# Projeto prático 1

**Franco Barpp Gomes**[1], **Patricia Abe Turato**[2]

{franco.b.gomes@gmail.com, patriciaturato@alunos.utfpr.edu.br}

## 1. Exercício 1

### 1.1. Decomposição inicial

O problema inicial se trata da impressão de um calendário de um ano como um todo. Como este possui 12 meses, é possível decompor o problema para a impressão de 12 calendários de mês, os quais são compostos de duas partes: cabeçalho e conteúdo (os dias).

### 1.2. Cabeçalho de um calendário mensal

O cabeçalho é a parte do calendário que indica o nome do mês, o ano, além do dia da semana correspondente a cada coluna. O programa desenvolvido por nós, nesse ponto, apresenta o recurso de centralização do cabeçalho para qualquer ano *int* (tipo int da linguagem C).

### 1.3. Conteúdo de um calendário mensal

O conteúdo, referente aos dias do mês, exige uma abordagem mais complexa. É necessário verificar se o ano inserido é bissexto (o que implica em mudanças no número de dias do mês de Fevereiro), obter o número de dias de cada mês e verificar o dia da semana correspondente a cada um dos dias.

#### 1.3.1. Obtenção do número de dias do mês

O problema de determinar o número de dias é relativamente simples, bastando usar a lógica que, até julho, os meses ímpares têm 31 dias, enquanto os pares têm 30 (com exceção de Fevereiro), e o contrário a partir de Agosto. Como dentro do programa Janeiro é o número 0, então a relação par-ímpar é alternada. Isso poupa muitas verificações se comparado a um *switch* com vários *cases*.

#### 1.3.2. Congruência de Zeller

Sobre a obtenção do dia da semana dos dias, necessário à impressão dos calendários mensais, o algoritmo para calcular o dia da semana de uma data específica — a Congruência de Zeller — foi utilizado somente para o primeiro dia do ano. Assim, sabendo o dia da semana inicial, o problema da obtenção dos próximos dentro do mesmo mês está resolvido, já que é uma simples incrementação dia após dia. Para a intercomunicação entre cada um dos meses, contudo, foi necessário retornar, na função de impressão do mês, o primeiro dia do próximo mês.

### 1.4. Interligando as funções

Para chegar ao resultado final, a função da impressão de um mês foi colocada dentro de um laço de repetição, incrementando o mês e atualizando o primeiro dia da semana do próximo pelo retorno do atual. Com isso, todos os meses serão impressos e o calendário anual estará completo.

## 2. Exercício 2

O teste de mesa e as explicações sobre os operadores estão em "./ex2/ex2.c".

### 2.1. Resposta 1:

Considerando o funcionamento do operador $<<$ (left shift) e a lógica do código, o número de linhas do triângulo original (com SIZE $1 << 4$), é 16. Com SIZE $1 << 7$, o triângulo terá 128 linhas e, consequentemente, mais triângulos internos.

### 2.2. Resposta 2:

Podemos destacar 2 outras lógicas para chegar a resultados próximos:

- **Triângulo de Pascal:** O resultado do algoritmo mostrado, um triângulo de Sierpinski, pode ser resolvido montando um triângulo de Pascal, uma expansão infinita dos coeficientes binomiais, e substituindo os números ímpares por "* "e os pares por 2 espaços.
- **Jogo do Caos:** Nesse caso, a representação será aproximada e dependente de um meio gráfico. O algoritmo básico é:
    1. Forme um triângulo equilátero com os vértices A, B e C
    2. Escolha um ponto aleatório dentro do triângulo
    3. Escolha um vértice aleatório
    4. Trace uma reta entre o ponto escolhido e o vértice
    5. Marque o ponto médio da reta
    6. Volte ao passo 2

  Quanto mais iterações, mais a imagem formada se assemelha a um triângulo de Sierpinski.

## 3. Exercício 3

### 3.1. Enunciado

A comunicação entre as duas pontas é baseada na relação entre um caractere e o número binário correspondente na tabela ASCII. De acordo com o enunciado, a forma de Anna e Bob enviarem suas mensagens era: se a contagem de bits 1 do caractere a ser enviado for ímpar, um bit 1 deve ser colocado na posição 7 do byte, para que assim todos os caracteres saiam da ponta inicial com contagem par.

### 3.2. Verificação de bits

Como todo caractere codificado tem número de bits 1 par, se torna simples descobrir se a letra foi corrompida, já que somente um bit pode ser alterado e, no caso, isso tornaria a contagem ímpar.
A base da resolução do problema é achar uma forma de descobrir quantos bits 1 o byte

do caractere contém. Isso foi feito por operadores bit-a-bit. Para descobrir se, em determinada posição $n$, o bit é 1, é possível fazer um AND bit-a-bit do caractere com $1 << n$. Se for retornado um valor diferente de zero, significa que a posição $n$ do byte é 1. Tornando o trabalho mais fácil, podemos usar *right shift* por $n$ casas nesse retorno para que o número se torne 0 ou 1, apenas incrementando-o ao contador.
A partir desse ponto, a solução é direta.

### 3.3. Codificação

De letra a letra, se tiver contagem de bits 1 ímpar o caractere é incrementado de 128 e então impresso, adicionando um 1 ao seu bit mais significativo, caso contrário é impressa diretamente.

### 3.4. Decodificação

Números com contagem de bits 1 par e menores que 128 são impressos diretamente, enquanto os maiores ou iguais a 128 (ímpares antes da codificação) são corrigidos. Aqueles que tiverem contagem ímpar são claramente corrompidos, como explicado anteriormente, e substituídos por "*".

## 4. Exercício 4

### 4.1. Uso de Constantes

Nesse problema, o uso de constantes é muito útil, tanto durante o desenvolvimento quanto durante a leitura. Os tamanhos mínimos e máximos, além do código de cada operadora e as mensagens de erro podem ser definidas ao início do programa. Assim, o código se torna mais literal sem a necessidade de explicações desses dados por meio de comentários ou atribuição a variáveis globais, adicionando também a possibilidade de edições de forma mais simples.

### 4.2. Coleta de dados

O programa deve ler um número de cartão de crédito e processar seus dados, além de conferir se este é válido. A melhor forma por nós encontrada foi a leitura caractere a caractere, o que possibilita a verificação de cada um como uma letra ou não, além da montagem de um número com os algarismos. Esse recurso é interessante, pois, dessa forma, é possível verificar a existência de caracteres não-numéricos até mesmo entre números.
Obter um algarismo a partir de seu caractere correspondente também não é uma tarefa difícil, já que a posição relativa a '0' de '$n$' (com n de 0 a 9) na tabela ASCII é o número em si (por exemplo, '1' - '0' = 1).
De algarismo a algarismo, podemos montar um *unsigned long long int* para armazenar esses dados.

### 4.3. Erro - Caracteres inválidos

Durante a leitura do *input* pelo método mencionado, se o dígito não estiver entre '0' e '9', então é não-numérico. Logo, o erro será acionado.

### 4.4. Verificação de operadora

Uma boa estratégia foi a divisão inicial do *unsigned long long int* do cartão para obter os 4 primeiros dígitos, tamanho máximo de prefixo a ser verificado para descobrir a operadora. Assim, com um número menor, o uso de operações grandes e custosas, como $cartao/10^{15}$, é diminuído.
Em sequência, foi possível dividir as operadoras entre cada tamanho de cartão para então testar o prefixo, retornando o código da operadora definido no início do código.

### 4.5. Algoritmo de Luhn

Usando a operação *modulus* dentro de um laço de repetição, cada dígito é separado e processado da esquerda para a direita, somando para determinar a validade. Esse algoritmo foi implementado em uma função, que retorna 0 para inválido e 1 para válido, simplificando seu uso.

## 5. Fontes

- <https://en.wikipedia.org/wiki/Zeller%27s_congruence>
- <https://en.wikipedia.org/wiki/Sierpi%C5%84ski_triangle>
- <https://en.wikipedia.org/wiki/Chaos_game>