# Direcionamentos básicos de programação

Este tutorial tem como objetivo dar alguns direcionamentos para os erros comuns cometidos por pessoas iniciantes.

## 1 Direcionamento 1: Tratar múltiplos casos de testes.

Muitas vezes, os problemas requerem que o programa trate múltiplos casos de testes contidos uma mesma entrada. Em geral, existem três formas comuns de se definir múltiplos casos de testes em uma mesma entrada. Nas seções seguintes, são explicadas cada forma e como você pode tratá-las.

### 1.1 Problemas onde o número de casos de teste é fornecido

Este tipo de problema tem, em geral na primeira linha, o número de casos testes que a entrada possui. É o caso mais fácil que pode ser resolvido com uma estrutura de repetição "for":

```
...
int num_testes,i;
scanf("%d",&num_testes);
for(i=0;i<num_testes;++i){
  //Trata cada caso de teste.
}
...
```

### 1.2 Problemas onde a entrada termina com um valor especial (sentinela)

Em vez de o número de casos de teste ser dado de forma explícita, você deve ler casos da entrada até encontrar um valor especial, descrito no enunciado do problema, que sinaliza o fim da entrada. Como o número de casos de teste não é conhecido a priori, não é possível determinar um limite para o número de iterações. Uma forma comum de tratar esse tipo de problema é ter um loop infinito, e um teste dentro do loop que verifica a condição de parada. Por exemplo, se um problema define que o caso de teste termina com uma entrada 0, então:

```
...
int n;
while (true) {
  scanf("%d",&n);
  if (n == 0)
    break; // Fim da entrada encontrado, sai do loop
    ...
}
...
```

### 1.3 Problemas onde a entrada termina com o fim do arquivo

Há também problemas onde o número de casos não é dado, e nem há um valor sentinela que indique o fim da entrada. Nesses casos, você deve ler todos os casos de teste até o fim do arquivo. Assim como os problemas em que o fim da entrada é indicado por um valor sentinela, nesses problemas é impossível definir um número de iterações a priori. Logo, usar um loop infinito e uma condição de parada também é uma idéia interessante. Basta alterar a condição de parada para algo que teste se o fim do arquivo foi alcançado.

Note que você tem que testar se chegou ao fim do arquivo apenas depois de tentar ler um caso de teste extra. Isso acontece porque, mesmo que você já tenha lido todos os casos de teste do arquivo de entrada, o stdin (C) só vai saber que o fim do arquivo foi alcançado quando você tentar ler alguma coisa e não conseguir. Lembre-se disso, pois é importante.

```
...
int entrada;
while (true) {
  scanf("%d",&entrada);
  if (feof(stdin))
    break;
    ...
  }
...
```

## 2 Direcionamento 2: Referências básicas

Há vários tipos de livros úteis sobre projeto de algoritmos. Recomendamos fortemente três deles:

- Introduction to Algorithms, de Thomas Cormen
- Introduction to Algorithms: A Creative Approach, de Udi Manber
- The Algorithm Design Manual, de Steven Skiena

O livro do Cormen é o livro oficial da disciplina, pois ele abrange um grande acervo de tópicos em detalhes. Entretanto, ele não é muito bom para ensinar a resolver problemas. Por isso, indicamos o livro do Manber e do Skiena.

O livro do Manber é certamente um dos melhores livros sobre projeto de algoritmos já escritos, apesar de não ser muito conhecido. Ele não cobre muita coisa, mas o que cobre ele cobre extremamente bem, e ele dá uma boa base para atacar mesmo os problemas não cobertos. É um ótimo livro para ensinar a pensar em como resolver problemas.

O livro do Skiena cobre um pouco mais de conteúdo que o do Manber, mas com um pouco menos de detalhe. Ele é recomendado por alguns motivos. O primeiro é que ele dá ênfase ao fato de que vários problemas algorítmicos podem ser resolvidos por redução a um problema clássico. Isso é importante na vida de quem lida com algoritmos. O segundo é que ele serve como um excelente guia de referência por mostrar um catálogo de problemas clássicos quase completo.

A dica é: Use o livro do Cormen. Se algo não ficou claro ou não é coberto pelo livro oficial, procure nos outros. Lembre-se que isso é apenas uma dica para te direcionar e que o ideal é ler os três livros :)

# 3 Direcionamento 3: Projetar algoritmos

Não existe uma receita para projetar algoritmos mas algumas práticas podem facilitar a vida. A primeira delas é entender o problema. Não entender direito o problema a ser resolvido pode te fazer projetar o algoritmo errado. Leia o problema diversas se for preciso mas entenda o problema.

Depois de dominar o problema, pense em algoritmos para resolvê-lo. Rabiscar cadernos, utilizar quadros e discutir com outras pessoas faz parte do processo. O objetivo é que, ao final deste processo, você tenha uma ideia de boas soluções para o problema. Nesta etapa é essencial entender os custos dos algoritmos em termos assintóticos de tempo e espaço para poder escolher a solução que melhor te atenda.

Por exemplo, suponha que um problema é encontrar a n-ésima permutação de um conjunto de elementos. Pensando em formas de resolvê-lo, você encontra duas formas: uma em tempo linear e outra em tempo exponencial em relação ao tamanho do conjunto. Qual escolher? A resposta é depende. Se o problema define um número máximo de elementos no conjunto que é baixo (suponhamos 5, que dá 5! permutações) e o algoritmo exponencial é mais rápido de implementar, a escolha do mesmo pode ser uma boa ideia. Entretanto, suponha que o número máximo de elementos no conjunto é alto (suponhamos 1.000, que dá 1.000! permutações). Mesmo que um algoritmo exponencial seja mais rápido de implementar, ele não vai rodar no pior caso. Logo, é importante estar ciente de que o algoritmo linear é a única solução viável dentro das que você enumerou.

Depois de pensar na solução, você tem que implementá-la em alguma linguagem de programação. Nesta etapa, é importante conhecer os tipos básicos da linguagem e as funções da biblioteca padrão (STL). Usá-las é bom pois economiza tempo e te oferece um código de qualidade e que foi bem testado. Entretanto, às vezes, é preciso entender o funcionamento das funções da STL. Por exemplo, você pode considerar que a função de comparar duas strings tem custo constante mas, na verdade, ela é linear no tamanho da menor string. Dependendo do número de operações de comparação de string que você fizer, uma parte considerável do tempo do algoritmo pode estar na STL que você está utilizando. Fique atento nas funções básicas que você utilizar e avalie o impacto que elas têm no algoritmo que você projetou.

Por fim, teste seu código e direito! Muitos problemas vem com restrições. Teste seu algoritmo no limite dessas restrições. Por exemplo, veja o que acontece com seu programa quando a entrada já inicia com um sentinela ou está vazia (não tem casos de teste). Se o exercício for uma permutação e o número máximo de elementos é 1.000, teste seu algoritmo com 1.000 elementos. Não seja preguiçoso e teste. Caso encontre um erro, revise o código. Se você tem certeza que o programa está correto, consulte outras pessoas. É comum outros verem erros que você não vê no seu próprio código.

# 4 Direcionamento 4: Modularização

Nos "Trabalhos práticos", é essencial que você modularize seu código. Modularizar significa dividir seu código em funções bem definidas. Um exemplo trivial é um trabalho que envolva a soma de dois números, A e B. Inicialmente, você pode fazer da seguinte forma:

```
...
printf("%d",A+B);
...
```

Entretanto, você poderia modularizar a soma como uma função. Por exemplo, você poderia criar uma função "Soma_Numeros(int A,int B)", conforme abaixo:

```
int Soma_Numeros(int A, int B){
  return A+B;
}
```

Dada esta função (Soma_Numeros), você poderia chamá-la diversas vezes, sempre quando necessário. Por exemplo, no código principal:

```
...
printf("%d",Soma_Numeros(A,B));
...
```

Um terceiro ponto é você separar o código em múltiplos arquivos. Por exemplo, você poderia querer construir uma mini biblioteca de operações matemáticas básicas com soma, multiplicação, divisão e subtração de inteiros. Para isso, você deve criar um arquivo de "header" (cabeçalho) para definir as estruturas presentes na sua mini biblioteca. Para a mini biblioteca de operações matemáticas, podemos criar um arquivo "operacoes.h", conforme abaixo:

```
#ifndef OPERACAO
#define OPERACAO
int Soma_Numeros(int A,int B);
int Subtrai_Numeros(int A,int B);
int Multiplica_Numeros(int A,int B);
```

```
6 int Divide_Numeros(int A,int B);
7 #endif
```

Observe que definimos apenas os cabeçalhos das funções neste arquivo. Isso porque ele funciona apenas como uma lista do que existe na mini biblioteca. Além do cabeçalho das funções, é comum colocar no arquivo ".h" as estruturas abstratas de dados, como "struct" e "typedef" (Abordado mais abaixo). No código, existem as flags "#ifndef OPERACAO", "#define OPERACAO" e "#endif". Essas flags são úteis para evitar a inclusão do cabeçalho mais de uma vez pelo compilador, podendo gerar erros. Dessa forma, a primeira flag verifica se o nome definido existe (em nosso caso, "OPERACAO"). Se não existir, ele define este nome através da segunda flag. A última flag serve para definir o escopo de atuação do "#ifndef", ou seja, tudo que estiver entre "#ifndef" e "#endif" será afetado.

Dado o arquivo de cabeçalho, você deve implementar as funções para ele. Neste caso, crie um arquivo com o mesmo nome do header, porém com extensão ".c". No nosso caso, esse arquivo chamará "operacoes.c", que é definido abaixo:

```
1  #include "operacoes.h"
2  int Soma_Numeros(int A, int B){
3    return A+B;
4  }
5  int Subtrai_Numeros(int A, int B){
6    return A-B;
7  }
8  int Multiplica_Numeros(int A, int B){
9    return A*B;
10 }
11 int Divide_Numeros(int A, int B){
12   return A/B;
13 }
```

Tudo certo. Sua mini biblioteca está pronta. Para utilizá-la em outros programas, basta incluir o arquivo do cabeçalho ".h", conforme abaixo:

```
1 #include "operacoes.h"
2 int main(){
3   int A=2,B=2;
4   printf("%d",Soma_Numeros(A,B));
5   ...
6 }
```

Você já sabe agora o básico para modularizar seus programas. O exemplo acima é didático mas, ao longo dos TPs, você sentirá necessidade de modularizar seu código. Por exemplo, um trabalho pode exigir que você use uma "Lista Encadeada". Para isso, você poderia criar uma "lista.h", que definirá as estruturas e as funções de operações sobre as listas e uma "lista.c", onde ficarão as implementações das operações.

# 5 Direcionamento 5: Fazendo um Makefile

A primeira pergunta é: O que é um Makefile? É um arquivo que define as diretrizes de compilação do seu código. Muita das vezes, quando o código, em "C" está em um arquivo só, basta executar o comando abaixo:

```
1 gcc <flags> <arquivo.c> -o <binario>
```

O problema surge quando existem vários arquivos a serem mesclados para gerar o código final. Isso ocorre quando o código está dividido em diversos arquivos para uma melhor modularização (Direcionamento 4). Para isso, existe um programa "Make" que, quando invocado (*make*), chama um arquivo chamado "Makefile" dentro do diretório corrente. O arquivo "Makefile" contem as instruções de compilação do programa. A sintaxe básica de um arquivo "Makefile" é:

```
1 <nome da regra>: <dependencias>
2 <TAB> <comando>
```

Cada regra é lida e as dependências são arquivos ou regras. No exemplo da mini biblioteca no Direcionamento 4, com os arquivos "main.c", "operacoes.h" e "operacoes.c", poderíamos fazer criar o seguinte arquivo, com nome "Makefile", dentro do diretório onde estão os fontes do programa:

```
1 operacoes: operacoes.o main.o
2 <TAB> gcc operacoes.o main.o -o operacoes
3 main.o: main.c operacoes.h
4 <TAB> gcc -g -c main.c
5 operacoes.o: operacoes.h operacoes.c
6 <TAB> gcc -g -c operacoes.c
```

A primeira linha define o objetivo que queremos: o binário "operacoes". Entretanto, para alcançar esse objetivo (executar a regra de "operacoes"), é necessário os arquivos "operacoes.o" e "main.o", que são códigos já traduzidos para linguagem de máquina (binário) mas que ainda não foram ligados com as funções externas a ele. Por exemplo, main.o sabe que a função "Soma_Numeros" está definida em algum endereço de memória mas não sabe exatamente qual é o endereço. Chamados esse binário de "objeto(.o)". A pergunta é: porque tudo não é passado diretamente para código de máquina? A resposta é simples: eficiência. Quando a tradução do código para binário ocorre, isso gera um custo que, em algumas vezes, é caro (demora muito tempo). Dessa forma, não é eficiente traduzir todo o código se apenas uma pequena modificação for feita. Por isso, o programa é dividido em vários ".o" e, quando uma modificação é feita em apenas um deles, apenas o ".o" em questão é compilado novamente. Tudo que você precisa saber é isso. Para maiores detalhes, curse as disciplinas de "Software Básico" e "Compiladores".

Definimos, então, como criar os arquivos "operacoes.o" e "main.o" nas linhas 3 e 5. Observe que a regra "main.o" já tem suas dependências resolvidas e já pode ter o seu comando executado. O comando invoca o "gcc" para compilar seu programa. Entretanto, observe que não podemos compilar o programa inteiro pois "main.c" usa funções que não são implementadas dentro do arquivo (Soma_Numeros(int,int)) mas sabe que elas existem em um outro arquivo, através do "operacoes.h" que está incluído em seu código. Por isso,

a compilação deve ser feita parcialmente, produzindo um binário tal que, quando todas as partes forem mescladas, as ligações sejam feitas corretamente. Para isso, utilizamos a flag "-c". O mesmo é feito para a regra "operacoes.o" (a flag "-g" é explicada no Direcionamento 6). Ao final, a regra "operacoes" passa a ter seus pré-requisitos preenchidos e executa o seu comando, que é ligar todos os arquivos ".o" (objetos). Para executar esse código, apenas digite "make". A figura 1 mostra um exemplo do make executado em um terminal Linux:

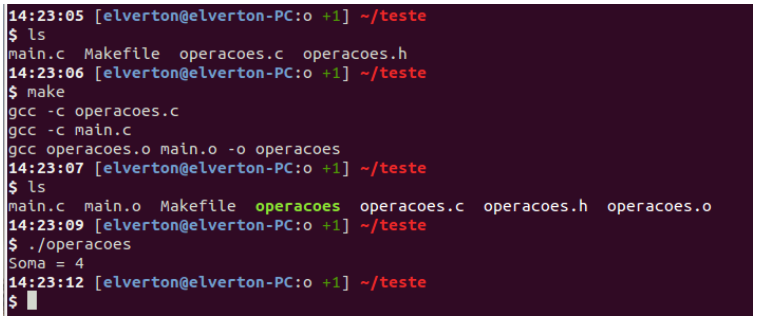


Figura 1: Executando um Makefile no terminal Linux.

Para mais detalhes, utilize o código do tutorial do "Make" no moodle da disciplina.

# 6 Direcionamento 6: Utilizando o Valgrind.

Um modo interessante de verificar se seu programa contém alguns erros comuns envolvendo o uso de memória é utilizar uma ferramenta chamada Valgrind. Essa ferramente é capaz de detectar problemas tais como leitura de variaveis que não foram devidamente inicializadas, falhas de segmentação ou vazamentos de memória — quando você esquece de chamar `free()`. Para depurar o código, é necessário incluir o parâmetro "-g" nas opções passadas para o `gcc`. Isso faz com que ele inclua informações no binário que é gerado que permitem que o valgrind mostre a linha de código em que ocorreu cada erro. Usar o valgrind é bastante fácil, bastando invocá-lo com o comando "valgrind" seguido pela forma como você executaria o trabalho prático normalmente no terminal do Linux. Por exemplo:

```
1 valgrind ./operacoes
```

Ao digitar esse comando em um terminal linux, contendo o binário "operacoes" compilado para a plataforma (Leia o Direcionamento 5), obtemos a saída mostrada pela figura 2.

```
14:36:56 [elverton@elverton-PC:o +1] ~/teste
$ valgrind ./operacoes
==4480== Memcheck, a memory error detector
==4480== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4480== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==4480== Command: ./operacoes
==4480==
Soma = 4
==4480==
==4480== HEAP SUMMARY:
==4480==     in use at exit: 0 bytes in 0 blocks
==4480==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==4480==
==4480== All heap blocks were freed -- no leaks are possible
==4480==
==4480== For counts of detected and suppressed errors, rerun with: -v
==4480== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
14:36:58 [elverton@elverton-PC:o +1] ~/teste
```

Figura 2: Utilizando o Valgrind para avaliar o programa “operacoes”.

Primeiramente, o valgrind checa os possíveis erros de memória durante a execução do código. No caso do programa “operacoes”, nenhum erro foi encontrado e a saída do código aparece conforme mostrado. Ao final, ele mostra um sumário das alocações feitas pelo programa. Essas alocações se referem ao heap, que é uma área da memória criada quando um programa é invocado e é dedicada para alocações dinâmicas. Como não foi utilizada nenhuma alocação dinâmica, temos 0 “allocs” e 0 “frees”.

Entretanto, para ficar mais interessante o uso do valgrind, vamos criar um código chamado “faz_nada.c”, conforme mostrado abaixo:

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
  int *A;
  A = (int*)malloc(sizeof(int));
  if(*A == 1){
    *A = 1;
    printf("%d\n",*A);
  } else{
    *A = 0;
    printf("%d\n",*A);
  }
  return 0;
}
```

Tente encontrar dois problemas neste código ... Ok! O primeiro é usar uma variável não inicializada na comparação. O segundo é não desalocar a memória alocada para a variável A, mais comumente chamado de vazamento de memória. A figura 3 mostra a compilação desse código e qual é a saida do Valgrind para ele.

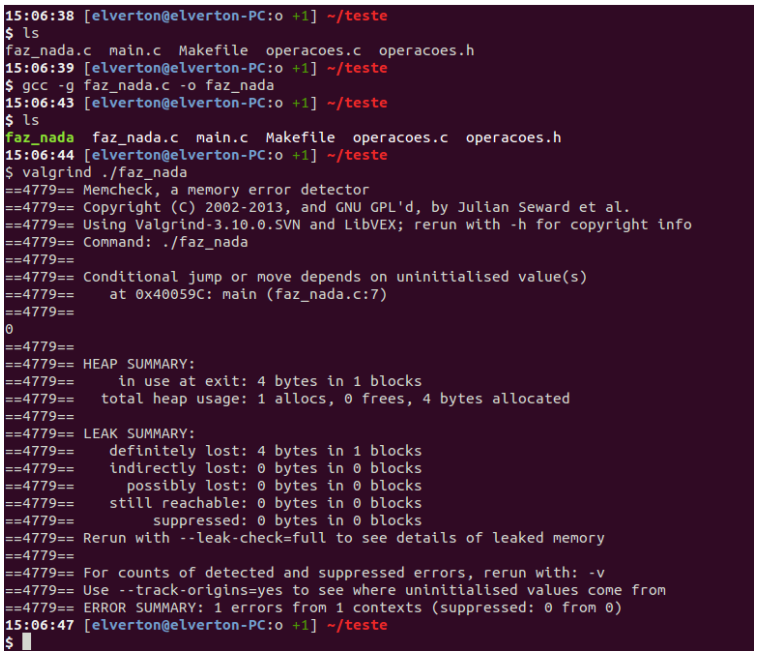


Figura 3: Utilizando o Valgrind para avaliar o programa “faz_nada”.

A saída do Valgrind mostra alguns avisos que ajudam a identificar os dois erros citados. O primeiro aviso é “Conditional jump ...”, que significa que um desvio condicional depende de uma variável não inicializada. Observe que o valgrind informa a linha do erro, que é justamente a linha da comparação. Após o término do programa, o Valgrind emite um sumário do uso do heap. Veja que foram alocados 4 bytes (ou 32 bits que é o tamanho do inteiro A alocado) mas eles não foram desalocados, segundo a linha do “total heap usage”. Para desalocar um endereço de memória alocado em “C”, você pode usar a função free(), conforme mostrado abaixo:

```
1 ...
2 A = (int*)malloc(sizeof(int));
3 ...
4 free(A);
5 ...
```

Lembre-se sempre de usar o Valgrind nos trabalhos práticos pois ele é utilizado pelos monitores para verificar o quão bom está sua alocação dinâmica de memória.