principal.c
> gcc –o prog converte.o principal.o
```

A opção `-c` do compilador `gcc` indica que não queremos criar um executável, apenas gerar o arquivo objeto (com extensão "`.o`" ou "`.obj`"). Depois, invocamos `gcc` para fazer a ligação dos objetos, gerando o executável.

1.6. Ciclo de desenvolvimento

Programas como editores, compiladores e ligadores são às vezes chamados de "ferramentas", usadas na "Engenharia" de Software. Exceto no caso de programas muito pequenos (como é o caso de nosso exemplo), é raro que um programa seja composto de um único arquivo fonte. Normalmente, para facilitar o projeto, os programas são divididos em vários arquivos. Como vimos, cada um desses arquivos pode ser compilado em separado, mas para sua execução é necessário reunir os códigos de todos eles, sem esquecer das bibliotecas necessárias, e esta é a função do *ligador*.

A tarefa das bibliotecas é permitir que funções de interesse geral estejam disponíveis com facilidade. Nosso exemplo usa a biblioteca de entrada/saída padrão do C, `stdio`, que oferece funções que permitem a captura de dados a partir do teclado e a saída de dados para a tela. Além de bibliotecas preparadas pelo fornecedor do compilador, ou por outros fornecedores de *software*, podemos ter bibliotecas preparadas por um usuário qualquer, que pode "empacotar" funções com utilidades relacionadas em uma biblioteca e, dessa maneira, facilitar seu uso em outros programas.

Em alguns casos, a função do ligador é executada pelo próprio compilador. Por exemplo, quando compilamos o primeiro programa `prog.c`, o ligador foi chamado automaticamente para reunir o código do programa aos códigos de `scanf`, `printf` e de outras funções necessárias à execução independente do programa.

Verificação e Validação

Outro ponto que deve ser observado é que os programas podem conter (e, em geral, contêm) erros, que precisam ser identificados e corrigidos. Quase sempre a verificação é realizada por meio de testes, executando o programa a ser testado com diferentes valores de entrada. Identificado um ou mais erros, o código fonte é corrigido e deve ser novamente verificado. O processo de compilação, ligação e teste se repete até que os resultados dos testes sejam satisfatórios e o programa seja considerado validado. Podemos descrever o ciclo através da Figura 1.4.

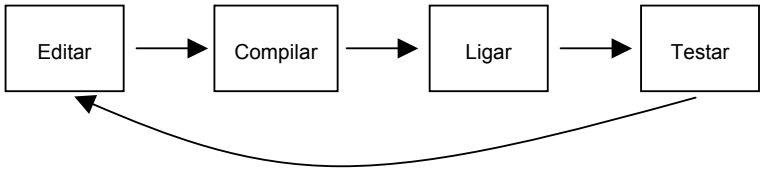

Figura 1.4: Ciclo de desenvolvimento.

Este ciclo pode ser realizado usando programas (editor, compilador, ligador) separados ou empregando um "ambiente integrado de desenvolvimento" (*integrated development environment*, ou IDE). IDE é um programa que oferece janelas para a edição de programas e facilidades para abrir, fechar e salvar arquivos e para compilar, ligar e executar programas. Se um IDE estiver disponível, é possível criar e testar um programa, tudo em um mesmo ambiente, e todo o ciclo mencionado acima acontece de maneira mais confortável dentro de um mesmo ambiente, de preferência com uma interface amigável.