C: "A" Linguagem de Programação

[Versão Integral]

Publicado originalmente em OLinux - http://www.olinux.com.br

Créditos:

- Elias Bareinboim - [eliasb@olinux.com.br] – Aulas: 1-9, 13, 17, 19, 20, e 25;
- André Duarte de Souza - [asouza@olinux.com.br] – Aulas: 10-12, 14, 22, 40-42;
- Paulo Henrique Baptista de Oliveira - [baptista@olinux.com.br] – Aula: 15;
- Diego Lages - [lages@olinux.com.br] – Aula: 16;
- Rodrigo Hausen - [cuco@olinux.com.br] – Aulas: 18, 21, 23, 24, 26, 28-31, e 37;
- Rodrigo Fonseca - [rodrigo@olinux.com.br] – Aula: 27;
- Gerson Garcia dos Santos – [gersongs@olinux.com.br] – Aulas: 38 e 39.

Edição:

- Davi Tavares Ferreira - davi@olinux.com.br
- Cristian Privat – csprivat@ieg.com.br

sendo LISTADOS."

Aula 1

Introdução:

Estamos começando uma seção de programação do nosso site e daqui para frente regularmente serão colocados artigos e tutoriais de temas relacionados a programação para o ambiente Linux (UNIX) neste espaço.

Com o intuito de uma introdução teórica a tal ambiente, utilizaremos a linguagem C, a linguagem de programação mais utilizada no mundo. Vários de nossos artigos tem base em outros artigos/sites (a maioria do exterior, senão todos) como fonte de pesquisa. Procure por estes na área de links, sempre estarão disponíveis e serão muito úteis.

Não espante-se com alguns termos e vários conceitos novos, o ato de programar é mais fácil do que parece, não se assuste, com um pouco de leitura e tempo para praticar você pode tornar-se um bom programador C.

"Programar é o ato de pensar utilizando-se algumas padronizações (pré-definições). Aqui tentaremos direcionar como você deve canalizar seus pensamentos para resolver os problemas propostos e lhes colocaremos a frente das padronizações da linguagem C".

Programar é só isso, resolver problemas da vida real de forma "entendível" ao computador. Mostrar-lhe como você fez em uma linguagem que ele entenda.

Um bom programador C, o que você tende a ser tornar, é algo que está em falta no mercado. Se tem a idéia que o mercado de programação está saturado e tem programadores saindo pelas janelas. Não é verdade. O que não falta por aí na verdade são programadores de "linguagens" visuais e programas populares de propósito superficial e de rápida implementação que não servem para o "trabalho pesado", está área sim, está saturada.

Irremediavelmente, quando uma empresa precisa fazer um programa "pesado" (algo robusto e flexível ao mesmo tempo, bem modelado e que utilize toda potencialidade possível de sua máquina) ela tem que utilizar o C, e a falta programadores desta área é notória. É uma mão de obra muito escassa e que cada vez mais ganha destaque e valor no mercado.

Nós próximos artigos também ensinaremos programação visual, front-end gráfico que faz o programa parecer mais amigável, mas isso é casca, perfumaria ... na verdade não importa muito. O importante realmente são os conceitos, é o que está acontecendo por trás, como as coisas funcionam e justamente isso que será nosso enfoque.

Histórico do C:

Em 1973, Dennis Ritchie, pesquisador da Bell Labs rescreveu todo sistema UNIX para uma linguagem de alto nível (na época considerada) chamada C (desenvolvida por ele), para um PDP-11 (o microcomputador mais popular na época). Tal

situação de se ter um sistema escrito em linguagem de alto nível foi ímpar e pode ter sido um dos motivos da aceitação do sistema por parte dos usuários externos a Bell e sua popularização tem relação direta com o exponencial uso do C.

Por vários anos o padrão utilizado para o C foi o que era fornecido com o UNIX versão 5 (descrito em The C programming Language, de Brian Kernighan e Dennis Ritchie - 1978. Se você está interessado em saber mais, vá para a introdução). Começaram a surgir diversos diversas implementações da tal linguagem e os códigos gerados por tais implementações era altamente incompatíveis. Não existia nada que formalizasse essas compatibilizações e com o aumento do uso das diversas "vertentes" da linguagem surgiu a necessidade de uma padronização, algo que todos devessem seguir para poderem rodar seus programas em todos os lugares que quisessem.

O ANSI (American National Standards Intitute, Instituto Americano que até hoje dita diversos padrões) estabeleceu em 1983 um comitê responsável pela padronização da linguagem. Atualmente, a grande maioria dos compiladores já suportam essa padronização.

Trocando em miúdos, o C pode ser escrito em qualquer máquina que se utilize de tal padrão e rodar em qualquer outra que também o faça. Parece inútil? Não, na verdade isso é a semente de grande parte dos avanços tecnológicos que toda programação nos tem proporcionado no mundo de hoje. Com o tempo isso ficará mais claro.

Características da Linguagem C:

1. C é uma linguagem que alia características de linguagens de alto nível (como pascal, basic) e outras de baixo nível como assembly. O que isso quer dizer? Que C junta flexibilidade, praticidade e poder de manipulação da máquina diretamente e, por isso, não traz as limitações dessas linguagens, como dificuldade de uso, limitações na operação, etc. C permite liberdade total ao programador e este é responsável por tudo que acontece, nada é imposto ou acontece simplesmente ao acaso, tudo é pensado pelo programador e isso significa um bom controle e objetividade em suas tarefas, o que não é conseguido em outras linguagens que pecam em alguma coisa.

   C é uma linguagem que podemos chamar de estruturada. Não queremos nos ater muito a nomenclaturas, mas ling. estruturadas são linguagens que estruturam o programa em blocos para resolver os problemas. A filosofia básica de uma linguagem estruturada é dividir para trabalhar, você divide um problema em pequenas partes que sejam possíveis de serem feitas. Por exemplo, temos um problema grande que é impossível de se resolver sozinho, dividimos este problema em pequenas partes simples que se integrando os pedacinhos acabam resolvendo o problema maior complexo.

2. Outra coisa atrelada a essa idéia básica de dividir para trabalhar é que as divisões sejam feitas de modo que os módulos sejam independentes ao máximo do contexto onde se está inserido.

O que isso quer dizer? Que os módulos tem que ter um caráter geral, independente, para que futuramente possa ser encaixado em outro problema que possa necessitar da tarefa que esse módulo realiza. Você precisa apenas saber o que a rotina faz e não como ela faz, isso pode ser útil para compartilhar código. Imagine que daqui a 1 ano, você tenha diversas funções desenvolvidas e deseje criar um programa com algumas similaridades do que você já criou. Simples, basta pegar as funções que você nem lembra mais como funcionam mas sabe o que elas fazem e encaixar no lugar certo do seu novo programa, economizará muito código e tempo.

Vamos a um exemplo, tem-se um problema de matemática de arranjo ou combinação (probabilidade? matemática combinatória?) Também não sou muito bom nisso:) e precisa-se calcular a fatorial de determinado número. Uma vez feita a função de fatorial, não importa como ela trabalha. O que me interessa é só passar como para metro para esta

função o número que eu quero ter o seu fatorial calculado e receber o resultado pronto. Você só se preocupou uma vez em saber como funciona tal função para criá-la.

Outra situação, imagine você trabalhando em um grande projeto e seu trabalho ser parte de um problema maior. Você sabe o que tem, o que é recebido pelo seu módulo, o que ele deve trabalhar e qual saída deve gerar. Só com essas informações deve-se fazer o trabalho e futuramente entregar a alguém que não sabe como ele funciona e sim apenas as especificações pedidas. Ele simplesmente fará a inserção no contexto do programa. Note, nem você sabe o que o programa todo faz ou como funciona. Nem o responsável por juntar sabe como o seu programa funciona por dentro e sim apenas o que ele faz e onde deve ser inserido. Isso é bastante comum para quem trabalha com programação.

Compilada ou Interpretada:

C é uma linguagem compilada, utiliza de um compilador C para ser executado. Ao contrário de outras linguagens que utilizam de um interpretador para tal. Na concepção da linguagem é que se decide se ela vai ser compilada ou interpretada,

pois é um detalhe de implementação (que não deixa de ser importante). A priori qualquer uma poderia ser interpretada ou compilada.

Na verdade, quem faz um programa ser executado é também um programa, só que um programa avançado que lê todo código fonte (o que foi escrito pelo programador) e o traduz de alguma forma para ser executado, isso acontece em todas linguagens.

A diferença básica é que um interpretador lê linha a linha do fonte, o examina sintaticamente e o executa. Cada vez que o programa for executado esse processo tem de ser repetido e o interpretador é chamado. Já um compilador, lê todo programa e o converte para código-objeto (código de máquina, binário, 0's e 1's) e pronto. Sempre quando tiver que ser executado é só chamá-lo, todas instruções já estão prontas para tal, não tem mais vínculo com seu código-fonte.

Bem pessoal, muitos conceitos novos já foram dados por hoje, tempo para reflexão !:)

Espero que tenham gostado. Aguardo seus comentários e sua participação nos nossos fóruns!

Aula 2

O compilador:

Terminamos a coluna da semana passada falando de compiladores, espero que tenham gostado.

Façamos uma revisão rápida:

Usar um compilador, uma linguagem *compilada* não é o mesmo que usar uma linguagem interpretada, como Perl. Existem diferenças na forma de execução. Porém, a implementação é opção de quem criou a linguagem.

Um programa em C é elaborado em dois passos básicos:

- O programa é escrito em texto puro num editor de texto simples. Tal programa se chama "código fonte" (source code em inglês).
- Passamos o código fonte para o compilador que é o programa que gera um arquivo num formato que a máquina entenda.

O compilador trabalha, em geral, em duas fases básicas:

1. O compilador gera, através do código fonte, um código intermediário que ajuda a simplificar a "gramática" da linguagem. Como o processamento subsequente agirá: um arquivo "objeto" será criado para cada código fonte.
2. Depois, o linkador junta o arquivo objeto com a biblioteca padrão. A tarefa de juntar todas as funções do programa é bastante complexa. Nesse estágio, o compilador pode falhar se ele não encontrar referências para a função.

Para evitar a digitação de 2 ou 3 comandos você pode tentar:

$ gcc fonte.c

ou

$ cc fonte.c

que evitará esse inconveniente.

Na maioria dos computadores, isso gerará um arquivo chamado "a.out", para evitar isso você pode utilizar a opção -o, assim fica:

gcc -o destino fonte.c

O seu programa se chamará destino e será o derivado do fonte chamado fonte.c.

Erros:

Erros são provocados sempre pelo programador. Existem basicamente dois tipos de erros:

- Lógico
- Sintaxe

Os erros de sintaxe são os melhores que podem acontecer. Você é capaz de identificá-lo durante a compilação do programa. Ou seja, não existe um efeito desagradável na execução do programa, ele não pode fazer algo que seja impossível ou não determinado previamente na linguagem (==gramática).

Em geral, quando os compiladores encontram um erro não terminam imediatamente, mas continuam procurando e ao final do programa mostram **indicativamente** onde os erros se encontram e do que se tratam. É a mesma coisa que uma correção ortográfica, você nunca pode falar nada errado ortograficamente nem gramaticalmente, o compilador não deixa.

É bastante traumatizante para alguém, fazendo seu primeiro programa, obter um erro e ver diversas linhas de erro e avisos "estranhos". Não se assuste nem se desmotive, pode ser um simples ponto e vírgula ou um detalhe bobo. Com o tempo e a experiência você começará a se acostumar e aprender a lidar com isso.

Caso não tenha erros de escrita, o compilador transformará o seu código fonte (texto puro) em código de maquina (os tais 0's e 1's, o que a máquina entende) e você poderá executa-lo. Bem, e se a lógica do programa estiver errada? Este tipo de erro não pode ser detectado pelo compilador.

É como se você falasse todas as coisas certas, desse todas as ordens entedíveis para um empregado seu por exemplo porem com inconsistência de dados.

Exemplificando hipoteticamente:

1. Você pode falar com seu empregado e ele não entender o que você está expressando e assim ele não conseguirá executar a tarefa e reclamará. Por exemplo, falando em **japonês** com ele! (erro sintático)

2. Você explica tudo certo para o seu empregado, ele entende tudo porém a explicação é inconsistente. Por exemplo, manda ele ir para uma rua que não existe ou comprar algo que não existe (erro de lógica).

Tais erros podem acarretar algumas conseqüências graves:

- Que o programa termine repentinamente e às vezes avise alguma coisa
- Que o programa funcione incorretamente e gere dados inconsistentes

Detalhes Importantes:

Alguns detalhes interessantes a frisar para você que está começando a programar, por exemplo, caso você tenha algum erro e conserte-o você é *obrigado* a compilar o programa novamente para que este tenha efeito, caso contrário o executável não reflitirá o atual código fonte, essa conversão *não é automática*.

Outro detalhe importante, o C, como tudo no mundo Linux/UNIX, difere letras maiúsculas e minúsculas (case sensitive em inglês). Ou seja, você tem que obedecer esse critério e atentar-se a isso. O comum é colocar tudo em minúsculas, com exceção das strings que ficam a gosto do programador.

Outra coisa importante, o C (como a maioria das linguagens atuais) exige que se faça uma listagem de todas as variáveis do programa previamente. Ou seja, não existe uso dinâmico de variáveis e tudo que você usa tem que ser previamente declarado.

É como na construção de um prédio, todo material (tijolos, cimento, madeira) tem que ser calculado e comprado antes pois na hora que você precisa usar um tijolo, você não fala, quero um tijolo e ele aparece na sua mão.

Exercício I:

Bem, algumas perguntas básicas que você deveria responder depois a primeira e segunda coluna sobre C Básico.

1. A linguagem C é alto ou baixo nível? Qual a relação de C com outras linguagens de programação nesses termos?

2. Qual a filosofia da linguagem estruturada? Qual a vantagem? Exemplifique.

3. A linguagem C é compilada ou interpretada? Qual diferença? Qual é melhor?

4. O que é um compilador?

5. Como um programa C roda?

6. Como se compila um programa em C?

7. As letras maiúsculas e minúsculas são diferenciadas?

8. Quais os tipos de erros que existem em C e qual é o preferível ?

Bem, por enquanto é só. Até a próxima semana.

Aula 3

Palavras Reservadas:

Fala moçada! O que acharam da semana passada? Não se preocupem se não entenderam alguns detalhes, e' só perguntar. Muitas das questões tratadas nas colunas passadas são teóricas demais e demora um tempo para se acostumar. A programação em si é mais fácil do que parece. Nesta coluna, trataremos de algumas coisas interessantes, entre elas as palavras reservadas do C.

O que vem a ser palavras reservadas? São palavras que têm significado especial na linguagem. Cada palavra quer dizer alguma coisa e em C as instruções são executadas através do uso de palavras chaves previamente determinadas que chamaremos de palavras reservadas. A priori, elas não poderão ser usadas para outras coisas, alem do determinado para o C, mas entenderemos mais a fundo o que isso quer dizer com o tempo.

Para o C conseguir trabalhar ficaria complicado distinguir quando uma dessas palavras especiais forem usadas como variáveis ou com instruções. Por isso, foi determinado a não utilização desta para outra coisa senão comando e afirmativas.

Abaixo está a lista dessas palavras. Relembrando, o C entende, tais palavras **apenas** em letras minúsculas (não funcionará se você colocar em maiúsculas).

auto, break, case, char, continue, default, do, double, else, entry, extern, float, for, goto, if, int, long, register, return, short, sizeof, static, struct, switch, typedef, union, unsigned, while, enum, void, const, signed, volatile.

Todo conjunto de palavras reservadas acima são o conjunto das instruções básicas do C. Aparentemente, parecem poucas e você na prática usará apenas algumas poucas delas. Tal fato acontece pois uma das facilidades do C é a utilização muito natural de bibliotecas que funcionam como acessórios para o C (como plugins do seu netscape). Tais bibliotecas (conjunto de funções) não fazem parte intrínseca do C, mas você não encontrará nenhuma versão do C sem nenhuma delas. Algumas são até tratadas como parte da linguagem por serem padronizadas.

Bibliotecas:

Como dito anteriormente, funções são uma forma genérica de resolvermos problemas. É como uma caixa preta, você passa os dados para ela e recebe o resultado.

Supondo que tenho uma função de realizar soma, eu só me preocupo em passar para ela os números que desejo ver somado e a função se preocupa em me entregar o resultado, o que acontece lá dentro é problema dela.

Através deste método, dividimos os programas em pedaços de funcionalidades, genéricos e pequenos de preferência, com intuito de utiliza-lo futuramente em situações que sejam convenientes.

Assim como soma, pode-se fazer uma função de subtração, multiplicação, divisão e várias outras e juntando-as se cria a tal famosa biblioteca.

As bibliotecas em si podem ser utilizadas por vários programas. Estes utilizam os subprogramas(funções!) desta mesma biblioteca tornando o programa gerado de menor tamanho, ou seja, o executável só vai possuir as chamadas das funções da biblioteca e não o incluirá no executável.

Só para esclarecer, tenho uma biblioteca que desenha botões em janelas(GTK faz isso). Na hora que se for criar uma agenda, por exemplo, utilizo as funções desta biblioteca sem precisar rescrever estas mesmas funções neste programa.

Em geral, utilizamos algumas funções já prontas para fazer determinadas tarefas que são consideradas básicas. O

programador não costuma fazer uma rotina que leia diretamente do teclado ou imprima na tela um caracter.

Isso já existe e é bem implementado (uma coisa interessante de se entender em programação é, o que já existe de bem feito e pode ser utilizado deve ser utilizado). Seu sistema não será menos digno ou pior se você utilizar uma rotina que todo mundo utiliza em vez de ter a sua própria. O que importa e a finalidade do programa e o quão bem implementado ele esteja.

Tais funções que falamos, básicas, fazem parte da biblioteca C padrão (citada acima). Todo compilador C a possui, ele faz parte da padronização ANSI C. Seu compilador, independente do sistema que você utiliza, deve possuir. Outras bibliotecas a mais, além das padronizadas pelo ANSI, também vem junto com seu compilador porém não é recomendado a utilização, caso você queira escrever programas portáveis (que rode em todas as plataformas). Podemos aqui citar programação gráfica, de rede, etc como casos que são "perigosos" para programação portável. Não estou dizendo que você não deve programar para estas áreas, futuramente haverão tutoriais para essas áreas por aqui, porém deve atentar-se que tal programação é peculiar a plataforma que você está utilizando e fazer algo padronizado portável torna-se quase impossível.

Printf():

Se desejamos citar uma função *invariável* e já consagrada que não é da linguagem C, porém já pode até ser considerada é a função printf(). Ela está contida na biblioteca padrão de entrada/saída (a barra eqüivale a palavra e tal biblioteca se chama stdio.h, o .h eqüivale a terminação de biblioteca, header em inglês).

A função printf quer dizer print-formated, ou imprimir formatado. A maneira mais simples de imprimir algo é:

printf("algum texto aqui!");

Bem, comecemos então. Caso você não queira imprimir uma coisa fixa, um texto, mas sim algo que varie durante a execução de um programa, digamos uma variável. Chamemos de controladores de seqüência os caracteres especiais que significarão as variáveis que serão impressas pela função. O lugar onde o controlador for colocado é o lugar onde a variável será impressa. Por exemplo, caso queiramos imprimir um inteiro.

printf ("O inteiro vale: %d, legal!", algum_inteiro);

A saída será:

O inteiro vale: 10, legal!

onde 10 é o valor dado a variável chamada "algum_inteiro" (sem aspas)

Exemplo básico de um programa em C:

```
/*************************************************************/

/* Primeiro exemplo */


/*************************************************************/

#include <stdio.h> /* Aqui incluímos a biblioteca */

/* C padrão de Entrada/Saída */

/*************************************************************/

main () /* Comentários em C ficam nesse formato! */
```

```
{

printf ("Bem, é nosso exemplo número %d em C !", 1);

printf ("Bacana, que desse exemplo número %d surja o %d .. \n", 1, 1+1);

printf ("E depois o %d ! \n", 3);

printf (" A criatividade tá em baixa e venceu de %d X %d o autor !", 1000,  0);

}
```

Saída:

```
 Bem, é nosso exemplo número 1 em C !
Bacana, que desse exemplo número 1 surja o 2 ..
E depois o 3 !
A criatividade tá em baixa e venceu de 1000 X 0 o autor !
```

Bem pessoal, último detalhe, usem o fórum de programação, tem muita gente boa lendo. Vale a pena, sua dúvida pode ser de outra pessoa, uma vai ajudando a outra e cria-se uma grande comunidade, a interação é essencial !

Por hoje é só.

Aula 4

Comparação entre Linguagens:

Fala galera! Tranqüilo? Bem, hoje faremos algo bem interessante para os programadores que não estão familiarizados com a linguagem C, porém já são programadores de outras linguagens.

Se você nunca programou você pode pular esta parte pois não será de grande utilidade para você.

Bem, mostraremos uma tabela comparando as mais populares linguagens: o C, nosso alvo, o Pascal, a linguagem mais didática que eu conheço, que 9 entre 10 alunos de programação há 2 anos atrás aprendiam e o Basic, como o Pascal, mas num passado um pouco mais remoto. Tanto Pascal como Basic foram muito úteis e são legais de se programar porém tem suas limitações e sejamos sinceros, já passaram do seu tempo.

C	Pascal	BASIC
=	:=	=
==	=	=
*,/	*,/	*,/

/,%	div,mod	DIV,MOD
printf ("...");	writeln('...');	PRINT"..."
scanf ("...", &a)	readln(a);	INPUT a
for (x = ..;...;)	for x := ...to	FOR x = ...
{ }	begin end	NEXT x
while(...) { }	while ... do begin end	N/D
do { while(...)	N/D	N/D
N/D	repeat until (...)	REPEAT UNTIL ...
if (...) else	if ... then ... else	IF .. THEN ... ELSE
switch(...) { case: ... }	case ... of end;	N/D
/* ... */	{ ... }	REM ...
*	^	?!$
union	N/D	N/D
struct	record	N/D

O N/D quer dizer Não Disponível.

Repare, as expressões de loop do C são mais robustas do que nas outras linguagens. Além disso, a entrada e saída do C é mais flexível. E, em relação ao Basic, o C pode executar todas operações com strings e várias outras.

Exercícios II:

Vamos lá, de novo, algumas perguntas para você testar seus conhecimentos. Caso você acerte menos de 1, releia os artigos anteriores:

Escreva um comando para imprimir "Olá mundo!" Escreva um comando para imprimir o numero 101 de três formas diferentes. Porque existem tão poucas palavras reservadas no C ? O que é uma função? Cite alguma. O que é uma biblioteca? Cite alguma e sua função. Caso você seja programador em Basic e/ou Pascal, trace suas perspectivas em relação ao C citando seus diferenciais.

Esqueci de falar, essa aula foi mais relax com objetivo de vocês fazerem uma revisão de tudo que aconteceu até aqui. Caso não estejam entendendo, voltem, releiam e realmente entendam. Quem entendeu tudo, descanse essa semana. Semana que vem voltaremos ao normal.

Ok. Por hoje é isso. Qualquer problema, continuem utilizando nossos fóruns. Comentários, sugestões e observações mandem-nos e-mail!

Aula 5

Introdução

Como foram de descanso? Semana passada realmente estava meio mole, e resolvi aumentar o nível esta semana. Vamos ver a opinião de vocês no final, vou adiantando que hoje muita coisa é referência e de cultura geral, mas tem muitos conceitos novos e necessários.

Falamos de alguns conceitos de Sistemas Operacionais, características poderosas herdadas dos velhos e robustos sistemas Unix e várias outras coisas.

Caso tenham problemas para entender de primeira, releiam o texto, é sempre o recomendado. Muitos não fazem isso, me mandam e-mail, eu aconselho e tudo termina bem !:)

Vale ressaltar que dúvidas são comuns e sempre nos depararemos com elas, mas transpassaremos e chegaremos ao nosso objetivo.

Vamos lá ..

Quem realmente controla os programas?

Quem controla as ações mais básicas de um computador, é o Sistema Operacional. É o que podemos chamar de camada de software, que faz a interface (comunicação) entre os usuários e o Hardware(parte física da máquina, placas, circuitos, memórias). O objetivo básico é controlar as atividades do Hardware e prover um ambiente agradável para o usuário do sistema, que ele possa trabalhar com maior grau de abstração(se preocupar menos com problemas relativos ao tipo de máquina, como ela funcione e pensar mais no problema real que tem de ser resolvido).

O Sistema Operacional(SO), tem dois componentes básicos importantes para o usuário : sistema de arquivos e interpretador de comandos. O SO é que coordena todo tráfego de saída e entrada de informações. Por exemplo, ele que determina o que vai para a tela, o que vem do teclado, movimentos do mouse, etc...

O Interpretador de comandos aceita toda interação com o usuário, e faz todo trabalho de gerenciamento, analisando o que vem do usuário, por exemplo, e entregando para o SO de forma que ele entenda. Ele é uma interface, digamos assim, entre usuário e SO. Faz com que eles falem a mesma língua, para simplificar a história.

O interpretador de comandos é conhecido no mundo Linux como SHELL (pode variar para outros sistemas).

Interfaces Gráficas

Atualmente, como cheguei a citar em artigos anteriores, existe uma tendência da programação Gráfica, baseada em Interfaces(GUI, Graphical User Interface), só que não considero das melhores ênfases que se deve dar a programação. Não que seja totalmente inútil, realmente há situações que ela se torna necessária. Para alguns sistemas como para

supermercados, você precisa fazer um programa core (explicação abaixo), e dar para as atendentes um ambiente agradável de programação, e de rápido e fácil aprendizado. Vale ressaltar que abordaremos programação gráfica em próximas colunas, agradando gregos e troianos.

Uma questão interessante é separarmos os termos, Back-end e Front-end. Back-end, é a parte core(a parte interna, o código que executa realmente as tarefas). A parte gráfica é conhecida como Front-end, é uma capa, como se fosse a casca, embalagem, a apresentação do trabalho. Não há dúvidas, que situações existam que essa "cara bonita" necessariamente deve aparecer.

As linguagens de programação e o C

Quando várias das linguagens de programação antigas foram desenvolvidos, elas foram criados para rodar em computadores de grande porte(grandes máquinas de tamanhos de prédios, depois andares, depois salas...), Sistemas Multi-Usuários(vários operadores usando o sistema ao mesmo tempo), time-sharing((tempo compartilhado), todos esses usuários dividiam uma fração(parte) da unidade de processamento) e eram incapazes de interagir diretamente com o usuário(vários detalhes técnicos são citados meramente como curiosidades para aprendizado histórico, não se preocupe caso tenha fugido algo).

O C é diferente de tudo isso pois possui um modelo de implementação de forma bem definida, e melhor desenhado. Podemos dizer que o C é um primor em termos de design. Todos mecanismos de entrada/saída de dados não são definidos como partes fixas da linguagem, imutáveis e embutidos no código. Existe um arquivo padrão, que deve ser declarado(explicaremos, mas podemos entender como chamado, utilizado) no começo do programa, que ensina ao C como ele deve se comportar em situações de entrada/saída, no computador que ele está rodando.

Quando mudamos de arquitetura, é só mudar este arquivo. Ele é conhecido como Biblioteca C padrão(ainda explicaremos melhor isso tudo). Ela é padrão no sentindo de implementar(ter para executar) um conjunto de instruções que todos os computadores e SO's devem implementar, mas são especialmente adaptados ao seu sistema atual. Basicamente, ela existe em todos os tipos de sistemas, funciona de forma padrão, mas, é internamente escrita para a arquitetura de computador que você está utilizando(arquitetura == tipo). Lembre-se, não se vive simplesmente da arquitetura Intel, i386. Existem diversas outras por aí, uma característica marcante do C é sua portabilidade(capacidade de rodar em múltiplas plataformas).

Dispositivos de Entrada/Saída

O Sistema de Arquivos do Linux pode ser considerado parte ativa do Sistema de Entrada/Saída, fluxo de dados, em sistemas Unix-Like(Linux também). Em vários SO's, ele próprio coordena os recursos/movimentação entre entrada/saída, isso tudo através de arquivos ou fluxo de dados.

O C faz isso tudo implícito (herdado dos Unix). O arquivo de onde o C geralmente pega a entrada, chama-se *stdin*, que quer dizer Standard Input(ou entrada padrão). Ele geralmente costuma ser o Teclado. O arquivo correspondente para saída, é conhecido como *stdout*, ou Standard Output (saída padrão). O monitor de vídeo faz esse papel.

É importante deixar claro, que o monitor e nem o teclado, são arquivos. Eles são periféricos reais(não é possível reler algo que foi enviado ao monitor ou rescrever para o teclado).

Devemos representa-los através de arquivos, mas devemos entender, que o arquivo de entrada padrão, teclado, serve apenas para leitura. Já o monitor, de saída padrão, simplesmente trabalha com escrita. Nota-se, características de escrita e leitura bem definidos.

Porque utilizá-los ?

Uma boa explicação para utilizarmos os tais arquivos é nos abstermos de como funcionam a fundo, é realmente essa "camada" de abstração. Pensarmos simplesmente em utiliza-los (tudo ficará claro com o tempo). Com isso criamos essa camada que pode ser utilizado com diversos outros tipos de periféricos, de forma igualmente transparente e fácil.

Os nomes do arquivos de dispositivos, em geral são conhecidos por abreviaturas de seus nomes originais. A impressora, por exemplo, é conhecida como LP, ou Line Printer.

Bem pessoal, por hoje é só. O que acharam ? Espero que tenham gostado, aguardo o feedback e muitas mensagens nos fóruns.

Aula 6

Só aviso aos navegantes, essa aula vai ser um pouco grandinha mas semana que vem piora! Estou brincando, vamos lá que tem muito o que falar .. fiquem tranqüilos, tudo é normal e o aprendizado sempre faz parte de uma fase, depois as coisas vão se encaixando e fica tudo mais tranqüilo, todos conseguirão superar.

Extensões de arquivo

Existe uma convenção usada pelos compiladores para denominarmos os programas escritos em C e todos elementos que se referem ao conjunto programa. As denominações encontradas são:

- O fonte do programa: nome_do_programa.c
- Arquivo Objeto: nome_do_programa.o
- Arquivo no formato executável: nome_do_programa ou nome_do_programa.exe (não Linux)
- Arquivo de cabeçalho: Nome_da_Biblioteca.h
- Biblioteca: libNome_da_Biblioteca.so libNome_da_Biblioteca.a

O final do arquivo após o ponto (c,o,a,so ou nada) identifica o tipo de arquivo para o compilador. Ele usa isso para gerar o executável. Os arquivos objeto e executável podem ser deletados sem problemas, são fáceis de serem gerados. Já o .c é o fonte do programa, ele que contém o código real do que o programa fará. Para executar o compilar, você deve:

cc -o nome_do_executável nome_do_programa.c
Que chama o CC, ou, C Compiler, em geral o GCC, compilador C da GNU.

Detalhes que fazem a diferença na hora de arrumar o sistema

Um detalhe interessante de se saber é como parar a execução de um programa. Todo sistema tem sua forma particular de fazer isso. Em geral, devemos pressionar Ctrl (tecla Control) junto com C, ou junto com Z, ou mesmo D.

Outra, o comando *ls* é equivalente ao *DIR* do DOS e permite listar os arquivos do diretório atual. O comando cd, change dir, também é equivalente a outros sistemas, serve para trocar de diretório.

Pergunta bastante comum é quanto aos ambientes de desenvolvimento para se fazer um programa em C. Um editor recomendável e muito simpático chama-se VIM (veio do famosíssimo VI). Eu acho ele bastante prático, tem indicação de comandos e se integra bastante confortavelmente com o sistema. Existem outros, como o emacs (muito adoram) e alguns mais gráficos como o rhide , que parece com o Turbo C, bem prático. Outro que tem sido bem comentando, é para X,

chama-se Code Crusader, de uma checada na CNET, tem sido um dos mais baixados por lá. Vale reforçar o XWPE, também muito bom e que foi ao ar em nosso artigo de programas de algumas semanas atrás.

Exercícios

Bem, alguns exercícios antes de prosseguirmos, para analisarmos como estamos. Até agora parece que a margem de acertos e entendimento está relativamente alta.

1) Para que serve um Sistema Operacional? Quais suas funcionalidades?
2) Explique o mecanismo de direcionamento de saída/entrada? Faça uma relação e cite exemplos.
3) Se você tiver um programa que quer chamar de "meuprimeiroteste", sem aspas, como você deveria chama-lo? Quais

seriam os arquivos gerados pelo compilador, após utiliza-lo?
4) Como você deveriam rodar esse programa?

Aula 7

Continuando..

O arquivo de cabeçalho mais comum existente hoje é o padrão de entrada/saída, da biblioteca C padrão.

Para incluir um cabeçalho, adicione ao código fonte:
#include cabecalho.h
em cima do programa.

#include "meu_cabecalho.h"
para incluir um arquivo pessoal, que esteja no diretório atual.

#include <arquivo.h> quando ele se encontra em um dos diretórios padrão, como /usr/include.

A diretiva #include é um comando para o pré-processador, que falaremos em outra semana.

Existem situações que não precisamos dizer cabeçalho algum ou ter bibliotecas chamadas explicitamente, todo programa utiliza a biblioteca c padrão, libc. Sempre. Ou seja, programas que utilizam de suas funções não precisam que a chamem explicitamente.

```
#include <stdio.h>

main() {
printf ("Biblioteca C padrão incluida\n");
printf ("Estamos brincando em OLinux!");
}
```

Um programa que use funções matemáticas, como por exemplo "seno", precisará adicionar os cabeçalho da respectiva biblioteca matemática.

```
#include <stdio.h>
#include <math.h>

main ()

{
double x,y;

y = sin (x);
printf ("Biblioteca matemática okay!");
}
```

Existem variações de sistema para sistema. Isso em relação a mouse, desenhos de janelas, propriedades do terminal e várias outras situações. Necessariamente, você terá que procurar no manual do sistema (man função, uma ajuda e tanto) ou do compilador.

Não importa quantas bibliotecas você usará, sua limitação será em geral memória, toda biblioteca adiciona o fonte e o código objeto. Sistemas Operacionais como o Linux tem a capacidade de distinguir e pegar apenas as funções que

interessam. Já outros, são obrigados a adicionar todos os códigos na memória, e depois de já ter perdido tempo e espaço.

Perguntas

1. Como uma biblioteca é incorporada a linguagem C?
2. Qual nome da principal biblioteca em C?
3. É possível adicionar novas funções, com o mesmo nome das biblioteca C padrão?

A forma de um programa em C

C é uma das linguagens mais flexíveis em termos de formato já existentes. É uma linguagem livre de formato. Isso quer dizer que não existem regras de onde quebrar linhas, botar chaves, e assim vai. Isso é ao mesmo tempo vantajoso mas perigoso. A vantagem é que o usuário tem a liberdade para criar um estilo que melhor lhe atenda. A desvantagem é que programas "bizarros" podem aparecer, e de impossível manutenção. Algumas dicas de se escolher uma estrutura bem definida:

* Programas maiores são de gerenciamento possível só se estiverem bem organizados.

* Programas são entendidos somente se os nomes de variáveis e funções condizem com o que fazem.

* É mais fácil achar partes de um programa se ele tiver bem ordenado, é impossível num programa totalmente desorganizado e "macarrônico" se achar algo.

Simplesmente regras rígidas na hora de escrever um programa não significam que ele terá êxito. Experiência na programação, muito bom senso e clareza de idéias são fatores decisivos na hora de se programar.

Bem, é por aí, Abraços.

Aula 8

Bem, estou gostando de receber os e-mails de vocês, continuem mandando. Quanto ao conteúdo, falaremos de variáveis! Ou seja, uma assunto interessante. Essa aula juntará com a próxima, será uma seqüência importante e conceitos indispensáveis para programação em qualquer linguagem. Vamos lá ..

O que são variáveis?

São espaços reservados que guardam valores durante a execução de um programa. Como o nome diz, elas tem a capacidade de "variar" no tempo. Em geral, são exatamente um pedaço de memória (o tamanho depende do que se esteja guardando) no qual botamos alguma informação e podemos nos referir a ela, utilizá-la, alterá-la ou fazermos o que bem entendermos durante a execução de um programa.

Toda variável tem um nome pela qual é chamada (identificada) e um tipo (o que ela guardará). Os nomes podem ser de uma letra até palavras. Obrigatoriamente deve começar por uma letra ou underscore (o sinal de menos achatado: "_"). O restante pode ser letras de A até Z, maiúsculas, minúsculas, números e o underscore. Exemplos: e, variável _essa_e_uma_variavel, tambem_variavel. Vale ressaltar que ela é sensível a case, o que significa que diferencia maiúsculas e minúsculas.

As variáveis possuem tipos, como dito anteriormente. Os tipos dizem ao compilador que tipo de dado será armazenado. Isso é feito com o intuito do compilador obter as informações necessárias sobre quanto de memória ele terá que reservar para uma determinada variável. Também ajuda o compilador com uma lista de variáveis em um lugar reservado de memória, para que ele possa fazer referências, checar nomes e tipos e que possa determinar erros.

Basicamente possibilita uma estrutura bem definida do que é usado e uma arrumação conveniente na memória.

Tipos de dados em C

Existem vários tipos de dados em C. Todos eles são palavras reservadas. O C é bem flexível e possibilita a criação de tipos (iremos ver isso em outra oportunidade). Agora vamos aos tipos mais usuais, já dá para o começo.

- char: Caracter unitário (só 1 mesmo)
- short int: inteiro "pequeno"
- int: um inteiro (em geral 32 bits)
- float: número real
- double: número real grande

Existem também os tipos *void*, *enum*, *volatile* que citaremos futuramente. Não podemos esquecer dos ponteiros, que falaremos também em futuros capítulos.

Outro detalhe, existe a possibilidade de se colocar a palavra *unsigned*, que significa não sinalizado, antes de um tipo. Isso quer dizer que um inteiro, por exemplo, que varia entre -32.768 até 32767 (isso calcula-se através do tamanho máximo que ele pode ocupar em memória) variaria só de 0 em diante. Ou seja, o espaço do menos seria utilizado para mais um número, ficaríamos com 0 até 65535, gastando o mesmo espaço de memória só que de forma diferente.

Declarando minha primeira variável

Para declarar uma variável, precisamos apenas dizer o tipo da variável, seguida de uma lista dos identificadores (nomes), separado por vírgula. Por exemplo:

char ch;
int i,j;
double x,y,z;

É muito interessante a forma que o C trabalha com variáveis, desde a declaração, passagem de parâmetros e todos tipos de operações com variáveis, ele checa, e torna os problemas referentes a isso mais facilmente de serem detectados.

Variáveis Globais x Locais

Existem basicamente dois tipos de varáveis, as locais e as globais. As locais são aquelas que só o pedaço (pedaço == função e será explicado na próxima aula) do programa que a pertence poderá alterá-la. Já a global, todo programa poderá mexer com ela. Citaremos exemplos mais específicos depois, fique tranqüilo se foi muito abstrato.

Inicializando variáveis

As variáveis em C, em geral, não são **necessariamente** zeradas quando são declaradas. Vamos pensar, as variáveis são pedaços de memória, quando resolvo declarar alguma variável, estamos associando um nome com um pedaço da memória.

Este pedaço de memória pode estar sujo, ou algo assim, e acaba interferindo no que vou utilizar numa variável que seria "limpa". Por isso, torne uma prática dar valores (iniciar) as variáveis antes de utilizá-las.

Podemos fazer isso de duas formas, inicializando ao declará-las, assim:

```
int i = 0;
float j = 2.0;
```

Ou podemos:

```
int i;
float j;

i = 0 ;
```

j = 2.0 ;

Normalmente você deve utilizar a declaração que torne seu programa mais claro para você. Lembre-se, clareza é muito importante em um programa, artigo passado falamos disso.

Escolhendo variáveis

Ao se escolher uma variável, você deve pensar:

- Qual tipo de variável é preciso para aquele dado;
- Escolher um nome claro para ela;
- Qual será a área de atuação da variável (ficará mais claro na próxima semana)

Mediante a isso, basta declara-la.

Algumas variáveis já se tornaram consagradas, como o char ch, ou o int i,j,k, para controle de laços.

Exercícios

- O que é uma variável?
- Indique como é declarado duas variáveis inteiras de nome w e z?
- Quais das variáveis seguintes são inválidas: 3ant, %12, _vri, Ab_1v, 0abc
- Qual diferença entre float e double?
- Quais as possibilidade de declaração de variáveis em termos de locais em C?

Bem, é isso pessoal, próxima aula tem mais ..

Aula 9

Pessoal, nessa aula falaremos sobre um conceito muito importante, que é o de função. Caso não estejam entendendo alguma coisa até aqui, mesmo após ler esta aula, voltem e releiam, as bases da programação estão sendo vistas.

Aproveito para agradecer os emails que tenham recebido, continuem mandando e usem o fórum também.

O que são funções?

Como o nome diz, intuitivamente, são "coisas" que desenvolvem tarefas, funções! Brilhante, vamos lá. Funções são caixas pretas, onde você passa algum tipo de dado, e espera receber algum tipo de saída. Explicando tecnicamente, são módulos ou blocos de código que executam uma determinada tarefa. Essa é a melhor definição.

Para que servem elas?

As tais funções existem, por dois motivos básicos: depuração de erros, quando se quebra um problema em pedaços menores, fica mais fácil detectar onde pode existir um problema. Outro motivo é a reutilização. É visível que grande parte dos códigos que existem em um programa são repetidos, só se diferenciando as variáveis que são passadas a eles.

Trazendo para a vida real, duas situações. A primeira, eu tenho que montar um carro. Posso fazer uma máquina que eu passe **todas** as peças e ela me retorne o resultado. Ou posso criar uma máquina que gerencie várias outras pequenas que desempenham tarefas diferentes que, juntando-as, eu obtenho o carro pronto. Intuitivamente, gostaríamos da primeira, mas devemos pensar que uma grande caixa preta é pior de se depurar do que várias outras pequenas e de tarefas específicas. Imagine se acontece algum problema na caixa preta grande, teríamos que abri-la toda, mexer em várias coisas e tentar chegar a uma conclusão em relação ao problema. Já em um monte de caixas pequenas especializadas,

detectaríamos o problema muito mais facilmente, só pelas tarefas que elas realizam. Podemos citar não só questão de problemas, como performance, entendimento e confiabilidade.

Para que servem? (II)

Outra situação. Imagine que eu precise fazer uma calculadora. Eu poderia fazer um conjunto de operações (função), que em um bolo de código calculasse todos os tipos de operações matemáticas desejáveis em minha calculadora no momento. Agora pense, depois de 1 ano, eu preciso de 2 operações matemáticas das 15 que minha calculadora antiga fazia, o que fazer ? Agregar o bolo de código com 15 funções, 13 delas desnecessárias? A modularização serve para o reaproveitamento de código, devemos chegar a pedaços razoáveis e especializados de código que nos resolvam problemas e que possamos utiliza-los depois.

Lembre-se, isso é uma prática não muito fácil, depende da experiência do profissional e como ele faz a análise inicial do problema, quebrando-os em menores pedaços e chegando a módulos pequenos e ao mesmo tempo usuais.

Nomes

Bem, podemos dar nomes as funções assim como em variáveis, lembre-se da aula passada. Letras de A até Z, sem preocupação de case (maiúscula/minúscula), de 0 a 9 e com underscore (aquele menos achatado, "_"). Precisa começar por um caracter ou underscore. É sensitivo a case, ou seja, funções com o mesmo nome, mas letras diferentes (em case) não são consideradas iguais. Podemos exemplificar: esta_e_uma_funcao e Esta_e_uma_funcao, o "E" ("e") é diferente!

Cara de uma função

Bem, falamos muito de função, mas não mostramos sua cara, então veja:

```
tipo nome(tipo1 var1, tipo2 var2, ...)
{
código1;
.
.
.
códigoN;
}
```

A cara de uma função é essa, vou exemplificar:

```
void diminuir(int parcela1, int parcela2)
{
int total;
total = parcela1 - parcela2;
printf ("A subtracao vale: %d",total);
}
```

Ignore a palavra void por enquanto, acredite que ela quer dizer nada pelo momento. Quando a chamo, diminuir(5,3);, eu recebo a mensagem da subtração de 5 por 3, e retorno ao meu programa. Conseguiu materializar?

Note que as chaves ({ e }) delimitam o que é da função (bloco) e o que não é.

A função main()

A função main() é a função principal de um programa. Ou seja, todo programa tem que ter a função main(), caso contrário o compilador reclama e não gera o executável.

Um programa começa executando a função main(). E um programa termina, quando esta função termina. Porém, dentro

da função main() você pode chamar (executar) outras funções. Falaremos mais sobre o main() adiante.

Chamando funções

Bem, podemos chamar (executar) as funções do ponto que desejamos, desde que ela já tenha sido declarada. Ela desvia o fluxo do programa, por exemplo:

```
main()
{
int a=10,b=3;
ação1;
ação2;
diminuir(a,b);
ação3;
}
```

Nota: neste exemplo ação 1, 2 e 3, podem ser qualquer comando (Até mesmo outra função).

O programa desviará o fluxo, após a "ação2", quando ele chamar a função diminuir. isto suspenderá temporariamente a execução do programa para poder executar a função diminuir, até que a mesma termine (retorne).

Tipos de funções

Existem basicamente, dois tipos de funções. Aquelas que retornam alguma coisa a quem a chamou e aquela que não retorna nada.

Começando pelas que não retornam, elas simplesmente realizam tarefas, como o exemplo anterior. Ela faz uma série de passos, e retorna o fluxo ao programa principal, sem interferir em nada em sua execução, a não ser pelo tempo perdido e saída na tela.

Outra opção são funções que retornam um valor de um tipo. Lembre-se, como declaramos uma função? tipoX nome(tipo1 var1,tipo2 var2); e assim vai. Ou seja, o tipoX eqüivale ao que a função vai retornar. Vamos entender:

```
int diminuir(int parcela1, int parcela2)
{
int total;
total = parcela1 - parcela2;
return total;
}

main()
{
int a=10,b=3,total;
ação1;
ação2;
total = diminuir(a,b);
printf ("A subtracao vale: %d",total);
ação3;
}
```

O efeito é exatamente o mesmo, só que agora o programa principal é que estará jogando a mensagem na tela e a variável do programa, chamada total, que terá o valor da subtração (resultado, tipo int, retornado de diminuir(a,b)). Aos poucos vamos juntando as peças.

Vale ressaltar, o que determinou a saída da função. No caso, foi a chamada ao comando return, que interrompe o fluxo do bloco que está sendo executado (saindo deste bloco) e volta aquele imediatamente anterior. Não é necessário chegar até a última linha da função, o return pode estar na 1a, 2a, onde quer que seja.

O que é o main?

main() também é uma função, é a função que é obrigada a existir em **todos** os programas. Como já disse, tal função é procurada e a primeira a ser executada em todo programa.

Ela pode retornar um valor de tipo int. Ou seja, retorna um número, em geral para o sistema operacional, com o código de sucesso ou indicando qual o erro (número do erro) ocorreu durante a execução do programa.

Exercícios

Vamos testar alguns conhecimentos.

- Escreva uma função que pegue dois números, ache a multiplicação entre eles e jogue na tela.
- Escreva o mesmo programa, só que agora ele deve passar para uma variável do programa principal, que irá somar o resultado a 3 e dividir por 2.
- Explique qual a diferença entre funções que retornam ou não valores.
- Qual utilidade do tipo de uma função?
- Qual tipo de função devemos utilizar quando não queremos retornar nada?

Bem, é isso pessoal, espero que tenham compreendido. Reforçando, mande e-mail se necessário e postem dúvidas no fórum.

Aula 10

Muito bem, desta vez eu vou escrever o artigo de programação desta semana, tendo em vista que o Elias não pode, pois anda ocupado, trabalhando demais (legal, conta outra). Estou brincando hein Elias?

Nesta semana estaremos falando sobre os comandos de controle (if,while,for) que irão lhe dar maior poder de programação. Depois desta aula, você já estará capacitado a criar seus primeiros programas e impressionar seus amigos.

Verdadeiro e Falso

Bom, vamos começar falando sobre o conceito de verdadeiro e falso, pois são utilizados constantemente em programas.

Para a linguagem C, quando uma variável contém o valor 0 (zero), ela está com o valor falso. Qualquer outro valor diferente de zero (positivo ou negativo), significa verdadeiro.

Generalizando, qualquer coisa diferente de zero é "verdadeiro" e qualquer coisa igual a zero é "falso".

Você deve estar se perguntando agora qual a utilidade disto. Veremos a seguir.

O Comando de Controle IF

O comando de controle **if** (em português: **se**) é o mais famoso de todas as linguagens de programação, em todas ele existe. Sua finalidade, é verificar se uma **condição** é "verdadeira", e caso seja, então executar determinada tarefa (que pode ser uma função).

O formato do **if** é o seguinte:
if (**expressão**) {
tarefa1
}

NOTA: O comando **if** deve ser escrito **sempre** em letras minúsculas, caso contrário não irá funcionar, todos comando do C são sensíveis a case.

A **expressão** do **if** na verdade é a **condição** lógica para que ele execute a **tarefa1**. Esta **expressão** caso seja numérica, pode ser comparada assim:

```
int a;
int b;
if (a comparação b) {
tarefa1
}
```

tarefa1 só será executada se a comparação entre **a** e **b** retornar verdadeira.

Onde **comparação** pode ser:
== : Retorna "verdadeiro" se o primeiro número (variável) for igual ao segundo. Falso caso contrário.
!= : Retorna "verdadeiro" se o primeiro número (variável) for diferente do segundo. Falso caso contrário.
> : Retorna "verdadeiro" se o primeiro número (variável) for maior que o segundo. Falso caso contrário.
>= : Retorna "verdadeiro" se o primeiro número (variável) for maior ou igual ao segundo. Falso caso contrário.
< : Retorna "verdadeiro" se o primeiro número (variável) for menor que o segundo. Falso caso contrário.
<= : Retorna "verdadeiro" se o primeiro número (variável) for menor ou igual ao segundo. Falso caso contrário.

Exemplo:

```
main() {
int primeiro=10;
int segundo=5;
if (primeiro>3) printf("primeiro maior que 3");
if (primeiro!=segundo) printf("primeiro diferente do segundo");
}
```

NOTA: Quando for executar apenas uma tarefa (como o printf no exemplo) não é necessário o uso das chaves. Caso você queira executar mais de um comando, use as chaves (**{** e **}**).

No exemplo acima a **expressão** (primeiro>3) vai retornar "verdadeiro" e irá imprimir na tela "primeiro maior que 3".

Podemos ainda usar o termo **else** (traduz-se para: **senão**) ao final do **if** caso a **expressão** seja falsa, para executar outra tarefa. Exemplo:

```
main() {
int x=10;
int y=2;
if (x>y) printf("x maior que y");
else printf("x menor que y");
}
```

Traduzindo: **se** x maior que y imprima na tela "x maior que y", **senão** imprima na tela "x menor que y".

Como dissemos antes, o que é zero é "falso" e o que é diferente de zero é "verdadeiro". Você pode usar o retorno de uma função como **expressão** para o **if**. O símbolo **!** (not, negação) inverte o valor lógico da **expressão**. Por exemplo:

```
int maior(int a, int b) {
if (a>b) return 1;
else retun 0;
}
```

```
main() {
int a=10;
int b=20;
if (!maior(a,b)) printf("a é menor que b");
}
```

Traduzindo: Se **a não (!)** for maior que **b** imprima na tela "a é menor que b".

Ainda sobre o **if** podemos destacar os operadores lógicos **and** e **or**. Estes dois são representados pelos símbolos **&&** e ||, respectivamente. O primeiro é um **e** e o segundo um **ou**. São usados como neste exemplo:

```
main() {
int a=10;
int b=20;
int c=30;
if (a<b && c>b) printf("c é o maior de todos");
if (a==10 || a==5) prnt("a é igual a 5 ou 10");
}
```

Traduzindo: Se **a** for menor que **b**, **e c** for maior do que **b** imprima na tela "c é o maior de todos". Se **a** for igual a 10 **ou a** for igual a 5 imprima na tela "a é igual a 5 ou 10".

Pode-se usar mais de um **and** e mais de um **or**, inclusive em conjunto, tornando a lógica um pouco mais complexa.

O Comando de Controle While

O **while** (em português: **enquanto**) serve para se deixar um programa em **loop** (rodando a mesma coisa, travado de propósito) até que a **expressão** que o prende seja verdadeira. O **while** tem dois formatos:

```
while (expressão) {
tarefa1
}

do {
tarefa1;
} while (expressão);
```

Exemplo:

```
main() {
int b=2;
int a=0;
while (a!=b) {
a=a+1;
}
}
```

Traduzindo: **enquanto a** for diferente de **b** some 1 ao valor atual de **a**. Ou seja, em duas execuções de "a=a+1", o valor de **a** igual ao de **b** que é 2.

NOTA: A expressão "a=a+1" não é uma expressão matemática válida. Porém, em programação isto vale, pois a variável a esquerda do símbolo de igualdade (=) na verdade está recebendo o valor da expressão à direita do igual (=). Ou seja, a posição de memória da variável **a** é preenchida com o resultado de "a+1".

Exemplo2:

```
main() {
int a=2;
```

```
do {
a=a-1;
} while (a!=0);
}
```

Traduzindo: **faça** "a=a-1" **enquanto a** for diferente de zero. Ou seja, tire 1 de **a** até que seu valor seja zero.

O Comando de Controle FOR

Imagine o **for** como um **while** com limites definidos. Este é o formato do **for**:

```
for (valor_inicial;condição;passo) {
tarefa1;
}
```

valor_inicial é o valor inicial de uma variável numérica qualquer.
condição indica qual é a condição de parada do **for** (expressão de verdadeiro ou falso), é um while.
passo é o que deve ser executado ao final de uma execução de **tarefa1** cuja condição do **for** ainda não foi satisfeita.

Exemplo:

```
main() {
int i;
for (i=0;i<10;i=i+1) printf("teste");
}
```

Traduzindo: comece com **i** igual a zero. Enquanto **i** for menor que 10, some 1 ao valor de **i** ("i=i+1") e imprima na tela "teste". Ou seja, imprime a "teste" 10 vezes na tela (com **i** variando de zero a 9).

A variável dentro do **for** pode ser usada dentro do próprio **for**, assim:

```
for (x=0;x<10;x=x+1) {
a=a+x;
}
```

Exemplo de Programa

```
int maior(int a, int b) {
if (a>b) return 1;
else return 0;
}

main() {
int a=10;
int b=5;
int c=8;
while (!maior(b,a)) {
b=b+1;
if (b==c) printf ("b vale o mesmo que c");
}
```

Traduzindo: Enquanto **b** não for maior do que **a** some 1 ao valor atual de **b** e se **b** agora for igual a **c** imprima na tela "b vale o mesmo que c". Ou seja, o valor de **b** vai crescendo (6,7,8,...). Quando **b** chega em 8, o **if** irá ativar o printf que imprimirá na tela "b vale o mesmo que c". O programa termina quando **b** chega a 11, pois **b** vai ser maior do que **a**, o que fará sair do **while**.

Exercícios

1) Escreva uma função que recebe dois números inteiros e verifica se são iguais. Se forem iguais retorna "verdadeiro" (diferente de zero). Caso contrário retorna "falso" (zero).
2) Use a função do item 1) em uma programa que verifica se 10 é diferente de 5, sem alterar a função do item 1).
3) Faça um programa que some os 10 primeiros números naturais usando o comando **for**.
4) Faça o mesmo programa que no item 3), porém usando o comando **while**.

Com isso concluímos nosso curso de C desta semana. Foi um pouco puxado, mas tentei explicar tudo o mais claro possível. Não deixe de reler este artigo e praticar, estes comandos e conceitos aprendidos hoje são os **mais** importantes em qualquer linguagem de programação.

Espero que tenham gostado. Qualquer dúvida postem no fórum, ou mandem e-mail. Críticas e sugestões também são bem vindas.

Aula 11

Estou de volta, agora o Elias saiu para o carnaval mais cedo. Nossos artigos estão sendo escritos muito antes de propósito, assim podemos viajar no carnaval.

Nesta aula de hoje, estaremos praticando os comandos de controle ensinados na aula passada. Se você quer se tornar um bom programador tem que se acostumar a praticar sempre que possível. Então mãos a obra!

Conhecendo melhor a função printf

Vamos então aprender a função printf um pouco mais a fundo. Sabemos que os tipos básicos de variáveis são: int (inteiro), char (caractere) e float (real). Para imprimir uma variável de um tipo destes, usamos a referência abaixo:

%d -> Para variáveis do tipo int (inteiros)
%c -> Para variáveis do tipo char (caractere)
%f -> Para variáveis do tipo float (real)

Exemplo:

```
#include

main() {
int x;
char y;
float z;
x=10;
y='2';
z=5.2;
printf("int: %d \n char: %c \n float: %f \n",x,y,z);
}
```

Traduzindo: O **%d** é o primeiro que aparece entre aspas, logo será substituído pelo valor da primeira variável depois das aspas (x). O **%c** é o segundo que aparece entre as aspas, logo será substituído pelo valor da segunda variável depois das aspas (y). O mesmo ocorre para o %f que será substituído pelo valor de z.

NOTA: O **\n** indica quebra de linha. Ou seja, quando o printf encontra o **\n** ele pula linha.

A função scanf

Assim como o printf, o scanf também funciona a base de %d, %f e %c. O scanf serve para pegar algo digitado do teclado.

Exemplo:

```
#include

main() {
int valor;
scanf("%d",&valor);
printf("Você digitou: %d\n",valor);
}
```

NOTA: Sempre use o **&**, está ali de propósito. Mais tarde no curso explicaremos o porque disto.

Ou seja, no scanf você apenas coloca o(s) tipo(s) que quer pegar e em que variável vai ficar o valor (não esquece do **&**).

Vamos aprender então a fazer um programa de perguntas.

Programa de Perguntas

Vamos fazer um simples programa de perguntas que poderá ser ampliado. Vou listar o código e explica-lo.

```
#include

main() {
char resposta;
int valor;


printf("Você usa Linux? [s/n]");
scanf("%c",&resposta);
getchar();
if (resposta=='n') print ("Então o que você está esperando? Compre já!\n");
else {
printf("Muito bom. Há quantos anos você o usa?");
scanf("%d",&valor);
getchar();
if (valor>=2) print("%d anos? Você deve ter uma boa experiência
então\n",valor);
else {
printf("Legal! Você conhece o site OLinux? [s/n]");
scanf("%c",&resposta);
getchar();
if (resposta=='s') printf("Continue nos acompanhando então :)");
else printf("Ainda não? Então visite: http://www.olinux.com.br. Você não irá
se arrepender :)");
}
}
}
```

Explicando:
Primeiro ele pergunta: "Você usa Linux?".
Se a resposta for não (n) então imprima: "Então o que você está esperando? Compre já!"
Senão, se a resposta for sim:

imprime: "Muito bom. Há quantos anos você o usa?"
Se for maior ou igual a 2 anos imprime: "%d anos? Você deve ter uma boa experiência então" (onde %d vão ser os anos que usa)
Senão, se for menos de 2 anos:
imprime: "Legal! Você conhece o site OLinux? [s/n]"
Se a resposta for sim (s) imprime: "Continue nos acompanhando então :)"
Senão imprime: "Ainda não? Então visite: http://www.olinux.com.br. Você não irá se arrepender :)"

NOTA: A função getchar() serve para limpar a entrada do teclado. No Linux às vezes é necessário usá-la, pois o Linux não tira o Enter do buffer (memória reservada) do teclado.

AND e OR mais a fundo

Vamos praticar o AND e OR do if.

Exemplo:

```
#include

main() {
int a;
int b;
int c;
int fim;
a=1;
b=2;
c=3;
fim=0;
while (!fim) {
if ((a>3 || b>3) && c>b) fim=1;
a=a+1;
b=b+1;
c=c+1;
}
}
```

Traduzindo: **Enquanto** não for fim (fim igual a zero, fim é "falso", lembram?):
Se **a** maior que 3 **ou b** maior que 3, **e c** maior que **b** então fim é igual a um, ou seja, fim é "verdadeiro".

Explicando: O while fica em loop testando o if e fazendo:
a=a+1;
b=b+1;
c=c+1;

Em duas rodadas do while a expressão do if torna-se verdadeira e ele torna fim igual a um, ou seja, chegou ao fim.
Agora o while irá sair pois !fim (não fim) será "falso", pois fim é "verdadeiro" (não verdadeiro = falso).

Use quantos parênteses forem necessários para a expressão do if ou while. Mas lembre-se, quanto mais AND e OR, mais complexa fica a lógica.

Exercícios

1) Faça um programa que pegue duas notas de um aluno e calcule a média deste aluno e a imprima na tela.
2) Faça um programa de matemática que pergunte somas simples e que fica em loop enquanto não acertar as perguntas, imprimindo na tela se a pessoa acertou ou não.

É isso aí. Criem mais programas e pratiquem sempre. Já viram o poder que têm nas mãos?

Qualquer dúvida escrevam no fórum ou mandem e-mail para mim. Críticas e sugestões também são bem vindas. Até semana que vem.

Aula 12

Nesta semana iremos praticar funções e comandos de controle, para que semana que vem possamos ensinar novos conceitos, prosseguindo com o curso.

É preciso estar consciente da utilização do que foi ensinado até agora, pois são as partes mais importantes da programação. Por isso a insistência. Queremos ter certeza de que estarão aptos a prosseguir, pois não pode haver insegurança.

Vamos então a uma revisão aprofundada.

Printf: Revisando

Imprimir um texto na tela:
printf("texto na tela");

Imprimir um texto na tela com quebra de linha (**\n**: pular para a próxima linha):
printf("texto na tela com quebra de linha\n");

Imprimir o valor da variável **x** do tipo inteiro (**int**) na tela:
printf("O valor de x é: %d",x);

Imprimir o valor da variável inteira **x** e depois o da variável inteira **y** na tela:
printf("O valor de y é: %d e o valor de x é: %d",y,x);

Imprimir o valor da variável inteira **y** e depois o da variável inteira **x** na tela:
printf("O valor de x é: %d e o valor de y é: %d",x,y);

Imprimir o valor da variável inteira **x** na tela e depois o da variável real **f**:
printf("x inteiro vale: %d, f real vale: %f",x,f);

Imprimir o valor da variável caracter **a** , depois o valor da variável inteira **b** e depois o valor da variável real **c**:
printf("a vale: %c, b vale: %d e c vale: %f",a,b,c);

Scanf: Revisando

Digamos que tenhamos declarado em nosso programas as 3 seguintes variáveis:
int a;
char b;
float c;

Vamos usar o scanf para pegar um valor do teclado para a variável inteira **a**:
printf("Entre com um número inteiro: ");
scanf("%d",&a);

Agora para a variável caractere **b**:
printf("Entre com uma letra ou símbolo: ");
scanf("%c",&b);

Por último para a variável real **c**:
```
printf("Entre com um número real: ");
scanf("%f",&c);
```

Declarando funções

Vamos criar a função **digaoi** que imprime na tela o texto "oi":
```
void digaoi() {
printf("oi\n");
}
```

Vamos criar a função **maior** que tem dois inteiros como entrada e verifica se o primeiro é maior que o segundo. Se for, retorna 1 (**verdadeiro**). Senão retorna 0 (**falso**):
```
int maior(int a, int b) {

if (a>b) return 1;
else return 0;
}
```

Vamos agora criar uma função **main** (principal) que usa estas duas funções:

```
#include

void digaoi() {
printf("oi\n");
}

int maior(int a, int b) {
if (a>b) return 1;
else return 0;
}

main() { int x=5;
int y=10;
int z=2;
digaoi();
printf("x maior que y? resposta: %d\n",maior(x,y));
}
```

Usando o IF

Vamos criar uma função **main** que use a função **maior** e o **if**:

```
main() {
int x=10;
int y=2;
if (maior(y,x)) printf("y maior que x\n");
else printf("x menor ou igual a y\n");
}
```

Podemos fazer usando negação:
```
main() {
int x=10;
int y=2;
if (!maior(y,x)) printf ("y não é maior que x\n");
else printf("y maior que x\n");
```

```
}
```

Usando o FOR

Vamos usar o **for** de maneira simples em uma função **main**:

```
main() {
int i;
int k=10;
printf("Eu sei contar até nove: ");
for (i=0;i<k;i++) printf("%d",i);
printf("\n");
}
```

Traduzindo: Imprime na tela "Eu sei contar até nove: ". Depois inicia o **for** com **i** igual a zero, fazendo **i++** enquanto **i** seja menor que **k** (i<K).

NOTA: O **i++** eqüivale a **i=i+1;**. Pode ser usado para qualquer variável inteira.

Explicando: O valor de **i** é impresso na tela enquanto seu valor for menor que o valor de k (que é 10). Logo, a cada iteração é feito **i++** (i=i+1). Sendo assim, quando **i** chega a 10 (valor de k), **i** não é mais menor (**<**) que **k** e o **for** para.

Usando o WHILE

Vamos fazer o mesmo programa que fizemos pro **for** só que pro **while** de duas maneiras diferentes:

```
main() {
int i=0;
int k=10;
printf("Eu sei contar até nove, de outra maneira: ");
while (i<k) {
printf("%d",i);
i++;
}
}
```

Agora com o **do**:

```
main() {
int i=0;
int k=10;
printf("Eu sei contar até nove, de outra maneira: ");
do {
printf("%d",i);
i++;
} while (i<k)
}
```

Exercícios

1) Faça um programa qualquer que use todas as funções revisadas hoje. Será que você consegue?

Quem quiser pode mandar o programa para que nós analisemos.

Por essa semana é só. Continuem praticando. O Elias deve voltar semana que vem com assunto novo. Não percam.

Qualquer sugestão ou crítica mandem e-mail para mim. Dúvidas ponham no fórum de programação.

Aula 13

Vamos discutir sobre vetores, de uma ou mais dimensões. Falaremos também de strings.

Matrizes

Matriz é um conjunto de variáveis referenciadas pelo mesmo nome. É referenciada através de um índice e disposta na memória de forma contígua (contínua). Cada "pedaço" é uma célula.

Por exemplo, temos uma matriz de inteiros, de 2x10:

5 12093 6 9

1 2 39382 1019

Matrizes de uma dimensão

Definimos matrizes unidimensionais da forma:

tipo nome_da_variavel[quantidade_de_elementos];
float temperatura[10];

Tais matrizes também são conhecidas por vetores. Um detalhe interessante é que toda matriz começa pelo elemento 0, sempre. Aprenda a se acostumar com isso.

Declarando de forma básica um vetor: *int vetor[20];*. Para alcançar o elemento número 1, seria: *inicio = vetor[0];* . Já o de número 20, teríamos: *fim = vetor[19];* .

Matrizes de várias dimensões

Declaramos uma matriz de duas dimensões da seguinte forma: *int matriz[20][20];*. Para aumentar o número de dimensões: *int matriz_auxiliar[20][30][30]..[30];*. Para acessar os elementos, temos: *variável = matriz[2][3];* . Para atribuirmos um valor, fazemos: *matriz[19][18] = 10;* .

Strings

Strings são definidas em C como vetores de caracteres. Por exemplo: *char endereco[50];*. Para atribuirmos valores a elas não podemos:

nome = "Elias"; sobrenome = "Bareinboim";

Temos que copiar caracter a caracter ou utilizarmos funções que já façam exatamente isso (em nosso exercícios temos isso). A biblioteca especial para tratamento de strings em C chama-se *string.h*. Dica: usar função strcpy

Para imprimirmos uma string fazemos: *printf("%s",nome);*. Ou seja, utilizamos o %s. Lembre-se, o %c é um caracter somente.

Todas strings em C terminam com \0. É imprescindível ter um marcador para fazermos operações com strings. A variável nome contém "Elias", então:

E l i a s \0

0 1 2 3 4 5 6 7 8 9

Para leitura de dados podemos também utilizar rotinas já prontas ou implementarmos as nossas próprias (teremos exercícios exatamente disso).

Em funções devemos nos referenciar a vetores pelo seu nome absoluto, na próxima aula entenderemos tudo muito melhor, fiquem calmos !:) Exemplificando, *char nome[30];*, quando se passa para uma função, passamos nome e em outro campo o número de elementos.

Exercícios

- Crie uma matriz 5x5 de inteiros . Acesse o elemento da linha 3 e coluna 4 e coloque numa variável chamada tmp. Ponha o valor 20 na primeira coluna e última linha.
- Faça uma função que retorne o elemento de número 22.
- Declare uma matriz de inteiros. A dimensão deve ser 20x40x80. Acesse o elemento de índice 2/5/10 e coloque-o numa variável chamada tmp2.
- Faça uma função que retorne o elemento de número 1024.
- Escreva uma função para ler um vetor de qualquer tipo e tamanho. Faça uma para imprimir também.
- Faça uma outra função para procurar um elemento (tipo básico de dados) qualquer (a se passar posição de início de procura) e retornar a posição da primeira ocorrência.
- Com a função acima, faça uma terceira que retorne o número de ocorrência de um elemento em vetor.
- Escreva um programa que copie strings. O nome da função chama-se copia_string e deve receber a origem e destino. Por exemplo, copia_string(destino,origem).

Bem, os desenhos estão feios mas espero que tenha sido claros. Qualquer dúvida usem o fórum ou mande-me e-mail.

Aula 14

Alguém viu o Elias por aí? Pois é, lá vou eu de novo escrever o artigo desta semana. Hoje faremos uma revisão da aula passada sobre vetores e matriz. Também estaremos iniciando um novo assunto: ponteiros. Eles têm várias finalidades e são um assunto bastante importante quando se fala em C.

Veremos porque ainda neste artigo. Mas antes faremos uma breve revisão da aula passada.

Vetores

Vetores são variáveis que guardam mais de um valor. Estes valores são acessados através de índices. Estes índices sempre começam de zero. Exemplo:

int teste[4]={ 5, 4, 3, 2 };

Então a posição de índice zero do vetor teste contém o valor inteiro 5. Acessamos assim:
printf("%d\n",teste[0]);
Isto imprime na tela o número 5, que é o valor da posição zero do vetor teste.

Para alterar um valor é a simples. Exemplo:

```
int teste[2]={ 10, 20 };
printf("%d\n",teste[1]);
teste[1]=30;
printf("%d\n",teste[1]);
```

No primeiro printf o valor de teste[1] é 20, ele o imprime na tela. No segundo printf o valor de teste[1] é 30 (pois fizemos teste[1]=30), e o printf o imprime na tela.

Matrizes

Matrizes são vetores com mais de uma dimensão. Exemplo:

```
int teste[5][10];
```

Colocamos um valor (neste exemplo 20) na linha zero e coluna 5 assim:

```
teste[0][5]=20;
```

Para ler é a mesma coisa:

```
printf("%d\n",teste[0][5]);
```

Isto irá imprimir na tela o valor 20 que definimos acima.

Ponteiros

Ponteiros, como o próprio nome diz, é um tipo de variável que aponta para outra (de um tipo qualquer). Na verdade um ponteiro guarda o endereço de memória (local onde se encontra na memória) de uma variável.

Vamos iniciar um exemplo simples para entendermos melhor. Imagine que tenho a seguinte variável:

```
int teste=20;
```

Sabemos que a variável **teste** é do tipo inteiro (**int**) e armazena o valor **20**.

Agora digamos que queremos uma ponteiro para a variável **teste**. Então criamos um ponteiro para apontar para uma variável do tipo inteira (ou seja, apontar para teste), assim:

```
int teste=20;
int *p;
```

Nota: Um ponteiro é definido usando o asterisco (*) antes do nome da variável.

Neste exemplo a variável **p**, por incrível que pareça, **não** é do tipo inteiro (**int**). A variável **p** é do tipo ponteiro, que aponta para uma variável do tipo inteiro (**int**).

Agora façamos **p** apontar para a variável **teste**:

```
int teste=20;
int *p;

p=&teste;
```

Nota: O símbolo **&** indica o endereço de memória da variável.

Fazendo p=&teste estamos dizendo que **p** irá armazenar o endereço de memória da variável **teste**. Ou seja, **p** não armazena o valor 20, mas sim o endereço de **teste** que, este sim, armazena o valor 20.

Mas então como chegar ao valor 20 usando a variável **p**, já que este aponta para **teste**?

Simples, basta fazer assim:

```
int teste=20;
int *p;

p=&teste;
printf("%d\n",*p);
```

Nota: Para acessar o valor de uma variável apontada por um ponteiro, usa-se o asterisco (*****) antes da variável ponteiro.

Neste exemplo ***p** retorna 20, que é o valor de **teste**, variável o qual armazenamos seu endereço de memória em **p**. Logo o printf irá imprimir 20 na tela.

Outro exemplo:

```
char algo[5] = { 5, 4, 3, 2, 1 };
char *c;

c=&algo[2];
```

Neste exemplo, colocamos em **c** o endereço do terceiro elemento de **algo**. Fazendo isso, dizemos automaticamente que c[0]=3, c[1]=2 e c[2]=1.

Se tivéssemos feito c=&algo[3], então estaríamos fazendo c[0]=2 e c[1]=1.

Quando fizemos:
char algo[5];
É a mesma coisa que:
char *algo;
Porém desta última maneira o tamanho de algo ainda não tem tamanho definido. Isto se faz com a função malloc() que veremos na próximas aulas.

Nota: Nunca use um ponteiro como vetor a menos que você tenha definido seu tamanho. No Linux isto retornaria uma mensagem de **segmentation fault** pois estaria invadindo uma área de memória não reservada para o atual programa. Em outros sistemas operacionais isto é aceitável podendo travar o mesmo.

Nota: Para matrizes de dimensão maior que um (ou seja, não vetor) não é válido dizer, por exemplo, que char teste[2][2] eqüivale à char **teste. Isto só vale para vetores.

Outra explicação

Se você ainda não entendeu como funcionam ponteiros imagine que variáveis são casas. As casas podem armazenar um valor, dependendo de seu tipo. Além de valores normais, uma casa destas pode armazenar um endereço de outra casa. Nesta outra casa você encontraria o valor desejado.

Pode não estar claro aonde se utiliza ponteiros e o porque de usa-los. Entretanto acompanhe nossos artigos e você vai entender porque ponteiros são muito importantes e qual sua finalidade.

Exercícios

1) Escreva um programa que peça ao usuário um número que deve ser digitado do teclado. Guarde este número em uma variável. Depois faça um ponteiro apontar para a variável que guardou este número e imprima-o na tela, acessando este pela variável ponteiro.

2) Crie uma variável **x** do tipo caractere (**char**) e armazene nela um valor inicial. Crie agora uma variável ponteiro chamada **ptr** para apontar para **x**. Agora, note a diferença do ponteiro para o tipo imprimindo:

```
printf("%d\n",x);
printf("%p\n",ptr);
```

Onde **"%p"** pede para imprimir o endereço de memória armazenado em **ptr**.

3) Tente usar um ponteiro como vetor sem usar a função malloc() e veja o que ocorre.

Por essa semana é só. Na próxima o Elias volta (espero :). Qualquer dúvida usem o fórum ou mandem e-mail. Sugestões e críticas também são bem-vindas.

Até a próxima.

Aula 15

Alguém viu o Elias por aí? E o André Souza? Pois é, estou estreando nessa semana como colunista do curso de C. Espero que o curso não perca uma identidade por minha causa pois os dois são excelentes professores de C. Deixando de papo furado, hoje a aula continuará sobre o assunto ponteiros. Vamos explicar como alocar (e desalocar) uma área de memória, falaremos sobre porque um ponteiro pode ser tratado como um vetor e vice-versa e introduziremos strings.

Malloc e Free

Quando se declara um ponteiro em C, nós temos um "ponteiro", um "indicativo" para uma única posição de memória. Isso é o suficiente se você quiser que esse ponteiro apenas te diga onde está uma variável.

Porém, ponteiros são usados muitas vezes quando não se sabe o tamanho de um vetor no início de um programa. Permita-me dar um exemplo. Digamos que você queira fazer um programa para ler as notas da prova de matemática de uma turma de 5ª série e fazer a média. Você deseja fazer um programa genérico que será usado em um colégio que possui turmas de 5ª de número variado de alunos. Então você tem 2 opções: descobrir qual o tamanho da maior turma e declarar no início de seu programa um vetor de inteiros com esse número (**int notas[maior_nº_de_alunos]**) ou declarar um ponteiro para inteiros que depois será um vetor, quando o usuário, entrar com o número de alunos de cada turma a cada vez (**int *notas**).

A primeira opção não é maleável. Sempre poderá surgir uma turma com mais alunos que o seu máximo. E se você colocar o máximo como 1 milhão, por exemplo, e nunca (ou quase nunca) estiver usando esse número de alunos, você está fazendo com que seu programa desperdice memória. Por isso, a segunda opção é a mais indicada. E ela é possível através de dois comandos de C: **malloc** e **free**. Veja abaixo como seria o exemplo com ponteiros.

```
void main()
{
int *notas;
int numero;
int i;
printf("Entre com o nº total de alunos: ");
scanf("%d", &numero);
notas = (int *) malloc(numero * sizeof(int));
}
```

A linha do malloc pode ser vista como: alocamos um vetor que tem número posições de inteiros. Agora notas pode ser acessada como um vetor qualquer.

```
for(i=0; i<numero; i++) {
printf("Digite a nota do aluno %d ",i+1);
scanf("%d",¬as[i]);
printf("\nA nota do aluno %d é %d:", i+1, notas[i]);
```

}

Uma observação é que eu pergunto a nota do aluno 1, mas na verdade, essa nota é armazenada na posição 0 do vetor, pois vetores em C, começam na posição zero.

Você pode perguntar, será que posso alocar muita memória a vontade. A resposta é depende da capacidade física da sua memória. Outra dúvida é: será que posso ir alocando memória sem parar. A resposta é: use sempre o comando free (para desalocar) se não for mais usar uma determinada área de memória. A sintaxe do free é simples, como tudo na vida é mais fácil destruir que construir.

free(notas);

Ponteiro = Vetor?

Outra característica de C é que um vetor pode ser acessado como um ponteiro e vice-versa. Para aprendermos isso, é necessário sabermos primeiro como é a aritmética dos ponteiros. Como um exemplo, vale mais que mil explicações, vamos a ele. Digamos que tenhamos duas variáveis: a primeira é um vetor e a segunda é um ponteiro para esse vetor.

```
int vet_notas[50];
int *pont_notas;
```

Agora vamos apontar pont_notas para vet_notas[50].

```
pont_notas=vet_notas;
```

Não é necessário o símbolo & porque os vetores em C, são representados internamente como um ponteiro para a sua primeira posição e o número de posições nele.

Portanto, caso queiramos imprimir a primeira e a décima nota de nosso vetor, temos duas opções:

```
print ("A primeira nota é: %d", vet_notas[0]);
print ("A primeira nota é: %d", *pont_notas);
print ("A décima nota é: %d", vet_notas[9]);
print ("A décima nota é: %d", *(pont_notas+9));
```

Eu sei que fui rápido! Vamos por partes. Acho que a primeira e terceira linhas, vocês entenderam, não? É a impressão de posições de vetores. Lembre-se que a posição 9 em um vetor na verdade guarda o 10º nº, pois em C os vetores começam no zero.

Agora a segunda e quarta linhas são as novidades. Como eu disse, a primeira posição de um vetor e de um ponteiro que aponta para esse vetor são a mesma coisa porque na verdade o ponteiro só pode apontar para a primeira posição de um vetor e não mais posições! Já a 4ª linha é mais difícil de entender de primeira. O que fizemos foi aritmética de ponteiros. Peguei o ponteiro pont_notas e somei nove. Porém esses nove significam nove posições de memória. Portanto, ele foi cair na décima posição do vetor. Daí coloco um * na frente e consigo pegar seu conteúdo!

Vamos a mais exemplos para fixar. Abaixo vou fazer uma equivalência entre vetor e ponteiro.

```
vet_notas[0]==*(pont_notas);
vet_notas[1]==*(pont_notas+1);
vet_notas[2]==*(pont_notas+2);
.
.
.
```

Strings:

Outro assunto interessante de C são strings. String nada mais são do que vetores de caracteres. Vamos a um exemplo:

```
char nome_vetor[10];
char *nome_ponteiro;
```

As duas variáveis, nome_vetor e nome_ponteiro são semelhantes, sendo que a única diferença é que nome_vetor não pode ter mais de 10 caracteres enquanto que nome_ponteiro pode ter a vontade desde que você aloque as suas posições (com malloc). Veja o exemplo:

```
void main(void)
{
char nome1[10];
char *nome2;
printf("Nome 1: ");
scanf("%s", nome1);
printf("Nome 2: ");
nome2 = (char *) malloc(10 * sizeof(char));
scanf("%s", nome2);
printf("Nome 1: %si\n", nome1);
printf("Nome 2: %s", nome2);
}
```

Como você pode observar, tivemos que alocar 10 posições antes de dar um scanf, pois nome2 só era um ponteiro para um caracter. Qualquer tentativa de ler um nome maior que 1 caracter iria falhar!

Operações com Strings

Existem algumas funções da biblioteca padrão de C para manipular strings. Vamos a elas.

strcmp compara duas strings e retorna se uma é maior que outra.

```
resultado=strcmp(string1,string2);
if(resultado==0) {
printf("Duas strings são exatamente iguais!\n");
else
if(resultado>0) {
printf("string1 é maior que string2!\n");
else
printf("string2 é maior que string1!\n");
```

strlen retorna o comprimento de uma string em nº de caracteres.

```
printf("Entre com seu nome: ");
scanf("%s",nome);

tamanho=strlen(nome);
printf("Seu nome tem %d caracteres", tamanho);
```

Semana que vem, continuaremos com ponteiros. Leiam e releiam o artigo para fixar bem os conceitos. É muito importante para as próximas aulas. Até lá!

Aula 16

Outro dia estava lendo um e-mail de uma lista de discussão sobre programação, e deparei com o seguinte e-mail (escrito por um brasileiro):

I have a little problem and I'm sure someone can help me.

I am trying to insert a text from a entry in a text(box), but I have to say the lenght of the text to be inserted. How can I get it???

```
Problem:
void on_btn_Ok_pressed (GtkButton *button, gpointer user_data)
{
char *texto;
entry1 = GTK_ENTRY(lookup_widget((button),"entry1"));

text1 = GTK_TEXT(lookup_widget((button),"box_Nome"));

texto = (gtk_entry_get_text(entry1));

gtk_text_insert(text1,NULL,NULL,NULL,texto, ????????????);
gtk_entry_set_text(entry1,"");

}
```

Thanks I apreciate your help and time.

Para quem não sabe inglês, o que ele quer saber é como achar o tamanho da string 'texto', e colocar no lugar do '????????????'.

Eu não sei se esta pessoa está lendo o nosso curso de C, mas realmente ainda não tocamos bem nós tópicos de manipulação de strings em C.

No artigo desta semana estaremos tocando no assunto de manipulação de strings em C e também de ponteiros com funções.

Existem inúmeras funções para manipulação de strings em C, das quais as mais usadas são: **strcpy**,**strcmp**,**strlen**,**strcat**.

strcpy:

Esta função serve para copiar uma string para outra em C. Os parâmetros desta função são a string fonte e a string destino, respectivamente. Exemplo:

```
char *string_retorno;
char string_fonte[] = "EXEMPLO DE MANIPULACAO DE STRINGS";
char *string_destino;

string_destino = (char *)malloc(80);

string_retorno = strcpy(string_destino,string_fonte);

printf("String Fonte = %s, String Retorno = %s",string_fonte,string_retorno);
```

Após a execução deste pequeno trecho de programa, a saída para a tela deverá ser:

String Fonte = EXEMPLO DE MANIPULACAO DE STRINGS, String Retorno = EXEMPLO DE MANIPULACAO DE STRINGS

Neste exemplo, a string 'string_retorno' será um ponteiro para a variável string_destino. Esta string de retorno da função strcpy serve somente para verificar se a cópia das strings foi feita com sucesso, exemplo:

```
if (strcpy(string_fonte,string_destino) == NULL)
{
printf("Erro ao copiar as strings\n");
}
```

strcmp:

Esta função serve para comparar duas strings. Esta função retornará 0 (zero), se as strings forem iguais, e diferente de zero se as strings forem diferentes.

Exemplo:

```
char string1[]="STRING";
char string2[]="STRING";
char string3[]="sTRING";

if (strcmp(string1,string2) == 0)
{
printf("As strings são iguais\n");
}
else
{
printf("As strings são diferentes\n");
}

if (strcmp(string1,string3) == 0)
{
printf("As strings são iguais\n");
}
else
{
printf("As strings são diferentes\n");
}
```

A saída deste pequeno trecho de programa deverá ser:

As strings são iguais
As strings são diferentes

strlen:

Esta função retorna o tamanho de uma string. O único parâmetro desta função é a string da qual você quer saber o tamanho. Neste exemplo, vamos resolver o problema que foi mostrado no início deste artigo:

```
void on_btn_Ok_pressed (GtkButton *button, gpointer user_data)
{
char *texto;
entry1 = GTK_ENTRY(lookup_widget((button),"entry1"));
```

```
text1 = GTK_TEXT(lookup_widget((button),"box_Nome"));

texto = (gtk_entry_get_text(entry1));
/* gtk_text_insert(text1,NULL,NULL,NULL,texto, ????????????); */
gtk_text_insert(text1,NULL,NULL,NULL,texto, strlen(texto));
gtk_entry_set_text(entry1,"");

}
```

strcat:

A função strcat faz é pegar a string fonte e concatenar no final da string destino. Exemplo:

```
char *string_retorno;
char string_final[] = "EXEMPLO DE MANIPULACAO ";
char string_inicial[] = "DE STRINGS";
string_retorno = strcat(string_final,string_inicial);

printf("String Final = %s, String Inicial = %s, String Retorno = %s",string_final,string_inicial,string_retorno);
```

Dessa forma a saída é: **String Fonte1 = EXEMPLO DE MANIPULACAO, String Fonte2 = DE STRINGS, String Retorno = EXEMPLO DE MANIPULACAO**

Hoje ficaremos por aqui porque é a minha estréia na seção de programação. Até semana que vem com mais ponteiros.

Aula 17

Bem, iremos falar nessa semana sobre estruturas, uniões e enumerações.

O que é uma estrutura?

É um conjunto de variáveis dentro de um mesmo nome. Em geral, uma variável é de um tipo específico, por exemplo, temos uma variável do tipo inteira e estamos fechados a nos referenciar aquele nome que lhe foi dado sempre por um número do tipo inteiro, logicamente. Já as estruturas, dentro de um mesmo nome podemos nos referenciar a uma gama de variáveis pré-definidas.

Declaração

Podemos criar definições de estruturas, nas quais podemos utilizá-las como "molde" (tipo) para futura utilização. Existe uma ligação lógica entre os elementos de uma estrutura. Podemos exemplificar com uma estrutura que contenha nome, telefone e saldo na conta corrente.

```
struct molde_conta
{
char nome[50];
int telefone;
float saldo ;
};
```

Bem, declaramos o tipo de dado, o molde para utilização no futuro. Repare que a linha foi terminada com ponto e vírgula, como em um comando comum. Definido o molde, devemos agora declarar a variável que utilizará desse molde.

```
struct molde_conta conta;
```

Agora sim, temos a variável conta com a definição declarada em molde_conta. Uma outra opção é a declaração direta, por exemplo, já na definição do molde, declaramos as variáveis de forma embutida. Assim:

```
struct molde_conta
{
char nome[50];
int telefone;
float saldo;
} conta, conta2;
```

Continuamos com o molde chamado molde_conta e além disso declaramos o conta e conta2 como esse tipo, o equivalente a:

```
struct molde_conta conta conta2;
```

Na memória, as variáveis dentro da struct são seqüenciais, o nome da struct só é o endereço da primeira posição de memória de tal struct, os elementos são as posições exatos do endereço de memória. Espero que todos estejam entendendo, essas definições de memória, posição e tal são altamente ligado aos conceitos de ponteiros.

Podemos também declarar apenas a variável do tipo struct, sem necessidade do seu molde. Teríamos então:

```
struct {
char nome[50];
int telefone;
float saldo;
} conta;
```

Assim, temos a variável conta, exatamente igual ao exemplo anterior. Isso é útil para quando não precisarmos de mais de uma vez, tal variável.

Utilização do tipo

Utilizamos um operador, chamado de operador ponto, para nos referenciar a struct. No exemplo acima, caso queiramos botar um valor no telefone, utilizamos:

```
conta.telefone = 10;
```

Sempre é assim, nome_da_estrutura.nome_da_variável, e assim a utilizamos como uma variável comum, com exceção da especificação do nome da estrutura. Através desse tipo de referenciação, se faz o cálculo automático da posição de memória do elemento. Para uma string, podemos:

```
strcpy(conta.nome,"meunomeentraaqui");
```

Para contar o número de caracteres de nome dentro da estrutura conta, podemos fazer:

```
for (i=0,conta.nome[i],++i) ;
printf ("o nome tem -> %d letras \n",i);
```

A utilização é idêntica. Muito fácil não?

Podemos fazer atribuição de structs, do tipo conta2 = conta, e os valores serão idênticos. Um tipo bem utilizado em programação, são as matrizes de structs. Podemos pensar que um banco, precisa de um conjunto desse tipo de structs, claro que de forma muito mais complexa, para referenciar-se aos seus clientes. Teríamos então:

struct molde_conta conta[100];

Agora teremos um vetor do molde_conta (lembre-se da definição de molde lá do começo), cada um idêntico que o conta tradicionalmente declarado lá em cima, com exceção de seu identificador de posição.

Union

O tipo Union não será muito discutido, pois vamos explicá-lo de forma superficial. É muito parecida sua utilização com a de struct, é como um caso dela, só que as variáveis compartilham de uma mesma área de memória. Por exemplo:

```
union molde_indice{
int numero;
char caracter[2];
};
```

Aqui fazemos o mesmo que nas structs, declaramos o seu molde. Agora para utilizá-la:

union molde_indice índices;

Agora quando eu desejo utilizar um dos elementos, normalmente faço: indices.numero, onde detenho desse elemento. O compilador sempre aloca o espaço da maior variável (no caso int = 2 bytes) para a união completa. Parece não fazer muito sentido esse tipo, só que é bastante interessante quando precisamos fazer conversões de tipos de forma freqüente. Temos como se fosse duas facetas de uma mesma posição de memória, de forma bem prática. Quando queremos o número de forma do tipo caracter, dizemos indices.caracter[0] e temos a primeira parte do número, e com [1] da mesma forma.

Se você não entendeu muito bem dessa parte, sem problemas, não será altamente aplicada no momento. Na hora mais conveniente, se necessário, explicaremos de novo.

Enum

Enumerações são associações de inteiros com nomes, simplesmente por conveniência. Declaramos:

```
enum molde_fruta { banana, maca, pera, uva, melancia, mamao, goiaba};
enum molde_fruta fruta;
```

Agora a variável fruta pode receber um dos valores possíveis do molde_fruta, que na verdade são inteiros seqüenciais. Só isso, é importante frisar, enums são na verdade inteiros e não strings, são úteis para facilitar a vida de quem está programando.

Caso queiramos fazer um programa para uma feira, podemos nos referenciar através dos nomes das frutas e utilizá-las em todo programa, em vez de chamarmos a banana de 0, maca de 1, e assim por aí. Podemos também fixar valores para os elementos de uma enum, através do sinal de igual e o valor, logo após o tipo.

É bastante útil também para ser índice de uma string, que poderá se referenciar aos nomes propriamente ditos que elas se chamam. Lembre-se, nunca podemos chegar e tentar utilizá-las como strings, elas são inteiros. Estou repetindo isso porque é bem comum as pessoas fazerem confusão e não entenderem que só é uma "máscara" para um valor inteiro.

Bem pessoal, é isso, qualquer dúvidas estamos aí. A aula foi um pouco corrida, fiquem a vontade para dúvidas. Ainda temos que falar de ponteiros mais a fundo, em relação a tudo que foi dito aqui e passagem de parâmetros por referência

a funções.

Aula 18

Apresentação

Respeitável público que acompanha este curso de programação... tenho o imenso prazer de anunciar que, a partir de hoje, estarei escrevendo para vocês! Sou Cuco Veríssimo, às vezes chamado pelo (verdadeiro) nome de Rodrigo Hausen, e espero poder ensinar a vocês várias facetas e nuances do mundo do C. Apresentações feitas, vamos ao que interessa!

No cardápio desta semana teremos passagem de parâmetros para funções, em dois sabores diferentes. Mas primeiro, vamos saborear uma entrada, que nos dará a motivação para definirmos algumas funções, antes de passarmos ao prato principal.

Aperitivo: o que nós queremos fazer?

Vamos dizer que, por algum motivo, precisamos declarar uma função qualquer num programa em C. Recordando das aulas anteriores, o que fazemos? A gente vê o que a função deve receber como entrada (ou seja, os seus parâmetros) e como será a saída, isto é, qual será o resultado da função após manipularmos os dados de entrada.

Primeiro Exemplo: suponhamos que nós queremos construir uma função que diga qual é o maior de dois números inteiros. Pense um pouco... como você faria isto? Para simplificar, eu faria assim:

```
int maior(int a, int b) {
if (a>b)
return a;
else /* se a nao for maior do que b */
return b;
}
```

Isso é bem fácil e nós já aprendemos como fazer na **aula 12** do nosso curso. Um programa que chamasse esta função poderia ser do tipo:

```
void main () {
printf ( "O maior número entre 10 e 20 é: %d",maior(10,20) );
}
```

Lembre-se de incluir a definição da função maior no programa! Se quiser, compile e verifique o resultado.

Segundo Exemplo: queremos trocar o valor de duas variáveis **a** e **b**, do tipo float. Então faríamos, no meio de um programa:

```
main() {
float a=10.5, b=17.1;
float temp;

temp = a;
a = b;
b = temp;
```

```
}
```

A variável **temp** é usada para armazenar o valor de **a** temporariamente. No final do programa, teremos os valores de **a** e **b** trocados. Funciona, mas é muito mais elegante fazer algo do tipo:

```
main() {
float a=10.5, b=17.1;

troca(&a,&b);
}
```

Então, devemos definir como será esta função troca que fará a "mágica" para nós (na verdade, não é mágica; é tecnologia). Para podermos aprender a fazer isto, vamos precisar de alguns conceitos, que são o nosso "prato do dia".

Passagem de parâmetros: dois sabores distintos

Quando nós dizemos quais e de que tipo são os dados que a função vai manipular, nós estamos passando parâmetros para a função. Isto nós já fazemos há tempos com as funções que nós mesmos definimos e com as outras definidas nas bibliotecas padrão do C (como as funções printf, scanf). O que há de novo é que esta passagem pode ser feita de dois modos: **por valor** e **por referência**.

Passagem por valor: é quando os valores das variáveis de entrada de uma função são copiados "para dentro" dela. Entenda assim: eu digo para a função o que tem armazenado nas tais variáveis, e ela manipula os **valores copiados** dentro da função. A função maior, do primeiro exemplo, recebe os parâmetros **a** e **b** passados por valor. Até hoje, todas as funções declaradas por nós eram assim.

Passagem por referência: quando a função é informada da **localização** das variáveis na memória. Curiosamente, estas palavras nos fazem lembrar de... **PONTEIROS** (lembrem-se da **14ª** e **15ª** aulas)! Isso mesmo: quando passamos um parâmetro por referência, nós enviamos à função um ponteiro para uma variável. Ganhamos com isso a possibilidade de alterar o valor da variável. Agora nós podemos fazer a nossa tão desejada "mágica" do Segundo Exemplo

Pondo tudo em pratos limpos

Quais os ingredientes dos quais precisaremos para passar os parâmetros por referência? Voilá! Os endereços das variáveis na memória. Só para lembrar, fazemos isso com o operador **&**, seguido do nome da variável. Por exemplo, temos uma variável chamada **resposta** do tipo **char** e quero que seja colocado nesta variável um caractere teclado pelo usuário; para isso, usamos a seguinte linha de código:

```
scanf ("%c",&resposta);
```

Você deve estar pensando agora: "então era para isso que serve o **&** antes da variável na chamada da função scanf...". Sim, senhor, na verdade, estamos passando para a função scanf o endereço da variável, para que ela possa colocar dentro dessa posição de memória o valor lido do teclado.

Agora, vejamos como declarar uma função com passagem de parâmetros por referência. Voltando ao nosso segundo exemplo, temos que definir que os parâmetros de entrada são ponteiros:

```
void troca(float *a, float *b)
```

Do mesmo modo como declaramos uma variável do tipo ponteiro na **aula 14**. Depois, manipulamos **a** e **b** como sempre tratamos os ponteiros. Pense um pouco e tente ver se você consegue fazer sem a minha ajuda (não leia o código abaixo neste momento; pense!). Conseguiu? Então verifique se você pensou de algum modo parecido com o meu:

```
void troca(float *a,float *b) {
float temp;
```

```
temp = *a;
*a = *b;
*b = temp;
}
```

Primeiro, declaro uma variável temporária **temp**. Depois, copio para **temp** o **valor da variável apontada** por **a**, para "dentro" da **variável apontada por a** o **valor da variável apontado** de **b**, e por fim, para dentro da **variável apontada** por **b** o **valor** de **temp**. Incluindo esta função num programa, poderíamos ter algo do tipo:

```
main(){
float a=10.5, b=17.1;
printf ( "Antes: a=%f, b=%f\n",a,b );
troca(&a,&b);
printf ( "Depois: a=%f, b=%f\n",a,b );
}
```

Compile este programa (volto a lembrar, inclua a função **troca** antes da função **main**). Ao executá-lo você verá os valores de **a** e **b** antes e, em seguida... veja só: nada em uma mão, nada na outra... num passe de mágica os valores de **a** e **b** estão trocados!

A última garfada: detalhes sobre vetores, matrizes e strings

Já reparou que quando nós usamos scanf para ler uma string nós não usamos o & antes do nome da variável? Agora você pode responder o porquê disso: uma string em C é nada mais nada menos que um vetor de caracteres. Como um vetor sempre pode ser acessado como um ponteiro (**aula 15**), as **strings são ponteiros**. Vamos exemplificar:

```
main() {
char palavra[255];
char *mesmapalavra;
mesmapalavra = palavra;
printf ( "Digite uma palavra: ");
scanf("%s",mesmapalavra);
printf ( "Palavra digitada: %s\n",palavra);
}
```

O uso do ponteiro **mesmapalavra** é, obviamente, redundante, mas serve para mostrar o que foi dito antes. Se na linha onde está escrito **scanf("%s",mesmapalavra);** eu tivesse escrito **scanf("%s",palavra);** o resultado seria exatamente o mesmo, já que eu faço **mesmapalavra** apontar para o vetor **palavra**. Em nenhum dos dois casos eu uso o &.

Como vetores e ponteiros são, de certo modo, intercambiáveis, eu posso criar uma função com passagem de parâmetros por referência usando um vetor como parâmetro de entrada. Como exemplo, podemos ter uma função para multiplicar por -1 um vetor de inteiros:

```
void neg(int vetor[], int tamanho) {
int i;
for (i=0;i<tamanho;i++)
vetor[i] = vetor[i] * (-1);
}
```

Veja que, neste caso, o parâmetro **vetor** é passado por referência, enquanto o parâmetro **tamanho** (que indica o tamanho do vetor) é passado por valor. Se eu tivesse escrito:

```
void neg(int *vetor,int tamanho)
```

Seria a mesma coisa. A chamada da função **neg**, dentro de uma rotina **main**, seria:

```
main() {
int numeros[] = { 4, -2, 31, -46, 85, 4 };
int tamanho=6;

neg(numeros,tamanho);
}
```

Exercícios

1) Considere a função **double sin(double x)**, que calcula o seno do ângulo x, incluída na biblioteca matemática padrão. O parâmetro para essa função é passado por valor ou por referência? E a função **gets(char *s)**, da biblioteca **stdio.h**, que lê o que o usuário digita no teclado e coloca na string **s**?

2) Faça uma função que, dados 3 números tipo float, **x**, **y** e **z**, faça **x = x/z**, **y = y/z** e **z = 1**. Chame esta função de **projeção**.

3) Faça uma função que transforma todos os espaços de uma string s no caracter #. Dicas: use a função strlen (aula 16) para achar o tamanho da string; percorra todos os caracteres da string até o último, verificando se cada um é o caracter ' ' (1 espaço entre ' '). Se for, troca pelo caracter '#'.

4) (Desafio) Faça uma função que inverte uma string **s**.

Isto é tudo por hoje, pessoal. Pratiquem com estes exercícios, e não se decepcionem se não conseguirem acertar de primeira. Não desistam: antes de se desesperarem, dêem uma parada, pensem um pouco, com bastante calma, e tentem novamente. Os que conseguirem acertar todos até o terceiro, muito bem! O coelho da páscoa irá presenteá-los com muitas guloseimas! Os que não conseguirem, por favor, tentem estudar um pouco mais as aulas anteriores.

Aula 19

Ola pessoal. Nesta aula começaremos a falar sobre arquivos. Abordaremos o tema de forma geral e seguiremos sempre pelo padrão C ANSI, compatilizando entre plataformas.

Espero que todos os conceitos anteriormente discutidos estejam claros. O que já falamos no curso, de forma resumida:

- Noções básicas da Linguagem, overview geral ;
- Palavras reservadas, funções e bibliotecas ;
- Comparação de construções com outras linguagens ;
- Conceitos básicos do Sistema Operacional, como funciona fluxo de dados(entrada/saída) ;
- Tipos de arquivos, bibliotecas, cabeçalhos, ambientes de desenvolvimentos e compilação de programas ;
- Noções aprofundadas de bibliotecas, formatação de um programa em C e necessidades de organização ("aprendendo a pensar") ;
- Falamos sobre o que são variáveis, tipos de dados, declaração de variáveis, globais, locais, inicialização destas e procedimento básico para escolha destas ;
- O que são funções, para que elas servem, nomenclatura, a "cara" delas, a função main, modos de chama-las e seus tipos ;
- Comandos de controle da linguagem C (if, for e while) ;
- Funções com comandos de controle ;
- Vetores, matrizes, string e seu relacionamento ;
- Ponteiros(conceitos básicos) e relação com os itens acima ;

- Ponteiros: como alocar e desalocar uma área de memória, como um ponteiros pode ser equivalente a um vetor e o que são strings dentro deste conceito ;
- Funções de strings e relacionamento com ponteiros ;
- Structs, unions e enums ;
- Funções: passagem de valor por valor e por referência, mais conceitos de vetores, strings e ponteiros ;

Bem, agora começaremos a discutir os conceitos de arquivos propriamente dito. Vamos começar falando sobre o relacionamento de streams e arquivos.

Existe um nível de programação, uma interface de abstração chamada stream onde podemos nos referenciar, independente do tipo de dispositivo real associado. O dispositivo propriamente é o arquivo.

Stream é um dispositivo lógico e comporta-se de forma semelhante para qualquer dispositivo associado. Ou seja, divisão clara: stream dispositivo lógico, arquivo dispositivo físico.

Streams texto

Uma stream de texto é uma stream que passa somente caracteres. Uma stream pode converter caracteres nova linha como retorno de carro/nova linha(CR/LF). Provavelmente não haverá relação do número de caracteres escritos(ou lidos) com o número do dispositivo externo, por esse motivo. E o inverso igualmente.

Streams Binária

Nas streams binárias existem seqüências de bytes, onde normalmente há correlação entre o número de bytes lidos(ou escritos) com o número efetivamente encontrado no dispositivo externo. Os bytes simplesmente passam e não a conversões de alguma natureza que possa interferir no processo.

Os arquivos

Os arquivos em C podem ser qualquer coisa, arquivo em disco, terminal, impressora, fitas, etc. Associa-se um arquivo a um stream quando abrimos um arquivo e devem ser trocadas informações entre ele e o programa.

Os arquivos, apesarem de serem referenciados por streams, não tem sempre os mesmos recursos, eles possui dependências com a natureza do periférico associado a ele. Por exemplo, num teclado não podemos escrever caracteres, logicamente, numa fita, dispositivo seqüencial, não podemos localizar um pedaço específico dando sua localização randômica. Fica claro ai a natureza do dispositivo interfere diretamente em seu comportamento, entrada/saída, seqüencial/randômico, etc.

Para desassociar um arquivo a uma stream devemos fechar o arquivo explicitamente. Caso um arquivo for fechado e houver alguma coisa associada a sua stream que não tenha sido efetivamente escrita no seu arquivo, haverá um processo de flushing(descarga), que encerrará a operação, limpando o buffer e jogará todo "resto", evitando que o pedações de informações sejam perdidos. Quando um programa fecha de forma não regular, os dados não são gravados e são perdidos.

Cada stream associada associada a um arquivo tem uma estrutura do tipo FILE. Essa estrutura é definida no cabeçalho STDIO.H.

Essa divisão de streams e arquivos no C, foram criadas basicamente com a utilidade de criação de uma camada de abstração, uma interface que facilite a programação e deixe o programador simplesmente preocupado com o que deve ser feito, e não necessariamente como isto está sendo efetivamente acontecendo.

Funções básicas de manipulação de arquivos

As funções básicas de manipulação de arquivos se encontram no cabeçalho STDIO.H. Vale ressaltar as funções chamadas pela letra f são as padronizadas pela ANSI e as que não começam por tal letra são as do UNIX.

Estrutura FILE

Utilizamos um ponteiro do tipo FILE, para nos referenciar ao arquivo físico que estamos abrindo, seria a stream. Ele contém informações como nome, status, posição atual, permissões, dono, etc. Chamamos este ponteiro de descritor de arquivo e sempre o utilizamos quando nos referenciamos ao arquivo a ele associada. Conseguiram observar? Agora, sempre quando quisermos por exemplo ler um arquivo, utilizamos o ponteiro, que é a stream, que está associada ao arquivo físico real.

Na declaração temos: FILE *fp; /* fp = file pointer */

fopen

A função fopen associa a um ponteiro do tipo FILE um arquivo físico, ou seja, faz a associação físico<=>lógico.

O protótipo dela é : FILE *fopen (const char *path, const char *mode);

Onde path é um ponteiro para uma string que contém o nome do arquivo, seja ele no diretório correntes ou com caminhos alternativos. Já o ponteiro da string de mode, diz como o arquivo será aberto (ex.: escrita, leitura, append). A função devolte um ponteiro do tipo FILE, ou seja, tudo se encaixa.

Podemos exemplificar com:
```
FILE *fp;
if((fp = fopen("test","w")) == NULL){;
printf("Erro ao abrir arquivo!");
exit(1); /* devolvendo erro ao SO */
}
```

Modo de abertura de arquivos: r: Abre um arquivo para leitura w: Abre um arquivo para escrita a: Abre um arquivo para apendar, nos eu final

Podemos inserir o qualificador **b**, em qualquer das opções e a operação será efetivada em modo binário, caso contrário usará o conceito de streams texto. Também podemos inserir o qualificador +, que complementa o sentido da operação, por exemplo w+ eqüivale a leitura e escrita, assim como r+.

Alguns detalhes interessantes. Se você abrir um arquivo com permissão de escrita somente, se o arquivo já existir, ele será sobreposto (ou seja, apagará o velho e criará um novo). Se um arquivo for aberto para operação de leitura/escrita, ele será criado caso não exista e utilizado normalmente caso já exista.

fclose

É necessário se desassociar a stream ao arquivo físico ao se terminar um programa. Utilizamos da função fclose para isso. Ele ainda descarregar obuffer(como citado acima) e garante que o arquivo foi fechado de forma correta, pelo menos pelo programa. Uma falha fechando uma stream pode causar vários danos a integridade dos dados. Além disso, ele libera o ponteiro para o tipo FILE associado ao arquivo, que poderá ser utilizado para outros arquivos.

O protótipo é: int fclose(FILE *stream);

A stream citada acima é o óbvio ponteiro para o tipo FILE. Repare que a função retorna um inteiro, que significa o código da operação. Caso retorno 0, significa como tradicionalmente, que a operação foi bem sucedida. Caso contrário houveram erros, como disco cheio, corrompido ou inexistente. Além de diversos outros problemas que podem ter

ocorrido. Vamos nos aprofundar nisso depois.

Voltando..

Bem pessoal, a aula foi um tanto quanto teórica e introduziu conceitos bastante novos. É bem importante que os entendam. Na próxima aula falaremos de funções para manipulação de arquivos. Começaremos depois falando sobre as funções básicas EM UNIX, estou realmente pensando em começar a aprofundar nessa área. Estou bem inclinado e quero o feedback de vocês, assim como dúvidas e sugestões para fechamento desse módulo, que estará ocorrendo nas próximas semanas. É isso aí pessoal, até a próxima.

Aula 20

Fala pessoal, tudo bem? Como foi o feriadão? Estudaram muito? :)

Bem, na aula passada citamos um histórico do curso. Gostaria da opinião do pessoal, em relação a um listão de exercícios, com respostas, para que possamos avaliar o conteúdo no final do curso, o que acham? Em primeira mão, em breve, estou "pensando" em dar um curso com acompanhamento intensivo. O que é isso? Eu marcaria exercícios, corrigiria e teríamos aulas para tirarmos dúvidas e trocarmos idéias.

Além disso, disponibilizar o material de forma organizada, exercícios e as tão esperadas provas. Também um livro "oficial" de consulta. Bem, acredito que a demanda seja alta, eu já conheço mais ou menos quem costuma participar, então dêem sua opinião. Mas lembrando, isso é só uma idéia, e um piloto, não sei se será possível (em termos de tempo, grana, etc.) a execução de tal projeto.

Outra coisa que queria perguntar também (isso aqui tá parecendo bate papo não!?), que tema vocês acham interessante para o próximo curso? Vou deixar essa discussão em aberto nas próximas semanas. Estava pensando em programação avançada para ambiente Unix, onde teria muitas coisas cascas, realmente a minha preferida. Também poderia mudar um pouco, e dar um curso de perl, banco de dados, e depois noções de xml, é interessante e o mercado esta pedindo muitíssimo (para quem não conhece, seria um programador voltado para web). Uma opção seria discutir estrutura de dados em C, seria também meio que a continuação natural deste curso, algoritmos, otimização, etc. Outra, programação gráfica, ou xlib, ou gtk. Bem, e tem algumas outras coisas, que eu vou pensando mais para frente, afinal, este curso ainda não acabou!

Bem, vamos continuar a parte de arquivos nesta aula, pelo que eu recebi de feedback, por enquanto "está tranqüilo", então vamos continuar neste ritmo. Falamos de arquivos físicos, streams, texto e binária, a estrutura do tipo FILE e abertura/fechamento de arquivos.

Vamos falar agora propriamente dita da manipulação dos arquivos em C, leitura e escrita.

Escrevendo em um arquivo

Para escrevermos em um arquivo em C, utilizamos as funções fputc e putc. Eles escrevem na stream, um caracter por vez. O protótipo é idêntico:

int fputc(int c, FILE *stream);

As duas funções fazem exatamente a mesma coisa, só que uma, a putc(), deve ser implementada em termos de macro, para manter compatibilidade. As versões antigas utilizavam o putc(), e para manter tudo funcionando, em novas implementações se fez isso. Caso alguém tenha algum problema com isso, me de um toque. Não acredito, mas, tem

maluco para tudo.

Lendo um caracter de arquivo

Igualmente, definido pelo padrão C ansi, temos as funções getc e fgetc. O protótipo é:

int fgetc(FILE *stream);

Igualmente a dupla fputc e putc, a putc é implementada como macro. Elas servem para ler caracteres da stream passada, que será o retorno da função. Caso chegue ao final do arquivo retornará EOF (End of File) ou erro.

Também vale relembrar, que a função getchar() é equivalente a getc(stdin), ou seja, lê caracteres da entrada padrão. Já putchar(), é equivalente a putc(caracter, stdout), ou seja, escreve no dispositivo padrão de saída.

As funções citadas fazem parte da stdio.h, e podem ser utilizadas de forma complementar, ou até com outras funções de streams. Essa é uma das características que tornam o C uma das linguagens mais versáteis e poderosas do mundo (em minha opinião a mais, mas isso é detalhe).

Bem, exemplificando eu vou disponibilizar este fonte, peguem e testem. Outro, peguem aqui, dêem uma lida nestes exemplos, eles estão muito ilustrativos.

Bem, agora falaremos de leitura de arquivos de forma mais geral, através de strings.

Escrevendo em um arquivo linha a linha

A função fputs() é igual a função putc, só que em vez de trabalhar caracter a caracter, ela trabalha linha a linha, geralmente uma situação de conforto para o programador. O protótipo dela segue:

int fputs(const char *s, FILE *stream);

Repare que passamos um ponteiro para char, onde passaremos a string (a linha) e especificamos qual stream estamos utilizando. Note que é idêntica a função puts, só que deixa especificar a stream, ao contrário da puts que utiliza stdin. protótipo: int puts(const char *s);

Lendo arquivo linha a linha

Você também, analogamente, pode ler arquivos linha a linha. Repare o protótipo da função:

char *fgets(char *s, int size, FILE *stream);

Ela retorna um ponteiro para string recebida, ela para até achar o fim da linha ou tiver alcançado o tamanho de size-1. Stream especifica de onde está sendo lido, e o retorno vai sempre para o ponteiro que está recebendo. Caso tenha sido encontrado o final de linha ou EOF, o \0 é armazenado depois do último caracter lido.

Estou disponibilizando os dois programinhas, pegue-os e de uma olhada, tem muitíssimas dicas, arquivo 1 e arquivo 2.

Voltando no arquivo..

Bem, sabemos que existe um descritor de arquivo, a tal estrutura do tipo FILE, definida no cabeçalho stdio.h. Bem, nesta descrição, quando é montada a estrutura em memória, existe um de seus campos que indica a posição onde o ponteiro está posicionado para leitura/escrita. É assim que fazemos para ler caracteres e escreve-los. Existe a função chamada fseek, que existe para reposicionar o indicador de posição do descritor de string. Isso mesmo, ela só muda para onde o ponteiro está direcionado. Você pode correr ele para o início do arquivo, final, ou uma posição definida por

sua pessoa mesmo.

O protótipo da função é: int fseek(FILE *stream, long offset, int whence);

O que isso quer dizer? Bem, você passa qual stream está trabalhando, o descritor logicamente, passa quantos bytes deseja percorrer a partir de uma posição predeterminada. Ou seja, o campo whence, você passa onde que é a posição base para início de calculo, por exemplo, existem a possibilidade de passar as macros: SEEK_SET, SEEK_CUR e SEEK_END, que você posiciona no início do arquivo, posição corrente ou no final do arquivo. Aí você passa quantos bytes deseja trabalhar a partir dessa posição. O retorno da função é 0 caso tenha obtido sucesso e diferente caso não o tenha.

Existem as funções ftell() (protótipo: long ftell(FILE *stream);), que retorna justamente onde está posicionado o indicador de posição da stream e a função rewind() (protótipo: void rewind(FILE *stream);) que volta ao início do arquivo, é como um (void)fseek(stream, 0L, SEEK_SET).

De uma olhada nesse exemplo simples para se ter uma idéia de como funciona.

Bem, foi bastante conteúdo esta aula, espero que tenham gostado, qualquer dúvida estamos aí. Principal pessoal .. vejam os exemplos, eles estão muito fáceis realmente, são os clássicos de qualquer aula de programação desta matéria e tem de ser vistos. A próxima aula continuamos, alguns detalhes vou explicar por lá..

Aula 21

Olá amigos! Estou aqui novamente para escrever-lhes mais algumas linhas em nosso curso de programação. O Elias, para variar, vai ficar tranqüilo, sem nada para fazer... estou brincando, hein Elias? Ele, bem enrolado como está (é?), me deixou encarregado de escrever hoje para vocês.

Como quem gosta de ficar parado é lagarto, e quem anda para trás é caranguejo, vamos dar mais uns passos à frente no assunto iniciado há duas aulas. Já sabemos como abrir e fechar um arquivo e algumas funções relacionadas à sua leitura e escrita, trabalhando caracteres e linhas individualmente. Nesta aula, vamos ver mais algumas funções para acesso e verificação do estado de arquivos, e daremos uma polida em nossos conceitos.

Mantendo tudo sob controle

Vamos relembrar: um arquivo em C é uma stream, um "fluxo" de dados. Como tudo em computação, uma stream possui um tamanho finito. Se efetuarmos diversas leituras nela, indo da primeira posição em frente, alcançaremos um ponto chamado EOF, que é o fim do arquivo (em inglês, End Of File). Deste ponto, só podemos retroceder; nem pense em ler mais alguma coisa do arquivo! E como sabemos que chegamos ao fim do arquivo? Simplesmente perguntando. Na biblioteca padrão temos a função feof que nos retorna se o arquivo já terminou. O protótipo dela é:

int feof(FILE *stream)

Ela retorna 0 se o arquivo não chegou ao fim ou 1 caso contrário (lembre que em C o valor inteiro que corresponde a uma afirmação falsa é 0 e uma afirmação verdadeira corresponde a um inteiro diferente de zero).

Por exemplo, para determinar se chegamos ao final da stream apontada pela variável Cadastro (que já deverá ter sido aberta com fopen), podemos fazer o seguinte teste:

```
if (feof(Cadastro))
printf ("Fim do arquivo");
```

Aqueles que são bons observadores, já devem ter notado que os exemplos dados na última aula já fazem uso desta função (propositalmente). Sugiro a todos que dêem mais uma olhada neles.

Uma outra função que é útil para verificar como está uma stream é a ferror. Novamente, como tudo em computação, um acesso a uma stream pode ser efetuado com sucesso, ou pode falhar (pessimistas acham que a probabilidade de falhar é maior do que a de acertar). Para ter certeza de que eu não tenha resultados imprevistos na hora em que um acesso der errado, eu testo o indicador de erro da stream logo após tê-la acessado. Se for 0, não há erro; se for diferente de zero, "Houston, temos um problema". O protótipo de ferror é:

int ferror(FILE *stream)

Vamos dar uma olhada em um pedacinho de código só para fixarmos esta nova função. Neste exemplo, a stream apontada por Lista já deverá estar aberta, e Item é uma string:

```
fgets(Item, 40, Lista);
if (ferror(Lista)) {
printf ("Apertem os cintos: ocorreu um erro no último acesso!");
exit(1);
}
```

Note bem: o indicador de erro poderá mudar se for feito uma nova leitura ou escrita antes de testá-lo. Se quiser testar, teste logo após executar um acesso.

Escreveu, não leu...

Já notou que as funções fgets e fputs se parecem muito com as suas primas gets e puts, sendo que estas lêem do teclado e escrevem na tela e aquelas fazem o mesmo, só que relativamente a streams? Seria bom se nós tivéssemos algo do tipo printf e scanf para arquivos; e nós temos! Usando fprintf e fscanf nós fazemos o mesmo que já fazíamos na tela, mas agora faremos em streams. Veja como elas estão definidas nas bibliotecas padrão:

fprintf (FILE *stream, char *formato, ...)

fscanf (FILE *stream, char *formato, ...)

Na prática, podemos usar estas duas funções que atuam em streams do mesmo modo como usamos suas primas, já velhas conhecidas nossas, mas acrescentando antes o ponteiro para a stream. Assim, se eu tinha um programa que lia do teclado dados digitados pelo usuário, posso fazer uma mudança para ler de um arquivo. De modo similar, isso também vale para a tela. Por exemplo, sejam dois arquivos apontados pelas variáveis Saída (aberto para escrita) e Dados (aberto para leitura):

se eu tinha: scanf("%d",&numero);
posso fazer: fscanf(Dados,"%d",&numero);

se eu tinha: printf("Tenho %d unidades do produto: %s\n",quant,prod);
posso fazer: fscanf(Saida,"Tenho %d unidades do produto: %s\n",quant,prod);

Experimente usar isto nos programas que você já fez, só para praticar: ao invés de usar de scanf e printf, leia dados com fscanf de um arquivo e escreva-os em um outro arquivo com fprintf. O arquivo gerado poderá ser aberto inclusive em editores de texto, e você poderá ver o resultado. Mas ATENÇÃO: fprintf e fscanf devem ser aplicados a streams texto (ou seja, abertas sem o qualificador b, explicado na **aula 19**).

Até agora, todas as funções que lêem e escrevem em um arquivo se referem a streams texto. É uma boa hora para se questionar a utilidade das tais streams binárias (**aula 19**).

Podemos querer ler e/ou escrever uma stream tendo controle total dos bytes dela, para, por exemplo, copiar fielmente

um arquivo, sem alterar nada nele. Assim, abriremos o arquivo usando o já mencionado qualificador b. E para acessar os dados, teremos duas novas funções, fread e fwrite, definidas da seguinte forma:

size_t fread(void *Buffer, size_t TamItem, size_t Cont, FILE *Fp)
size_t fwrite(void *Buffer, size_t TamItem, size_t Cont, FILE *Fp)

O tipo size_t é definido na biblioteca STDIO.H e, para simplificar, podemos dizer que eqüivale a um inteiro sem sinal (unsigned int). Lembre-se de que void* é um ponteiro qualquer, ou seja, um ponteiro sem tipo. Assim, eu posso usar um int*, char*, ...

A função fread opera do seguinte modo: lê da stream apontada pela variável Fp tantos itens quantos forem determinados pela variável Cont (cada item tem o tamanho em bytes descrito em TamItem) e coloca os bytes lidos na região de memória apontada por Buffer. A função fwrite funciona de modo similar, porém gravando o que está na região de memória apontada por Buffer na stream Fp.

As duas funções retornam o número de bytes efetivamente lidos ou escritos do arquivo, respectivamente.

Para determinar o tamanho em bytes de um determinado tipo de dado que queremos ler ou gravar, é possível usar o operador sizeof. Como exemplo, podemos querer gravar na stream binária apontada pela variável Dados o valor da variável inteira X:

fwrite(&X,sizeof(int),1,Dados);

O tamanho de um inteiro é determinado por sizeof(int). Note que o número de bytes que foram efetivamente gravados, retornado pela função, pode ser desprezado (e geralmente o é).

Seja a struct Pessoa definida por (lembra-se das aulas sobre struct?):

```
struct Pessoa {
char nome[40];
int idade;
};
```

Definamos a variável Aluno do tipo struct Pessoa. Podemos ler um registro de um aluno de um arquivo com o comando:

fread(&Aluno,sizeof(struct Pessoa),1,Dados);

Um exemplo de utilização das funções fread e fwrite pode ser encontrado neste arquivo (aula3.c).

Desviando o curso dos acontecimentos

Finalizando a nossa coluna, apresento a vocês uma função que poderá ser muito útil para "automatizar" programas. A função freopen redireciona as streams padrão (stdin, stdout, stderr) para um arquivo qualquer que nós quisermos designar. O seu protótipo é:

FILE *freopen(char *NomeArq, char *Mode, FILE *stream)

Onde NomeArq é uma string contendo o nome de um arquivo a ser associado à stream. A string Mode indica como o arquivo será aberto, e segue os mesmos moldes de fopen. A função retorna NULL se ocorrer algum erro no redirecionamento. Assim, se quisermos redirecionar a saída padrão para o arquivo saida.txt em um programa qualquer, podemos adicionar a seguinte linha no início do programa:

freopen("saida.txt","w",stdout);

A "automatização" de um programa pode ser feita se redirecionarmos a entrada padrão (stdin) para um arquivo que designarmos. Seja um programa que leia dados do teclado para uma finalidade qualquer, podemos fazer, no início do

programa:

freopen("meuarq.dat","r",stdin);

Se criarmos o arquivo meuarq.dat em um editor de texto qualquer, podemos colocar nele os dados que seriam esperados que o usuário do programa digitasse.

Espero que possamos continuar este assunto em outras oportunidades, já que desejo passar para vocês exercícios para praticar estas funções e conceitos apresentados hoje, mas isso fica para uma outra vez. Por hora, analisem o que foi apresentado hoje e procurem aplicações onde possamos utilizar estes conhecimentos. Sentir-me-ia muito grato se houver algum retorno (e-mails) de vocês. Por favor, digam-me as suas dúvidas e suas opiniões sobre esta coluna (aí eu escreverei mais...).

Aula 22

Como vocês devem ter percebido o Curso de C está quase terminando. Por isso esta semana estarei ensinando a usar uma ferramenta muito importante para programação: o **make**.

Antes de falar dele vejamos uma opção do compilador GCC que nos irá ser útil.

Gerando Objetos

Para gerar objetos com o gcc usamos a opção **-c**. Por exemplo:
$ gcc -c programa.c

A linha acima irá gerar o arquivo **programa.o**. Este é o objeto gerado pela compilação do arquivo **programa.c**. No entanto, tal objeto não é executável. Veremos no **make** qual sua utilidade.

Make e Makefile

O **make** é uma ferramenta muito importante quando se faz um projeto de programação. Sua finalidade é diminuir o tempo de compilação quando já se compilou o programa pelo menos uma vez. Isto é feito compilando-se apenas o que foi alterado, sem precisar recompilar o programa todo de novo.

O arquivo que o **make** usa para saber o que compilar e como é o **Makefile**. Este arquivo deve ser feito pelo programador. E é isso que irei ensinar a fazer: arquivos **Makefile**.

A estrutura de um arquivo Makefile é a seguinte:

[CONSTANTES]
all: [DEPENDÊNCIAS]
[TAB} [COMANDO PARA COMPILAR}

[SEÇÃO]: [DEPENDÊNCIA]
[TAB} [COMANDO PARA COMPILAR]

Analisando:

[TAB] é simplesmente a tecla TAB. Ou seja, é preciso colocar [COMANDO PARA COMPILAR] como um parágrafo novo.

[COMANDO PARA COMPILAR] é o que deve ser executado para compilar o programa ou parte dele.

[SEÇÃO] é uma parte do Makefile que pode ser executada. A seção **all** é especial e deve sempre existir no Makefile. Quando se executa o **make** ele procura esta seção.

[DEPENDÊNCIA] são os arquivos que o [COMANDO PARA COMPILAR] precisa que exista (dependa), ou seções que precisam ser executadas antes da atual.

[CONSTANTES] são variáveis constantes que você define para utilizar dentro do Makefile, que podem aparece no [COMANDO PARA COMPILAR]. Estas constantes podem ser variáveis do ambiente.

Vejamos um exemplo bem simples, onde temos os arquivos **inclui.h** e **teste.c**:

inclui.h:
```
void imprime() {
printf("Isto é só um teste\n");
}
```

teste.c:
```
#include <stdio.h>
#include "inclui.h";

main() {
imprime();
}
```

Makefile:
```
all: inclui.h
[TAB] gcc teste.c -o teste
```

Neste exemplo, o programa teste.c apenas chama a função imprime() que está em inclui.h. Para compilar, basta deixar estes 3 arquivos no mesmo diretório e executar:
```
$ make
```

Para rodar:
```
$ ./teste
```

Vamos analisar este Makefile:

Desmembrando a seção **all**, neste caso, vemos que:
[DEPENDÊNCIA] = inclui.h
[COMANDO PARA COMPILAR] = gcc teste.c -o teste

Ou seja, a seção **all** depende do arquivo **inclui.h**. Isto quer dizer que se alterarmos o arquivo **inclui.h** (a data deste, salvando-o, por exemplo), ao executarmos o make novamente ele irá perceber esta mudança e sabendo que a seção **all** depende deste arquivo irá executar o [COMANDO PARA COMPILAR} desta seção.

E tudo um jogo de dependência. Vejamos a seguir este exemplo um pouco modificado.

Se quisermos criar uma seção para a dependência do arquivo **inclui.h** e criar uma [CONSTANTE] que define o nome do executável fazemos o Makefile assim:

Makefile:
```
NOME=programa
all: parte2
```

parte2: inclui.h
[TAB] gcc teste.c -o $(NOME)

Ao executar o **make**, ele irá ver a dependência do **all** (parte 2) e irá executá-la. Ou seja, será executado:

gcc teste.c -o programa

Isto porque definimos a constante **NOME** como sendo **programa**. Assim sendo, o nome do arquivo executável agora é **programa**. Para executá-lo, fazemos:
$ ./programa

Usando Objetos no Makefile

Vimos na primeira página deste artigo como gerar objetos. Para que servem? Servem exatamente para diminuir o tempo de compilação, pois objetos são arquivos C já compilados, mas não executáveis. Vejamos um exemplo para entender o que isto quer dizer.

Imagine que temos um projeto de um jogo. Nele temos os arquivos:

cor.c, que contém a paleta de cores e funções de manipulação de cores.
video.c, que contém as funções de controle de vídeo.
timer.c, que contém funções que ativam um cronômetro.
eventos.c, que contém funções que controlam eventos como tecla pressionada, ou botão do mouse pressionado, etc.
audio.c, que contém funções de controle de áudio.
ia.c, que contém funções de inteligência artificial.
main.c, que contém o programa principal.

O mais correto neste projeto é criar vários objetos e depois unir tudo em um único executável. Fazemos isso criando um Makefile para o projeto. Este Makefile pode ser assim:

Makefile:
LIBS=-lX11 -lm
INCLUDES=/usr/X11R6/include

all: cor.o, video.o, timer.o, eventos.o, audio.o, ia.o, main.o
[TAB} gcc cor.o video.o timer.o eventos.o audio.o ia.o main.o -o jogo $(LIBS) $(INCLUDES)

cor.o: cor.c
[TAB] gcc -c cor.c $(INCLUDES)

video.o: video.c
[TAB] gcc -c video.c $(INCLUDES)

timer.o: timer.c
[TAB] gcc -c timer.c $(INCLUDES)

eventos.o: eventos.c
[TAB] gcc -c eventos.c $(INCLUDES)

audio.o: audio.c
[TAB] gcc -c audio.c $(INCLUDES)

ia.o: ia.c
[TAB] gcc -c ia.c $(INCLUDES)

main.o: main.c
[TAB] gcc -c main.c $(INCLUDES)

Neste exemplo, vemos que temos uma seção para cada arquivo C. Cada uma destas gera o objeto respectivo do seu fonte. Por exemplo a seção **cor.o** depende de **cor.c** e gera o arquivo **cor.o** ao executar seu [COMANDO PARA COMPILAR}.

Ao executarmos:
$ make

Ele irá procurar a seção **all** e irá executar suas seções dependentes que foram alteradas. Como nada foi compilado ainda ele irá executar todas as seções dependentes. Fazendo isso, são gerados os objetos: cor.o, video.o, timer.o, eventos.o, audio.o, ia.o e main.o.

Agora que a seção **all** tem todas suas dependências executadas, ele executa o próprio **all**, ou seja:

$ gcc cor.o video.o timer.o eventos.o audio.o ia.o main.o -o jogo $(LIBS) $(INCLUDES)

O que essa linha acima faz na verdade é juntar todos os objetos em um executável, que é o arquivo **jogo**.

Agora, imaginemos que eu alterei o cor.c para adicionar uma função a mais, ou alterar alguma variável ou função. Se eu rodar o **make** ele irá notar que só o arquivo **cor.c** foi alterado (pela data) e irá executar apenas a seção **cor.o**. Ou seja, irá criar apenas o arquivo objeto **cor.o**. Logo em seguida executará a seção **all**, juntando novamente todos os objetos no arquivo executável**jogo**.

Note que isto facilita a compilação do projeto, pois agora eu só preciso compilar o que foi realmente alterado. Pode parecer inútil, porém imagine um jogo que demore horas para compilar todo. O programador não vai tentar compilar ele completamente toda vez que alterar um detalhe no programa. É muito mais fácil quebrar em objetos usando o **make**, que compila somente o objeto necessário.

Detalhes

Alguns detalhes para geração de objetos:

Pode-se executar uma das seções sem que as outras sejam executadas, assim:

$ make [SEÇÃO]
Exemplo:
$ make cor.o

Sempre que fizer um arquivo C que depende de uma função de outro, use o comando extern.
Exemplo:

inclui.c:
```
void imprime() {
printf("Teste\n");
}
```

teste.c:
```
#include <stdio.h>
extern void imprime(void);

main() {
imprime();
}
```

Note que neste caso o teste.c depende do inclui.c. Então o Makefile ficaria:
Makefile:
all: teste.o inclui.o
[TAB] gcc teste.o inclui.o -o teste

teste.o: teste.c inclui.o
[TAB] gcc -c teste.c

inclui.o: inclui.c
[TAB] gcc -c inclui.c

Caso exista arquivos header (com extensão .h) deve-se adicioná-los como dependências também.

Perceba que o Makefile pode ser usado para outros fins que não o de compilar um programa. Por isso, normalmente em projetos encontramos no Makefile uma seção chamada **install** que copia os arquivos necessários para determinados diretórios (normalmente, subdiretórios de /usr/local).

Veja um exemplo da seção **install** para o exemplo do jogo:

install:
[TAB} cp jogo /usr/local/bin

Ou seja, esta seção copia o arquivo **jogo** para /usr/local/bin, que está no path.

Concluímos aqui o artigo desta semana. Semana que vem provavelmente falaremos do GDB, debuggador (depurador) de arquivos executáveis, que ajuda a encontrar onde acontecem problemas no programa que geram uma mensagem de: Segmentation Fault. Depois disso, terá uma série com vários exercícios, com a possibilidade de você testar seus conhecimentos. Estou para combinar com o Elias, veremos ainda...

Aula 23

Saudações, amigos! Estou de volta à nossa coluna hebdomadária para dar uma última revisada nas streams em C, nossas velhas conhecidas. Daqui a poucas semanas, estaremos colocando no ar uma lista com exercícios, e uma prova para vocês avaliarem os seus conhecimentos.

Na última aula sobre streams, a de **número 21**, houve um equívoco na hora de colocar a página no ar, o que acarretou em um link quebrado para o arquivo aula3.c (obrigado por avisar, Denny!). Estou pedindo ao Elias que conserte isso na página, mas parece que ele está meio enrolado com seus compromissos... Correm boatos de que ele está trancado há três dias, sem comer ou dormir, em um escritório no Rio de Janeiro, com um laptop no colo. ô Elias, sai daí rapaz! Qualquer dia desses você vai acabar tropeçando na sua própria barba.

Deixando um pouco de lado as brincadeiras, vamos ao que interessa?

Relembrar é viver

Quem se lembra de todos os conceitos para manipular arquivos em C levanta o dedo! Quem não se lembra, deve olhar as aulas de número **19**, **20** e **21**. De qualquer modo, aí vai uma "cola" do que nós já vimos:

ARQUIVO EM C = STREAM Streams texto ou binárias Descritor de arquivo: FILE * Abrir arquivo: FILE *fopen(char *nomearq, char *modo) Fechar arquivo: fclose(FILE *arq) Ler, escrever: fgets, fputs, fread, fwrite, ...

Andando numa stream: fseek, rewind

Colocando em prática

Vamos começar a nossa malhação mental desde já. Vamos fazer um pequeno editor de texto, que não chega aos pés do StarWriter. O código fonte está aqui (editor.c).

Começamos definindo o descritor do arquivo: FILE *arquivo.

Na função main, abrimos a stream usando fopen no modo "a+", que quer dizer que iremos adicionar mais dados a um arquivo texto (a de Append). Se o arquivo especificado não existir, ele será automaticamente criado.

Esperamos que o usuário faça uma opção (função Escolhe). Se for a primeira opção, chama a função MostraArq(), que exibe na tela o arquivo. O primeiro passo desta função é mover o indicador de posição do arquivo para o início usando rewind. Isto é necessário porque, quando abrimos um arquivo com o modo "a", o indicador do arquivo é posicionado ao seu final. Depois, lemos linha a linha até o final, quando feof deverá retornar um valor não nulo, saindo do laço de repetição. Então, movemos o indicador de posição do arquivo para o final. Observe que isto não seria necessário, porque teoricamente lemos o arquivo até o fim. Digo "teoricamente" porque se tentarmos abrir um arquivo que não esteja em formato texto, ele pode não ser lido até o fim.

Se o usuário optar por adicionar dados ao arquivo, é chamada a função EscreveArq. Como o indicador de posição do arquivo já está no final, o programa aguarda que o usuário digite uma linha de texto, e grava esta linha no arquivo. Há um pequeno macete nesta função, que é explicado em um comentário no fim do programa. Simples, não?

Com este programa, poderíamos criar até uma pequena lista de pessoas, com nomes, telefones, endereços e idade, fazendo assim uma mini agenda. É só adicionar os dados em ordem. Para consultar a lista, é só selecionar a opção "1-Ver Arquivo". Um arquivo destes seria mais ou menos assim:

Fulano de Tal
5555-3124
Rua da Moca, 171
32

Beltrano da Silva
333-3333
Rua da Conceição, 65 apto. 804
16

Só teríamos um problema: na hora de acessar o cadastro de uma pessoa qualquer, teríamos que percorrer os cadastros de todas as pessoas anteriores (assim, para sabermos o telefone de Beltrano da Silva, teríamos que ler o arquivo do início até a posição onde está o seu cadastro, entendeu?). Podemos resolver este problema? Claro que sim. Faremos isto na próxima seção.

Mini-agenda usando arquivo binário

O problema mencionado anteriormente fica resolvido se alterarmos a organização do arquivo. Mas, primeiro, vamos apresentar alguns conceitos.

Lembra-se da época em que quase tudo era feito manualmente, e quando as pessoas queriam, por exemplo, marcar uma consulta com um médico, tinham que preencher uma ficha de papel com seus dados, como nome, endereço, telefone, etc? Então você entregava a ficha para o médico (ou a secretária) e ele colocava-a numa gaveta, junto com outras.

Agora, voltemos para a época atual. Os seus dados seriam armazenados num arquivo em um computador, e não mais na gaveta. Ao invés de preencher um ficha, você digita os seus dados, e eles são armazenados num REGISTRO. O seu nome, telefone e endereço, estão contidos no registro, e são chamados CAMPOS.

Resumindo, podemos dizer que um registro é formado de campos. Um arquivo pode conter um ou mais registros. Em C, nós podemos definir um registro como sendo uma struct. Assim, um registro para uma pessoa poderia ser definido do seguinte modo:

```
struct pessoa{
char nome[40];
char endereco[80];
char telefone[15];
int idade;
}
```

Todos os registros ocupam partes da memória do computador de mesmo tamanho. Se nós quisermos saber exatamente o tamanho do registro de uma pessoa, nós usamos o seguinte pedaço de código:

```
tamanho = sizeof(struct pessoa);
```

Se os registros forem gravados em um arquivo, o tamanho do arquivo será um múltiplo do tamanho dos registros (número de registros vezes o tamanho do registro). Assim, se eu quiser ler, por exemplo, o terceiro registro de uma pessoa de um arquivo, é só adiantar o indicador do arquivo 3 registros a partir do início, ou seja, andar adiante **3 vezes o tamanho do registro**. Em C:

```
fseek(arquivo,3*sizeof(struct pessoa),SEEK_SET);
```

Para que possamos manipular o arquivo com esta facilidade, é melhor que este arquivo seja aberto no modo binário (modo "rb+"). Deste modo, usamos fwrite para gravar os registros e fread para ler.

Sabendo de tudo isso, fica mais fácil entender o código fonte do programa. Pegue-o aqui (agenda.c).

Exercícios

1) Melhore o programa editor.c para que o usuário possa incluir ao invés de 1 linha, 3 linhas de cada vez. Peça que o usuário digite primeiro um nome, depois o telefone, e por fim um endereço (você pode fazer três gets: gets(nome), gets(telefone), gets(endereco)). Grave com fprintf os dados no arquivo. As variáveis nome, telefone e endereço devem ser strings de tamanho 40, 15 e 80 caracteres, respectivamente. Chame esse programa melhorado de listanomes.c

2) Melhore o programa listanomes.c do exercício anterior para que a visualização do arquivo seja executada de 3 em 3 linhas, aguardando que o usuário pressione Enter após a exibição de cada 3 linhas.

3) Melhore o programa agenda.c para que ele possa exibir os registros de pessoas que sejam mais velhas do que uma idade que o usuário digitar. É só percorrer o arquivo do primeiro ao último registro verificando se o campo uma pessoa.idade é maior do que a idade escolhida, exibindo o registro caso verdadeiro.

4) Melhore o programa agenda.c para apagar um registro qualquer do arquivo. Para isto, peça que o usuário escolha um registro, troque o registro a ser apagado com o último registro e chame a função ApagaUlt.

Enfim, chegamos ao final desta aula. Hoje o dia foi bem puxado, e espero que vocês consigam fazer todos os exercícios. Se tiverem algum problema intransponível com os programas, enviem-me uma mensagem que eu responderei o mais rápido possível ou coloquem-na no fórum (totalmente reformulado por sinal, bom trabalho).

Se vocês conseguirem melhorar o programa agenda.c - exercícios 3 e 4 - vocês terão feito nada mais nada menos que um GERENCIADOR DE BANCO DE DADOS rudimentar. Viram como vocês conseguem ir longe com o que vocês já sabem?

Aula 24

Olá amigos! Com esta aula de hoje estaremos terminando o nosso curso básico de C(só faltará mais uma aula). Esperamos que vocês tenham gostado das aulas e que tenham aprendido a base da programação nessa linguagem. Mas não precisa chorar; o curso terminou(falta só mais uma), mas nós continuamos disponíveis para eliminar eventuais dúvidas que ainda possam existir.

O Elias falou-me que infelizmente não poderia comparecer para esta aula porque ele teria um compromisso urgente, mas eu acho que, na verdade, ele não escreveu esta aula porque ele deve ficar emocionado em despedidas.

Para fechar com chave de ouro, hoje eu apresentarei para vocês algumas informações que serão muito úteis para organizar programas, permitindo que sejam feitos sistemas de forma estruturada e com melhor legibilidade.

A importância de um código legível

Quando se desenvolve um programa em qualquer linguagem, é extremamente importante que ele seja feito de modo que, ao se examinar o código fonte posteriormente, ele possa ser compreendido da melhor forma possível. Já imaginou se você precisa corrigir um erro em um programa que você escreveu há 6 meses, do qual você já nao se lembra mais dos detalhes de implementação, e você não consegue entender quase nada do código porque foram usadas construções do tipo:

```
*(++q)=(**(p++)==0)?x&&y:x&&z;
```

Por favor, não me pergunte o que isso significa, porque eu não sei! E o mais incrível é que essa linha é sintaticamente CORRETA em C. No momento que uma pessoa escreve uma linha de código dessas, o funcionamento desse código parece extremamente lógico e previsível, mas depois de um tempo, nem ela própria entende o que se passa. Imagine então se o coitado que tiver de alterar um programa desses não for o próprio autor...

Lembre-se sempre disto: um código bem escrito, com espaçamento adequado, nomes de variáveis e de funções bem sugestivos e outros recursos que permitam entender o funcionamento do programa é muitas vezes mais importante que um código compacto. Vamos então à algumas regras de ouro da programação legível.

Espaçamento e indentação

Para quem não sabe o que significa, indentação é o modo como alinhamos a parte esquerda de cada linha do código fonte de modo que cada bloco do programa possa ser identificado facilmente pelo programador. Quando escrevemos uma condição, por exemplo, devemos recuar a(s) linha(s) abaixo do if um pouco para a direita. Um exemplo:

```
if (a>10) {
printf ("A variável é maior que 10.\n");
printf ("Digite um novo valor: ");
scanf ("%d",&a);
}
```

O mesmo vale para outros comandos, como for, else, while, ou para declarações de funções. A regra a ser seguida é, a cada colchete aberto ({), recuar para a direita. Quando se fecha o colchete, deve-se recuar para a esquerda.

Repare também que foram inseridos espaços no código antes e depois dos parênteses, que possibilitam uma melhor visualização do mesmo.

As vezes, podemos nos deparar com programas que, apesar de compilarem normalmente, podendo até funcionar bem, não seguem nenhuma dessas regras de indentação. Corrigir erros lógicos em programas assim é uma tortura. Por sorte, em quase todas as distribuições do Linux, existe um programa chamado indent, que analisa um programa escrito em C

e coloca indentação e espaços automaticamente no código. Para tanto, deve-se executar o indent da seguinte forma (você deve estar em um terminal):

indent programa.c

Onde programa.c é o nome do arquivo que contém o código fonte a ser indentado.

Nomes de variáveis e funções

É bem óbvio que uma variável ou função com o nome adequado pode fazer-nos compreender melhor o que faz uma parte do código, ou qual o valor que ela pode conter. Veja este exemplo:

```
int t() {
int s, i;
s=0;
for (i=0;i<20;i++)
s=s+p[i];
return s;
}
```

O que este código faz? Talvez você consiga descobrir rapidamente, mas para a maioria das pessoas, seria melhor escrevê-lo assim:

```
int total() {
int soma,item;
soma=0;
for (item=0;item<20;item++)
soma = soma + preco[item];
return soma;
}
```

Agora ficou bem melhor! Esse código é uma função que retorna o total de uma lista de preços com 20 itens.

Nomes de constantes

Um outro problema que pode afetar a compreensão e manutenção de um programa é a presença de constantes numéricas. No exemplo anterior, são somados os itens de uma lista com 20 elementos, o que não está muito claro no código. Nós poderíamos definir uma variável MAX_ITENS que contivesse o valor 20, trocando a linha:

for (item=0;item<20;item++)

por:

for (item=0;item<MAX_ITENS;item++)

Ou, melhor ainda, poderíamos definir MAX_ITENS como sendo uma constante no início do programa. Colocaríamos, então, a sua definição logo abaixo dos "#include" no início do programa, usando o que nós chamamos de diretiva de compilação (um comando para orientar o compilador). Usaremos para isso a diretiva #define:

Incluindo as bibliotecas necessárias:

```
#include <stdio.h>
#include <stdlib.h>
```

A linha abaixo define MAX_ITENS como sendo 20:

```
#define MAX_ITENS 20
```

Com isso, teremos uma constante de valor 20 com um nome bem sugestivo. Além disso, temos outra vantagem. Imagine agora se nós tivéssemos aumentado a lista de preços para 40 itens. Só teríamos que mudar para:

```
#define MAX_ITENS 40
```

Se a constante MAX_ITENS for usada várias vezes no nosso programa, isso pode representar uma economia considerável de tempo, já que só tenho de alterar uma linha.

Dividir para conquistar

Ninguém faz um sistema ou programa complexo por inteiro, de uma vez só. Ele é feito em várias partes separadas, que são unidas posteriormente. Nós já fizemos isso, definindo várias funções em um programa e chamando na função main essas funções.

Vejamos, por exemplo, o programa agenda.c da **aula 23**. Poderíamos ter colocado todo o código dentro do main(), o que teria resultado numa bagunça completa, apesar de termos um programa funcionando. Foram definidas várias funções que permitem que se compreenda o que está escrito na parte principal do programa de forma bem clara. Hoje nós vamos alterar um pouco a forma do programa.

Poderíamos melhorar ainda mais o modo como organizamos o programa, colocando, em arquivos separados, a parte principal do programa (arquivo newagenda.c), as definições dos registros (newagenda.h), e as funções (defagenda.h). Pegue esses três arquivos para acompanhar melhor a explicação.

Como juntamos isso tudo? Usando a nossa velha amiga, a diretiva #include, que instrui ao gcc para juntar o código da biblioteca com o código do programa. Quando usamos essa diretiva para bibliotecas padrão do C, usamos < > para informar qual a biblioteca. Por exemplo:

```
#include <stdio.h>
```

Porém, quando nós usamos com as funções definidas por nós mesmos (bibliotecas construídas pelo programador), usamos o nome do arquivo entre aspas. Assim, para incluirmos o arquivo newagenda.h, que são as definições das funções usadas no programa newagenda.c, usamos:

```
#include "newagenda.h"
```

Teremos também um arquivo onde definiremos as constantes usadas no programa e a estrutura do registro. Esse arquivo chama-se defagenda.h, e também deve ser incluído em newagenda.c, antes da inclusão de newagenda.h, já que este último usa as definições. Ficaremos então com as linhas em newagenda.c:

```
#include "defagenda.h"
#include "newagenda.h"
```

Precisamos agora informar às funções que o ponteiro para arquivo, FILE *agenda, a que elas se referem, é o mesmo que o ponteiro de mesmo nome no arquivo defagenda.h. O mesmo ocorre para a variável que contém o nome do arquivo, char *nomearq. Para isso, colocaremos no início do arquivo newagenda.h:

```
extern FILE *arquivo;
extern char *nomearq;
```

Repare na palavra extern antes das definições das variáveis. É isso que informa ao compilador que FILE *arquivo e struct pessoa existem e estão declaradas em um arquivo externo. No programa newagenda.c também temos umas definição idêntica para FILE *arquivo, com o mesmo propósito.

Chegamos ao fim do nosso curso de C (fim parte 1, na próxima semana tem mais), que não tem a pretensão de ser um curso avançado, mas básico, que mostre os fundamentos desta linguagem. Espero que estas aulas possam abrir-nos a visão para horizontes mais amplos da programação em geral, e que tenhamos o interesse de aprender cada vez mais.

Amigos, foi muito bom escrever estas aulas para vocês. Espero que nos encontremos novamente em futuros cursos neste site. Agradecemos a todos que nos enviaram suas sugestões, críticas e dúvidas, para que pudéssemos criar aulas cada vez melhores.

Um grande abraço a todos que nos acompanharam neste curso, e boa sorte a todos!

É o que desejam os autores,
André Souza
Diego Lages
Elias Bareinboim (Sim! Ele está aqui!)
Paulo Henrique B. de Oliveira
Rodrigo Hausen

PS.: Sei que vocês devem estar entusiasmados com o que já sabem fazer em C, mas, por favor, não passem noites em claro programando.

Aula 25

Tudo bom com vocês? A algum tempo eu não apareço, agradeço os emails e o feedback que temos recebido e sinto muito pela ausência, realmente estou enrolado por aqui.

Bem, voltando, esta aula vamos falar rapidamente sobre "projeto de software", isso como um todo. Já falamos sobre programação em si, muitos exemplos, etc.. agora vamos falar da parte de projetos.

Não tenho intenção de dar um curso completo de Engenharia de Software (ciência que estuda a fundo tais tópicos), simplesmente quero passar rapidamente sobre alguns tópicos que merecem destaque num curso de programação. Explicar o que é, importância, porque usar, como usar. Só para entendermos melhor, existe um estudo que diz que um projeto deve-se caracterizar por: 40% análise, 20% codificação, 40% testes. Ou seja, a parte de projeto é considerável, análise, especificação, etc. e *deve* existir. Outros estudos dizem que a análise sendo diminuída, a parte de codificação aumenta de forma considerável e os erros (e tempo do projeto) crescem assustadoramente. É o mesmo que tentarmos construir um prédio sem projeto gráfico, só pensando em tijolo após tijolo.. realmente "grotesco".

Aproveito para pedir que os que não mandaram e-mail falando sobre o tipo de curso que desejam para as próximas semanas da **área de programação**, que enviem o quanto antes seus emails, estamos realmente decidindo o tema e a sua opinião conta muito. Relembrando as opções:

- programação avançada em C para ambiente Unix;
- programação com Xlib (gráfica);
- programação com GTK (gráfica);
- programação voltada para redes;
- programação orientada para web (c, perl, php, python);
- programação com banco de dados (poderia ser continuação do tópico acima);
- projeto, especificação e análise de software (Engenharia de Software)

É por aí, esses são os mais pedidos, e o que estamos pensando realmente .. !:) Continuem mandando os emails, para que possamos começar a elaborar o próximo curso.

Não poderia deixar de agradecer o Rodrigo Hausen (mais conhecido como Cuco), ele entrou nesta reta final de curso e abrilhantou ainda mais nosso staff de contribuintes, valeu Cuco (hoje você que está de férias hein? se eu tivesse passado esse tempo todo que você falou que eu estava de férias, seria outra coisa..).

Na próxima aula teremos alguns exercícios elaborados por alguns de nós da equipe, essa aula vai ser importante para ensinar a forma de se estruturar um projeto, vale a pena acompanhar, principalmente quem tem a "ambição" de fazer um projeto maior. Vamos lá...

Bem, esta aula é escrita com base no livro: Software Engineering: A Practitioner's Approach, do famoso Mr. Pressman.

O Software

Pego de um livro didático: "Software é:(1) instruções (programas de computador) que, quando executadas, produzem a função e o desempenho desejados; (2) estruturas de dados que possibilitam que os programas manipulem adequadamente a informação; e (3) documentos que descrevem a operação e o uso dos programas". Sem dúvida existem outras definições mais completas, mas por enquanto é o suficiente, vamos nos aprofundar..

Engenharia de Software

A arte e a ciência da Eng. de Software encerram uma ampla gama de tópicos. Criar grandes programas é semelhante a projetar um grande prédio, ou seria um conjunto de prédios? Uma cidade? Um estado? Enfim, há tantas partes envolvidas que parece quase impossível fazer tudo funcionar junto, harmoniosamente. Evidentemente, o que faz a criação de um grande programa possível é a aplicação dos métodos de engenharia adequados. Neste artigo, serão vistos vários tópicos que são interessantes para um conhecimento básico, bem superficial sobre Eng. de Software, que é um mundo, talvez o mais importante na Ciência da Computação.

Fritz Bauer, na primeira grande conferência dedicada a Eng. de Software propôs sua definição:

"O estabelecimento e uso de sólidos princípios de engenharia que se possa obter economicamente um software que seja confiável e que funcione eficientemente em máquinas reais".

É unânime entre os profissionais da área que a Engenharia de Software é fundamental para o desenvolvimento do software, isso já é uma realidade, ainda bem. A Eng. de Software é parte integrante e ativa da Engenharia de Sistemas, e abrange três elementos básicos: métodos, ferramentas e procedimentos (explicaremos mais adiante) - que junto possibilitam um gerenciamento do processo de desenvolvimento, oferecendo um guia para construção de software de alta qualidade, fator mais relevante na natureza do software.

Os *métodos* explicam como se fazer algo para construir o software. Os *métodos* envolvem um conjunto de tarefas bem amplo, como o planejamento, estimativas do projeto, requisitos de software, de sistemas, projeto de estrutura de dados, arquitetura do programa, algoritmo de processamento, codificação, teste e manutenção. Além disso, são introduzidos critérios para avaliação de qualidade de software.

As *ferramentas* proporcionam apoio aos *métodos*, de forma mais automatizada possível. Existem conjunto de ferramentas integradas, que possibilitam "trânsito" de dados entre fases do projetos, que são conhecidos como ferramentas CASE (links no final do artigo).

Os *procedimentos* são aqueles que constituem um elo de ligação entre os *métodos* e as *ferramentas* e possibilitam uma visão mais ampla do projeto, um desenvolvimento sustentável. Eles definem a seqüência em que os *métodos* serão aplicados, as saídas que devem ser entregues, os controles que ajudam a assegurar a qualidade e a coordenar as mudanças, e os marcos de referência que possibilitam aos gerentes de software avaliarem o andamento do projeto.

Gerenciamento de Projetos

O gerenciamento de projetos de software representa a primeira camada no processo de Eng. de Software. O gerenciamento compreende atividades que envolvem medição, estimativa, análise de erros, programação de atividades e controle.

A medição possibilita que gerentes e profissionais entendam melhor o processo de eng. de software e o produto que ele produz (o software em si!). Usando mediadas, as métricas de produtividade e qualidade podem ser definidas.

Estimativas

O planejador do projeto de software deve estimar 3 coisas antes que um projeto comece: quanto tempo durará, quanto esforço será exigido e quantas pessoas estarão envolvidas. Além disso, o planejador deve prever recursos (hardware e software) necessários e avaliar riscos envolvidos no projeto.

A declaração do escopo (dados quantitativos e delimitadores do sistema) ajuda o planejador a desenvolver estimativas usando técnicas específicas.

Dentro dessa declaração de escopo ele deve extrair todas funções de software importantes (esse processo chama-se decomposição), que serão úteis para a elaboração do projeto.

Planejamento

O risco é parte inerente de todos os projetos de software, e por essa razão deve ser analisado e administrado. A análise dos riscos inicia-se com a definição e é seguida pela projeção e avaliação. Essas atividades definem cada risco, sua probabilidade de ocorrência e seu impacto projetado. Com essas informações em mãos, a administração e monitoramento dos riscos podem ser levadas a efeito, ajudando a controlar tais riscos.

Depois de levantada as rotinas básicas do sistema, os riscos e mediante a quantificação de mão de obra prevista, inicia-se a determinação do cronograma. Cada etapa deve ser descrita, prazos, mão de obra envolvida e grau de dependência entre fases. O ideal é a elaboração de um mapa, com esses dados explicitados graficamente. Além disso, um mapa de pessoa/tarefa pelo tempo também é interessante. De Tarefa/tempo também vale a pena.

Usando o gráfico como projeto, o gerente pode mapear o funcionamento, detectar falhas e aferir produtividade.

Em geral, considera-se num projeto de software a possibilidade de comprar partes externas a esse pacote. Particularmente, opto pelo desenvolvimento interno ao máximo, com intuito de diminuir tarefas como auditoria de código, com enfoque em segurança e coesão de código. A reengenharia de partes não herméticas de um sistema também contam. Essa posição contraria várias grandes empresas de software, mas é uma filosofia de onde trabalho, e concordo plenamente.

Em relação ao sistema de gerenciamento como um todo (conhecido como tracking), existe um ditado tirado do Pressman que diz: "Os projetos atrasam-se em seu cronograma um dia de cada vez". Frase sábia, uma vez que os dias de programação não necessariamente vão atrasar todo projeto, mas o conjunto deles sem dúvida alguma.

Para evitar isso, existe algumas formas, como: (1) através de reuniões esporádicas sobre a situação do projeto, em que cada membro da equipe relate sua situação, (2) verificando se os marcos de referência de projeto formais foram atingidos na data programada, (3) comparando-se cada data de início, a real e planejada e fazer seu deslocamento, conferindo a evolução das fases, (4) reunindo-se informalmente(importante) e adquirindo informações subjetivas sobre seus respectivos progressos em suas áreas (uma das maneiras mais interessantes mas que depende de experiência do gerente).

Mediante a uma avaliação da situação, há um leve controle sobre quem trabalha no projeto, caso contrário, o gerente deve focar em soluções após detecção do problema, redefinição de papéis, alocação de mão de obra, reestruturação de agenda, enfim, prazo para redefinições.

Links

Bem, aqui vão os links de ferramentas CASE:

- DOME: http://www.htc.honeywell.com/dome/
- ArgoUML: http://www.ics.uci.edu/pub/arch/uml/index.html
- ZMECH: http://www.xcprod.com/titan/ZMECH/
- DIA: http://www.lysator.liu.se/~alla/dia/
- MagicDraw: http://www.nomagic.com/magicdrawuml/

Entendendo o mundo da Informática..

Vale a pena, para ilustrar o que tem acontecido na indústria de software, contar uma tirinha do nosso amigo dilbert.. (não leia o final, faça a leitura seqüencial).

Chefe do Dilbert - Dilbert, você vai para a área de vendas. Geralmente quem vai para a área de vendas é porque deverá ser gerente num futuro próximo, mas não é seu caso.

Dilbert - Mas por quê?

Chefe do Dilbert - É porque eu não gosto da gerente de vendas, nem de você...

(Dilbert vai para a área de vendas)

Gerente de Vendas - Dilbert, o seu curso de vendas começa em uma hora.

(Dilbert, ansioso, vai para o curso de vendas)

Instrutor de Vendas - Bem, hoje começa nosso curso de vendas. Bem, como nossos produtos são extremamente caros e de péssima qualidade, vocês vão fazer praticamente todo o trabalho da empresa, terão que justificar todo faturamento de nossa empresa. Alguma dúvida? Curso encerrado.

(Dilbert sai com cara de ...)

Sem comentários hein? Bem, para ficar claro, já que não queremos ser como nessa situação, o que tem acontecido com todas as grandes software houses do mundo, grandes, sem ter que citar exemplos, vamos considerar qualidade do software um fator importante e intrínseco a qualquer projeto de tal natureza.

Como essa aula se estendeu bastante, eu continuo com o tema na semana que vem.. ou com os exercícios, vamos ver. O curso está terminando, não se esqueçam do e-mail.

Aula 26

Introdução

Amigos, hoje daremos mais um passo rumo a conclusão do nosso curso básico de C. Notemos que este curso é somente um curso inicial para que sejam aprendidos alguns conceitos fundamentais para que se possa construir algo mais

elaborado a partir desses. Estamos dando os passos iniciais em direção à conhecimentos mais profundos de programação. Mas a nossa caminhada não terminará aqui. Continuaremos indo adiante, cada vez com maior especialização, aprendendo aspectos diversos da programação, como a criação de interfaces gráficas com o usuário e, talvez até elaboração de programas que fazem o uso de redes de computadores. Mas lembremo-nos de que toda caminhada de mil léguas começa sempre com o primeiro passo.

Hoje, na conclusão do curso, apresentamos duas formas básicas de organizar dados na memória, consagradas pela sua praticidade e utilização: a pilha e a lista. Veremos as maneiras de se criar e manipular estas estruturas usando a linguagem C.

Estruturas de Dados

A forma como organizamos nossos dados na memória do computador é chamada de ESTRUTURA DE DADOS. Por exemplo, quando nós definimos uma struct em C, estamos definindo uma estrutura de dados que é um aglomerado de informações. Uma pilha e uma lista encadeada são outros dois exemplos de estruturas de dados, que passaremos a estudar adiante.

O que é uma pilha?

Imagine que você tem um monte de livros espalhados em cima de uma mesa. Esta não é a melhor maneira de se armazená-los, não é mesmo? Podemos organizá-los, colocando-os empilhados uns sobre os outros, formando uma pilha. Esta forma de organização de objetos físicos nos fornece uma abstração que pode ser aplicada para as informações que nós queremos guardar no computador.

Se nós temos uma pilha de livros, para adicionar um livro a esta pilha, nós simplesmente o colocamos no topo da pilha, e assim vamos aumentando a altura da pilha. Se nós quisermos retirar um livro da pilha, temos duas situações: se o livro está no topo, retiramo-lo sem dificuldade da pilha; se o livro não está no topo, devemos retirar da pilha todos os outros livros que estão sobre ele. Podemos proceder de modo análogo com os dados na memória do computador.

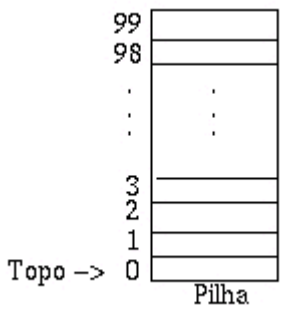
Vamos supor que nós queremos criar uma pilha de caracteres na memória do computador. Devemos primeiro especificar o tamanho da pilha, que pode ser de, por exemplo, 100 caracteres. A pilha pode ser, então, um vetor de caracteres:

```
char Pilha[100];
```

Como nós sabemos onde está o topo da pilha? Fácil: teremos um indicador para o topo da pilha, ou seja, a próxima posição vazia. Como ainda não inserimos nada na pilha, o topo aponta para a base da pilha, que é o primeiro elemento do vetor:

```
int topo=0;
```

Podemos representar esta pilha, graficamente, da seguinte forma:



Os números 0, 1, 2, ..., 98, 99, correspondem à posição do vetor, totalizando 100 posições. Vemos que topo aponta para a primeira posição, que é zero.

Para inserir um elemento na pilha, criamos uma função que recebe como entrada um caractere e coloca-o no vetor Pilha, na posição correspondente ao topo. Veja que o topo deve ser incrementado de 1, já que foi inserido 1 elemento. Chamaremos esta função de PUSH (empurrar, em inglês):

```
void Push(char elemento) {
if (topo==100) {
printf ("Pilha cheia!\n");
}
else {
Pilha[topo]=elemento;
topo++;
}
}
```

Note que temos que tomar o cuidado para não exceder o número máximo de elementos na pilha.

A operação de retirada só pode ser efetuada no último elemento inserido, ou seja, o elemento de número topo-1 no vetor (lembre-se de que a variável topo aponta para a próxima posição vazia). Ou então, decrementamos a variável topo de 1 e retiramos o elemento da pilha. Veja que não podemos retirar mais do que colocar na pilha. Criaremos para isso a função POP, que devolve o elemento que foi extraído da pilha:

```
char Pop() {
if (topo==0) {
printf ("Pilha vazia!");
}
else {
topo--;
return Pilha[topo];
}
}
```

Compile e execute este programa (pilha1.c) e veja como as operações são efetuadas. Repare que as operações de inserção e retirada de elementos da pilha alteram a ordem desses elementos.

Listas

Nós já aprendemos que um vetor é uma lista de elementos. Hoje, nós usamos esta lista trabalhando-a como uma pilha. Mas também nós podemos acessar os elementos desta lista de modo direto, como nós já fizemos em aulas anteriores.

Podemos, por exemplo, conceber um programa que armazene uma lista de preços para o usuário. Cada item desta lista terá um nome com até 40 caracteres e um preço. Vamos definir então uma struct para armazenar estes dados:

```
struct _item {
char nome[40];
float preco;
};
```

Para facilitar as coisas, definiremos o tipo Item como sendo a struct definida acima:

```
typedef struct _item Item;
```

Definamos então que a nossa lista conterá, no máximo 100 itens. Fazemos isso do seguinte modo:

Item Lista[100];

Um elemento qualquer da lista pode ser acessado diretamente pela sua posição na lista. Digamos que nós queremos inserir um elemento na posição 39 da lista. Faremos, então, deste modo (lembra-se da função strcpy?):

strcpy(Lista[39].nome,"Banana"); // copia para o nome do item a string "Banana"
Lista[39].preco=1.15; // coloca o preco de 1.15 reais

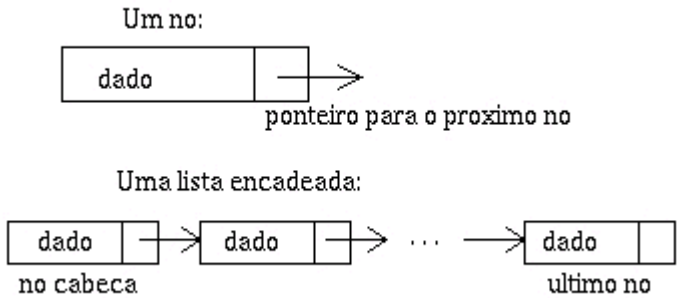
Não iremos entrar em mais detalhes acerca das listas definidas com vetores, pois já apresentamos esta estrutura em aulas anteriores. Para ver um exemplo completo, compile e execute este arquivo (lista.c). Este exemplo define funções para inserir e efetuar buscas na lista.

Listas (continuação)

A lista apresentada anteriormente apresenta alguns inconvenientes. Por exemplo, para apagar um elemento da lista, teremos que localizar a posição desse elemento e mover para essa posição algum outro elemento que está na lista (por exemplo, o último), ou colocar nessa posição algum dado inválido. Na verdade, o que estamos fazendo é somente "tapear" o usuário, pois não estamos removendo esse elemento de verdade, e sim mudando a informação contida naquela posição do vetor. O ideal seria se pudéssemos liberar a memória correspondente ao elemento removido para que pudéssemos usá-la posteriormente.

Outro inconveniente é que a lista está limitada ao número máximo de elementos definidos antes de começarmos a inserção na lista. Se o usuário quiser inserir mais de 100 elementos na lista, teremos que alterar a linha do programa que define o número máximo de elementos e recompilar o programa.

Para evitar esses inconvenientes, podemos usar uma estrutura de dados chamada de LISTA ENCADEADA. Cada item de uma lista encadeada é chamado de NÓ, e é composto por um dado qualquer e um ponteiro para o próximo nó da lista. O primeiro elemento da lista é chamado de NÓ CABEÇA ou NÓ RAIZ, e devemos ter um ponteiro que nos diga onde ele está. Representamos graficamente um nó e uma lista encadeada do seguinte modo:



Repare que o ponteiro para o próximo nó do último nó da lista encadeada não aponta para ninguém. Ele é o que chamamos de ponteiro nulo (NULL POINTER).

Vamos agora fazer a nossa lista de preços com uma lista encadeada. Recomendo a todos que peguem agora o arquivo listaenc.c e o acompanhem junto com o texto.

Devemos definir o nosso nó agora como:

```
struct _no {
char nome[40];
float preco;
struct _no *proximo;
};

typedef struct _no No;
```

Repare que em nosso exemplo, os nossos dados são uma string, char nome[40], e um preço, float preço. O ponteiro para o próximo nó de definido como o ponteiro para a próxima struct, struct _no *proximo;

Devemos ter, então, um ponteiro para o nó raiz:

```
No *Raiz=NULL;
```

Como ainda não foi inserido nenhum elemento na lista, não pode haver um nó raiz. Logo, ele deve ser definido como NULL (ponteiro nulo).

Criaremos a rotina de inserção de nós na nossa lista encadeada, chamando-a de InsereNo. Primeiro, verificamos se há nó raiz. Se não houver, criamos um. Se já houver nó raiz, percorremos a lista da raiz o último elemento.

Como faremos para percorrer os nós na lista encadeada? Preste bastante atenção: cada nó possui um ponteiro para o próximo. Assim, vamos definir as variáveis:

```
No *Atual;
No *Proximo;
```

A variável Atual aponta para o nó atual. Para achar o próximo nó, fazemos:

```
Proximo = Atual->proximo;
```

Ou seja, colocamos na variável próximo o valor do ponteiro para o próximo elemento (Atual->proximo). Fazemos então com que Atual aponte para o Próximo elemento. Repetimos esse procedimento até que Atual->proximo seja um ponteiro nulo, ou seja, quando Atual apontar para o último elemento:

```
while (Atual->proximo!=NULL) {
Proximo = Atual->proximo;
Atual = Proximo;
}
```

Para inserir um novo nó no final, devemos primeiro alocá-lo em memória com a função malloc. Agora é só informar ao último nó o ponteiro para o nó inserido:

```
Atual->proximo = Inserido;
```

É importante que o ponteiro para o próximo do nó Inserido seja nulo, para que indiquemos que ele agora é o último nó. Devemos também colocar os dados no nó Inserido:

```
strcpy(Inserido->nome,nomeitem);
Inserido->preco = precoitem;
Inserido->prox = null;
```

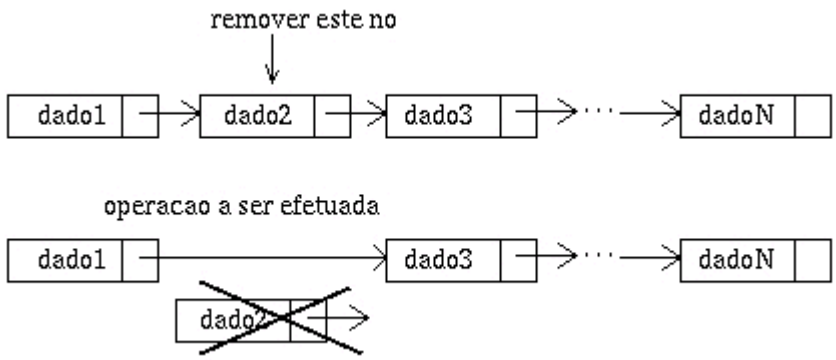
E está terminada a inserção. Ufa!

Para buscar um elemento na lista encadeada, devemos simplesmente percorrer a lista do primeiro ao último nó, como já foi descrito acima, verificando se o nó atual é o nó que procuramos. Veja a função BuscaNo no programa.

Para finalizar, olharemos a função ApagaNo. Para remover um nó, devemos primeiro percorrer a lista encadeada, como já vimos, procurando o nó. Se ele for encontrado, temos 2 situações: se for o nó raiz, devemos fazer com que a variável Raiz, que aponta para esse nó, aponte para o nó sucessor à raiz. Devemos então desalocar o nó removido usando free:

```
if (Atual==Raiz) {
Raiz = Raiz->proximo;
free(Atual)
}
```

Se não for o nó raiz, devemos informar ao nó anterior ao que será removido que ele deve apontar para o nó sucessor ao que será removido. Veja a figura:



Para isso, teremos que armazenar na variável Anterior, durante a busca, o ponteiro para o nó anterior. Com esta informação, fazemos:

```
Anterior->proximo = Atual->proximo;
free(Atual);
```

Note que não precisamos nos preocupar se o próximo nó ao que será removido não existe. Se não houver próximo nó, a variável Próximo conterá o valor NULL, e teremos que o nó Anterior passará a ser o último nó da lista.

Conclusão

E, por fim, terminamos essa aula de nossa fase final do curso de C básico. As estruturas vistas hoje são implementadas em muitos programas que nós usamos no dia-a-dia, e servem de base para muitas outras usadas inclusive(principalmente) em Bancos de Dados. Não se preocupem se vocês não entenderem de imediato tudo o que foi mostrado hoje. Quando eu estava começando a programar em C, demorei alguns dias para poder assimilar isso. Não desistam. Pratiquem, treinem, analisem as linhas de código uma por uma até que tenham certeza de que os conceitos foram fixados.

Para quem se interessou nesta aula de estruturas de dados, recomendo ler o capítulo 20 de:

C Completo e Total
Herbert Schildt
3a. edição
Makron Books

É um bom livro de introdução à programação C, cobrindo até alguns aspectos um pouco mais aprofundados.

Em breve começaremos um outro curso de programação, dando seqüência a este. Solicito a todos que enviem-me sugestões de quais assuntos deveremos tratar no próximo curso. Algumas pessoas já nos enviaram sugestões como interfaces gráficas para o usuário ou um curso mais aprofundado de estruturas de dados. De qualquer forma, o assunto ainda está em aberto e todos aqueles que quiserem colaborar com suas opiniões ou críticas serão bem-vindos.

Para todos que nos tem acompanhado durante este curso, um grande abraço!

Aula 27

Na aula de hoje vamos dar uma breve demonstração de como usar o depurador C fornecido pelo Linux, que é o depurador GNU Gdb. Mais uns dos tópicos importantes que não foram discutidos e estamos inserindo nesta fase final.

O Gdb é uma ferramenta muito útil para encontrar erros em seu código fonte.

O Gdb fornece uma interface de alto nível para facilitar sobremaneira a tarefa de depuração de código fonte, principalmente em se tratando de programas extensos.

O Gdb permite que você determine breakpoints, ou seja, pontos do seu programa nos quais a execução é interrompida por algum erro, seja de sintaxe, do sistema operacional, etc. Então, quando o seu programa abortar a execução e você não souber onde está o erro, através da determinação de um breakpoint ficará muito mais fácil descobrir onde o mesmo se encontra, pois o gdb é que vai encontrá-lo e informar via mensagens de erro onde está o problema e de que tipo é o mesmo.

Para usar o Gdb, e preciso antes de mais nada recompilar o programa com as opções de depuração oferecidas por ele.

Opções de Depuração

Para o Gdb funcionar, ele precisa de algumas informações que normalmente são removidas pelos compiladores C e C++. Estas informações são do tipo: tabela de símbolos, tabela de mnemônicos, dentre outras.

O Gdb, assim como a maioria dos depuradores, precisa destas informações para poder estabelecer uma conexão entre o código executado e a localização real dos dados, geralmente um número de linha no código fonte.

Através da linha de comando abaixo, você estará informando ao seu compilador, seja gcc ou cc, que ele não deve remover as informações de depuração acima listadas quando da compilação do seu código fonte. Por exemplo:

```
$ gcc -g -c seuprograma.c
$ gcc -g -o seuprograma seuprograma.o
```

As linhas de comando acima compilam o código fonte contido no arquivo chamado seuprograma.c e colocam as informações de depuração no programa executável resultante, ou seja, seuprograma.

No caso de seu programa principal chamar outros programas, isto é, de seu programa ser composto por diferentes códigos fonte, será necessário que você compile cada um com a opção -g e depois faça a linkagem do programa completo coma opção -g também.

Usando o Gdb

Após ter compilado seu(s) programa(s) com as opções de depuração, ele estará pronto para ser depurado pelo Gdb.

Existem duas maneiras de usar o Gdb. Você pode executar o programa de dentro do ambiente do depurador e depois visualizar o que acontece ou você pode usar o depurador em modo post-mortem, ou seja, após seu programa ter sido executado e descarregado (morrido), um arquivo "core" será criado deixando informações sobre a execução do programa e seus erros (problemas). Assim, o Gdb verifica tanto o core quanto o executável do seu programa, a fim de descobrir o que está errado.

Neste breve tutorial é abordado a primeira maneira de depuração. A segunda maneira será tema de um segundo tutorial a ser publicado em breve.

A sintaxe básica padrão é a seguinte:

gdb seuprograma

Obviamente, seuprograma é o nome de seu programa. O Gdb roda em modo texto e lhe fornece um aviso (prompt) gdb, a partir do qual você insere e executa os comandos gdb.

Testando o Gdb

Para testar o Gdb, vamos usar um programa exemplo chamado bugmotif.c. Este programa tem um erro intratável: ele grava em um ponteiro NULL usando a função padrão do C para cópia de strings strcpy. Abaixo é listado o código do programa bugmotif.c.

Pegue as instruções e arquivos, aqui.

O programa deve compilar bem, desde que você salve no seu diretório "home" um arquivo de recurso (resource file) com o nome LinuxProgramming. Para compilar o programa, use a seguinte linha de comando:

$ gcc -g bugmotif.c -o bugmotif -I/usr/X11R6/include \ -L/usr/X11R6/lib -lXm -lXt -lX11

Para compilar e linkar este programa é necessário que você tenha instalado as bibliotecas LessTif e alguns outros headers do C. Por isso, aconselhamos que use um programa seu, do qual tenha conhecimento das bibliotecas necessárias para sua execução. Sugerimos que este fique apenas como um exemplo do passo a passo do uso do Gdb, poderia se ter utilizado um programa de atribuição simples de ponteiros não alocados.

Você deve rodar o programa de dentro do Gdb para obter mais informações sobre o que deu errado. O comando para isto é o seguinte:

$ gdb bugmotif

Ao encontrar um erro, o programa será abortado e o Gdb imprimirá uma mensagem de erro como a seguinte:

Program received signal SIGBUS, Bus error. 0x7afd12e4 in strcpy ()

Se o programa estiver rodando de dentro do Gdb, o depurador continuará rodando e você poderá tentar descobrir o que aconteceu de errado, mesmo que o programa tenha sido abortado. Para saber em que ponto o programa parou de executar, use o comando "backtrace", como mostrado a seguir:

(gdb) backtrace
#0 0x402062 in strcpy ()
#1 0x4006b670 in expose (w=0x8063228, event=0x0, region=0xbffff504) at PushB.c: 597
.
.
.

#12 0x80492db in crt_dummy ()

Este comando lhe mostra a seqüência de erros ocorridos durante a execução do programa.

Visualizando Arquivos de Código

O Gdb permite que você chame o arquivo no qual o erro ocorreu, desde que você tenha o código fonte, e determine pontos (linhas) onde você deseja que a execução seja interrompida. Um ponto de interrupção é, então, um ponto do seu programa onde você deseja que o Gdb interrompa a execução.

As opções de interrupção são geralmente em uma linha determinada do arquivo, ou no início ou final da função.

(gdb) break 23
Breakpoint 1 at 0x2164: file bugmotif.c, line 23.
(gdb) run
The program being debugged has been started already.
Start it from the begining? (Y or N) y

Deste modo, quando um erro ocorrer, no caso do programa bugmotif.c ocorrerá quando clicar no botão Gerar o bug, o gdb parará o programa na linha 23, que é o ponto de interrupção atual e exibirá uma mensagem de erro, como a seguir:

Breakpoint 1, bugCB (widget=0x4001d660, client_data=0x0, call_daat=0x7b03b25c
"") at bugmotif.c:23
23 strcpy(client_data, "This is a string.");

Já o comando "list" mostrará o código fonte ao redor do ponto de interrupção, como mostrado a seguir:

(gdb) list
18
19 { /* bugCB */
21
22 /* Copy data into NULL pointer. */
23 strcpy (client_data, "This is a string.");
24
25 } /* bugCB */
26
27

Neste ponto, você pode imprimir o valor das variáveis, conforme mostrado abaixo:

(gdb) p client_data
$1 = (String) 0x0 /* mostra o conteúdo da variável client_data, que neste caso é zero, pois gravou o conteúdo em um ponteiro NULL.

Através do comando "c" e prosseguida a execução:

(gdb) c
Continuing

Program received signal SIGBUS, Bus error.
0x7afd12e4 in strcpy ()

Após, o programa vai cair depois do ponto de interrupção.

Comandos Gdb

A seguir, são listados os principais comandos suportados pelo Gdb. Além dos comandos listados abaixo, o Gdb suporta uma variedade muito grande de comandos, que permitem realizar uma tarefa de muitas maneiras diferentes. Sugerimos que descubra outros comandos e faça testes.

Comando	**Significado**
bt	Apresenta rastreamento de pilha.
backtrace	Apresenta rastreamento de pilha.
info stack	Apresenta rastreamneto de pilha.
quit	Sai do ambiente do depurador.
run	Roda o programa.
run "args"	Roda o programa com os parâmetros da linha de comando "args".
break "func"	Determina o ponto de interrupção no início da função "func".
break "line_num"	Determina o ponto de interrupção no início da função "func".
list "filename.c:func"	Visualiza "filename.c".
run	Roda o programa.
c	Continua a partir do ponto de interrupção.
step	Executa a linha de programa seguinte.
print "var"	Imprime valor da variável "var".
help	Ajuda.

Tabela 9.1 Os comandos Gdb mais utilizados

Através do comando "help", o Gdb oferece também ajuda on-line.

O Gdb gráfico do Linux

O Linux fornece uma interface gráfica para o Gdb para aqueles que preferem o ambiente gráfico ao invés do modo texto. O xxgdb é o ambiente Gdb gráfico que vem na maioria da distribuições Linux atuais. Também é possível, mesmo com o ambiente gráfico, executar os comandos no modo texto do Gdb a partir da janela principal xxgdb.

Um ambiente que vem ganhando a preferência entre os desenvolvedores é o DDD, que é bastante simples e fácil de utilizar, roda em cima do X também.

Aula 28

Bem pessoal, nesta aula teremos alguns exercícios para revisão, serão 5. É interessante mensurar o grau de acertos de tais exercícios, para que se possa avaliar o entendimento das aulas anteriores (vejam o grau de dificuldade para executá-los).

Na próxima aula serão divulgados algumas respostas e mostraremos o funcionamento do DDD. O DDD (Data Display Debugger) é um front-end gráfico para o GDB, e como o pessoal achou um tanto quanto difícil sua utilização na aula anterior, não custa nada dar uma mãozinha.

Vamos aos exercícios, que é o que importa realmente. Ah, não esperem as respostas semana que vem, façam como se não houvesse nada, assim vocês realmente aprendem e não só ficam lendo programas (não é o objetivo básico de se propor exercícios).

Questão 1:

Seja um vetor v definido como:

int v[3] = {10,20,30}

Suponha que o vetor está na memória iniciando na posição 65522 e que cada inteiro ocupa 2 bytes na memória.

v =	10	20	30
posição	65522	65524	65526

Responda certo ou errado a cada uma das perguntas:

a) as expressões v e *v[0] são equivalentes

b) as expressões v+2 e &v[2] são equivalentes

c) as expressões *(v+2) e v[2] são equivalentes

d) as expressões *v+2 e v[0]+2 são equivalentes

Também com base no mesmo vetor, responda:

e) O que será escrito na tela se houver uma linha no programa como a abaixo?
printf ("%d",v[1]);

f) E se fosse incluída a linha abaixo?
printf ("%d",*v);

g) A seguinte linha exibe o que está na primeira posição do vetor? Por quê?
printf ("%d",v);

Questão 2:

Seja a struct abaixo:

```
struct data {
int dia, mes, ano;
};
```

Faça uma função que retorne um valor 1 se a data fornecida for pertencente a um ano bissexto ou 0 se a data fornecida não pertencer a um ano bissexto. Assuma que o ano é expresso em 4 dígitos (afinal, não queremos ter problemas com o bug do milênio, não é!?).

```
int bissexto(struct data d) {
... inclua o seu código ...
}
```

Lembrete: um ano é bissexto se ele é divisível por 4 mas não por 100, ou é divisível por 400.

Exemplo:

```
main () {
struct data x;
x.dia=11;
x.mes=12;
x.ano=1978;
printf ( "%d",bissexto(x) ); // deve aparecer 0
}
```

DESAFIO: Faça uma função dias_decorridos que retorna o número de dias entre uma data e outra. Use a sua função bissexto.

Questão 3:

Por que o programa abaixo não funciona? O que deve ser feito para que ele funcione como esperado?

```
#include <stdio.h>
#include <string.h>

struct pessoa {
char *nome;
int idade;
}

main () {
struct pessoa *elemento;
elemento = (struct pessoa *)malloc(sizeof(struct pessoa));
elemento->idade = 30;
strcpy(elemento->nome,"Fulano de Tal");
printf("%s\n",elemento->nome);
}
```

Questão 4:

Crie um programa para ler até 100 números inteiros do teclado e guardá-los em uma lista encadeada (**aula 25**). Depois, modifique o programa para tirar esses números da lista e colocá-los numa pilha, retirando-os em seguida da pilha e mostrando-os na tela. O que acontece com os números?

Os nós da lista encadeada são definidos da seguinte forma:

```
struct no *{
int numero;
struct no *prox;
}
```

A pilha pode ser definida como:

```
int pilha[100];
```

Questão 5:

Faça um programa para ler dois arquivos texto e juntá-los em um só, intercalando as suas linhas. O programa deve pedir que o usuário digite os nomes dos 3 arquivos. Exemplo:

digite o nome do primeiro arquivo de entrada: nomes.txt
digite o nome do segundo arquivo de entrada: telefones.txt
digite o nome do arquivo de saída: lista.txt

Se os arquivos estiverem da seguinte forma:

arquivo nomes.txt:

Juca Bala
Zé Mané
Disk-pizza do Bartolo

arquivo telefones.txt:

555-1234
9999-9876
0800-181121

O resultado deverá ser o arquivo lista.txt da seguinte forma:

Juca Bala
555-1234
Zé Mané
9999-8876
Disk-pizza do Bartolo
0800-181121

Bem pessoal, é por aí. Espero que se divirtam fazendo tais exercícios e semana que vem estaremos de volta. Continuem mandando os emails em relação ao próximo curso, estamos satisfeitos com os resultados e queremos que cada vez mais pessoas participem desta votação.

Aula 29

Olá, amigos! Suaram a camisa fazendo os exercícios da semana passada? Muito bem! Somente quem treina consegue alcançar o sucesso. E já que vocês se esforçaram tanto, hoje nós teremos as respostas dos exercícios para ver como vocês se saíram.

Mas antes disso, vamos ver hoje uma ferramenta muito útil para encontrar e corrigir erros em programas. Nós já vimos

o gdb, que é o debugador comumente distribuído com o gcc. O único problema do gdb é a sua dificuldade de utilização. Quem já trabalhou com algum ambiente gráfico para depuração de programas, tipo o ambiente do Delphi ou C Builder (num passado remoto, argh!), sente a diferença. E se você achava que o Linux não tinha nenhum depurador com uma interface bonita, então você estava redondamente enganado.

O DDD é um front-end gráfico para o gdb, isto é, ele é um programa que usa uma interface de janelas com o usuário para enviar comandos para o gdb. Ele "esconde" a tela de texto do gdb, mostrando ícones e botões, que são mais fáceis de usar. Em tempo: DDD, neste caso, significa Data Display Debugger, e não Discagem Direta à Distância, como muitos podem pensar.

Instalando o DDD

Antes de utilizar o DDD, ele deve ser instalado. Para instalar o DDD em um sistema Debian/GNU Linux, utilize a seguinte linha de comando:

apt-get install ddd

O pacote .deb do DDD já vem incluído nos CDs originais da Debian. Para distribuições tipo RedHat, recomendo que seja utilizado o ddd compilado estaticamente, distribuído em pacote .rpm, para evitar possíveis preocupações com dependências. Para fazer isso, vá ao endereço http://rpmfind.net/ e procure pelo pacote ddd-static. Baixe-o para o seu computador e instale-o com:

rpm -ivh <nome do pacote>

onde <nome do pacote> é o nome completo do pacote rpm contendo o DDD.

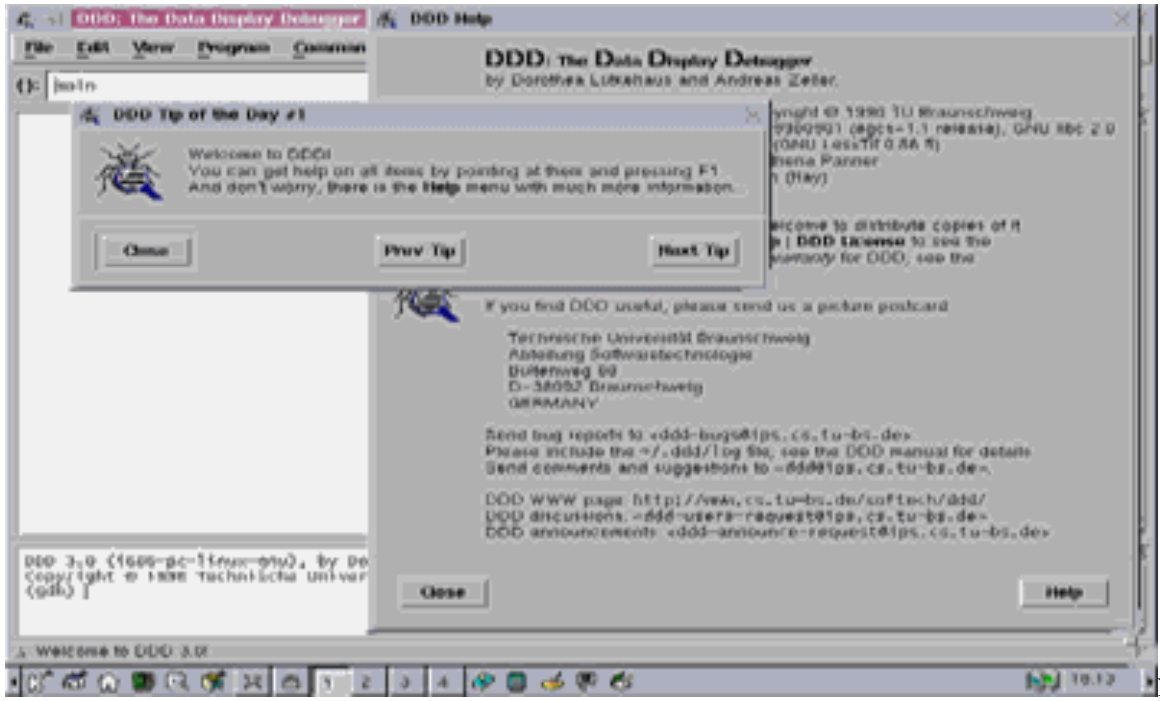
Para instalar em outras distribuições do Linux, como Mandrake, Slackware ou Suse, pegue o pacote rpm mencionado acima e converta-o para o sistema de pacotes da sua distribuição. Qualquer distribuição possui um conversor de rpm para o seu formato próprio. Feito isso, instale o pacote convertido.

Iniciando o DDD

Para utilizar o DDD, basta rodar o arquivo cujo nome é, por algum motivo que me foge à compreensão, ddd. No meu caso (Debian/GNU Linux), o path completo para esse arquivo é:

/usr/X11R6/bin/ddd

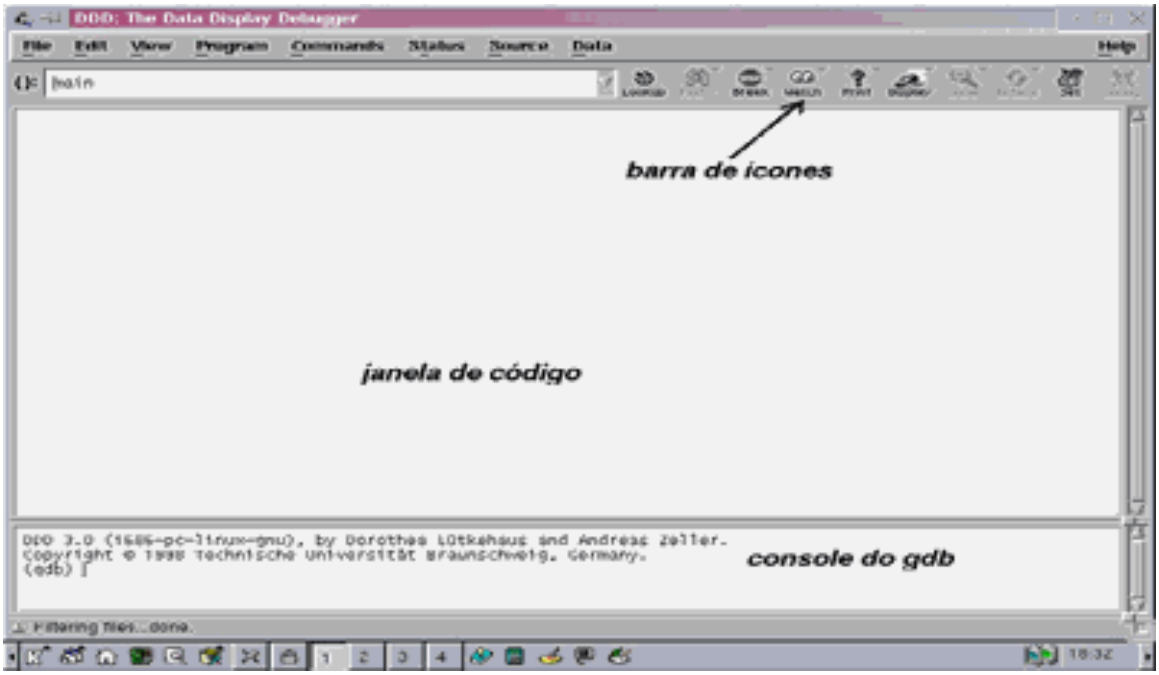
Na maioria dos sistemas, este diretório já deverá estar na lista dos caminhos a serem procurados.


Note que você deverá estar dentro do X para executar o programa! Se tudo foi instalado corretamente, ao executá-lo, devem aparecer algumas mensagens informando que foram criados alguns diretórios no seu home para armazenar informações de configuração do DDD. Após isso, aparece a janela do programa, uma janela contendo dicas de utilização e uma outra de informações:

Não se esqueça de mandar um cartão postal para o criador do programa!

Utilizando o DDD

Devemos fechar as janelas que estão sobre a nossa área de trabalho do DDD. Clique em ambas as janelas o botão Close. Agora nós temos a nossa tela livre para trabalhar com o DDD. Vemos, inicialmente, três partes principais: a barra de ícones, a janela de código e a janela do console do gdb.



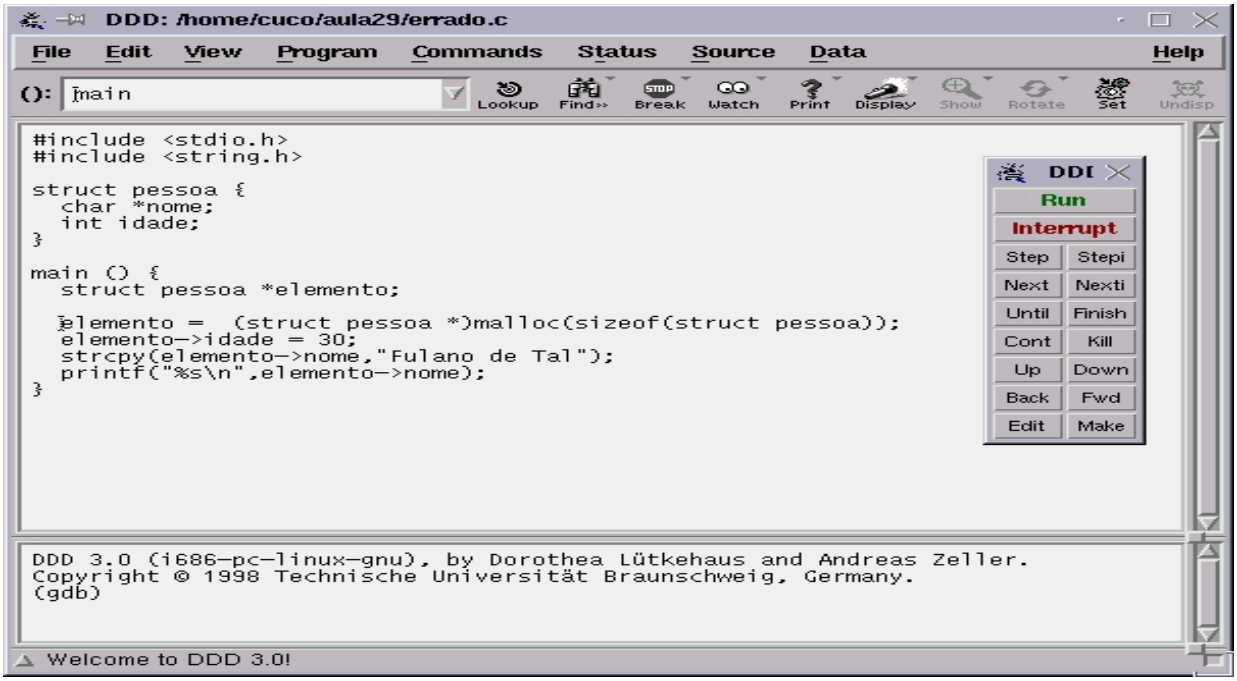
Devemos agora abrir o programa a ser depurado. A título de exemplo, pegue o programa errado.c, que é o programa da questão 3 da lista de exercícios. Como antes, quando usamos o gdb, devemos compilar o programa utilizando a opção -g do gcc, que gera código para depuração:

gcc -g errado.c -o errado

Isso irá gerar um arquivo executável cujo nome é "errado". Se você executá-lo, obterá um vistoso "Segmentation Fault" na sua tela. E agora, José?

No DDD, clique na opção File do menu de opções; aparecerá uma lista de opções, onde você deve selecionar Open Program. Procure o arquivo executável "errado" e clique em Open. Isso deve fazer com que o código do programa seja mostrado na janela de código. Outro modo de fazer isso seria executar o DDD já informando a ele qual o arquivo a ser aberto. No nosso caso:

ddd errado



Observe que apareceu também uma nova janela, contendo alguns botões que servem para executar, interromper e seguir passo a passo o programa. Vamos executar o programa clicando no botão Run. Como o nosso programa ainda contém erros, ao ser executado, aparece na janela do console do gdb:

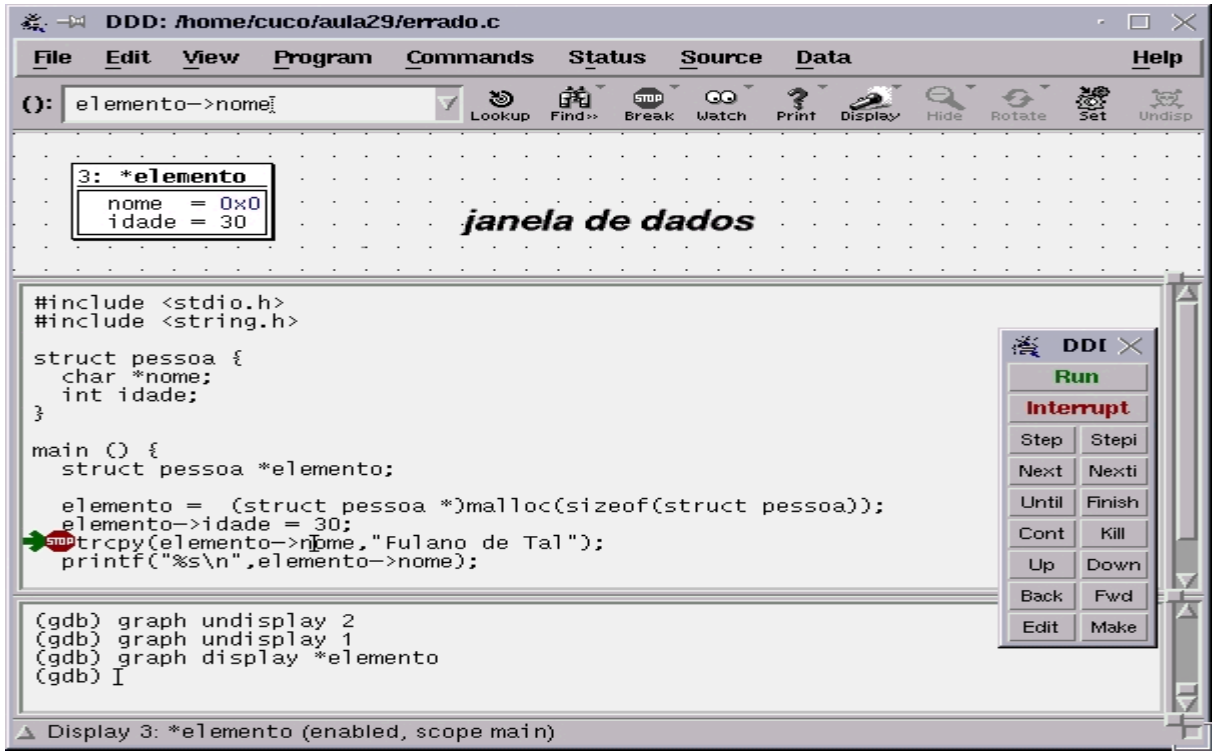
Program received signal SIGSEGV, Segmentation fault.
0x400523d7 in strcpy ()

Indicando que o programa foi interrompido devido a um problema, na função strcpy. Procuremos então a linha no nosso programa onde é chamada a função strcpy e coloquemos lá um ponto de parada, ou BreakPoint. Para fazer isso, clique primeiro um pouco à esquerda da linha que contém a chamada para a função strcpy, fazendo com que o cursor piscante se mova para lá, e depois clique o botão Break (o botão que tem a imagem de uma placa de parada). Deverá aparecer um símbolo indicando foi adicionado um ponto de parada naquela linha.

Execute o programa (botão Run). Como o programa não foi finalizado completamente da vez anterior, deverá aparecer uma janela perguntando se você pretende iniciá-lo novamente do começo. Responda sim para que possamos continuar a analisar o programa.

A execução será interrompida na linha onde foi adicionado o breakpoint, o que é indicado por uma seta verde à esquerda da linha. Nesse ponto, vamos começar a analisar os valores das variáveis para tentar achar o problema. Clique com o botão direito sobre o nome da variável elemento em alguma linha de código para que apareça um menu drop-down com várias opções para mostrar o valor daquela variável. Repare que se você passar a seta do mouse sobre qualquer variável, também serão mostradas algumas informações referentes a ela.

Ao selecionar a opção Display *elemento no menu drop-down, a janela do DDD será dividida em mais uma parte, que é a janela de dados. Nela, haverá uma caixa mostrando a variável elemento. Como ela é uma struct, são mostrados os nomes dos seus campos e os seus valores.



Repare que o campo nome da struct contém o valor 0x0, ou seja, o valor NULO. A partir deste ponto, passamos a desconfiar de alguma coisa. Como elemento->nome é um ponteiro para uma string, então isso nos indica ele é um ponteiro nulo! Isso deve estar causando o problema no nosso programa. Somente para tirar a dúvida, vamos observar o valor de elemento->nome. Selecione alguma parte do programa onde está escrito elemento->nome; isso é feito clicando e arrastando o ponteiro do mouse, fazendo com que uma barra preta apareça para indicar que o texto está selecionado. Clique com o botão direito do mouse sobre a barra preta e selecione Print elemento->nome. Na janela de console do gdb deve aparecer:

(gdb) print elemento->nome
$1 = 0x0

Ou seja, isso indica que elemento->nome é realmente nulo. Isso ocorre porque a posição de memória que deveria armazenar a string não foi alocada.

Vamos corrigir o erro do programa. Abra o programa errado.c em algum editor e adicione a linha abaixo, antes da chamada da função strcpy, para alocar memória para a string:

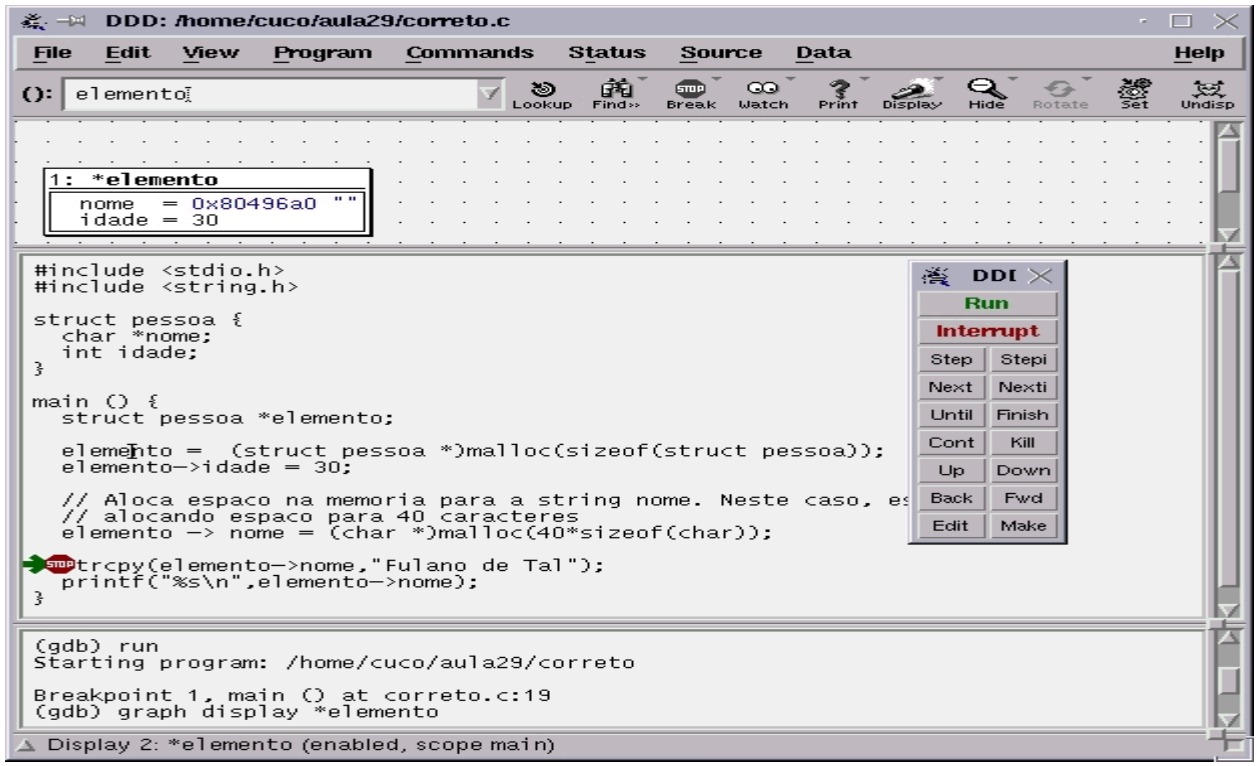
elemento->nome = (char *)malloc(40*sizeof(char));

Salve-o com o nome de correto.c. Ou, se preferir, pegue-o aqui: correto.c

Compile novamente o programa com a opção -g do gcc, feche o DDD e abra-o novamente carregando o executável "correto":

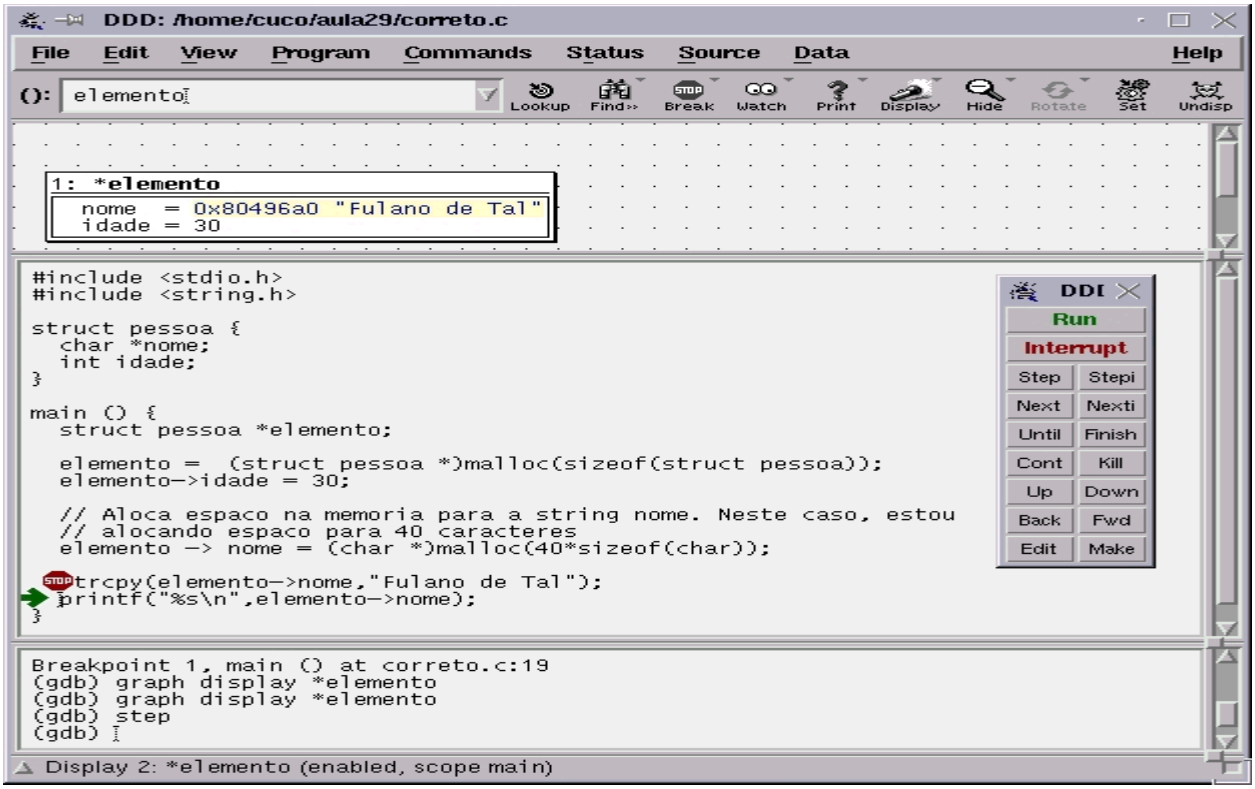
ddd correto

Vamos adicionar novamente um breakpoint na linha da chamada da função strcpy, usando o botão Break, e executar o programa clicando no botão Run. A execução será interrompida no ponto de parada. Agora, vejamos o conteúdo da struct elemento, do mesmo modo como fizemos antes. E lá está: o campo nome não contém mais o valor 0, e sim 0x80496a0, ou alguma coisa parecida, indicando que ele aponta para alguma posição de memória, ou seja, não é mais um ponteiro nulo.



Vamos dar um passo na execução do programa para verificar como ele se comporta, clicando no botão Step. A seta verde deve andar uma linha para baixo, indicando que as instruções pertencentes à linha de cima foram executadas e a linha indicada está para ser executada. Veja que alguma coisa mudou na janela de dados: agora, além de mostrar para que endereço de memória o campo nome aponta, também é mostrado o conteúdo daquela área de memória.

Como vemos que o nosso programa está se comportando bem, vamos continuar a execução dele até o final, clicando no botão Cont. Veja que na janela do console do gdb é mostrado tudo que seria exibido normalmente na tela durante a execução do programa; no nosso caso, é mostrada a string "Fulano de Tal". Logo depois, o programa é terminado e nenhuma variável mais é mostrada na janela de dados.



Acho que por hoje já é o bastante, amigos. E isso não é tudo sobre o DDD! Ele é, com certeza, um dos programas mais úteis que já foram feitos para Linux, possuindo, por isso, inúmeras funções. Correm rumores de que alguns usuários conseguiram fazer com que o DDD lavasse e passasse suas roupas, ao passo que outros instruíram-no a fazer pratos da culinária internacional. De qualquer modo, semana que vem teremos mais informações sobre como utilizá-lo para depurar programas, com mais um exemplo. Se até lá alguém descobrir como fazer com que o DDD lave, passe e cozinhe, por favor, mande-me uma mensagem!

E para aqueles que leram todo o artigo roendo as unhas de ansiedade para ver as respostas dos exercícios, aqui estão elas, merecidas por sinal:

RESPOSTAS DOS EXERCÍCIOS

A letra a da primeira questão é a única afirmativa errada. Vejamos por que: v é um vetor de inteiros; logo, a variável v é um ponteiro para a posição de memória onde encontra-se o primeiro elemento do vetor. Assim, v indica a primeira posição do vetor, ao passo que *v[0] seria o valor indicado pela posição de memória apontada por v[0], a primeira posição do vetor. Como os elementos do vetor são números inteiros, e não ponteiros, *v[0] não é nem reconhecido pelo compilador como sendo uma operação válida.

A seguir vêm as respostas para as letras e, f, g da primeira questão:

e) Será escrito 20 na tela, ou seja, o elemento que está na segunda posição. (Lembre-se: vetores começam com 0).

f) Seria escrito o conteúdo da primeira posição do vetor, ou seja, 10.

g) A resposta é não. Ela exibe o endereço de memória para o qual v aponta.

O erro no programa da primeira questão está no fato de não ter sido alocada memória para o nome da pessoa na struct elemento. Assim, antes de utilizar o campo nome da struct, deveria ser feito:

elemento->nome = (char *)malloc(40*sizeof(char));

Veja o programa correto no arquivo questao3.c.

As respostas para as questões 2, 4 e 5 estão aqui:
questao2.c, questao4.c, questao5.c

Quanto ao que acontece com os números na questão 4, deixo para vocês observarem o resultado. Compilem, executem o programa e vejam com seus próprios olhos.

Aula 30

Olá, amigos! Como têm passado? Como eu sei que vocês gostaram da aula anterior sobre DDD (vocês gostaram, não foi? Por favor, digam que sim...), hoje temos os toques finais para vocês se tornarem expertos em DDD; serão mostradas as utilidades de outros botões e mais algumas funções do menu, com mais um exemplo.

Relembrar é viver

Na aula passada, além da correção dos exercícios, vimos algumas noções básicas sobre a utilização do DDD: instalação, execução do programa, carregamento do código a ser depurado e função de alguns botões. Para aqueles que não viram a aula anterior ou não entenderam direito, aqui vai um pequeno glossário dos termos utilizados:

- **bug**: um erro, geralmente lógico, que é percebido somente quando se executa um programa;
- **debugar ou depurar**: eliminar os *bugs* do programa, ou seja, retirar os erros de modo que o programa venha a funcionar corretamente;
- **debugger, debugador ou depurador**: programa que ajuda na tarefa de eliminar os erros, mostrando como o programa a ser depurado é executado, o valor de suas variáveis, etc.
- **DDD**: front-end gráfico para o GNU debugger, ou gdb. O DDD esconde a cara do gdb, considerada feia por alguns, mostrando ícones e botões em vez de uma tela de texto;
- **Breakpoint ou ponto de parada**: é um ponto, em alguma parte de um programa, que pode ser estabelecido durante a depuração do seu código onde a sua execução é paralisada, podendo ser retomada posteriormente.

Caminhando um pouco mais

Vamos embora que esperar não é saber; quem sabe faz a hora e aprende DDD. Vamos aprender agora como fazemos para pular partes do programa que nós já sabemos que funcionam, e que não precisam ser testadas. Vamos pegar este programa exemplo e dar uma olhada nele: exemplo1.c. Ele calcula a média de alguns números.

A função **double media(double numeros[], int quantos)** recebe um vetor de doubles como entrada e a quantidade de números contidos no vetor, retornando a média desses números. Supondo que a função média já foi testada e aprovada com louvor pelo controle de qualidade, nós a usamos em um programa para imprimir 10 notas e a respectiva média. Após compilar e executar são mostrados os valores das notas e a seguinte média:

Média das notas: -1073741824

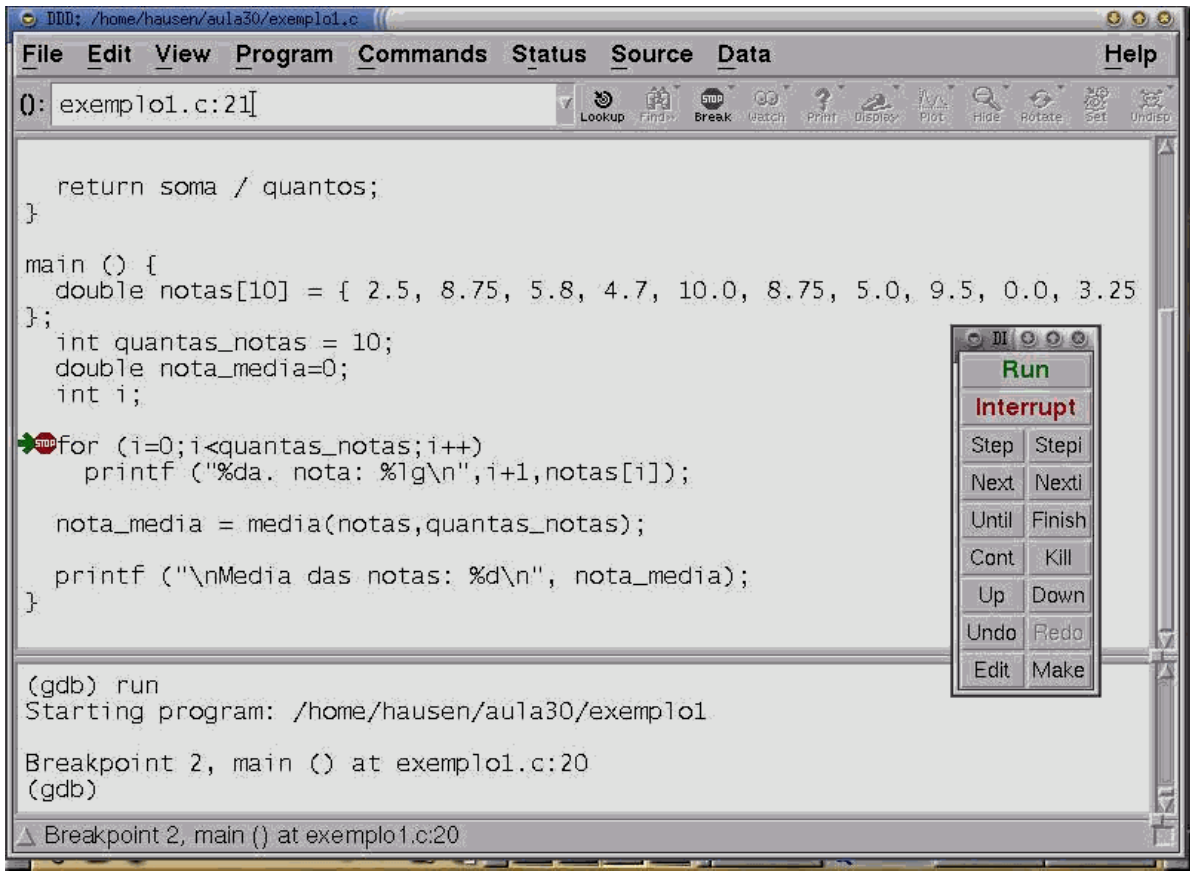
Se olharmos os valores das notas, veremos que a média mostrada está completamente errada. Para corrigir este problema, pediremos gentilmente que o DDD nos acuda. Primeiro, compilamos o programa com a opção -g do gcc e depois o carregamos no DDD (se você não se lembra como, veja a **aula 29**). Vamos colocar um breakpoint na linha onde são impressos os valores das notas para ver se os valores estão sendo alterados (selecione a linha abaixo e clique no botão break):

for (i=0;i<quantas_notas;i++);

Vamos executar o programa clicando o botão run e exibir os valores das notas, selecionando o nome da variável notas e escolhendo **print notas**. Obtemos:

$1 = {2.5, 8.75, 5.7999999999999998, 4.7000000000000002, 10, 8.75, 5, 9.5, 0, 3.25}

Que são os valores corretos das notas (até mesmo 5.7999999998 e 4.7000000002, pois são aproximações de 5.8 e 4.7).



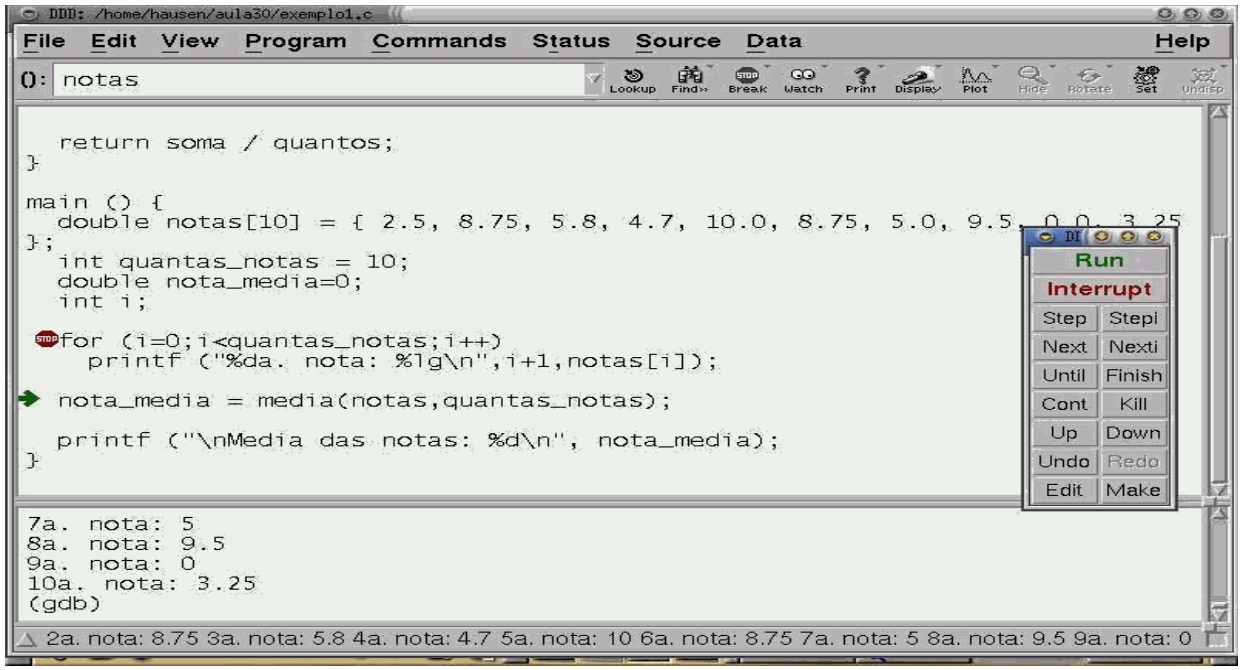
Para continuarmos a execução do programa, nós deveríamos clicar no botão step, para ir para a próxima instrução. Hoje faremos um pouquinho diferente: clicando no botão Until. Esse botão faz com que a execução adiante para uma linha de código que seja a maior do que a linha corrente, indicada pela seta verde. Clicando uma vez, iremos para a linha do printf, abaixo do for. Clicando mais uma vez, é impresso no console do gdb:

1a. nota: 2.5

E a linha corrente volta a ser o for. Até aí, parece que o comportamento do botão Until é o mesmo do botão Step. Porém, clicando mais uma vez naquele botão, o programa "dispara", executando várias linhas em seqüência e imprimindo no console do gdb:

2a. nota: 8.75
3a. nota: 5.8
4a. nota: 4.7
5a. nota: 10
6a. nota: 8.75
7a. nota: 5
8a. nota: 9.5

9a. nota: 0
10a. nota: 3.25



Isso economiza o seu dedo indicador, que ficaria exausto de tanto pressionar o botão do mouse para clicar em Step. Veja que o programa parou exatamente sobre a linha que segue o laço for:

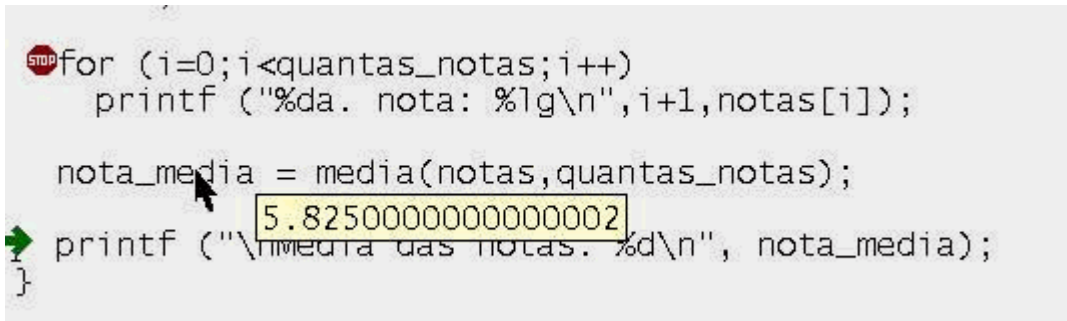
nota_media = media(notas,quantas_notas);

Imprimindo as notas (selecionando print notas), obteremos no console do gdb:

$4 = {2.5, 8.75, 5.7999999999999998, 4.7000000000000002, 10, 8.75, 5, 9.5, 0, 3.25}

Ou seja, os seus valores ainda estão corretos.

Vamos executar a função média e ver qual o valor retornado na variável nota_media. Se clicarmos no botão Step, entraremos na função media; mas nós não desejamos isso, pois sabemos que a função funciona perfeitamente bem. Vamos passar "por cima" da função clicando no botão Next. Observe que agora a linha corrente é a que se segue à função, e não a primeira linha de código desta. Pause o cursor por cima do nome da variável nota_media e observe o seu valor, que aparece em uma caixa.



O valor da variável nota_media está correto. Mas, surpreendentemente, quando clicamos no botão Step para executar a próxima linha, aparece na janela do console do gdb:

Media das notas: -858993459

Usando o nosso poder de dedução, percebemos que algo de podre está ocorrendo na linha onde a média das notas é impressa. Observando a linha com mais atenção, vemos que estamos usando o printf com a opção de formatação "%d", que é usada para números inteiros! Como nota_media está no formato **double**, corrigimos a linha para:

printf ("\nMedia das notas: %lg\n",nota_media);

Lembre-se que devemos usar "%lg" ou "%lf" para imprimir um número no formato double.

Compilando e executando o programa corrigido, obteremos o resultado correto:

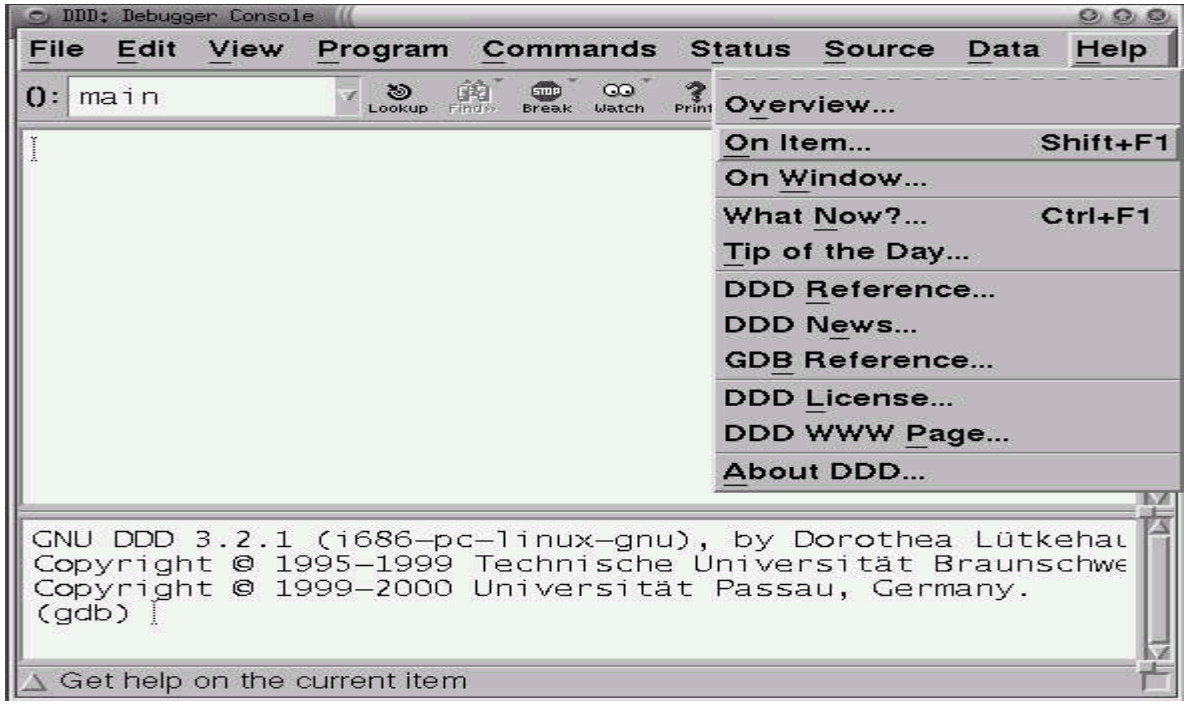
2a. nota: 8.75
3a. nota: 5.8
4a. nota: 4.7
5a. nota: 10
6a. nota: 8.75
7a. nota: 5
8a. nota: 9.5
9a. nota: 0
10a. nota: 3.25

Media das notas: 5.825

Help! I need somebody!

Como não dá para ficar descrevendo cada função do DDD em detalhe, aconselho a todos que pratiquem o seu uso, experimentem apertar cada um dos botões do DDD, mexer nos menus - com cuidado para não alterar as configurações - enfim, fuçar bastante no DDD. E o melhor amigo do usuário nessas horas é a ajuda do programa. Para obter auxílio, clique na opção Help da barra do menu. Dentre as várias opções disponíveis, as mais úteis são:

- **On Item**: ajuda sensível ao contexto, ou seja, após selecionar este tipo de auxílio, você deve clicar em algum item da janela do DDD para obter um texto explicativo sobre aquele item;
- **What now?**: indica o que você pode fazer após a execução de algum comando e quais as melhores alternativas;
- **DDD Reference**: referência completa sobre como utilizar o DDD.



Com esta aula terminamos o caminho das pedras deste excelente programa que é o DDD. Espero que com estas noções vocês possam encontrar e debugar os erros que eventualmente aparecem em qualquer programa, pois ninguém é perfeito. Na próxima semana teremos estrutura de dados mais a fundo. Vocês não vão querer perder!

Aula 31

Saudações a todos vocês, que nos acompanham semanalmente com a nossa humilde coluna de programação. Gostaram da moleza que foi a aula passada? É melhor não acostumar... hoje nós vamos pegar pesado. Vamos revisar as estruturas de dados que já foram apresentadas e aprender algunas cositas más. Como vocês devem lembrar, na **aula 26**, explicamos o que são estruturas de dados, e mostramos formas de organizar pilhas e listas na memória do computador. Hoje, veremos versões "turbinadas" da pilha e da lista encadeada.

Revisão

Recomendo àqueles que perderam a **aula de número 26** que a leiam inteiramente antes de começar esta, pois aqui só vamos dar uma pincelada de leve nos conceitos daquela aula.

Começando pela **pilha**, nós a definimos como sendo um vetor alocado em memória com tantas posições quantas fossem necessárias:

<tipo_da_pilha> Pilha[n_pos];

Onde <tipo_da_pilha> é o tipo de dados que a pilha deverá conter e n_pos é o número de posições que ela terá. Ela também deverá ter um topo, ou seja, a próxima posição livre, que inicialmente é a primeira posição da pilha:

int topo=0;

As operações que podem ser efetuadas com dados na pilha são duas: **push** e **pop**, que são respectivamente colocar um elemento no topo da pilha - e, evidentemente, atualizar o topo - ou retirar o último elemento colocado.

A **lista encadeada** é outra forma de organizarmos dados na memória do computador. Ela é uma lista que não tem o seu tamanho definido *a priori*, podendo aumentar ou encolher dinamicamente, ocupando mais ou menos memória no computador. É armazenada por meio de ponteiros, sendo que existe um ponteiro para uma posição especial,a primeira posição ou **nó** da lista, o **nó cabeça** ou **nó raiz**, para que nós tenhamos sempre alguma referência de onde começa a lista. Cada nó contém informações quaisquer, e pode ser definido da seguinte forma:

```
struct _no {
<tipo_dado_1>dado1;
<tipo_dado_2>dado2;
...
<tipo_dado_N>dadoN;
struct _no *proximo;
};

typedef struct _no No;
```

Onde <tipo_dado_1>...<tipo_dado_N> são os tipos das variáveis dado1 a dadoN. Repare que cada nó tem um ponteiro próximo, que é a referência para o próximo nó.

O nó raiz, inicialmente, é indefinido, já que não temos nenhuma informação na lista:

```
No *raiz=NULL;
```

Ele é alocado dinamicamente ao se inserir a primeira informação na lista e a partir daí a sua referência não muda mais. Assim como o primeiro nó, todos são alocados dinamicamente:

```
No *qualquerno;
qualquerno = (No *)malloc(sizeof(No));
qualquerno->dado1 = dado1;
...
qualquerno->dadoN = dadoN;
qualquerno->proximo = NULL;
```

Repare que definimos o próximo nó como NULL para podermos ter uma referência sobre onde acaba a lista. Ao inserirmos mais um nó, devemos também avisar ao nó anterior que existe mais um nó:

```
Anterior->proximo = qualquerno;
```

Para efetuarmos operações na lista, geralmente é necessário percorrê-la de nó em nó, começando pelo nó cabeça e terminando no nó desejado, ou no fim da lista, caso o nó não tenha sido encontrado:

```
No *Atual;

Atual = Raiz;
while (Atual != NULL) {
if (Atual->dado == DadoDesejado) break;
else
Atual = Atual->Proximo;
}
```

Em uma lista encadeada, qualquer elemento pode ser acessado, diferentemente da pilha, que só possui o topo acessível. Também relativamente à pilha, a lista encadeada possui um maior número de operações sobre os elementos: inserção, deleção, alteração, movimentações (mudança de lugar dos elementos).

Do modo como foi apresentada a pilha anteriormente, ela tinha mais uma desvantagem em relação à lista encadeada: enquanto o número de elementos que podem ser inseridos na lista, o número de elementos da pilha fica limitado ao seu tamanho. Mas será que isso não pode ser contornado?

Pilha com ponteiros

Já que usamos ponteiros para definir uma lista, porque não usamos também ponteiros para definir uma pilha? Assim, teremos algo que se parece com a primeira, mas tem o comportamento da segunda. Para simplificar, vamos supor que nós queremos uma pilha que pode conter uma quantidade qualquer de números inteiros. Definimos então uma posição da pilha como:

```
struct _PilhaPos {
int numero;
struct _PilhaPos *abaixo;
};

typedef struct _PilhaPos PilhaPos;
```

Repare o campo **abaixo** da posição da pilha: ele serve como uma referência para a posição que esta abaixo dela na pilha.

Neste instante, vou dar 5 segundos para os mais espertos pensarem: o que mais falta para definir a pilha? 5... 4... 3... 2... 1... Exatamente: O TOPO DA PILHA!

```
PilhaPos *Topo=NULL;
```

Apesar da definição ser muito parecida, não confunda o topo da pilha com o nó cabeça da lista encadeada. O topo da pilha muda cada vez que é feito um push ou pop. E já que estamos falando das operações da nossa pilha, vamos defini-las:

```
void Push(int num) {
PilhaPos *NovoTopo;

NovoTopo = (PilhaPos *)malloc(sizeof(PilhaPos));
NovoTopo->numero = num;
NovoTopo->abaixo = Topo;
Topo = NovoTopo;
}
```

Perceberam a manha? Cada vez que faço um push, crio um novo elemento que será inserido "em cima" do topo da pilha. Depois, atualizo o topo.

O pop também é definido facilmente:

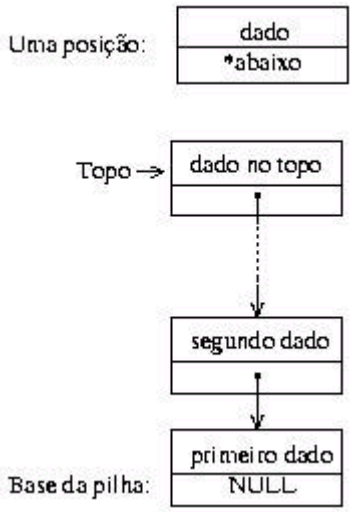
```
int Pop {
int num;
PilhaPos *TopoAntigo;

if (Topo == NULL) {
printf ("Pilha vazia!\n");
return 0;
}
num = Topo->numero;
TopoAntigo = Topo;
Topo = TopoAntigo->abaixo;
free(TopoAntigo);
```

```
return num;
}
```

Essa função é simples: ela pega os dados que estão no topo, "desce" o topo da pilha, elimina o topo antigo e retorna os dados contidos nele. Esta função tem o cuidado de verificar se a pilha está vazia (Topo = NULL).

Uma forma gráfica de nós enxergarmos a pilha definida dessa forma está abaixo:



Lista encadeada ordenada

Depois de turbinarmos a pilha, vamos também incrementar a lista encadeada. Na nossa vida, precisamos de muitas coisas ordenadas. E um dos melhores ordenadores que existem, se não o melhor, é o computador. Vamos agora pô-lo à nossa mercê para que possamos obter o máximo proveito dele.

Um exemplo bem prático é uma agenda de telefones: temos as pessoas e os respectivos telefones ordenados por nome. Quando nós inserimos uma pessoa na nossa agenda de telefones, vamos até a página que contém as pessoas cujos nomes iniciam por uma certa letra e colocamos o seu nome lá. Geralmente, as páginas estão ordenadas em ordem lexicográfica (de A a Z).

Podemos fazer um programa que "imite" esse comportamento, criando uma lista encadeada ordenada pelo nome da pessoa. Definiremos cada nó como:

```
struct _no {
char nome[40];
char telefone[15];
struct _no *proximo;
};

typedef struct _no No;
```

O nó raiz é definido do mesmo modo como foi mostrado na **aula 26**. As rotinas de inserção, busca e remoção de nós da lista encadeada são quase iguais, exceto que temos mais um teste, além de checar se a lista chegou ao fim, testamos também se o nó atual é lexicograficamente superior ao nó que buscamos. Para entendermos melhor, vamos ver um exemplo. Uma lista encadeada ordenada contém os seguintes nomes:

Se buscarmos pelo nome Bruno na nossa lista ordenada, somente passaremos pelos nós Ana, Beto e Carla para saber que o Bruno não está na lista. Vejamos bem: em ordem alfabética, Ana e Beto vêm antes de Bruno. Quando eu chego no nó que contém o nome Carla, não posso mais encontrar Bruno na lista, porque Carla vem depois de Bruno na ordem alfabética, ou em outras palavras, Carla é lexicograficamente superior a Bruno.

Para fazermos o teste referido anteriormente, usamos a função **strcmp(string1,string2)**, que retorna um 0 se as duas strings são iguais, um valor maior do que 0 se **string1** é lexicograficamente superior a **string2** ou um valor menor do que 0 se **string1** é lexicograficamente inferior a **string2**.

Assim, para percorrer a nossa lista encadeada ordenada buscando por um nome fazemos agora:

```
Atual = Raiz;
while (Atual!=NULL) {
if (strcmp(Atual->nome,nome)>=0) break;
else
Atual = Atual->proximo;
}
```

A linha onde está o **if** quer dizer: "pare se o campo nome do nó atual é lexicograficamente superior ou é igual ao nome buscado". Se a condição não for satisfeita, vou para o próximo nó.

Um programa que faz exatamente o que foi mostrado nesta última parte da aula pode ser encontrado aqui: listaord.c

Dá para perceber que o uso de uma lista ordenada já melhora um pouco o modo como nós acessamos os nossos dados. Mas e se nós já tivermos os dados em um arquivo e quisermos pô-los em ordem?

Além disso, você pode pensar em uma maneira de, além de mostrar os nomes em ordem alfabética, mostrá-los em ordem inversa, ou seja, do último para o primeiro? Certamente deve haver um modo de fazer isso, mas com listas encadeadas fica meio difícil fazer, não?

Aula 32

E aí, pessoal, tudo "em riba"? Gostaria de saber o que vocês estão achando das aulas sobre estruturas de dados. Penso que alguns de vocês devem estar achando o assunto um pouco difícil de ser compreendido; mas não se preocupem: o que pode parecer complicado à primeira vista, pode se tornar claro com um pouco de prática. Plagiando um autor de lingua inglesa, eu diria que o caminho da prática leva ao palácio da sabedoria.

Hoje, em nossa penúltima aula de estruturas de dados, iremos cumprir a promessa feita na anterior e atender os pedidos de vários leitores que nos enviaram e-mails pedindo que frisássemos alguns assuntos relativos à organização de dados.

Antes de prosseguir, verifique se você já leu as aulas anteriores sobre estruturas: **aula 26**, **aula 29** e **aula 31**.

Se já leu, inspire e expire profundamente algumas vezes; relaxe... você está sentindo o seu corpo leve e a sua mente tranqüila... ao meu sinal, você começará a compreender tudo aquilo que está escrito. Um, dois, três...

Revisão

Na aula passada, vimos que as pilhas podem ser implementadas de forma semelhante à lista encadeada, usando ponteiros, o que nos possibilita empilhar um número indeterminado de elementos (na prática, esse número é geralmente limitado pela memória do computador). Para maiores detalhes, recorra à **última aula**.

Além de examinarmos as pilhas, também mencionamos um modo de construir uma lista encadeada de um modo tal que os elementos encontram-se em alguma ordem dentro dela. Chamamos essa lista de lista encadeada ordenada, ou simplesmente, lista ordenada. Observamos que, para obter uma listagem dos dados em ordem, basta percorrer a lista do nó raiz ao último nó, mas para recuperá-los em ordem reversa, a operação não é tão simples assim. Ou será que não?

Lista duplamente encadeada

Analisando um elemento de uma lista encadeada, vemos que ele possui duas partes distintas: uma que armazena uma certa quantidade de informação qualquer e outra que é um ponteiro, uma referência para o **próximo** elemento. Preste bastante atenção nesta palavra - próximo - e responda: por que a partir de um nó da lista encadeada conseguimos facilmente obter os nós seguintes? Justamente por causa da existência desse tal de **próximo**. E por que não conseguimos obter os anteriores? Porque falta-nos a referência para o elemento **anterior**.

Para termos um pouco mais de simetria, vamos contornar essa dificuldade incluindo um novo campo na definição dos nós da lista encadeada:

```
struct _no {
struct _no *anterior;
<INFORMAÇÃO>
struct _no *proximo;
};

struct _no No;
```

A parte grifada e em maiúsculas, identificada por <INFORMAÇÃO>, no elemento da lista, indica que ali devem ser incluídos um ou mais campos referentes aos dados que queremos armazenar.

O nó cabeça da lista duplamente encadeada é definido de modo similar ao respectivo nó da lista encadeada simples, com a única diferença que o seu ponteiro para o nó anterior deve ser NULL.

Como os dados contidos na lista podem ser obtidos em ordem normal ou inversa, também devemos guardar uma referência para o último nó da lista, o qual terá o seu campo **proximo** com o valor NULL.

Inicialmente, como a lista está vazia, o nó raiz e o último nó não existem, e isso pode ser definido da seguinte forma:

```
No *raiz=NULL;
No *ultimo=NULL;
```

Quando for inserido o primeiro elemento na lista, os dois nós deverão ser alocados, sendo que ambos apontam para o mesmo elemento. Caso haja inserção de mais elementos, os ponteiros **anterior** e **proximo** devem ser atualizados convenientemente:

1º passo - percorrer a lista e achar uma posição para inserir o novo nó. Terei então referências para os dois nós vizinhos ao nó que será inserido: **NoAnterior** e **NoPosterior**;

2º passo - alocar o novo nó:
```
NovoNo = (No *)malloc(sizeof(No));
```

3º passo - colocar a informação e atualizar os ponteiros:
```
NovoNo->INFORMACAO = AlgumaCoisa;
NovoNo->proximo = NoPosterior;
```

```
NovoNo->anterior = NoAnterior;
if (NoAnterior!=NULL)
NoAnterior->proximo=NovoNo;
if (NoPosterior!=NULL)
NoPosterior->proximo=NovoNo;
```

4º passo - verificar se o nó foi inserido na primeira ou na última posição, o que deve ocasionar uma atualização da referência para nó raiz ou para o último nó, respectivamente:

```
if (NovoNo->anterior==NULL)
raiz = NovoNo;
if (NovoNo->posterior==NULL)
ultimo = NovoNo;
```

As verificações sobre o NoAnterior e o NoPosterior são necessárias pois, caso o nó que está sendo inserido fique na última ou na primeira posição, NoPosterior ou NoAnterior serão nulos, e qualquer tentativa de manipular os seus dados acarretará a exibição da elegante mensagem **Segmentation fault**, ou algo do gênero, seguida do incômodo término prematuro do programa.

O processo de percorrer a lista duplamente encadeada é semelhante ao modo como percorremos a lista encadeada simples. O procedimento para parcorrer a lista encadeada "de trás para frente" também é parecido, só que começamos pelo **ultimo** nó e utilizamos o ponteiro **anterior**. Não entraremos em meiores detalhes sobre isso porque, pelo tempo que já estamos falando de lista encadeada, vocês já têm alguma noção de como isso é feito.

A deleção requer um cuidado maior para ser feita, necessitando verificar se o nó a ser apagado é o primeiro ou o último, reposicionando os ponteiros **ultimo** e **primeiro**.

Para um melhor esclarecimento sobre o assunto, analisem este programa, construído com base no respectivo código-fonte apresentado na aula anterior: listaord2.c.

Noções de Ordenação

Às vezes, temos informações armazenadas no computador que estão fora de ordem, e desejamos ordená-las. Uma maneira possível, se pudermos recuperá-las de um arquivo, seria a seguinte:

- abrir o arquivo;
- ler os dados do arquivo e colocá-los na memória em uma lista encadeada ordenada até que não haja mais dados no arquivo;
- listar, em ordem, os dados da lista encadeada ordenada e gravá-los de volta no arquivo na seqüência em que são listados.

Muito simples. E seria perfeito, se não fossem três problemas:

1) nem sempre os dados encontram-se em um arquivo, podendo estar na memória principal do computador. Gravar os dados em um arquivo, para depois relê-los, ordená-los, e gravá-los de volta é uma operação lenta (pense no caso da ordenação de uns dez mil registros). Caso façamos toda a operação na memória principal (sem usar arquivos), podemos não ter memória sificiente para, além de manter os dados, montar uma lista ordenada;
2) a operação de inserção em uma lista ordenada é lenta. Não parece, mas se você inserir um milhão de registros seguidamente, você perceberá;
3) mesmo que os dados a serem ordenados encontrem-se em arquivo, eles podem não caber integralmente na memória principal;

No nosso curso de C, resolveremos a primeira e a segunda inconveniências. O terceiro problema é mais difícil de resolver e necessita de conhecimentos mais avançados de estruturas de dados para que seja encontrada uma solução eficiente.

Hoje iremos resolver o primeiro problema com a utilização de um algoritmo que, mesmo não sendo muito eficiente, é melhor do que as pseudo-soluções apresentadas para resolvê-lo, e é bastante simples de ser implementado.

Para simplificar, vamos supor que estamos trabalhando com uma lista não ordenada de inteiros armazenada em um vetor:

int lista[N];

Onde N é o número de inteiros na lista.

O algoritmo para ordenação mostrado abaixo chama-se bubble sort, ou ordenação por bolha, que recebeu esse nome devido ao modo como os elementos mudam de posição até que estejam ordenados, "subindo" ou "descendo" na lista como bolhas em uma panela de água fervente. Implementado em C, ele pode assumir as seguintes feições:

```
void BubbleSort(int lista[],int num_eltos) {
int temp;
int i,j;
for (i=0;i<num_eltos;i++)
for (j=i+1;j<num_eltos;j++)
if (lista[i]>lista[j]) {
temp=lista[i];
lista[i]=lista[j];
lista[j]=temp;
}
}
```

A função BubbleSort efetua a ordenação em uma lista de inteiros, recebendo como parâmetros: a lista a ser ordenada e o seu número de elementos. Ela efetua a ordenação na própria lista, trocando dois elementos de posição se o primeiro for maior que o segundo, repetindo essa operação para todos os elementos da lista. Para ordenar a nossa referida lista de inteiros, basta chamar a função com os seguintes parâmetros:

BubbleSort(lista,N);

Para ordenar uma lista de strings, devo substituir a comparação lista[i]>lista[j] pela a respectiva construção com strcmp:

if (strcmp(string[i],string[j])<0)

Em uma lista encadeada, se for necessário trocar dois nós de posição, basta trocar as informações contidas neles, não sendo necessária a manipulação de ponteiros na hora do câmbio.

Pegue este exemplo, analise e compile-o: bubble.c. Além de mostrar o bubble sort em funcionamento, ele é um bom exemplo de como gerar números aleatórios.

Experimente alterar o valor de N para 100, 1000 e 10000. Veja que de 100 para 1000 a diferença no tempo gasto para ordenar não é tão grande. Porém, de 1000 para 10000 o tempo gasto aumenta em cem vezes, aproximadamente, para um incremento de apenas dez vezes no número de elementos!

Concluímos, por hoje, a nossa aula de C. Espero que tenham gostado do assunto, já que na próxima aula continuaremos falando sobre melhores métodos de ordenação. Introduziremos também uma última estrutura de dados chamada árvore.

Aula 33

Amigos, hoje terminaremos as aulas de estruturas de dados no nosso curso de C com um último algoritmo sobre ordenação que, como já vimos, é um assunto de bastante importância em computação. Espero que todos vocês tenham gostado dessas aulas e que pelo menos os conceitos básicos tenham sido aprendidos.

Revisão

Na aula passada vimos um modo de organizar informações em uma lista encadeada para que pudéssemos listá-las em ordem crescente ou decrescente. Chamamos essa estrutura de lista duplamente encadeada ordenada, e ela se parecia muito com a lista encadeada ordenada comum, exceto que se utilizava de dois ponteiros, anterior e próximo, para referenciar os dois nós adjacentes a um nó.

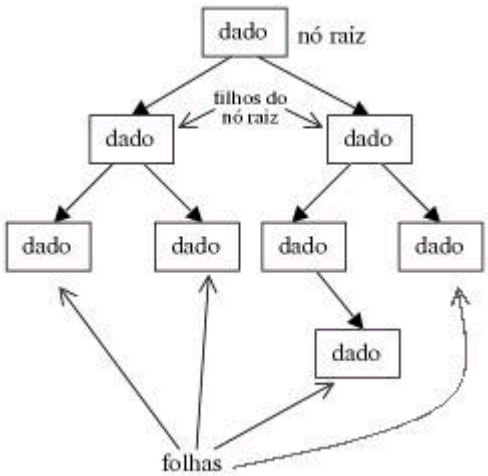
Também na última aula tivemos as primeiras noções de ordenação, analisando a importância de se ordenar os dados e os problemas que ocorrem quando tenta-se ordená-los utilizando métodos inadequados para isso. Foi mostrado um algoritmo de ordenação que resolvia um dos problemas, mas que, apesar de não ser muito eficiente, merecia destaque apenas por sua simplicidade. Hoje, veremos um algoritmo que, apesar de ser mais complexo, possui uma melhor eficiência. Para isso, precisamos aprender mais alguns conceitos.

Árvores

Geralmente quando mencionamos a palavra árvore em computação não estamos nos referindo ao vegetal, mas sim à uma estrutura de dados que possui uma organização peculiar. A definição formal de uma árvore é algo que foge ao escopo de nosso curso, e por isso iremos fazer uso de uma definição mais simples que atenderá aos nossos propósitos:

Uma **árvore** é uma estrutura de dados cujos itens são constituídos de uma informação qualquer e uma ligação a outros dois itens que estão localizados após o item atual. Cada item de uma árvore recebe o nome de **nó** e os itens que estão ligados a um nó qualquer recebem o nome de **nós filhos**. Reciprocamente, o nó que está localizado anteriormente e que referencia o nó atual recebe o nome de **pai**. O primeiro nó de uma árvore não possui nó pai, chamando-se **nó raiz**. Os nós que não possuem filhos são chamados de **folhas**.

Para entender melhor essa nomenclatura, vamos lançar mão de uma representação gráfica já consagrada de uma árvore:

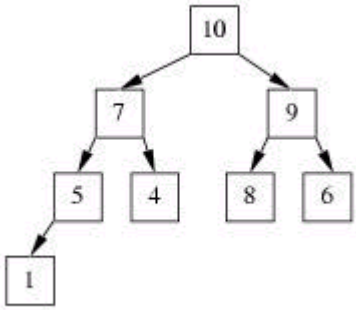


Na árvore da figura, cada nó tem no máximo dois filhos, que nós chamaremos de esquerdo e direito. Essa estrutura, com essas características, é chamada de árvore binária.

Uma árvore heap é uma árvore com as seguintes características:

- árvore binária;
- é armazenada em um vetor;
- os filhos de um nó na posição **i** encontram-se na posição **2*i**, para o filho esquerdo, e **2*i+1**, para o filho direito;
- os filhos são menores que o pai;

Um exemplo de árvore heap é a seguinte:



Veja que tanto o filho esquerdo como o direito (quando existem) de cada nó são menores do que o respectivo pai.

Em um vetor, a árvore heap da figura anterior ficaria armazenada do seguinte modo:

Veja a relação entre os nós e as posições no vetor: o nó que possui o número 7 está armazenado na posição 2. Seus filhos, os nós com os números 5 e 4, estão armazenados respectivamente nas posições 4 e 5.

Em C, definimos uma heap criando um vetor com o número adequado de posições e uma variável que informa o tamanho de elementos no vetor. Como em C, os vetores começam a partir do 0, podemos contornar isso alocando mais uma posição para o vetor, desprezando a posição 0. Para uma heap que armazene 100 números do tipo `float`, teremos:

```
float heap[101];

int tamanho=100;
```

Para obtermos, facilmente, a posicao no vetor dos nós filhos, definiremos duas funções que retornam-nos esse valor:

```
int filho_esq(int i) {
return 2*i;
}

int filho_dir(int i) {
return 2*i+1;
}
```

Voltando à árvore de exemplo, se trocarmos a informação do nó raiz, armazenando o número 3 naquele nó, violaremos uma das condições de integridade da heap (filhos menores que o pai). Para corrigir isso, definiremos uma operação chamada **pushDown** que nos garante que a integridade seja mantida.

A operação de pushDown verifica se um nó em uma posição dada é maior do que cada um de seus filhos. Se não for, troca a informação desse nó com o seu maior filho e efetua, recursivamente, outro pushDown na posição para onde o

nó se moveu, até que os filhos sejam menores que o pai ou até que aquela informação chegue em uma folha (que não possui filhos).

```
void pushDown(int i,float heap[],int tam) {
int ha_esq=0,ha_dir=0;

if (filho_esq(i)<=tam) ha_esq=1;
if (filho_dir(i)<=tam) ha_dir=1;

if (ha_esq&&!ha_dir) {
if (heap[i]<heap[filho_esq(i)]) {
troca(&heap[filho_esq(i)],&heap[i]);
pushDown(filho_esq(i),heap,tam);
}
}
else
if (ha_esq&&ha_dir) {
if ((heap[i]<heap[filho_esq(i)])||(heap[i]<heap[filho_dir(i)])) {
if (heap[filho_esq(i)]>heap[filho_dir(i)]) {
troca(&heap[filho_esq(i)],&heap[i]);
i = filho_esq(i);
}
else {
troca(&heap[filho_dir(i)],&heap[i]);
i = filho_dir(i);
}
pushDown(i,heap,tam);
}
}
}
```

A função pushDown precisa que se determine, através do parâmetro **i** o índice do elemento que será objeto da operação, além dos outros dois parâmetros que são o vetor **heap** e o tamanho **tam** desse vetor.

A função troca deve ser definida anteriormente no programa e, devido à sua simplicidade, não é exibida aqui. Ela simplesmente troca os valores de duas variáveis.

Preste bastante atenção nas variáveis ha_esq e ha_dir. Elas armazenam as informações sobre a existência ou não dos nós filhos esquerdo e direiro, respectivamente. Seus valores ficam determinados após uma verificação que analisa se o índice do nó esquerdo ou direito é maior do que o índice da última posição do vetor.

Após isso, essas informações são utilizadas para fazer a verificação de integridade. Se houver somente um filho à esquerda, testa se ele é maior que o pai, trocando-os de posição caso isso seja verdade. Se o nó possuir dois filhos, testa se um dos dois é maior que o pai e, caso positivo, efetua a troca com o maior deles. Por fim, caso haja troca, é efetuado outro pushDown na posição onde o nó pai está agora, notando-se, então, a recursividade do algoritmo. A recursividade é terminada quando não há nenhum nó filho ou quando todos os nós filhos são menores que o pai.

Frisamos que esse não é o único nem o melhor algoritmo para pushDown existente, porém é um dos mais didáticos. Apesar disso, encontra-se, muitas vezes, implementado de forma incompleta, resultando em uma ordenação que nem sempre organiza os dados corretamente. Nesta aula de C, estamos apresentando a vocês uma implementação didática, desenvolvida **exclusivamente** para a nossa coluna, tendo sida analisada e testada exaustivamente, encontrando-se livre de erros. Dificilmente encontra-se uma implementação desse algoritmo que atende a esses requisitos (sem falsas pretensões).

Concluindo nossas explicações, vamos explicar como o algoritmo de ordenação funciona em sua completude. O heap sort é efetuado em duas etapas:

1)montagem da árvore heap;
2)desmontagem da heap;

O procedimento de montagem da heap toma uma lista qualquer de dados não ordenados, e organiza-os na lista de modo que, eles satisfaçam no final todas as condições de integridade da heap. Neste ponto, já temos os dados em alguma ordem, mas que ainda não é a que desejamos.

O próximo passo consiste na desmontagem da árvore de modo organizado, efetuada em um determinado número de iterações. A cada iteração, a árvore é reduzida de tamanho e os dados são rearrumados, até que se tenha, por fim, uma lista ordenada.

Veremos, a seguir, os códigos para essas duas rotinas:

```
void montaHeap(float heap[],int tam) {
int i;
for (i=tam/2;i>=1;i--) {
pushDown(i,heap,tam);
}
}

void desmontaHeap(float heap[],int tam) {
int i;
int n=tam;
for (i=1;i<=n;i++) {
troca(&heap[1],&heap[tam]);
tam--;
pushDown(1,heap,tam);
}
}
```

A função montaHeap, responsável pela montagem da árvore, faz o que é necessário para que os dados sejam organizados para satisfazer as condições de integridade. Lembram-se de que o pushDown organiza os dados na heap? A montagem da árvore consiste, tão somente, de uma série de pushDowns desde a metade até o início da lista. Fazemos desde a metade e não desde o final porque, analisando a forma como a heap é organizada no vetor, vemos que da metade até o final temos apenas as folhas da árvore. Fazer pushDown nelas, além de desnecessário, é inútil. Porém, a ordem com que os pushDowns são executados (de trás para frente) deve ser respeitada.

A desmontagem do heap é ligeiramente mais complexa, porém também é fácil de ser compreendida. O que a rotina desmontaHeap faz é simplesmente tomar a raiz da heap, que é o maior elemento de toda a árvore, trocá-la de posição com o último elemento. Assim, colocamos o maior elemento no final. Reduzimos então o tamanho da heap e efetuamos um pushDown na nova raiz, que é a posição onde havíamos colocado o último elemento da lista, para que a heap tenha a sua integridade garantida. Repetimos esse procedimento até que o tamanho da heap seja reduzido a 0 (ou seja, o número de iterações é o número de elementos), quando teremos a lista com todos os elementos em ordem crescente.

Note que todas as rotinas de mnipulação da heap apresentadas recebem como parâmetros de entrada uma lista e o seu tamanho, para que funções tenham como manipular os dados de forma correta.

Para entender como o algoritmo funciona, na prática, analisem e compilem o seguinte código: heapsort.c. Para ter uma idéia da superioridade deste algoritmo sobre o bubble sort, tente alterar o numero de elementos para 100, 1000, 10000, ... e comparar com o tempo gasto pelo algoritmo de ordenação da última aula.

Finalmente, chegamos ao fim de nossas aulas sobre estruturas de dados em C. Esperamos que vocês tenham aproveitado bastante os conceitos apresentados e que possam, além de utilizar as estruturas e algoritmos descritos, encontrar neles soluções para problemas encontrados cotidiananamente.

E fiquem ligados a próxima aula! Nela teremos exercícios para poder fixar melhor os conhecimentos e aclarar possíveis

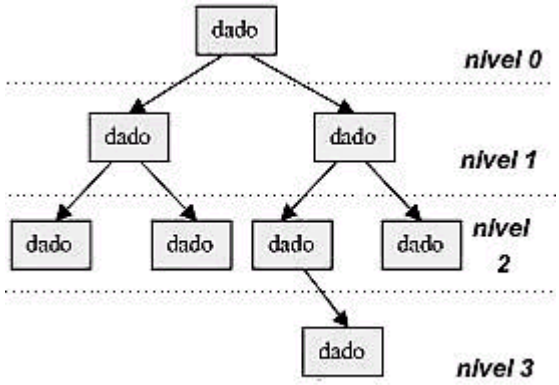
dúvidas.

Aula 34

Olá amigos. Espero que vocês não tenham ficado irritados com a nossa ausência de duas semanas; estávamos nos reestruturando para organizar melhor as novas aulas do curso que se seguirá a este. Mas hoje estamos de volta para concluir o assunto de estruturas de dados com um aula de exercícios para vocês praticarem.

EXERCÍCIOS

1) Um **nível** de uma árvore é o conjunto de nós que estão a uma mesma distância da raiz. Por exemplo, a árvore abaixo possui 4 níveis, de 0 a 3.



Quantos níveis possui uma árvore heap com: a) 7 elementos; b) 8 elementos; c) 15 elementos; d) *x* elementos?

Uma árvore heap com 100 níveis (de 0 a 99) possui, no mínimo, quantos elementos? E no máximo?

2) Implemente uma **lista duplamente encadeada ordenada** para armazenar informações sobre candidados às eleições. Os nós devem conter três campos de informação: o nome, o número de candidato (pode ser um inteiro) e a sua popularidade (um float). Você pode usar a seguinte estrutura para os nós:

```
struct _no {
char nome[40];
int numero;
float popularidade;
struct _no *anterior;
struct _no *prximo;
};
```

A lista encadeada deve ser ordenada por nome (use a função strcmp para comparar strings). Faça um programa que permita inserções, consultas, deleções, listagem em ordem alfabética e em ordem inversa e exibição do número de candidatos inscritos.

3) Altere o programa anterior para incluir uma opção para listar os candidatos por ordem crescente de número de inscrição. Para isso, defina uma função **ord_num** que aloca um vetor cujo tamanho é igual ao número de candidatos e ordena-o pelo método do bubble sort (ordenação por bolha). O vetor pode ser definido da seguinte forma:

```
struct _candidato {
char nome[40];
int numero;
float popularidade;
};

typedef struct _candidato candidato;

candidato *candidatos;
```

Dentro da função ord_num, você deve fazer:
`candidatos=(candidato *)malloc(NumCandidatos * sizeof(candidato));`

4) Inclua mais uma função **ord_pop** que ordene e liste os candidatos por ordem crescente de popularidade. Faça-a de modo semelhante à função anterior, mas usando heap sort.

5) A seguinte implementação (ou similar à essa) da operação de **push down** em uma árvore heap é comumente encontrada em várias referências que descrevem o funcionamento do heap sort. Supondo que as funções troca, filho_dir (que retorna o índice do filho direito) e filho_esq (idem, para o filho esquerdo) estão definidas corretamente e que TAMHEAP armazena o tamanho do heap, também de modo correto, por que o algoritmo não funciona?

```
void pushDown(int i, float heap[ ] ,int tam) {
if (filho_dir(i)<=TAMHEAP) {
if ((heap[i]<heap[filho_esq(i)])||(heap[i]<heap[filho_dir(i)])) {
if (heap[filho_esq(i)]>heap[filho_dir(i)]) {
troca(&heap[filho_esq(i)],&heap[i]);
i = filho_esq(i);
}
else {
troca(&heap[filho_dir(i)],&heap[i]);
i = filho_dir(i);
}
pushDown(i,heap,tam);
}
}
}
```

6) Exercício de cidadania: analise os candidatos às eleições, prestando bastante atenção naqueles que transmitem confiança, honestidade e que sejam realmente comprometidos com os interesses da população. Escolha os que passarem por esse crivo e, no dia das eleições, compareça à urna e vote! Lembre-se: apesar do voto ser eletrônico, nenhum programa de computador pode escolher um candidato para você.

Não são exercícios simples e sei que haverá um grande tempo gasto para elaboração das respostas. Caso achem muito difícil, não desanimem, existem coisas melhores na vida do que isso, como..... refazer a leitura, re-estruturas os pensamentos e tentar de novo !:)

Brincadeira.. semana que vem darei as respostas de tais exercícios, continuem postando no fórum e mandando email caso tenham comentários, dúvidas e sugestões, sobre este e o próximo curso, que está bem próximo. Vale ressaltar que estamos esperançosos que não tenham ficado aborrecidos com nossa ausência, reclamações, pra caixa postal do webmaster (/dev/null pra mim) !:)

Aula 35

Olá amigos! Como foram com os exercícios da semana passada? Espero que vocês tenham conseguido fazer todos. Pelo teor das mensagens que recebi, parece-me que não houve grandes dificuldades.

Como não poderia deixar de ser, hoje teremos a correção dos exercícios com comentários a respeito das soluções apresentadas.

Correção dos Exercícios

1) Os itens a e b foram feitos para serem respondidos por qualquer ser humano que já tenha passado da classe de alfabetização, bastando simplesmente olhar para a própria figura da árvore: com 7 elementos, a árvore tem 3 níveis; com 8 elementos, 4 níveis.

Para responder os itens c e d é necessário um pouco mais de esforço. Com 15 elementos, a árvore também tem 4 níveis. Para um número *x* qualquer de elementos, precisamos fazer umas contas: uma árvore heap com 1 nível tem 1 elemento; com 2 níveis tem, no mínimo 2 e no máximo 3 elementos; com 3 níveis, o número de elementos pode variar ente 4 e 7, e por aí vai. Se olharmos o número máximo de elementos, podemos observar que ele mantém uma relação com o número de níveis:

1 nível => 1 elemento = 2^1-1
2 níveis => 3 elementos = 2^2-1
3 níveis => 7 elementos = 2^3-1
...
n níveis => 2^n-1 elementos

Achamos a fórmula para calcular o número de elementos a partir do número de níveis. Para calcular o inverso, ou seja, quantos níveis para um número tal de elementos, precisamos usar um logaritmo na base 2. Para aqueles que não se afeiçoam muito à matemática ou que não sabem como operar com o logaritmo, podem ir direto à resposta final.

Sendo *x* o número de elementos:

$x = 2^n-1$
$x+1 = 2^n$
$\log_2(x+1) = \log_2(2^n)$
$\log_2(x+1) = n * \log_2 2$

Como $\log_2 2 = 1$,

$\log_2(x+1) = n$

Calculamos assim o número *n* de níveis para *x* elementos.

Vamos fazer uma análise sobre o resultado utilizando valores conhecidos. Com 7 elementos, temos quantos níveis? Basta utlizar a fórmula: $\log_2(7+1) = \log_2 8 = 3$. Está de acordo com o que nós já vimos.

Mas e se a árvore tiver 8 elementos? Peguem as suas calculadoras e me acompanhem: $\log_2(8+1) = \log_2 9 = 3{,}1699250014$. Como uma árvore pode ter 3,169 níveis!? Calma... a calculadora não enlouqueceu, e tampouco nós. Já sabemos que uma árvore com 8 elementos tem 4 níveis. Assim, como não podemos ter 0,1699250014 nível, temos 1 nível inteiro (**arredondamento "para cima"**). Conclusão:

O número de níveis de uma árvore heap com *x* é $\log_2(x+1)$ arredondado "para cima".

Só por curiosidade, em C, a função que calcula o arredondamento "para cima" de um número fracionário chama-se **ceil**. Ela está na biblioteca matemática **math.h** padrão.

Para respondermos as duas últimas perguntas deste exercício, usaremos algo que nós encontramos para achar a solução anterior: *n* níveis => máximo de 2^n-1 elementos. Tendo 100 níveis, uma árvore heap conterá, no máximo, 2^{100} elementos!

O número mínimo pode ser calculado da seguinte forma: pela forma como foi definida a árvore heap, todos os níveis **devem** ter o número máximo de elementos, exceto o último, que pode ter de 1 até o número máximo de elementos. Então, uma árvore heap com 100 níveis, possui, com certeza, 99 deles com o número máximo de elementos. Para ter mais um nível, é necessário ter, no mínimo, 1 elemento. Assim, uma árvore heap com 100 níveis tem, no mínimo, 2^{99}+1 elementos.

Alguns de vocês que estão nos acompanhando podem se perguntar para que serve esse monte de contas. Apesar de parecer inútil à primeira vista, saber o número de níveis de uma árvore heap a partir do número de elementos é importante para estimarmos o tempo de ordenação de uma seqüência por *heap sort*. Este algoritmo de ordenação apresentado na **aula 33** ordena uma série de números em um tempo que possui uma relação com o número de elementos multiplicado pela altura da árvore. Por isso, esse algoritmo é muito melhor que o de ordenação por bolha. onde o tempo para ordenar guarda uma relação com o número de elementos elevado ao quadrado! Se vocês não entenderam essa explicação, não se preocupem, pois isso é tópico de estudos de complexidade de algoritmos, que foge ao escopo do nosso curso. Nesta curta explicação, queremos somente mostrar a vocês os caminhos diversos aos quais a computação nos leva.

A solução para as questões 2, 3 e 4 é o programa candidatos.c. Ele foi feito aproveitando-se as rotinas apresentadas nas aulas 32 e 33, alteradas para poder comportar os tipos de dados requeridos. Compilem este programa e analisem o seu código para entender como ele foi implementado.

O problema da rotina apresentada na questão 5 é que ela somente verifica se a árvore possui um filho direito. Caso exista, faz a verificação de integridade da heap. Porém, uma verificação importante não é feita: checar se, mesmo não existindo um filho direito, existe um filho esquerdo satisfazendo as condições de integridade. Isso causa um problema na ordenação, fazendo com que nem sempre ela seja feita de modo correto; muitas seqüências de dados, ao serem ordenadas desse modo, ficam com o primeiro elemento trocado com o segundo. A implementação correta do algoritmo é a encontrada na **aula 33**.

Hoje tivemos uma redução no nosso ritmo normal só para apresentar as soluções dos exercícios não muito triviais. Na aula que vem, estaremos apresentando tópicos finais no nosso curso de C (é, hehe!). E preparem-se: a derradeira prova para testar os seus conhecimentos em programação nessa linguagem está chegando!

Aula 36

Olá amigos! Depois de uma pequena pausa para fazermos exercícios, hoje voltamos a apresentar bastante conteúdo para vocês. Preparem-se, pois a aula de hoje será bem "puxada". Vamos aparar as últimas arestas para que vocês possam desenvolver programas bem elaborados em linguagem C, utilizando os tópicos que iremos apresentar adiante.

Na reta final do nosso curso, apresentaremos algo que parece bastante estranho, os ponteiros que não apontam para nenhuma estrutura em especial, e observaremos uma peculiaridade bastante interessante da linguagem C, que são os ponteiros para funções.

Vamos arregaçar as mangas e prepararmo-nos para o que está por vir!

Ponteiros para qualquer tipo.

Só para aguçar a curiosidade, vamos mostrar primeiramente uma coisa que provavelmente vocês não esperavam que fosse possível. Vejam esta declaração:

void *alguma_coisa;

Como vocês já sabem, **void** indica, em C, algo que não tem tipo nenhum (não é inteiro, nem float, nem char, ...) e o operador * indica um ponteiro. O que vocês provavelmente não conhecem é a associação acima, que pode gerar a seguinte dúvida: como a linguagem nos permite criar um ponteiro que *não aponta para nada*?

Surpreendentemente, o ponteiro **alguma_coisa** realmente pode apontar para uma região de memória. Mais ainda, ele pode apontar para *qualquer* região de memória válida, que pode conter um *tipo de dados qualquer*. Assim, supondo que tenhamos definido a variável **alguma_coisa** como mostramos acima, podemos fazer as seguintes construções na nossa linguagem favorita:

```
int numero_inteiro;
float numero_fracionario;
char *uma_string_qualquer;

uma_string_qualquer=(char *)malloc(32*sizeof(char));
strcpy(uma_string_qualquer,"Veja só mãe, sem usar as mãos!");
numero_inteiro=2;
numero_fracionario=3.1415926;

alguma_coisa=&numero_inteiro;
printf ("O numero inteiro é: %d\n",*(int *)alguma_coisa);
alguma_coisa=&numero_fracionario;
printf ("O numero fracionario é: %f\n",*(float *)alguma_coisa);
alguma_coisa=uma_string_qualquer;
printf ("A string é: %s\n",(char *)alguma_coisa);
```

Note que, onde imprimimos a variável **alguma_coisa**, colocamos antes da variável o tipo de dados que ela deve conter entre parênteses. Isto é o que chamamos de *type cast* ou *coerção*, um nome bonito para dizer que estamos fazendo a conversão de um tipo para outro. Como **alguma_coisa** é do tipo **void ***, ou seja, não tem tipo definido, temos que informar, na hora de usar a variável, qual o tipo dos dados contidos. Por exemplo, no programa, usamos:

```
(char *)alguma_coisa
```

Para dizer que a variável continha um ponteiro para uma string. Já quando fizemos:

```
*(int *)alguma_coisa
```

Estávamos informando que a variável **alguma_coisa** continha um ponteiro para um inteiro, ou seja, fazendo uma coerção para **int *** e, além disso, pegando o valor do inteiro apontado por ela, através do primeiro "*".

Para ver melhor esta "mágica" (na verdade, diremos mais uma vez: isto não é mágica; é tecnologia), pegue este programa: ponteiro.c

É importante não confundir **void *** com ponteiro nulo. Um ponteiro nulo é um *ponteiro que não aponta para nada, ou seja, ele possui o valor NULL*, enquanto que **void *** é simplesmente um meio de nos referenciarmos a um ponteiro que não possui tipo definido.

Ponteiro para função

Filosofando um pouco, vamos analisar o que é um programa de computador, sem nos atermos a definições precisas. Um programa de computador é uma seqüência de instruções em alguma linguagem. Neste curso, a seqüência de instruções é descrita na linguagem C e gravada em um arquivo. Este arquivo é compilado, ou seja, traduzido para algo que o computador possa guardar em sua memória para ser executado. E o que o computador armazena? Isso mesmo, *dados*.

Conclusão: para um computador, programas, assim como as informações, são um conjunto de dados.

Como já sabemos, os ponteiros apontam para dados na memória. Podemos, então, fazer com que eles apontem para partes do nosso programa? A resposta é sim. Cada função contida em um programa pode ser referenciada por um ponteiro, bastando que nós façamos as coisas do jeito correto.

Vamos supor que a seguinte função seja definida em um programa:

```
int compara_float(void *a, void *b) {
if (*(float *)a==*(float *)b) return 0;
if (*(float *)a<*(float *)b) return -1;
if (*(float *)a>*(float *)b) return 1;
}
```

Essa função compara dois números do tipo **float** e retorna o valor 0 se os dois números são iguais, -1 se o primeiro é menor que o segundo e 1 se o primeiro é maior que o segundo. Note que definimos que os parâmetros são do tipo **void *** e fizemos as coerções, ou conversões de tipo, necessárias para que possamos enxergar **a** e **b** como sendo dois números do tipo **float**.

Podemos definir também outras funções de comparação para avaliar inteiros, strings, etc... que retornem os valores 0, 1 e -1 de acordo com a entrada. Para o nosso exemplo, vamos definir outras duas funções:

```
int compara_int(void *a, void *b) {
if (*(int *)a==*(int *)b) return 0;
if (*(int *)a<*(int *)b) return -1;
if (*(int *)a>*(int *)b) return 1;
}

int compara_string(void *a, void *b) {
return strcmp((char *)a,(char *)b);
}
```

A função **compara_int** funciona de modo similar à função **compara_float**. Na função **compara_string**, usamos a função **strcmp** da biblioteca padrão, que retorna os valores que nós já mencionamos, de acordo com a ordem lexicográfica das strings (veja mais explicações nas **aulas 31** e **16**). Todas as coerções necessárias foram feitas.

Vamos mostrar como podemos fazer uma função que avalie duas variáveis quaisquer, que podem ser dos tipos **int**, **float** ou **char**, e exiba uma mensagem informando qual delas é a menor. Essa função precisa receber os dois valores a serem comparados, além de uma referência para a função que os compara, para que possamos fazer uma correta avaliação.

```
void avalia(void *dado1, void *dado2, int (*func_comp)(void *a, void *b)) {
int valor;
valor = func_comp(dado1,dado2);
if (valor == 0) printf ("As duas variáveis são iguais.\n");
if (valor == -1) printf ("A primeira variável é menor que a segunda.\n");
if (valor == 1) printf ("A primeira variável é maior que a segunda.\n");
}
```

Analisemos a definição da função. **void *dado1** e **void *dado2**, como vimos no tópico anterior desta aula, são ponteiros que podem apontar para tipos de dados quaisquer. Assim, não precisamos nos importar com o tipo dos parâmetros passados.

Para definirmos que a rotina avalia receberá também um *ponteiro para função*, escrevemos: **int (*func_comp)(void *a, void *b)**. Vamos olhar essa construção com bastante cuidado, parte a parte:

- primeiro, definimos o tipo de retorno que a função terá - **int**
- depois, definimos o nome do ponteiro para a função dentro da rotina - **(*func_comp)**;
- por último, informamos quais são os parâmetros da função - **(void *a, void *b)**

Resumindo, a construção apresentada informa que teremos um ponteiro para uma função que retorna um inteiro, que pode ser referenciada pelo nome func_comp, e possui dois parâmetros, a e b, que não possuem um tipo definido (**void ***). Foi por esse motivo que definimos as funções **compara_int**, **compara_float** e **compara_string** aceitando como parâmetros duas variáveis **void ***.

Para chamarmos essa função por meio do seu ponteiro, fazemos como se estivéssemos normalmente chamando uma função:

```
valor = func_comp(dado1,dado2);
```

Tendo definido tudo isso, vamos ver como utilizar a rotina avalia, fazendo um programa que compare vários tipos de dados.

```
main() {
int num1 = 18, num2 = 45;
float num3 = 2.78, num4 = 1.25;
char str1 = "bala", str2 = "bolo";

printf ("Avaliando num1=%d e num2=%d:\n",num1,num2);
avalia(num1,num2,compara_int);
num2=18;
printf ("Avaliando num1=%d e num2=%d:\n",num1,num2);
avalia(num1,num2,compara_int);
printf ("Avaliando num3=%f e num4=%f:\n",num3,num4);
avalia(num3,num4,compara_float);
printf ("Avaliando str1=\"%s\" e str2=\"%s\":\n",str1,str2);
avalia(str1,str2,compara_string);
}
```

O programa completo pode ser encontrado aqui: avalia.c. Ele faz uma série de chamadas à rotina avalia. Cada vez que ela é chamada, passamos como parâmetros os dois valores a serem avaliados e o nome da função adequada à comparação deles.

Todo esse papo da aula de hoje é muito interessante, pois ele mostrou um aspecto um tanto obscuro da linguagem C. Mas creio que algumas pessoas podem ter ficado com dúvidas sobre a real utilidade dos ponteiros para qualquer tipo e dos ponteiros para função.

Uma boa aplicação para os temas apresentados na aula de hoje é no algoritmo *heap sort*. Podemos implementá-lo de modo que ele aceite dados de qualquer tipo, bastando que sejam definidas funções para comparação e troca dos valores nos vetores a serem ordenados. Passaríamos então, como parâmetros, além do vetor a ser ordenado, o nome da função de comparação dos elementos do vetor e o nome da função que faz a troca.

Como vocês já estão ficando "feras" em programação, deixarei isso para que vocês o façam. Na aula que vem, mostrarei para vocês como pode ser esse programa, além de apresentarmos os dois últimos assuntos do nosso curso de C.

Aula 37

Amigos, hoje chegamos ao fim do conteúdo de nosso curso básico de C. Fecharemos este curso com chave de ouro, apresentando um programa que implementa o algoritmo completo de heap sort, que permite que sejam feitas ordenações em qualquer tipo de dados. Também teremos mais dois tópicos finais, que são os ponteiros múltiplos e como passar parâmetros para programas em linha de comando. Este último aspecto permite que deixemos os *softwares* criados por nós com uma aparência mais "profissional". Lembre-se, as próximas aulas teremos exercícios mais cascas, fundamentais para testar seus conhecimentos e ambientar-se com programas mais complexos.

Resposta para o problema da aula passada

Na aula passada aprendemos que podemos criar ponteiros sem tipo predefinido, que podem apontar para qualquer posição de memória válida. Também mostramos como devem ser manipulados ponteiros para função. No fim da aula, foi proposto que fizéssemos um programa para ordenação de um vetor que aceitasse qualquer tipo de dados, para praticarmos os conceitos aprendidos.

O programa completo pode ser encontrado aqui: heapsort.c. Vamos discutir o seu funcionamento.

O algoritmo utilizado é o de ordenação por heap, apresentado na **aula 33**. O programa foi alterado para poder aceitar tipos quaisquer de dados.

Como qualquer algoritmo de ordenação, o heap sort precisa comparar os dados do vetor a ser ordenado. Sendo os dados de um tipo qualquer, preciso informar à função de ordenação **heapSort** como comparar os dados. A comparação pode ser efetuada por meio de uma função que sabe qual o tipo dos dados a serem comparados e que deve retornar um valor que informe qual é menor ou maior. Uma convenção amplamente aceita é fazer uma função de comparação **comparaFunc** que, dadas duas variáveis quaisquer, **a** e **b**, retorna:

- **-1**, se **a** é menor que **b**
- **1**, se **a** é maior que **b**
- **0**, se **a** é igual a **b**

Note que o conceito de ser menor, maior ou igual não se aplica somente a números. Obviamente, se **comparaFunc** for aplicada a dois números, ela deve retornar o resultado esperado. Por exemplo, se fizermos:

comparaFunc(2.5,3), será retornado -1, pois 2.5 é menor que 3
comparaFunc(1.9,0.2), será retornado 1, pois 1.9 é maior que 0.2
comparaFunc(6,6), será retornado 0, pois os dois númros são iguais

Analogamente, **comparaFunc** pode ser definida como uma função que compara *strings*, de acordo com a sua ordem alfabética. Exemplificando, podemos definir essa função de tal modo que:

comparaFunc("Ana","Julia"), retorne -1, pois "Ana" é "menor", ou seja, vem antes de "Julia", em ordem alfabética;
comparaFunc("Bernardino dos Santos","Ze Pedro"), também retorne -1, pois "Bernardino dos Santos" é "menor", ou seja, vem antes de "Ze Pedro", em ordem alfabética;
comparaFunc("Linux","LINUX"), retorne 0, pois apesar de uma string estar em maiúsculas e a outra estar em minúsculas, as duas contém as mesmas letras.

Enfim, o modo como os valores são avaliados é definido "a gosto do freguês", mas a função deverá respeitar os valores de retorno -1, 1 e 0 de acordo com o resultado da comparação. Que tal treinarmos um pouco e fazermos uma função **comparaString** parecida com essa que nós apresentamos? Mais adiante, ela será mostrada.

Voltando ao nosso programa de ordenação, a função **heapSort**, necessita ser informada do ponteiro para a função de comparação. Além disso, precisamos também dizer qual é a função que efetuará a troca dos elementos no vetor.

A função de comparação deve ser feita no formato:

```
int funcao_de_comparacao(void *a,void *b) {
// código da função...
}
```

Ela deve aceitar dois **void ***, pois o ponteiro para essa função será passado para **heapSort**, que não sabe qual é o tipo dos dados a serem ordenados. Dentro da função, devemos fazer a coerção para o tipo de dados que serão comparados, analisá-los e retornar o valor correspondente.

A função de troca também deve ser feita de modo similar, porém ela não tem valor de retorno. As coerções necessárias devem ser feitas. Para maiores esclarecimentos, vejam como isso foi implementado no código.

Resposta para o problema da aula passada (continuação)

Outro aspecto importante para a compreensão do programa diz respeito ao modo como os elementos do vetor são acessados. Normalmente, se queremos acessar um dado elemento de um vetor qualquer, fazemos:

vetor[posicao]

Porém também sabemos que a variável **vetor** é, na verdade, um ponteiro para a primeira posição do vetor. Assim, também podemos escrever:

vetor+**posicao**

Para obtermos o mesmo resultado.

Quando escrevemos isso de um ou de outro modo, o que é feito pelo computador é uma conta para calcular onde está o tal elemento na memória. O primeiro exercício da lista apresentada na **aula 28** ilustra isso.

Suponha que um número inteiro ocupa 2 bytes na memória. Se definirmos a variável:

int ***vetor**

E a fizermos apontar para a posição 65522 da memória, teremos que:

- **vetor[0]** é o inteiro que ocupa as posições 65522=65522+0*2 e 65523;
- **vetor[1]** é o inteiro que ocupa as posições 65524=65522+1*2 e 65525;
- ***(vetor+1)** é a mesma coisa que **vetor[1]**;
- **vetor[i]** é o inteiro que está na posição 65522+i*2 e na seguinte;
- ***(vetor+i)** é a mesma coisa que **vetor[i]**;

Como calculamos a posição de memória onde está um elemento de índice **i** do array **vetor**? Procedemos assim:

1. pegamos o endereço do primeiro elemento, que é apontado pela variável **vetor**. No caso anterior: 65522
2. obtemos a distância do elemento de índice **i** ao primeiro elemento; fazemos isso pegando o tamanho do dado (2 bytes no exemplo) e multiplicando pelo índice **i**, ou seja, **tamanho_do_dado*i**. No exemplo: tamanho_do_dado = 2, então o cálculo é **2*i**
3. somamos o endereço do primeiro elemento à distância calculada: **vetor**+**tamanho_do_dado*i**. Voltando ao exemplo, isso equivale a 65522+i*2.

É um pouco complicado. Ainda bem que o computador faz isso automaticamente quando temos um vetor de um tipo definido.

Quando chamamos a função **heapSort**, o ponteiro para o vetor que é passado como parâmetro é do tipo **void ***, ou seja, sem tipo definido. Assim, a posição de cada elemento no vetor *não pode ser calculada automaticamente*. Para

calculá-la, precisamos informar mais um parâmetro, que é o tamanho, em bytes, de um elemento da lista que queremos ordenar, e fazer as contas como mostrado acima.

A título de curiosidade, existe outra função de ordenação já implementada na biblioteca **stdlib.h** chamada **qsort**, que possui um desempenho um pouco melhor ao da nossa rotina heapSort. Essa rotina aceita os mesmos parâmetros que a heapSort, exceto o ponteiro para a função de troca. Para maiores informaçãoes, vejam na página do manual da **qsort**, executando o seguinte comando no shell do Linux: `man qsort`.

Ponteiros múltiplos

Vamos começar esta seção com uma pergunta: pode um ponteiro apontar para outro ponteiro? E o que será que isso significa?

Vamos pensar: um ponteiro é uma variável que armazena a posição de memória onde está algum dado, ou seja, o ponteiro contém uma referência para essa posição. Se esse dado for um outro ponteiro, teremos *uma variável que aponta para uma referência para uma posição de memória*... Confuso não? A figura pode explicar isso um pouco melhor.

[ponteiro.jpg]

Na posição de memória 33125 temos um número do tipo **float**, cujo valor é 2,78. Um ponteiro para float, que está na posição 65532, contém o valor que corresponde a esse endereço de memória, ou seja, 33125. Existe um outro ponteiro chamdo p1 *que é um ponteiro para um ponteiro* do tipo float, e contém o valor 65532, referenciando, assim, essa posição de memória.

Veremos, então, o que significa:

- **p1** - é um ponteiro para a posição 65532;
- ***p1** - é um ponteiro para a posição 33125, ou seja, para o número;
- ****p1** - é como acessamos o número 2.78;

Repare na existência dos ** no último item. Isso quer dizer que p1 é um **ponteiro para ponteiro**, ou um **ponteiro duplo**.

Para criar um ponteiro para ponteiro do tipo **float**, fazemos:

float **ponteiro;

E qual seria a utilidade de se utilizar ponteiros duplos? Uma das respostas pode ser: para fazer um *array* de *arrays*. Êpa! Está ficando complicado. Vamos exemplificar: suponhamos que temos 3 strings na memória:

```
char *nome1="Astrogildo";
char *nome2="Laurêncio";
char *nome3="Antuérpio";
```

Uma string é, na verdade, um array de caracteres. Agora, se quisermos colocar essas strings em um array, fazemos:

```
char **array_de_strings;
array_de_strings=(char **)malloc(3*sizeof(char *));
array_de_strings[0]=nome1;
array_de_strings[1]=nome2;
array_de_strings[2]=nome3;
```

Repare o que fizemos na linha em que alocamos o espaço para o array. Usamos **sizeof(char *)** para dizer que o tamanho de cada elemento do array é do tamanho de um ponteiro para uma string, ou **char ***; e fizemos **(char **)** para informar que iremos utilizar a área de memória alocada para armazenar um array de arrays.

Se fizermos:

- **printf("%p",array_de_strings);** - será exibido o endereço de memória onde está armazenado o array de strings.
- **printf("%s",*array_de_strings);** - será exibido "Astrogildo", que é a primeira string (**char ***) do array_de_strings. Equivale a **printf("%p", array_de_strings[0]);**
- **printf("%c",*array_de_strings[1]);** - será exibido o primeiro elemento da primeira string, que é a letra 'L'.

Observe este programa e veja como isso funciona: ponteiros.c

Ponteiros múltiplos (continuação)

Outro motivo para utilizarmos ponteiros duplos é quando necessitamos alocar uma matriz de 2 dimensões. Até agora, se quiséssemos alocar uma matriz de um tipo qualquer, como por exemplo, de inteiros, teríamos que fazer desta forma:

int matriz[N][M];

Onde N e M são as dimensões da matriz e devem ser números fixos. Porém, quando nós estamos fazendo um programa, nem sempre é desejável que utilizemos uma matriz com tamanho pré-estabelecido. Às vezes, quem deve determinar o tamanho da matriz é o próprio usuário, no momento em que o programa está sendo executado.

Para evitarmos ter que, a cada vez que o tamanho da matriz for alterado, editar o código e recompilar o programa, podemos alocá-la dinamicamente, utilizando **malloc**.

O problema é que, quando nós alocamos uma área para um vetor, estamos trabalhando com um ponteiro simples e as matrizes não são ponteiros simples.

Quando utilizamos o compilador **gcc**, durante a compilação, as nossas matrizes são "traduzidas" para ponteiros duplos. Utilizando ponteiros duplos, podemos fazer matrizes de tamanhos arbitrários, de modo que o usuário possa especificá-lo durante a execução do programa.

Para isso, se quisermos alocar uma matriz de inteiros de tamanho N por M, por exemplo, devemos fazê-lo seguindo os passos abaixo:

1. definir um ponteiro duplo:
   int **matriz;
2. alocar espaço para as colunas:
   matriz = (int **)malloc(N*sizeof(int *));
3. alocar espaço as linhas, uma de cada vez:
   for (i=0;i<N;i++)
   matriz[i]=(int *)malloc(M*sizeof(int));

O programa completo encontra-se aqui: matriz.c

OBSERVAÇÃO IMPORTANTE: o uso de ponteiros duplos para manipular matrizes não é um padrão da linguagem C! Esse artifício realmente funciona no **gcc** e talvez em muitos outros, mas isto não é garantido. Se você está desenvolvendo um programa que será compilado no gcc, os ponteiros duplos podem ser usados com segurança para alocar matrizes.

Isso é tudo? Não! Podemos também utilizar ponteiros triplos, ou seja, *um ponteiro para ponteiro para um outro ponteiro*, ou até mesmo ponteiros quádruplos, quíntuplos... Um ponteiro triplo pode ser definido do seguinte modo:

<*TIPO*> ***ponteiro_triplo;

Onde <*TIPO*> é o tipo dos dados que serão apontados. Essa construção pode ser utilizada para fazer referências a matrizes tridimensionais (uma matriz tridimensional tem três índices, onde cada um de seus elementos pode ser

acessado deste modo: **matriz[i][j][k]**).

O uso de ponteiros duplos dificulta um pouco a compreensão dos programas, mas muitas vezes é imprescindível utilizá-los. Felizmente, ponteiros triplos são utilizados em raríssimas ocasiões e geralmente podem ser substituídos por outras estruturas.

Quanto mais aumentam os asteriscos à esquerda de uma variável, mais esquisito se torna o seu uso, aumentando a dificulade de entendimento e a probabilidade de erros. Se isto servir-lhes de consolo, em toda a minha vida de programador eu tive a sorte de nunca ter visto um ponteiro quádruplo sendo utilizado em qualquer programa. Moral da história: ponteiro quádruplo é como cabeça de bacalhau - todo mundo sabe que existe, mas ninguém nunca viu!.

Parâmetros em linha de comando

A maioria dos programas que usamos aceitam parâmetros que mudam a forma como eles são executados. Esses parâmetros geralmente são digitados na mesma linha de comando que usamos para iniciar o programa. Um exemplo disso é o próprio gcc; para compilar um programa, usamos:

```
gcc programa.c -o programa
```

Além de digitarmos "gcc", digitamos também as strings "programa.c", "-o" e "programa". As três últimas são ditas parâmetros de linha de comando, e o gcc utiliza-as para saber qual arquivo deverá ser compilado e qual será o nome do executável gerado.

Também podemos fazer com que nossos programas aceitem parâmetros de linha de comando. Basta fazê-lo do modo correto.

Dentro de um programa, para que nós definamos que uma função qualquer aceite parâmetros, devemos colocá-los entre parênteses na definição da função, como por exemplo:

```
float soma(float num1, float num2) {
...
}
```

As variáveis **num1** e **num2** são os parâmetros dessa função. Não confunda os parâmetros de uma função com os parâmetros de linha de comando.

Parâmetros em linha de comando (continuação)

A função **main** de um programa C também aceita dois parâmetros: o primeiro é um inteiro, e o segundo é um vetor de strings (veja a seção anterior). Quando digitamos parâmetros de linha de comando, eles são passados para os parâmetros da função **main**. Para podermos manipulá-los, temos que definir os parâmetros da função **main**:

```
main (int numpar, char **pars) {
...
}
```

Quando o programa for executado, **numpar** conterá o número de parâmetros de linha de comando que foram passados para o programa e **pars** conterá as strings com cada parâmetro.

Há, porém, um detalhe importante. O padrão adotado pelos compiladores de C considera que a linha de comando digitada para executar um programa também é um parâmetro. Assim, **numpars** contém, na verdade o número de parâmetros passados para a o programa somado de 1, e a string **pars[0]** contém o nome do programa.

Para ficar mais claro, vejamos para um exemplo:

```
main (int numpar, char **pars) {

int i;

printf("Número de parâmetros passados: %d\n\n",numpar-1);
printf("Linha de comando: %s\n\n",pars[0]);

printf("Parâmetros passados:\n");
for (i=1;i<numpar;i++)
printf("%s\n",pars[i]);
}
```

Este programa pode ser encontrado aqui: parametros.c. Compile e execute-o.

Para exemplificar, executemos o programa no shell do Linux do seguinte modo:

```
./parametros alo mundo
```

Teremos então a segunte saída:

```
Número de parâmetros passados: 2

Linha de comando: ./parametros

Parâmetros passados:
alo
mundo
```

Se executarmos o programa digitando o caminho completo para o executável, que no meu caso encontra-se no diretório /home/cuco/aula37:

```
/home/cuco/aula37/parametros fim do curso
```

Vamos observar:

```
Número de parâmetros passados: 3

Linha de comando: /home/cuco/aula37/parametros

Parâmetros passados:
fim
do
curso
```

Conclusão

É... acabou. Após semanas de aulas, depois de um monte de conceitos apresentados, terminamos o nosso curso básico de C. E, principalmente, SOBREVIVEMOS!

Agradecemos a todos pela paciência e pelos e-mails enviados. Foram tantos que infelizmente ainda não pudemos responder a todos. Peço a todos que me desculpem pela demora em respondê-los; agora que o curso foi terminado terei mais tempo para poder ler e retornar as suas mensagens.

Continuem mandando suas sugestões por e-mail para o próximo curso a ser ministrado no nosso site. Estaremos considerando cada opinião.

Muito Obrigado e Boa Sorte a todos!

PS.: Ahá! Estão achando que vão escapar da prova? Preparem-se para as próximas semanas, é fundamental você aferir seus conhecimentos e ver a correção de exemplos mais complexos...

PS2.: Semana que vem teremos uma introdução de artigo diferente, aguardem.

PS3.: Repetindo, é fundamental que mandem sugestões com quais próximos cursos deseja ver no OLinux, mande email para o Elias com o subject "Tema da área de Programação".

PS4.: O Feedback em relação ao curso também é importante, mesmo email.

Aula 39

Introdução

Como não poderia deixar de ser, hoje teremos a tão aguardada correção dos exercícios, para que cada um de vocês possa avaliar o seu desempenho. A quantidade gigantesca de emails que recebemos, estamos gratos, e aí vai o gabarito.

Ressalva: a prova deveria ser feita em aproximadamente 4 horas, caso você tenha ficado na média, ou seja, 1 hora a mais ou a menos, tudo ok. Caso tenha saido desse valor, releia as aulas e tente refazer os exercícios. Estude para que possa nos desafios conseguir melhor desempenho. Semana que vem está chegando.

Questões 1 e 2

Questão 1 (estruturas de dados): Faça um programa que peça alguns números ao usuário, armazene-os em uma lista SIMPLESMENTE encadeada, inverta esta lista e liste os números em ordem inversa. A quantidade de números deve ser definida pelo usuário, da maneira que você quiser, mas o algoritmo de ordenação não deve supor nenhuma quantidade fixa pré-definida. Não é permitido que se usem estruturas de dados auxiliares. Trabalhe SOMENTE com ponteiros para os nós desta lista. Não esqueça de atualizar o ponteiro para o nó-raiz. Não é necessário armazenar o ponteiro para o último nó.

Gabarito: Verifique a resposta

Questão 2 (estruturas de dados e ordenação): Uma operação de merge é feita quando juntamos os elementos de duas listas ordenadas em uma terceira lista, sendo que esta terceira lista também está ordenada. O algoritmo de ordenação que utiliza este processo é chamado mergesort e é tão eficiente quanto o heapsort, apesar de exigir o dobro da memória, nas implementações mais simples. Implemente um programa que faça o merge de duas listas de inteiros, como descrito acima. O programa deverá conter uma função merge(), cujo protótipo é o seguinte:

void merge(int *lista1,int *lista2,int *lista3,int tam1,int tam2);

onde "lista1" é o ponteiro para a primeira lista, "lista2" é o ponteiro para a segunda e "lista3", o ponteiro para a lista resultante da operação de merge. Os argumentos "t1" e "t2" são os tamanhos da lista1 e da lista2, respectivamente. Antes de esta função ser chamada, deve ser alocada na memória uma lista de inteiros cujo tamanho é a soma dos

tamanhos da lista1 e da lista2 e deve-se fazer lista3 apontar para esta lista. O corpo do programa está abaixo:

```
#include <stdio.h>
#include <stdlib.h>

//prototipo da funcao:
void merge(int *lista1,int *lista2,int *lista3,int tam1,int tam2);

void main(void){
int t1, t2, t3; //os tamanhos das listas.
int *l1, *l2, *l3; // os ponteiros para as listas.
int i;

//Neste ponto deve-se fazer o preenchimento das listas,
//como voce quiser.
//Faca t1 e t2 conterem os tamanhos da lista1 e
//da lista2, respectivamente.

t3=t1+t2;

l3=(int*)malloc(t3*sizeof(int));
if(l3==NULL){


        printf("%s\n","Erro de alocacao de memoria");
        return; //sai do programa

}

merge(l1,l2,l3,t1,t2);

for(i=0;i<t3;i++){

        printf("%d\n",l3[i]);


}

}//fim main()
```

Gabarito: Verifique a resposta

Questão 3

Questão 3 (ponteiro para função): Implemente um interpretador rudimentar, que execute as 4 operações aritméticas elementares, além de mostrar resultados destas operações na tela. A sintaxe da linguagem entendida pelo interpretador é a seguinte (o que está escrito depois dos dois pontos não faz parte da linguagem):

sum op1 op2 :soma os operandos op1 e op2 e mostra o resultado na tela
sub op1 op2 :subtrai o operando op2 do op1 (faz op1-op2) e mostra o resultado na tela
mul op1 op2 :multiplica os operandos op1 e op2 e mostra o resultado na tela
div op1 op2 :divide o operando op1 pelo operando op2 e mostra o resultado na tela

Cada comando aceita exatamente 2 operadores. Todos os operadores devem ser considerados números em ponto flutuante. Os comandos estarão escritos em um arquivo-texto (um comando por linha), cujo nome será fornecido ao interpretador. O interpretador funcionará como descrito a seguir:

1. Abre o arquivo cujo nome foi fornecido;
2. Lê uma linha do arquivo;
3. Separa a linha lida em 3 sub-strings: a primeira contendo o nome do comando (sum, sub, mul ou div), a segunda contendo o segundo operando e a terceira contendo o terceiro operando.
4. Converte a segunda e a terceira sub-strings em números em ponto flutuante (float ou double, à sua preferência);
5. De acordo com a primeira sub-string, chama a função que executará a operação correspondente, através de um ponteiro para função;
6. Se não chegou ao fim do arquivo, volta para (2).

Eis o corpo do programa (mostrado de forma bem simplificada):

```
#include <string.h>

//Crie 4 funcoes que
//facam as operacoes basicas. A primeira
//funcao já está implementada.

float sum(float a, float b)
{
return a+b;
}

void main(void)
{
//Crie um ponteiro para funcao do tipo float.
//Suponha que voce chamou este ponteiro de "p"

//Supondo que a variavel str1 contem
//a string com o nome do comando, executo
//o codigo a seguir:

if(!cmp(str1,"sum")){

        //Atribua ao ponteiro "p" o endereco da
        //funcao "sum".

}

if(!cmp(str1,"sub")){

        //Atribua ao ponteiro "p" o endereco da
        //funcao "sub".


}

if(!cmp(str1,"mul")){

        //Atribua ao ponteiro "p" o endereco da
        //funcao "mul".


}

if(!cmp(str1,"div")){

        //Atribua ao ponteiro "p" o endereco da
        //funcao "div".
```

```
}

//Execute a funcao apontada por "p"

//OBS: parte do código acima estará em um loop,
//onde é feita a leitura do arquivo. O loop está
//omitido.

}
```

Gabarito: Verifique a resposta

Questão 4

Questão 4 (estruturas de dados): Aproveitando o interpretador implementado na questão anterior, construa um interpretador que, além de aceitar o conjunto de instruções acima, seja capaz de armazenar valores em variáveis, fazer operações com variáveis e escrever e ler valores de uma pilha.

Cada vez que uma operação aritmética (sum, sub, mul e div) é executada, o resultado ficará armazenado em uma variável chamada "ans". Existirão duas variáveis manipuláveis diretamente, chamadas "r1" e "r2". As operações aritméticas poderão receber como argumentos números em ponto flutuante ou as variáveis r1 e r2.

Existirá um comando "mov" que receberá como argumento uma das variáveis r1 ou r2 e armazenará o valor de ans na variável especificada. Haverá dois comandos para manipulação da pilha: push e pop, que receberão como argumento o nome da variável (r1 ou r2) cujo valor será guardado ou lido da pilha.

Nenhum resultado será mostrado na tela até que se execute o comando "show", que recebe um nome de variável (r1 ou r2) e o mostra na tela.

Eis um exemplo:

```
sum 5 2 (ans recebe 7)
mov r1 (r1 recebe o valor 7)
div 15 5
mov r2 (r2 recebe o valor 3)
push r2 (o topo da pilha fica com o valor 3)
mul r1 4
mov r1 (r1 vale 28)
push r1 (o topo da pilha fica com o valor 28)
pop r2 (r2 vale 28)
pop r1 (r1 vale 3)
sub r2 r1
mov r1 (r1 vale 25)
show r1 (mostra "25" na tela)
```

A variável ans deverá ser apenas de leitura. A execução deste interpretador será semelhante à do primeiro (abre o arquivo especificado, lê uma linha por vez, etc.). Porém, agora existe a diferença de que uma linha nem sempre tem 3 sub-strings, pois alguns comandos possuem só um argumento. Continue utilizando ponteiros para funções. As variáveis ans, r1 e r2 podem ser representadas internamente por variáveis globais.

Gabarito: Verifique a resposta

Quem tiver sugestões e quiser mandar o tema para o próximo curso, enviem. Qualquer comentário também é bem vindo, de qualquer natureza, sejam críticas, sugestões ou elogios.

Aula 40

Esta semana estaremos fazendo exercícios desafios muito práticos. Estaremos vendo um pouco de programação em Xlib (X Window System), aprendendo a compactar e descompactar imagem, além de listar e desmontar/montar devices utilizando chamadas ao sistema.

Exercício Desafio 1

Como sabemos, o arquivo /etc/mtab guarda as informações de quais devices estão montados no momento. Vamos então ver se o disquete está montado e se estiver perguntar ao usuário se ele quer demonstá-lo. Caso contrário, pergunte ao usuário se quer montá-lo.

Como fazer?

Procure pelo device do disquete (/dev/fd0) no /etc/mtab.

Caso encontre, pergunte ao usuário se quer desmontá-lo. Caso queira, faça a chamada ao sistema que desmonta o disquete, através da função system(), assim:
system("umount /dev/fd0");

Caso não encontre, pergunte ao usuário se quer montá-lo. Caso queira, faça a chamada ao sistema que monta o disquete, através da função system(), assim:
system("mount /dev/fd0");

Exercício Desafio 2

Listar todos os devices contidos no /etc/fstab exibindo por grupo. Exemplo:
ext2: /dev/hda3, /dev/hda2
iso9660: /dev/cdrom
auto: /dev/fd0

Depois perguntar ao usuário se quer montar os devices (perguntando de um em um).

montar: system("mount [device]"); desmontar: system("umount [device]");

OBS: Ignorar swap e proc

Exercício Desafio 3

Vamos ver um pouco de Xlib.

Arquivos:
main.c
color.h
video.h
Makefile
x.olinux
xpak.olinux

Baseado no código **main.c** em anexo e no arquivo de imagem **x.olinux**, terminar a função **draw()** do arquivo **main.c** de maneira que desenhe a imagem do arquivo **x.olinux** na janela.

A imagem tem os 2 primeiros bytes indicando largura e altura, o resto eh a imagem em si.

A funcao pix() desenha um pixel na tela. Exemplo:
pix(50,30,blue);
Isto desenha um ponto azul (blue) de coordenada x igual a 50 e coordenada y igual a 30.

Leia o vetor img[] a partir da posicao 2 (img[2]) até o final (img[66]) para desenhar a figura que foi carregada nele. Os dois primeiros bytes de img (img[0] e img[1]) indicam largura e altura da imagem, respectivamente. Para desenhar, faça um "for" dentro do outro, sendo o do eixo y o mais de fora, simulando assim, uma matriz "largura X altura" no vetor img[].

Exemplo: (preenchendo um retangulo de azul)
i=2;
for (y=0;y<altura;y++)
for (x=0;x<largura;x++) pix(x,y,blue); Detalhe: Baseie-se no código acima para desenhar a imagem (que é uma matriz largura X altura). Apenas desenhe onde img[] valer 1. Lembre-se de começar a desenhar depois dos 2 primeiros bytes de img.

Para compilar o programa, rode no xterm, no diretório onde colocou os arquivos:
$ make

Para rodar o programa, entre no xterm e no diretório onde compliou e execute: **$ ./olinux**

Exercício Desafio 4

Compacte o arquivo **x.olinux** (mudando o nome dele para **xpak.olinux**) de forma que fique com os seguintes bytes quando compactado:

8 8 2 1 4 0 5 1 2 0 3 1 1 0 6 1 3 0 4 1 4 0 4 1 3 0 6 1 1 0 3 1 2 0 5 1 4 0 2 1

Significando que a imagem tem dimensoes 8x8 (os dois primeiros bytes).

O resto eh a imagem compactada, sendo montada de 2 em 2 bytes. Por exemplo:

Se tenho a sequencia de bytes no x.olinux: **5 5 5 5 4 4 4 3 0 0 0**
Compacte em: **4 5 3 4 1 3 3 0**
Que significa: 4 **vezes** o **byte** 5, 3 **vezes** o **byte** 4, 1 **vez** o **byte** 3, 3 **vezes** o **byte** 0.

Faça tambem o descompactador.

No **xpak.olinux** (que voce irá criar) ele comecará assim: **8 8 2 1 4 0 5 1** Que descompactando irá ficar: **8 8 1 1 0 0 0 0 1 1 1 1 1**

E com o arquivo gerado (descompactado em **unpak.olinux**) teste-o no exercício anterior (desenhar a imagem) para saber se compactou e descompactou direito, comparando-o com o x.olinux (que deve ser exatamente igual).

Semana que vem tem a correção. Não se preocupem se errarem, o importante é tentar. E mesmo que errem, vendo a correção vocês aprendem mais. Vamos lá, tentem! E lembrem-se, bom programador é aquele que treinar sempre...

Aula 41

Na aula de hoje você confere a correção dos exercícios 3 e 4 da **aula passada**. Os exercícios 1 e 2 serão corrigidos na próxima aula. Veja seu desempenho!

Correção do Exercício Desafio 3 (main.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>

XEvent event;
int w,h;
Display *disp;
Window win;
Colormap cmap_padrao;
XSetWindowAttributes attr;
int sc;
int pixel[256];
GC gc;

#include "video.h"
#include "color.h"

void draw() {

        FILE *arq;
        char img[102];
        int i,x,y;
        int altura, largura;

        arq=fopen("x.olinux","rb");
        if (arq==NULL) {

                printf("Arquivo x.olinux nao existe\n");
                exit(0);

                }
                fread(img,1,66,arq);
                fclose(arq);

                largura=img[0];
                altura=img[1];

                i=2;
                /* 50 e +50 para ver mais no meio da janela */
                for (y=50;y<altura+50;y++)

                for (x=50;x<largura+50;x++) {
                if (img[i]==1) pix(x,y,blue);
                i++;
                }

}

main() {

        int fim=0;
```

```
        Window root;

        w=h=512;

        // Tenta abrir o Display 0 do X
        disp=XOpenDisplay(0);
        if (disp==NULL) {

                printf("Erro ao abrir display\n");
                exit(0);

                }
                // Screen (tela)
                sc=XDefaultScreen(disp);
                // Graphic Context
                gc=XDefaultGC(disp,sc);
                // Mapa de cores
                cmap_padrao=DefaultColormap(disp,sc);
                // Definicao das cores
                setc(0, 0, 0, 0); // preto
                setc(1, 0xffff, 0xffff, 0xffff); // branco
                setc(2,0,0,0xaaaa); // azul
                setc(3,0x9999,0x9999,0x9999); // cinza escuro
                setc(4,0xaaaa,0xaaaa,0xaaaa); // cinza
                setc(5,0,0,0x5555); // 0,0,5555 - fundo azul
                // Janela raiz do X
                root=DefaultRootWindow(disp);
                // Criando uma janela comum
                win=XCreateSimpleWindow(disp,root,0,0,w,h,0,0,pixel[1]);

                // Escolhe os dispositos que geram sinais (eventos) pro X
                XSelectInput(disp, win, KeyPressMask | ExposureMask);
                // Mapeia esses eventos na minha janela (garantia)
                XMapWindow(disp,win);

                // Desenhe na tela
                draw();

                while (!fim) {

                // Esvazia o buffer de eventos
                XFlush(disp);
                // Pega o proximo evento
                XNextEvent(disp,&event);
                // Se o evento for uma tecla pressionada, entao sai do programa
                if (event.type == KeyPress) fim++;
                // Se o evento for a exposicao da janela (focus) redesenhe
                if (event.type == Expose) draw();

                }

}
```

Correção do Exercício Desafio 4 (comp.c)

```
#include <stdio.h>
```

```
main() {

        FILE *arq;
        char img[66];
        int i,j;
        char quanto,byte;

        arq=fopen("x.olinux","rb");
        if (arq==NULL) {

                printf("Arquivo x.olinux nao encontrado\n");
                exit(0);

                }
                fread(img,1,66,arq);
                fclose(arq);

                arq=fopen("xpak.olinux","wb");

                // Escreve os 2 primeiros bytes (indica o tamanho)
                fwrite(&img[0],1,1,arq);
                fwrite(&img[1],1,1,arq);
                j=2;
                while (j<66) {

                quanto=1;
                byte=img[j];
                j++;
                if (img[j]==byte) {

                        while (img[j]==byte && j<66) {

                                j++; quanto++;

                                }

                                fwrite(&quanto,1,1,arq);
                                fwrite(&byte,1,1,arq);

                        }
                        fclose(arq);

}
```

Correção do Exercício Desafio 4 (unpak.c)

```
#include

main() {

        FILE *arq;
        char img[66];
        int i;
        char quanto,byte;
        arq=fopen("xpak.olinux","rb");
        if (arq==NULL) {
```

```
                printf("Arquivo xpak.olinux nao existe\n");
                exit(0);

                }
                // Pegando largura
                fread(&byte,1,1,arq);
                img[0]=byte;
                // Pegando altura
                fread(&byte,1,1,arq);
                img[1]=byte;

                // Descompactando
                i=2;
                while (i<66) { // Ja sabemos que descompactado vai ter tamanho 66

                fread(&quanto,1,1,arq);
                fread(&byte,1,1,arq);
                while (quanto != 0) {

                        img[i]=byte;
                        i++;
                        quanto--;

                        }

                        }
                        fclose(arq);

                        arq=fopen("unpak.olinux","wb");
                        fwrite(img,1,66,arq);
                        fclose(arq);

}
```

Aula 42

Finalmente, na aula de hoje, você confere a correção dos exercícios 1 e 2 da **aula 40**. Espero que todos tenham obtido um bom desempenho!

Correção do Exercício Desafio 1 (cd_disq.c)

```
#include <stdio.h>
#include <string.h>

#define CD 0
#define DISQ 1

char linha[50][1024];
int nlinhas=0;

abre_mtab() {
```

```
        FILE *arq;
        arq=fopen("/etc/mtab","rb");
        while (!feof(arq)) {

                if (!feof(arq)) {

                        fgets(&linha[nlinhas][0],1024,arq);
                        nlinhas++;

                        }

                        }
                        fclose(arq);

}

char tem(char *s) {

        int k;
        for (k=0;k<nlinhas-1;k++) {

                int i=0;
                // Vamos forcar a ser string a primeira palavra do arquivo,
                // ou seja, ate achar espaco.
                while (linha[k][i]!=' ') i++;
                linha[k][i]='\0';
                if (!strcmp(&linha[k][0],s)) {

                        // ok volta ao normal
                        linha[k][i]=' ';
                        return 1;

                        }
                        // volta ao normal
                        else linha[k][i]=' ';

                        }
                        return 0;

}

char cd_montado() {

        int i,j;
        for (i=0;i<nlinhas;i++) {

                if (tem("/dev/cdrom")) {

                        return 1;

                        }

                        }
                        return 0;

}
```

```
char disq_montado() {

        int i,j;
        for (i=0;i<nlinhas;i++) {

                if (tem("/dev/fd0")) {

                        return 1;

                        }

                        }
                        return 0;

}

p_montar(int i) {

        char q;
        if (i==CD) printf("Deseja montar o CDROM? [S/N] ");
        else printf("Deseja montar o Disquete? [S/N] ");
        scanf("%c",&q);
        getchar();
        if (q=='s' || q=='S') {

                if (i==CD) system("mount /dev/cdrom");
                else system("mount /dev/fd0");

                }

}

p_desmontar(int i) {

        char q;
        if (i==CD) printf("Deseja desmontar o CDROM? [S/N] ");
        else printf("Deseja desmontar o Disquete? [S/N] ");
        scanf("%c",&q);
        getchar();
        if (q=='s' || q=='S') {

                if (i==CD) system("umount /dev/cdrom");
                else system("umount /dev/fd0");

                }

}

main() {

        abre_mtab();
        if (!cd_montado()) p_montar(CD);
        else p_desmontar(CD);
        if (!disq_montado()) p_montar(DISQ);
        else p_desmontar(DISQ);

}
```

Correção do Exercício Desafio 2 (part.c)

```
#include <stdio.h>
#include <string.h>

#define EXT2 0
#define VFAT 1
#define NFS 2
#define ISO 3

char linha[50][1024];
int nlinhas=0;


char ext2[5][1024];
char next2=0;
char vfat[5][1024];
char nvfat=0;
char nfs[5][1024];
char nnfs=0;
char iso[5][1024];
char niso=0;

abre_fstab() {

        FILE *arq;
        arq=fopen("/etc/fstab","rb");
        while (!feof(arq)) {

                if (!feof(arq)) {

                        if (!feof(arq)) fgets(&linha[nlinhas][0],1024,arq);
                        nlinhas++;

                        }

                        }
                        fclose(arq);

}
```

Correção do Exercício Desafio 2 (continuação)

```
char ver_tipo(char *s) {

        int i=0;
        int j=0;
        char buffer[256];
        // primeiro espaco
        while (s[i]!=' ') i++;
        while (s[i]==' ') i++;
        // segundo espaco
        while (s[i]!=' ') i++;
        while (s[i]==' ') i++;
        // Achei, copia para buffer
        while (s[i]!=' ') {
```

```
                buffer[j]=s[i];
                i++; j++;

                }
                // para indicar fim de string
                buffer[j]='\0';
                // compara ignorando case
                if (!strcasecmp(buffer,"ext2")) return EXT2;
                if (!strcasecmp(buffer,"vfat")) return VFAT;
                if (!strcasecmp(buffer,"nfs")) return NFS;
                if (!strcasecmp(buffer,"iso9660")) return ISO;

                return -1;

}

perg_montar(char *s) {

        char c;
        char comando[256];
        printf("Device: %s\n",s);
        printf("Deseja monta-lo? [S/N] ");
        scanf("%c",&c);
        getchar();
        if (c=='s' || c=='S') {

                sprintf(comando,"mount %s",s);
                system(comando);

                }

}

separa_grupos() {

        int k,tipo;
        for (k=0;k<nlinhas-1;k++) {

                int i=0;
                // Vamos forcar a ser string a primeira palavra do arquivo,
                // ou seja, ate achar espaco.
                while (linha[k][i]!=' ') i++;
                // verifica o tipo
                tipo=ver_tipo(&linha[k][0]);
                // forca virar string
                linha[k][i]='\0';
                switch (tipo) {

                        // joga para o grupo certo
                        case EXT2: strcpy(&ext2[next2][0],&linha[k][0]);

                                next2++; break;

                                case VFAT: strcpy(&vfat[nvfat][0],&linha[k][0]);

                                nvfat++; break;
```

```
                                case NFS: strcpy(&nfs[nnfs][0],&linha[k][0]);

                                nnfs++; break;

                                case ISO: strcpy(&iso[niso][0],&linha[k][0]);

                                niso++; break;

                                default: break;

                                };
                                // volta ao normal
                                linha[k][i]=' ';

                        }
                        printf("EXT2:\n");
                        for (k=0;k<next2;k++) perg_montar(&ext2[k][0]);
                        printf("VFAT:\n");
                        for (k=0;k<nvfat;k++) perg_montar(&vfat[k][0]);
                        printf("NFS:\n");
                        for (k=0;k<nnfs;k++) perg_montar(&nfs[k][0]);
                        printf("ISO:\n");
                        for (k=0;k<niso;k++) perg_montar(&iso[k][0]);

}

main() {

        abre_fstab();
        separa_grupos();

}
```