UNIVERSIDADE ESTADUAL PAULISTA

FACULDADE DE ENGENHARIA DE ILHA SOLTEIRA
DEPARTAMENTO DE ENGENHARIA MECÂNICA
CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA

Tese de Defesa de Mestrado

NÚMERO DE PONTO FLUTUANTE BINÁRIO COM PRECISÃO ESTENDIDA EM CLASSE DE C++

Defesa de tese apresentado à Faculdade de Engenharia de Ilha Solteira – UNESP, como parte dos requisitos para obtenção do título de Mestre em Engenharia Mecânica.

Aluno: **Richardson Leandro Nunes**
Orientador: **João Batista Aparecido**
Data da Realização: **31 de março de 2008**
Horário: **8:30**
LOCAL: **Sala de Reuniões – M3**

Sumário

Índice de Figuras e Gráficos

Resumo

A execução de cálculos computacionais é limitada pela precisão que as linguagens de programação podem fornecer. Os compiladores convencionais possuem formatos de números incapazes de realizar cálculos que exigem grande precisão, porém, possuem ferramentas que possibilitam a criação de formatos extras.

Utilizando o conceito de classe, é possível criar objetos computacionais e implementar métodos. Visando solucionar problemas de precisão implementou-se uma classe onde o objeto é um número de ponto flutuante, o Sfloat. A classe implementada em C++ é composta de um arranjo de variáveis booleanas de tamanho arbitrário para representar os bits de um número de ponto flutuante e métodos de classe para representar operadores aritméticos e lógicos.

Os operadores binários aritméticos estão sobrecarregados, ou seja, os quatro operadores já existentes (“+”, “-“, “*” e “/”) podem utilizar números Sfloat como argumentos.

Os operadores binários lógicos relacionais (<, >, <=, >=, ==, !=) seguem o mesmo modelo dos binários aritméticos, sendo sobrecarregados para utilizar Sfloat como argumento.

Para somar, soma-se dígito a dígito os dois somandos. A subtração, na verdade, é a soma de um número positivo com um negativo, de um modo que pode ser executada da mesma maneira que o operador soma fazendo jogo de sinais. Na multiplicação, soma-se as multiplicações parciais de cada dígito de um dos fatores pelo outro fator, ou seja, a multiplicação é executada como um somatório

de multiplicações. A divisão forma os dígitos do quociente verificando sempre qual o maior número inteiro que pode multiplicar o divisor sem ultrapassar o valor do dividendo.

Sfloat foi utilizado para cálculos simples de soma, subtração, multiplicação e divisão. Também foi calculado o valor de uma série infinita truncada, apresentando resultados mais precisos.

1. Preâmbulo

O padrão IEEE 754 contém normas a serem seguidas pelos fabricantes de computadores e construtores de *software*s ao desenvolver números de ponto flutuante e no tratamento da aritmética binária. O padrão aborda questões relativas ao armazenamento, métodos de arredondamento, ocorrência de *underflow* e *overflow* e realização das operações aritméticas básicas. [1]

Existe também o padrão IEEE 854, que tem os mesmos objetivos, porém, relativos a uma padronização para uma base independente genérica, para aqueles que utilizam outra base diferente da binária. [2]

A vantagem deste procedimento é permitir uma maior portabilidade dos *software*s numéricos e criação de novos tipos de dados a fim de tratar problemas de precisão.

1.1 – Números de ponto flutuante

De acordo com [1], os números de ponto flutuante possuem o seguinte formato:

$N = \pm d_0.d_1d_2d_3\ldots .b^{\pm e}$

Onde *N* é o número, d_0, d_1, ... são os dígitos da mantissa, *b* é a base e *e* é o expoente da base. De maneira que um número de ponto flutuante tem três partes fundamentais: a mantissa, que é a seqüência de dígitos, a base e o expoente.

Veja alguns casos a seguir, no exemplo 1.1:

Exemplo 1.1:

N1 = -2.591 x 103 (que é o número –2591 na forma convencional), aqui d0 = 2, d1 = 5, d2 = 9, d3 = 1, b = 10, e = 3;
N2 = 1.0101 x 24 (10101 no sistema binário, e 21 no sistema decimal), neste caso d0 = 1, d1 = 0, d2 = 1, d3 = 0, d4 = 1, b = 2, e = 4;
N3 = 5.C6D x 163 (5C6 no sistema Hexadecimal), de maneira que d0 = 5, d1 = C (equivale a 13 no sistema decimal), d2 = 6, d3 = D, b = 16, e = 3;

As bases mais comuns são *10*, característica do sistema decimal, e *2*, utilizada principalmente na linguagem de baixo nível dos microcomputadores, mas também pode ser *8*, *16*, ou qualquer base que se queira utilizar.

Observa-se que o número de símbolos distintos necessários é igual ao valor da base que se utiliza. Por exemplo, a base decimal possui dez símbolos diferentes, de *0* a *9*. Estes símbolos serão sempre referenciados como dígitos.

O menor dígito sempre é o *0* (zero), de forma que o maior dígito é uma unidade menor do que o valor da base.

O exemplo a seguir demonstra o que ocorre quando um dígito é o maior possível e há a necessidade de se somar uma unidade a ele.

Exemplo 1.2:

Aqui utiliza-se os números N_1 e N_2 do exemplo 1.1. No número N_1 acrescenta-se uma unidade ao dígito $d_n = d_2 = 9$, e no número N_2 acrescenta-se uma unidade ao dígito d_1.

No caso do número N_1, faz-se o dígito $d_2 = 0$ e acrescenta-se uma unidade ao dígito d_{n-1}, que é o dígito $d_1 = 5$. Assim N_1 assume a forma $N_1 = -2.601 \times 10^3$.

No número N_2, d_n é o primeiro dígito da seqüência ($n = 0$), de modo que não há como acrescentar uma unidade ao dígito d_{n-1}, então anula-se d_n e este ocupará a posição do dígito d_{n+1}. O resto dos dígitos da mantissa deslocam-se uma unidade e no lugar de d_0 coloca-se o dígito 1 que passa a ser o novo primeiro dígito da mantissa. Finalmente, acrescenta-se uma unidade ao expoente e ($4 + 1 = 5$). De maneira que N_2 fica na forma 1.00101×2^5, pois agora $d_0 = 1$, $d_1 = 0$, $d_2 = 0$, $d_3 = 1$, $d_4 = 0$, $d_5 = 1$, $b = 2$, $e = 5$;

O número de dígitos d_n da mantissa fixa a precisão. A linguagem C++ disponibiliza como números do tipo ponto flutuante o tipo *double*, com 8 *bytes*, e o tipo *float*, de 4 *bytes*. O tipo *float* possui 23 dígitos binários na mantissa, que equivalem a 7 dígitos decimais, e o tipo *double* possui 52 dígitos binários na mantissa, o mesmo que 15 dígitos decimais. Os tipos *float* e *double* são práticos e eficientes em problemas onde a precisão não seja um fator crítico, entretanto falham quando há a necessidade de maior precisão.

1.2 – Número de ponto flutuante binário normalizado

Seja $F(b, t, m, M)$ um sistema de ponto flutuante que representa um subconjunto dos números reais, onde $b \geq 2$ é a base de representação, $t \geq 1$ é a precisão, e m e M são respectivamente o menor e maior expoente. O subconjunto $F \subset R$ contém números reais, de ponto flutuante, da forma $X = (-1)^s.b^e.(0.d_1d_2d_3...d_t)$, onde s é sinal (0 se o sinal for positivo e 1 se o sinal for negativo), e o expoente ($m \leq e \leq M$, com $m < 0$, $M > 0$ e $|m| \approx M$ e d_i são dígitos da mantissa na base b ($0 \leq d_i \leq b-1$, $\forall i$, $1 \leq i \leq t$).

No sistema binário a base é $b=2$. De modo que cada dígito pode se referir a apenas dois algarismos diferentes, sendo um deles o 0 (zero), e o outro o 1(um).

A representação de um número de ponto flutuante normalizado é a mostrada a seguir. Neste caso está representado um número de ponto flutuante binário de 4 *bytes*, ou seja 32 *bits*.

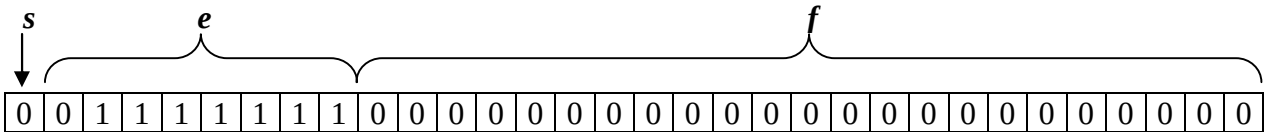


Figura 1.1 – Número de ponto flutuante de 4 *bytes*, armazenado na forma binária.

Os dígitos do número de ponto flutuante são representados em uma única seqüência.

O sinal, o expoente e a mantissa são os três componentes mais importantes do número de ponto flutuante normalizado, cada um tem sua posição determinada na seqüência.

O primeiro bit armazena o sinal *s*. Se for igual a *0*, o sinal é positivo. Se for igual a *1*, o sinal é negativo.

Os próximos *bits* da seqüência armazenam o valor do expoente *e*. Neste caso, há oito *bits*, que é a quantidade mais comum para um *float* de quatro *bytes*.

Os últimos vinte e três *bits* armazenam a mantissa *f*.

Na representação IEEE 754 [1], o ponto decimal, implicitamente, está entre os dígitos d_1 e d_2, o que permite o armazenamento de mais um dígito na mantissa.

O primeiro dígito é diferente de zero par assegurar a unicidade de representação, e manter sempre a precisão máxima suportada pala mantissa. Esta forma é a chamada forma normalizada.

O valor zero não pode ser normalizado e tem representação especial, com mantissa nula (todos *bits* iguais a zero) e expoente o menor possível (m-1).

Exemplo 1.3:

Considerando *F(2, 8, -4, 3)* e sendo X e Y, dois números da forma:

X=	0	0	1	0	1	1	1	0	0	1	1	0
Y=	0	0	1	0	1	1	1	0	0	1	1	1

$X = (-1)^0.2^2.(0.11100110) = (11.100110)_2 = (3.59375)_{10}$

$Y = (-1)^0.2^2.(0.11100111) = (11.100111)_2 = (3.609375)_{10}$

Observe que *X* e *Y* são dois números consecutivos neste sistema de representação, sendo que o número decimal 3.6 não teria representação exata.

1.3 – Relações

Para todo sistema da forma $F(b, t, m, M)$ são válidas as seguintes relações:

i) Variação da mantissa: $1/b \leq f < 1$

ii) Menor número positivo: $\lambda = b^{m-1}$

iii) Maior número positivo: $\Lambda = (1-b^{-1}).b^{M}$

iv) Número de elementos: $|F| = 2.(b-1).b^{t-1}.(M-m+1)+1$

v) Épsilon da máquina: $\varepsilon = (0.5).b^{t-1}$

ε (épsilon) é um valor tal que $(1+\varepsilon) = 1$; e normalmente é chamado de "zero da máquina", apesar de ser um valor positivo, porém, muito pequeno. Este valor pode ser obtido através do algoritmo:

```
EPS = 1
Repita
      EPS = EPS/2
EPS1 = EPS + 1
Até EPS1 = 1
```

Figura 1.2 – método de cálculo do ε (épsilon) da máquina

1.4 – *underflow* e *overflow*

Seja X um número real e X' uma aproximação de X, então, sabendo que λ é o menor número positivo do sistema de representação e Λ o maior número positivo, tem-se:

i) Se $|X| > \Lambda$; um caso de *overflow*, que deve ser tratado como uma exceção. Mais adiante será abordado o uso de exceções nas operações aritméticas;

ii) Se $0 < |X| < \lambda$, um caso de *underflow* que é considerado como $X' = 0$. Esta prática pode invalidar resultados, caso em que é chamado de *underflow* destrutivo. Pode causar por exemplo, uma divisão por zero, que é um caso grave.

iii) Se $\lambda < |X| < \Lambda$, com X necessitando de d dígitos de mantissa, causa um arredondamento para X' contendo t dígitos, sendo descartados (d – t) dígitos usando um dos métodos de arredondamento descritos a seguir.

1.5 – Métodos de arredondamento

Os métodos de arredondamento são utilizados para padronizar a forma de truncamento dos dígitos, em função do tamanho fixo da mantissa, por ocasião de seu armazenamento.

Os quatro tipos de arredondamento são descritos a seguir.

i) Em direção ao zero: arredonda-se seguindo em direção ao zero até o número mais próximo.

ii) Em direção a +∞: arredonda-se para o maior valor relativo;

iii) Em direção a -∞: arredonda-se para o menor valor relativo;

iv) Para o mais próximo, com opção de aproximação para o próximo número para (sistema decimal) em caso de empate. Este método é o recomendado por IEEE 754.

No exemplo a seguir, há um exemplo para cada método descrito acima, onde foi considerado que $d=5$ e $t = 3$:

Exemplo 1.3:

Na tabela a seguir, há quatro casos de números reais decimais e três casos de números binários que serão arredondados e truncados para terem precisão de três casas.

Tabela 1.1 – Arredondamento e truncamento de números decimais e números binários.

Número	Método (i)	Método(ii)	Método(iii)	Método(iv)
+0.34521	+0.345	+0.346	+0.345	+0.345
-0.34521	-0.345	-0.345	-0.346	-0.345
-0.3455	-0.345	-0.345	-0.346	-0.346
+0.3445	+0.344	+0.345	+0.344	+0.344
+1.101	+1.10	+1.11	+1.10	+1.11
+1.110	+1.11	+1.11	+1.11	+1.11
-1.011	-1.01	-1.01	-1.10	-1.10

1.6 – Padrão IEEE 754 para ponto flutuante.

A base 2 é utilizada como padrão.

As operações devem ser executadas com uso de dígitos de guarda e expoente deslocado, também chamado característica, tem por objetivo eliminar o sinal do expoente, por exemplo, se $m = -127$ e $M = 127$, (δ deve ser igual a 127. Desta forma a variação de expoente seria de 0 a 254.

Considerando o formato padrão para números de ponto flutuante da Figura 1, o padrão IEEE 754 [1] recomenda os seguintes números de *bits*, de acordo com a precisão usada, a expressão $(N = s + e + t)$, corresponde ao tamanho da palavra em *bits*:

i) Precisão simples: s = 1, e = 8, f = 23 (mais um escondido), N = 32 *bits*.

$X = (1.f * 2^{e-\delta})$, $\delta = 127$

ii) Precisão simples estendida: s = 1, e ≥ 11, f ≥ 32, n ≥ 43 *bits*.

$X = (0.f * 2^{e-\delta})$, $\delta = 127$

iii) Precisão dupla: s = 1, e = 11, f = 52 (mais um escondido), N = 64 *bits*

$X = (1.f * 2^{e-\delta})$, $\delta = 1023$

iv) Precisão dupla estendida: s = 1, e ≥ 15, f ≥ 64, n ≥ 79 *bits*

$X = (0.f * 2^{e-\delta})$, $\delta = 1023$

Existem três formatos comuns de números de ponto flutuante:

Precisão simples de *32 bis*;

Precisão dupla de *64 bits;*

Precisão estendida de *80 bits*.

1.7 – Configuração especial de *bits* – padrão IEEE 754

Alguns resultados de operações aritméticas não são suportados pela máquina, em virtude de divisão por zero, *overflow*, etc. Desta forma, esses resultados são representados de forma especial. Nos casos seguintes, o item (i) corresponde a uma representação especial para o zero, o item (ii) é um número com menor expoente possível, e os itens (iii) e (iv) são exceções tratáveis.

Tabela 1.2 – Configurações especiais de *bits*.

Item	Expoente	Fração	Significado
(i)	E = m – 1	F = 0	± 0
(ii)	E = m – 1	F ≠ 0	$\pm 0.f * 2^m$
(iii)	E = m + 1	F = 0	± ∞
(iv)	E = m + 1	F ≠ 0	NaN (Not a Number)

1.8 – Operações aritméticas de adição e subtração

Dados dois números X e Y representados na forma: $(-1)^s.b^e.(0.d_1d_2d_3...d_t)$ são definidas as seguintes regras para as operações de adição e subtração:

i) Escolher o número com menor expoente entre X e Y e deslocar sua mantissa para a direita um número de dígitos igual à diferença absoluta entre os respectivos expoentes;

ii) Colocar o expoente do resultado igual ao maior expoente entre os expoentes de X e de Y;

iii) Executar a adição ou subtração das mantissas e determinar o sinal do resultado;

iv) Normalizar o valor do resultado, se necessário;

v) Arredondar o valor do resultado, se necessário e

vi) Verificar se houve *overflow* ou *underflow*.

Exemplo 1.4:

Seja $F(10, 4, -50, 49)$, $\delta = 50$, com um dígito de guarda.

Se $X = 436.7$ e $Y = 7.595$, calcular a soma $(X + Y)$.

X=	0	5	3	4	3	6	7
Y=	0	5	1	7	5	9	5

i) $e_1 - e_2 = 53 - 51 = 2$ (deslocamento de dois dígitos do menor, no caso Y)

Y=	0	5	3	0	0	7	5	9

ii) Expoente do resultado: $e = 53$

iii) Adição das mantissas: $4\ 3\ 6\ 7\ (0) + 0\ 0\ 7\ 5\ (9) = 4\ 4\ 4\ 2\ (9)$, onde o valor entre parênteses é o digito de guarda.

iv) Normalização do resultado: não há necessidade, pois $d_1 = 4 \neq 0$.

v) Arredondamento do resultado: $4\ 4\ 4\ 2\ (9) \Rightarrow f = 4\ 4\ 4\ 3$

vi) Verificação de underflow e overflow: $e - 50 = 53 - 50 = 3 \leq 49$: Não há

Resultado:

X+Y=	0	5	3	4	4	4	3

Ou $X + Y = 0.4443 * 10^{53-50} = 444.3$

1.9 – Multiplicação

Dados dois números *X* e *Y* representados na forma: $(-1)^s.b^e.(0.d_1d_2d_3...d_t)$ são definidas as seguintes regras para a operação de multiplicação:

i) Colocar o expoente de resultado igual à soma dos expoentes de *X* e *Y*;

ii) Executar a multiplicação das mantissas e determinar o sinal do resultado;

iii) Normalizar o valor do resultado, se necessário;

iv) Arredondar o valor do resultado, se necessário e

v) Verificar se houve *overflow* ou *underflow*.

Exemplo 1.5:

Seja *F(10, 4, -50, 49)*, $\delta = 50$, com um dígito de guarda.

Se X = 436.7 e Y = 7.595, obter o produto (X * Y).

X=	0	5 3	4 3 6 7			
Y=	0	5 1	7 5 9 5			

i) Expoente: e = 53 + 51 = 104

ii) Multiplicação das mantissas: *(4 3 6 7) * (7 5 9 5) = (3 3 1 6 7 3 6 5)*, considerando quatro dígitos de guarda.

iii) Normalização do resultado: Ajustar expoente (e – δ) = 104 –50 = 54.

iv) Arredondando o valor do resultado: *(3 3 4 6 7 3 6 5)* $\Rightarrow$ *(3 3 1 7)*.

v) Verificando overflow e underflow: e – 50 = 54 –50 = 4 < 49. Não há.

Resultado:

X*Y=	0	5 4	3 3 1 7			

Ou X * Y = $0.3317 * 10^{54-50} = 3317$

1.10 – Divisão

Dados dois números *X* e *Y* representados na forma: $(-1)^s.b^e.(0.d_1d_2d_3...d_t)$ são definidas as seguintes regras para a operação de divisão:

i) Colocar o expoente de resultado igual à diferença dos expoentes de *X*, que é o dividendo, e *Y*, que é o divisor;

ii) Executar a divisão das mantissas e determinar o sinal do resultado;

iii) Normalizar o valor do resultado, se necessário;

iv) Arredondar o valor do resultado, se necessário e

v) Verificar se houve *overflow* ou *underflow*.

Exemplo 1.6:

Seja $F(10, 4, -50, 49)$, $\delta = 50$, com um dígito de guarda.

Se X = 436.7 e Y = 7.595, obter a divisão(X / Y).

X=	0	5	3	4	3	6	7
Y=	0	5	1	7	5	9	5

i) Expoente: e = 53 - 51 = 2

ii) Divisão das mantissas: *(4 3 6 7) / (7 5 9 5) = (4 7 6 9 8 3 5 4)*, considerando quatro dígitos de guarda.

iii) Normalização do resultado. Ajustar expoente (e + δ) = 2 + 50 = 52.

iv) Arredondando o valor do resultado: *(5 7 4 9 8 3 5 4)* $\Rightarrow$ *(5 7 5 0)*.

v) Verificando overflow e underflow: e –50 = 52 –50 = 2 < 49. Não há.

Resultado:

X/Y=	0	5	2	5	7	5	0

Ou X / Y = $0.5750 * 10^{52-50}$ = 57.50

2 – Tipos derivados de dados

C++ é uma linguagem de programação que permite a utilização de tipos derivados de dados. Os tipos derivados de dados são conjuntos formados pelos tipos básicos. Os tipos de dados básicos em C++ são, por exemplo, os inteiros, os reais, os caracteres, os lógicos e os complexos.

Por exemplo, pode-se utilizar um tipo derivado de dados para representar os números complexos sem incluir o próprio tipo complexo, utilizando apenas variáveis reais. Duas variáveis reais seriam suficientes, sendo que uma delas representaria a parte real e a outra a parte imaginária dos números complexos.

Em C++, há basicamente três tipos derivados de dados: uma estrutura (*struct*), uma união (*union*) ou uma classe (*class*).

Os tipos derivados de dados serão abordados com exemplos mais adiante.

2.1 – Estruturas

Ao armazenar informações relacionadas de diferentes tipos pode-se utilizar uma estrutura. Uma estrutura é um conjunto de informações relacionadas, chamadas membros, cujos tipos podem diferir. Agrupando dados em uma variável dessa forma, os programas se tornam mais simples reduzindo o número de variáveis que precisa gerenciar, passar para as funções e assim por diante. Este tópico examina como os programas criam e usam estruturas e os seguintes conceitos fundamentais:

- Uma estrutura consiste de um ou mais dados, chamados membros;

- Para definir uma estrutura dentro do programa, é preciso especificar o nome da estrutura e seus membros;
- Cada membro da estrutura tem um tipo, como *char* , *int* e *float*, que em C++ são variáveis de tipo caractere, inteiro e número de ponto flutuante, respectivamente;
- Cada membro precisa ter um nome exclusivo na estrutura;
- Após definir uma estrutura, o programa poderá declarar variáveis de tipo estrutura;
- Para alterar membros da estrutura dentro de uma função, os programas precisarão passar a estrutura para a função por endereço.

A compreensão das estruturas facilitará o entendimento de classes mais adiante.

Declarando uma estrutura

Uma estrutura define um gabarito que pode ser usada mais tarde através de um programa para declarar uma ou mais variáveis. Em outras palavras, em um programa pode ser definida uma estrutura e depois declaradas variáveis do tipo estrutura. Para definir uma estrutura, utiliza-se a palavra-chave *struct*, normalmente seguida por um nome e o símbolo de chave *{*. Após a chave, especifica-se o tipo e o nome de um ou mais *membros*. O último membro deve ser seguido pelo símbolo de chave fechada *}*. Neste ponto, opcionalmente, é possível declarar variáveis do tipo estrutura, como mostrado na figura a seguir.

```
Struct nome
{
      int nome_membro_1;          ⇐ Declaração de membro
      float nome_membro_2;        ⇐ Declaração de membro
} variável;                       ⇐ Declaração de variável
```

Figura 2.1 – Modelo de declaração de uma *struct*

A definição a seguir, por exemplo, cria uma estrutura chamada *funcionário* que contém informações sobre um funcionário:

```
Struct funcionário
{
      char nome[64];
      long ident_func;
      float salario;
      char fone[10];
      int ramal;
};
```

Figura 2.2 – Exemplo de declaração de *struct*

Na exemplo da figura acima, a definição da estrutura não declara qualquer variável do tipo estrutura. Após definir uma estrutura, pode-se declarar variáveis do tipo estrutura usando o nome desta (também chamado de *tag* de estrutura), como mostrado a seguir:

```
funcionário chefe, func, novo_func;
```

Figura 2.3 – Declarando uma variável do tipo estrutura.

Neste exemplo, o comando cria três variáveis da estrutura *funcionário*. Examinando programas em C++, pode-se encontrar declarações que precedem a *tag* de estrutura com a palavra-chave *struct*, como mostrado a seguir:

```
struct funcionário chefe, func, novo_func;
```

Figura 2.4 – Variável do tipo estrutura.

A programação em C requer a palavra-chave *struct*, de modo que alguns programadores podem incluí-la por hábito. Em C++, no entanto, o uso da palavra-chave *struct* é opcional.

Usando membros de estruturas

A estrutura permite que o programa agrupe em uma variável informações chamadas membros. Para atribuir ou acessar o valor de um membro, use o *operador ponto* (.) de C++. Os comandos a seguir, por exemplo, atribuem valores a diferentes membros de uma variável chamada *func* do tipo *funcionario*.

```
func.ident_func = 12345;
func.salario = 25000.00;
func.ramal = 102;
```

Figura 2.5 – Atribuindo valores a membros de estruturas.

Para acessar um membro da estrutura, especifique o nome da variável, seguido por um operador ponto e o nome do membro (*nome_estrutura.membro*). Os comandos a seguir, por exemplo, atribuem valores armazenados em diferentes membros de estrutura para as variáveis de um programa.

```
identificacao = func.ident_func;
ealario = func.salario;
escritorio = func.ramal;
```

Figura 2.6 – Copiando valores de variáveis membro de estruturas para variáveis do programa.

O programa *Function.CPP*, a seguir, ilustra o uso de uma estrutura do tipo *funcionário*.

```
#include <iostream.h>
#include <string.h>

void main (void)
{
       struct funcionario
       {
              char nome[64];
              long ident_func;
              float salario;
              char fone[10];
              int ramal;
       } func;

// Copia um nome para a string
       strcpy(func.nome, "João Silva");

func.ident_func = 12345;
func_salario = 25000.00;
func_ramal = 102;

// Copia um número de telefone para a string
strcpy(func.fone, "555-1212")

cout << "Funcionário: " << func.nome << endl;
cout << "Fone: " << func.fone << endl;
cout << "Ident. funcional: " << func.ident_func << endl;
cout << "Salario: " << func.salario << endl;
cout << "Ramal: " << func.ramal << endl;
}
```

Figura 2.7 – Exemplo de programa usando estrutura.

Como pode ser visto acima, o programa pode atribuir valores aos membros inteiros e de ponto flutuante da estrutura de uma forma direta, simplesmente usando o operador de atribuição. No uso da função *strcpy* para copiar arranjos de caracteres para os membros nome e fone, a não ser que sejam inicializados os membros da estrutura explicitamente, ao declarar a variável estrutura, será preciso copiar strings de caracteres para os membros desta. Posteriormente, neste tópico, serão inicializados os membros ao declarar uma variável.

Estruturas e funções

Se uma função não altera uma estrutura, a estrutura pode ser passada para a função por nome. Por exemplo, o programa *Exib_Fun.CPP*, a seguir, usa a função *exibe_funcionário* para exibir os membros de uma estrutura do tipo *funcionário*.

```
#include <iostream.h>
#include <string.h>

struct funcionario
{
       char nome[64];
       long ident_func;
       float salario;
       char fone[10];
       int ramal;
};

void exibe_funcionario(funcionario func)
{
       cout << "Funcionario: " << func.nome << endl;
       cout << "Fone: " << func.fone << endl;
       cout << "ident. Functional: " << func.ident_func << endl;
       cout "Salario: " << func.salario << endl;
       cout "Ramal: " << func.ramal << endl;
}

void main(void)
{
       funcionario func;

       // Copia um nome para a string
       strcpy(func.nome, "João Silva");

       func.ident_func = 12345;
       func.salario = 25000;
       func.ramal = 102;

       // Copia um número de telefone na string
       strcpy(func.fone, "555-1212");
exibe funcionário(func);
```

Figura 2.8 – Utilizando funções em estruturas.

Como pode ser visto, o programa passa a variável-estrutura *func*, por nome, para a função *exibe_funcionario*. Depois, dentro da função, *exibe_funcionário* exibe os membros da estrutura. Observe, no entanto, que o programa agora define a estrutura *funcionário* fora de *main* e antes da função *exibe_funcionário*. Como a

função declara a variável *func* como tipo *funcionário*, a definição da estrutura *funcionário* precisa preceder a função.

Funções que modificam os membros da estrutura

Quando uma função modifica um parâmetro, é preciso passa-lo para a função por endereço. Para que uma função altere o valor do membro de uma estrutura, é preciso passá-la para a função por endereço. Para passar uma variável-estrutura por endereço, basta preceder o nome da variável com o operador de endereço de C++ (&), como mostrado a seguir

```
Alguma_funcao(&func);
```

Figura 2.9 – Passando uma variável estrutura por endereço.

Dentro de uma função que modifica um ou mais membros, é preciso trabalhar com um ponteiro. Ao usar um ponteiro para uma estrutura, o modo mais fácil de referir-se a um membro de estrutura é empregar a seguinte sintaxe:

```
Variavel_ponteiro->membro = algum_valor;
```

Figura 2.10 – Usando ponteiro em estrutura.

Por exemplo, o programa *Muda_Msg.CPP*, a seguir, passa uma estrutura do tipo *funcionário* para a função *pede_ident_func*, que solicita ao usuário que informe o número de identificação de um funcionário e depois o atribui ao membro da estrutura *ident_func*. Para modificar o membro, a função trabalha com um ponteiro para a estrutura:

```
#include <iostream.h>
#include <string.h>

struct funcionario
{
	char nome[64];
	long ident_func;
	float salario;
	char fone[10];
	int ramal;
};

void pede_ident_func(funcionario *func)
{
	cout << "Informe a identificação functional: ";
	cin >> func->ident_func;
}

void main(void)
{
	funcionario func;

	// Copia um nome para a string
	strcpy(func.nome, "João Silva");
	pede_ident_func(&func);

	cout << "Funcionário: " << func.nome << endl;
	cout << "Ident: " << func.ident_fund << endl;
}
```

Figura 2.11 – Passando uma estrutura para uma função.

Como pode ser visto, dentro de *main* a variável-estrutura *func* é passada para a função *pede_ident_func* por endereço. Dentro da função, *pede_ident_func* atribui o valor que o usuário digita para o membro *ident_func* usando o seguinte comando:

```
	cin >> func->ident_func;
```

Figura 2.12 – Atribuindo um valor que o usuário digitou.

Inicializando os membros da estrutura

Ao declarar variáveis-estrutura, C++ permite inicializar os membros da variável. Por exemplo, a declaração a seguir cria e inicializa uma variável-estrutura do tipo *livro*:

```
Struct livro
{
       char *titulo;
       float preco;
       char *autor;
} livro_informatica = { “Segredos de C++”, 19.90, “Luis Alberto” }
```

Figura 2.13 – Declarando e inicializando uma variável-estrutura.

O comando inclui os valores iniciais da variável dentro de chaves.

2.2 – Classes de C++

Em C++, uma classe é muito similar a uma estrutura no sentido de agrupar os membros, que correspondem aos dados e as funções que operam nos dados. Um objeto é uma entidade, como um telefone ou um arquivo ou um livro, por exemplo. Uma classe de C++ permite definir os atributos (características) do objeto. No caso de um objeto *telefone*, a classe pode conter os membros de dados, como o número do telefone e o tipo (tom ou pulso), e as funções que operam no telefone, como *discar, atender e desligar*. Agrupando os dados e o código de um objeto em uma variável, a programação simplifica e amplia a reutilização do código.

Neste tópico serão explanados os seguintes conceitos fundamentais:

- Para definir uma classe, o programa precisará especificar o nome, os membros de dados e as funções (métodos) da classe;

- A definição de uma classe fornece um gabarito que os programas usam para criar objetos desse tipo de classe, de forma muito parecida com a criação de variáveis a partir dos tipos *int, char, etc*;
- Para atribuir valores aos membros de dados da classe utiliza-se o operador ponto;
- As funções membro (métodos) da classe também são chamadas através do uso o operador ponto.

Compreendendo objetos e a programação orientada aos objetos

Ao criar um programa, normalmente são usadas variáveis para armazenar informações sobre coisas diferentes do mundo real, como funcionários, livros ou até arquivos. Na programação orientada a objetos, o enfoque é nas informações que compõem um sistema e nas operações realizadas com estas informações. Dado um exemplo de um objeto *arquivo*, as operações poderiam imprimir, exibir, excluir ou modificar o arquivo. Em C++, usa-se uma classe para definir seus objetos, incluindo o maior número de informações que puder sobre o objeto. É possível, portanto, utilizar uma classe em muitos programas deferentes.

Uma classe de C++ permite que o programa agrupe dados e funções que realizam operações nos dados. A maioria dos livros e artigos sobre programação orientada a objetos refere-se às funções da classe como *métodos*. Como uma estrutura, uma classe de C++ precisa ter um nome único, seguido pelo símbolo *{*, um ou mais membros e o símbolo *}*, como mostrado a seguir:

```
Class nome_classe
{
      int dado_membro;                  // Dado membro
      void exibe_membro(int);           // Função membro
```

Figura 2.14 – Declarando uma classe.

Após definir uma classe, podem ser declaradas variáveis do tipo da classe (chamadas *objetos*) como mostrado a seguir:

```
nome_classe objeto_um, objeto_dois, objeto_tres;
```

Figura 2.15 – Declarando objetos.

A definição a seguir cria uma classe *funcionário* que contém variáveis de dados e definições de métodos:

```
class funcionario
{
	public:
		char nome[64];
		long ident_func;
		float salario;
		void exibe_func(void)
		{
			cout << "Nome: " << nome << endl;
			cout << "Ident: " << ident_func << endl;
			cout << "Salário: " <<salario << endl;
		};
}
```

Figura 2.16 – Declarando uma classe com variáveis e método.

Neste exemplo, a classe funcionário contém três variáveis-membro e uma função-membro. Dentro da definição da classe é utilizado o rótulo public. Mais a frente será visto que os membros da classe podem ser *private* ou *public*, o que controla como seus programas podem acessar os membros. No exemplo dado todos os membros são do tipo *public*, o eu significa que o programa pode acessar qualquer membro usando o operador ponto. Após definir a classe dentro do programa, pode-se declarar objetos (variáveis) do tipo da classe, como mostrado a seguir:

```
Funcionario func, chefe, secretaria;
```

Figura 2.17 – Declarando objetos do tipo da classe funcionário.

O programa EmpClass.CPP, a seguir, cria dois objetos *funcionário*. Usando o operador ponto, o programa atribui valores aos membros de dados. O programa usa então o membro exibe_func para exibir as informações do funcionário:

```
#include <iostream.h>
#include <string.h>

class funcionario
{
	public:
		char nome[64];
		long ident_func;
		float salario;
		void exive_func(void)
		{
			cout << "Nome: " << nome << endl;
			cout << "Ident: " << ident_func << endl;
			cout << "Salario: " << salario << endl;
		};
};

void main(void)
{
	funcionario func, chefe;

	strcpy(func.nome, "João Silva");
	func.ident_func = 12345;
	func.salario = 25000.00;

	strcpy(chefe.nome, "Joaquim Matos");
	chefe.ident_func = 101;
	chefe.salário = 101101.00;

func.exibe_func();
	chefe.exibe_func();
}
```

Figura 2.18 – Métodos operando sobre membros de uma classe

O programa anterior declara dois objetos *funcionário*, *func e chefe* e depois usa o operador ponto para atribuir valores aos membros e chamar a função *exibe_func*.

Declarando métodos de classe fora da classe

Na classe *funcionario* anterior, a função foi definida dentro da própria classe. À medida que as funções se tornarem maiores, defini-las dentro da classe poderá congestionar a definição da classe. Como alternativa, é possível colocar o protótipo de uma função dentro da classe e depois definir a função fora dela. Na classe, a definição com o protótipo de função, torna-se o seguinte:

```
Class funcionario
{
       public:
              char nome[64];
              long ident_func;
              float salario;
              void exibe_func(void)       // Prototipo da função
};
```

Figura 2.19 – Declaração de protótipo de função.

Como diferentes classe podem usar funções de mesmo nome, é preciso preceder os nomes das funções declaradas fora da classe com o nome da classe e o operador de resolução global (::). Neste caso, a definição da função *exibe_func* torna-se o seguinte:

```
Void funcionario::exibe_func(void)
{
       cout << "Nome: " << nome << endl;
       cout << "Ident: " << ident_func << endl;
       cout << "Salario: " << salario << endl;
};
```

Figura 2.20 – Definição de função fora da classe.

O código precede a definição da função com o nome da classe (*funcionário)* e o operador de resolução de escopo (::). O programa *ClassFun.CPP*, a seguir, move

a definição da função *exibe_func* fora da classe usando o operador de resolução global para especificar o nome da classe:

```
#include <iostream.h>
#include <string.h>

class funcionario
{
	public:
		char nome[64];
		long ident_func;
		float salario;
		void exibe(void);
};

Void funcionario::exibe_func(void)
{
	cout << "Nome: " << nome << endl;
	cout << "Ident: " << ident_func << endl;
	cout << "Salario: " << salario << endl;
};

void main(void)
{
	funcionario func, chefe;

	strcpy(func.nome, "João Silva");
func.ident_func = 12345;
fund.salario  = 25000.00;

strcpy(chefe.nome, "Joaquim Matos");
chefe.ident_func = 101;
chefe.salario = 101101.00;

func.exibe_func();
chefe.exibe_func();
```

Figura 2.21 – Exemplo de programa utilizando método implementado fora da classe.

2.3 – Dados públicos e privados em classes de C++

Os membros de classe públicos e privados controlam os membros de classe que os programas podem acessar diretamente usando o operador ponto. Os programas podem acessar os membros públicos a partir de qualquer função. Por outro lado, seus programas somente podem acessar os membros privados usando funções de

classe. Deste modo, os membros de classe privados permitem que os objetos controlem o modo de um programa usar seus dados membros. Neste tópico examina os membros públicos e privados em detalhe e enfoca os seguintes conceitos fundamentais:

Para controlar como os programas usam os membros de classe, C++ permite definir membros como públicos ou privados.

Os membros privados permitem que uma classe oculte informações sobre a classe que o programa não precisa conhecer ou acessar diretamente.

Classes que usam membros privados permitem acesso a eles por maio de *funções de interface*.

Como foi abordado no tópico anterior, ao definir uma classe, deve-se colocar, dentro da definição, tantas informações sobre um objeto quanto puder. Desse modo, os objetos se tornarão autocontidos, o que pode ampliar sua reutilização em diversos programas.

Compreendendo a ocultação de informações

Uma classe contém dados e métodos (funções). Para usar uma classe, os programss precisam conhecer as informações que ela armazena 9seus dados-membro) e os métodos que manipulam os dados (as funções). Os programas não precisam saber como os métodos funcionam; devem conhecer somente a tarefa que os métodos executam. Supondo, por exemplo uma classe chamada *arquivo*.

Idealmente, o programa só precisa saber que a classe fornece os métodos *arquivo.imprimir*, que imprime uma cópia formatada do arquivo atual, e *arquivo.excluir*, que apaga o arquivo. O programa também não precisa saber como esses dois métodos funcionam. Em outras palavras, o programa deve tratar a classe como uma "caixa-preta". Ele sabe quais métodos chamar e os parâmetros para os métodos, mas não conhece o processamento real que ocorre dentro da classe (a caixa preta).

As informações são ocultadas e só estão disponíveis ao acesso direto de um programa as informações mínimas de classe. Os membros de classe privados e públicos de C++ ajudam a ocultar informações. Anteriormente, o rótulo *public* foi usado para tornar todos os membros de classe públicos ou visíveis ao programa inteiro. portanto, qualquer membro de classe poderia ser acessado diretamente, usando o operador ponto, como mostrado abaixo.

```
Class funcionario
{
public:
            char nome[64];
            long ident_func;
            float salario;
            void exibe_func(void)
}
```

Figura 2.22 – Dados publicos em uma classe.

Ao criar classes podem-se ter membros cujos valores a classe usará internamente para efetuar seu processamento, mas que não podem ser acessados diretamente por um programa. Esses são *membros privados* e devem ser ocultados do programa. Por padrão, se o rotulo *public* não for incluído, C++ pressupõe que todos os membros de classe são privados. Seus programas não podem acessar membros de classe privados usando o operador ponto. Somente as funções de membro de classe podem acessar os membros de classe privados. Quando as

classes são criadas os membros podem ser separados em *private* e *public*, como mostrado a seguir:

```
Classe alguma_classe
{
	public:
		int alguma_variável;
		void inicializa_privada(int, float);
		void exibe_dados(void);
	private:
		int valor_chave;
		float numero_chave;
}
```

Figura 2.23 – Declarando dados públicos e privados.

Os rótulos *public* e *private* permitem definir facilmente os membros que são privados e os que são públicos. Neste exemplo, utiliza-se o operador ponto para acessar os membros públicos, como mostrado aqui:

```
alguma_classe objeto;                    // Cria um objeto
objeto. Alguma variável = 1001;
objeto.inicializa_privado(2002, 1.2345);
objeto.exibe_dados();
```

Figura 2.24 – Acessando membros públicos.

Se o programa tentar acessar os membros privados *valor_chave* ou *número_chave* usando o operador ponto, o compilador gerará erros de sintaxe.

Como regra geral, os dados-membro da classe serão protegidos do acesso direto do programa tornando-os membros privados. Não é possível atribuir valores aos membros diretamente usando o operador ponto; é preciso invocar um método da classe para atribuir esses valores. Impedindo o acesso direto aos dados-membro, garante-se que sempre serão atribuídos valores válidos aos dados-membro da classe. Supondo, por exemplo, que o objeto *reator_nuclear* do programa usa a variável-membro chamada *derretido*, que sempre deve conter um valor na faixa

de 1 a 5. Se o membro *derretido* for público, o programa pode acessar seu valor diretamente, alterando-o como quiser. Por exemplo, o comando a seguir atribui o valor 101 (que está fora do intervalo de 1 a 5) para o membro de classe *derretido*:

```
reator_nuclear.derretido = 101;
```

Figura 2.25 – Atribuição de valor inadequado a uma variável-membro.

Se a variável for privada, pode-se usar um método de classe, como *atribuir_derretido*, para atribuir valor ao membro. Como mostrado aqui, a função *atribui_derretido* pode testar o valor que o programa quer atribuir ao membro para garantir que seja válido:

```
int ataque::atribui_derretido(int valor)
{
	if((valor > 0) && (valor <= 5))
	{
		derretido = valor;
		return(0);		// Atribuição bem sucedida
	}
	else
	{
		return(-1);		// Valor inválido
}
```

Figura 2.26 – Método de classe.

Os métodos de classe, que controlam o acesso aos dados-membro, são *funções de interface*. Ao criar classes, podem ser usadas funções de interface para proteger os dados da classe.

Compreendendo os membros públicos e privados

As classes de C++ contêm dados e métodos. Para controlar os membros que seus programas podem acessar diretamente usando o operador ponto, C++ permite definir membros públicos e privados. Seus programas podem acessar diretamente qualquer membro público usando o operador ponto. Por outro lado, somente os métodos da classe podem acessar os membros privados. Como regra, a maioria dos membros da classe dever ser protegidos tornando-os privados. Depois, o único modo dos programas poderem atribuir um valor a um dado-membro será usar uma função de classe, que pode examinar e validar o valor.

Usando membros públicos e privados

O programa Oculta.CPP, a seguir, que define um objeto do tipo *funcionário*, ilustra o uso de membros públicos e privados, como mostrado aqui:

```
class funcionario
{
        public:
                int atribui_valores(char *, long, float);
                void exibe_funcionario(void);
                int altera_salário(float);
                long obtem_ident(void);
        private:
                char nome[64];
                long funcionario_id;
                float salario;
};
```

Figura 2.27 – Usando membros públicos e privados.

A classe protege cada um de seus dados-membro tornando-os privados. Para acessar um dado-membro, o programa precisa usar uma das funções de interface pública. O programa a seguir implementa Oculta.CPP:

```
#include <iostream.h>
#include <string.h>

class funcionario
{
public:
              int atribui_valores(char *, long, float);
              void exibe_funcionario(void);
              int altera_salário(float);
              long obtem_ident(void);
       private:
              char nome[64];
              long funcionario_id;
              float salario;
};

int funcionario:: atribui_valores(char *emp_nome, long emp_id, float emp_salario)
{
       strcpy(nome, emp_nome);
       funcionário_id = emp_id;

       if(emp_salário < 50000.00)
       {
              salário = emp_salário;
              return(0);                    // Bem sucedido
       }
       else
       {
              return(-1);                   // Salário invalido
}

void funcionário::exibe_funcionário(void)
{
       cout << "Funcionário: " << nome << endl;
       cout << "Ident: " << funcionário_id << endl;
       cout << "Salário: " << salário << endl;
}

int funcionário::altera salário(float new_salário)
{
       if(new_salário < 5000.00)
       {
              salario = new_salario;
              return(0);                    // Bem sucedido
       }
       else
       {
              return(-1);                   // Salário invalido
}

long funcionario::obtem_ident(void)
{
```

```
        return(funcionario_id);
}

void main(void)
{
        funcionario func;

        if(func.atribui_valores("Joaquim Matos", 101, 1010.0) == 0)
        {
                cout << "Valores de funcionário atribuídos" << endl;
                func.exibe_funcionário();
                if(func.altera_salário(3500.00) ==0)
                {
                        cout << "Novo salário atribuído" << endl;
                        func.exibe_funcionário();
                }
        }
        else
        {
                cout << "Salário inválido especificado" <, endl;
}
```

Figura 2.28 – Programa usando membros públicos e privados.

Embora o programa anterior seja longo, suas funções são simples. O método *atribui_valores* inicializa a classe de dados privados e usa um comando if para garantir que atribui um salário válido. O método *exibe_func* exibe os dados-membro privaods. Os métodos *altera_salário* e *obtem_ident* são funções de interface que fornacem ao programa o acesso a dados privados. Se o programa Oculta.CPP for editado para tentar acessar diretamente um dado-membro privado usando um operador ponto a partir de *main*, o compilador gerará erros de sintaxe.

Funções de interface

Para reduzir erros potenciais, o acesso do programa aos dados da classe pode ser limitado definindo estes como privados. Deste modo, um programa não poderá acessar os dados-membro da classe usando o operador ponto. Em vez disso, a classe deverá definir funções de interface com as quais o programa poderá

atribuir valores aos membros privados. As funções de interface, por sua vez, poderão examinar e validar os valores que o programa está tentando atribuir.

2.4 – Funções construtora e destrutora

Ao criar objetos, uma das operações mais comuns que os programas farão é inicializar os dados-membro do objeto. O único modo dos programa terem acesse aos dados-membro privados é usando uma função da classe. Para simplificar o processo de inicializar os dados-membro da classe, C++ permite definir uma função *construtora* especial, uma para cada classe, que C++ chama automaticamente toda vez que um objeto for criado. De forma similar, C++ oferece uma função *destrutora* que é executada quando o objeto é descartado. Este tópico examina as funções construtora e destrutora em detalhes e aborda os seguintes conceitos fundamentais:

- As funções construtoras são métodos que facilitam aos programas a inicialização dos dados-membro da classe;
- As funções construtoras têm o mesmo nome que a classe; entretanto, o nome delas não é precedido pela palavra-chave *void*;
- As funções construtoras não retornam um tipo;
- Cada vez que o programa cria uma variável de classe, C++ chama a função construtora da classe, se ela existir;
- À medida que o programa é executado, muitos objetos podem alocar memória para armazenar informações quando um objeto for descartado,

C++ chamará uma função destrutora especial que irá liberer essa memória, em certo sentido, fazendo a limpeza após o objeto terminar;

- As funções destrutoras têm o mesmo nome que a classe, precedido com o caractere til (~);
- As funções destrutoras não retornam um tipo e, como na função construtora, não são precedidas pela palavra-chave *void*.

Criando uma função construtora simples

Uma função construtora é um método de classe com o mesmo nome que a própria classe. Por exemplo, se for usada uma classe chamada *funcionário*, o nome da função construtora também é *funcionário*. Da mesma forma, para uma classe chamada *cachorros*, o nome da função construtora é *cachorros*. Se seu programa definir uma função construtora, C++ a chamará automaticamente toda vez que um objeto do tipo de classe correspondente for criado. No programa ConStru.CPP, a seguir, é definida uma classe chamada *funcionário*. O código também define uma função construtora chamada *funcionário*, que atribui os valores iniciais do objeto. Uma função construtora não pode retornar um valor; portanto, não pode ser declarada como *void*.

```
class funcionario
{
      public:
            funcionario(char *, long, float);  // Função construtora
            ~funcionario(void);                // Função destrutora
            void exibr_func(void);
            int altera_salário(float);
            long obtem_ident(void);
      private:
            char nome[64];
            long ident_func;
            float salario;
};
```

Figura 2.29 – Declarando funções *construtora* e *destrutora.*

A função destrutora é definida como qualquer método de classe é definido. Assim:

```
funcionario::funcionario(char *nome, long ident_func, float salario)
{
	strcpy(funcionario::nome, nome);
	funcionário:: ident_func = ident_func;
	if(salário < 5000.00)
	{
		funcionario::salario = salario;
	}
	else
	{
		// Salário inválido especificado
	funcionario::salario = 0.0;
	}
}
```

Figura 2.30 – Corpo de uma função construtora.

Como pode ser visto, a função construtora não retorna um valor para o chamador. Além disso, ela não usa o tipo *void*. Neste exemplo, a função usa o operador de resolução global e o nome de classe antes de cada membro. O programa a seguir implementa ConStru.CPP completo:

```
#include <iostream.h>
#include <string.h>

class funcionario
{
	public:
		funcionario(char *, long, float);  // Função construtora
		~funcionario(void);                // Função destrutora
		void exibr_func(void);
		int altera_salário(float);
		long obtem_ident(void);
	private:
		char nome[64];
		long ident_func;
		float salario;
};

funcionario::funcionario(char *nome, long ident_func, float salario)
{
	strcpy(funcionario::nome, nome);
	funcionário:: ident_func = ident_func;
	if(salário < 5000.00)
```

```
        {
                funcionario::salario = salario;
        }
        else
        {
                // Salário inválido especificado
        funcionario::salario = 0.0;
        }
}

funcionário::~funcionario(void)
{
        cout << "Destruindo o objeto para " << nome << endl;
}

void funcionario::exibe_func(void)
{
        cout << "Funcionário: " << nome << endl;
        cout << "Ident: " << ident_func << endl;
        cout << "Salário: " << salario << endl;
}

void main(void)
{
        funcionario func("Joaquim Matos", 101, 10101.00);

        func.exibe_func();
}
```

Figura 2.31 – Programa completo usando funções construtora e destrutora.

O programa ConStru.CPP segue a declaração do objeto *func* com parênteses e os valores iniciais do objeto, exatamente com uma chamada de função. Ao usar funções construtoras, o programa precisa passar parâmetros para elas quando declara um objeto.

Se o programa tivesse criado vários objetos *funcionário*, poderia-se inicializar os membros de cada objeto usando a função construtora, como mostrado a seguir.

```
funcionario func("Joaquim Matos", 101, 1010.00);
funcionario secretaria("Jerusa Silva, 57, 2000.00);
funcionario gerente("Jane Silva", 1022, 3000.00);
```

Figura 2.32 – Declarando e inicializando vários objetos.

Especificando valores de parâmetro-padrão para as funções construtoras

C++ permite especificar valores de parâmetros-padrão para as funções, que serão usados quando o usuário não os especificar. As funções construtoras não são exceção. Os programas podem especificar valores-padrão exatamente como fariam para qualquer função. Por exemplo, a função construtora *funcionário*, a seguir, usa o salário-padrão de 10000.00 se o programa não especificar um salário quando criar o objeto. No entanto, o programa precisa especificar um nome de funcionário e um número de identificação:

```
Funcionario::funcionario(char *nome, long ident_func, float salario = 10000.00)
{
	strcpy(funcionario::nome, nome);
	funcionário::ident_func = ident_func;
	if(salário < 5000.00)
	{
		funcionário:;salário = salário;
	}
	else
	{
		// Salário inválido especificado
		funcionário::salário = 0.0;
	}
}
```

Figura 2.33 – Inicializando um objeto com parâmetros-padrão

Sobrecarregando as funções construtoras

C++ permite que seus programas sobrecarreguem as definições de funções especificando funções alternativas para tipos de parâmetros diferentes. C++ permite também sobrecarregar as funções construtoras. O programa SobreCon.CPP, a seguir, sobrecarrega a função construtora *funcionário*. A primeira definição de função construtora requer que o programa especifique um nome, número de identificação e salário de *funcionario*. A segunda definição de função construtora pede que o usuário digite o salário desejado se o programa não especificar um, como mostrado a seguir.

```
funcionario::funcionario(char *nome, long ident_func)
{
	strcpy(funcionario::nome,nome);
	funcionário::ident_func = ident_func;

	do
	{
		cout << "Informe o salário para " << nome << "menor que 5000.00: ";
		cin >> funcionario::salario;
	} while (salario >= 5000.00);
}
```

Figura 2.34 – Sobrecarga de função construtora.

Dentro da definição da classe, o programa precisa especificar ambos os protótipos de função, como pode ser visto nas linhas 7 e 8 do programa SobreCon.CPP, a seguir.

```
#include <iostream.h>
#include <string.h>

class funcionario
{
	public:
		funcionario(char *, long, float);  // Função construtora
		funcionário(char *, long);
		~funcionario(void);                // Função destrutora
		void exibr_func(void);
		int altera_salário(float);
		long obtem_ident(void);
	private:
		char nome[64];
		long ident_func;
		float salario;
};

funcionario::funcionario(char *nome, long ident_func, float salario)
{
	strcpy(funcionario::nome, nome);
	funcionário:: ident_func = ident_func;
	if(salário < 5000.00)
	{
		funcionario::salario = salario;
	}
	else
	{
		// Salário inválido especificado
	funcionario::salario = 0.0;
	}
}

funcionario::funcionario(char *nome, long ident_func)
{
	strcpy(funcionario::nome,nome);
	funcionário::ident_func = ident_func;
```

```
        do
        {
                cout << "Informe o salário para " << nome << "menor que 5000.00: ";
                cin >> funcionario::salario;
        } while (salario >= 5000.00);
}

funcionário::~funcionario(void)
{
        cout << "Destruindo o objeto para " << nome << endl;
}

void funcionario::exibe_func(void)
{
        cout << "Funcionário: " << nome << endl;
        cout << "Ident: " << ident_func << endl;
        cout << "Salário: " << salario << endl;
}

void main(void)
{
        funcionario func("Joaquim Matos", 101, 10101.00);
        funcionário gerente("Jane Silva", 102);

        func.exibe_func();
        gerente.exibe_func();
}
```

Figura 2.35 – Programa completa exemplificando sobrecarga de função construtora.

Quando o programa SobreCon.CPP for compilado, uma mensagem na tela pedirá que o usuário digite o salário para Jane Silva. Quando o fizer, a execução do programa continuará exibindo informações sobre os dois funcionários.

Funções destrutoras

Da mesma forma que a função construtora é chamada automaticamente quando é criado um objeto de classe, C++ permite definir uma função destrutora, chamada quando o objeto é destruído. Uma função destrutora é útil quando há a necessidade de liberar a memória alocada por um objeto quando o programa ainda se encontra em execução.

2.5 – Sobrecarga de operador

O tipo de uma variável especifica o conjunto de valores que ela pode armazenar e um conjunto de operações que se pode executar com ela. Por exemplo, usando uma variável do tipo *int*, o programa pode adicionar, subtrair, multiplicar e dividir os valores. Quando uma classe é definida dentro dos programas, está essencialmente definindo um novo tipo. A *sobrecarga de operador* é o processo de alterar o significado de um operador (como o sinal de mais(+), que C++ normalmente usa para a adição para uso por uma classe específica. Neste tópico será definida uma classe *string* e serão sobrecarregados os operadores de mais e de menos. Para os objetos string, o operador de mais acrescentará caracteres especificados no conteúdo atual da string. De modo similar, o operador de menos removerá cada ocorrência de um caractere especificado do arranjo. Este tópico aborda os seguintes conceitos fundamentais:

Deve-se sobrecarregar um operador somente quando isso de fato tornar o programa mais fácil de entender;

Usa-se a palavra-chave *operator* para sobrecarregar um operador;

Ao sobrecarregar um operador, especifica-se uma função que C++ chama toda vez que a classe usa o operador sobrecarregado. A função, por sua vez, executa a operação correspondente;

Quando um operador é sobrecarregado para uma classe específica, o significado do operador muda somente para essa classe. O resto do programa continuará a usar o operador para efetuar as operações-padrão;

C++ permite a sobrecarga da maioria dos operadores; no entanto, existem quatro operadores que não podem ser sobrecarregados.

A Tabela 2.1 relaciona os operadores para os quais C++ não permite sobrecarga.

Tabela 2.1 – Operadores que não poder ser sobrecarregados.

Operador	Propósito	Exemplo
.	Operador de membro de classe	Objeto.membro
.*	Ponteiro para operador membro	Objeto.*membro
::	Operador de resolução global de escopo	Nomeclasse::membro
?:	Operador de expressão condicional	c = (a > b) ? a : b;

Sobrecarregando os operadores de mais e menos

Quando um operador é sobrecarregado para uma classe, a função desse operador não muda para os outros tipos de variáveis. Por exemplo, se o operador mais for sobrecarregado par a classe *string*, a função do operador não mudará quando for preciso somar dois números. Quando o compilador C++ encontrar o operador dentro do programa, ele determinará as operações a executar com base no tipo da variável correspondente. A definicção de classe, a seguir, cria uma classe *string*. Esta contém um dado membro, que é a própria string de caracteres. A classe contém vários métodos diferentes e atualmente não define quaisquer operadores, como mostrado a seguir.

Quando um operador é sobrecarregado, utiliza-se a palavra-chave *operator* dentro do protótipo e definição da função para dizer ao compilador C++ que a classe usará esse método como um operador. Por exemplo, a definição de classe, a seguir, usa-se a palavra-chave *operator* para atribuir os operadores de mais e de menos para as fuções *str_anexa* e *chr_menos* dentro da classe *string*:

```
class string
{
	public:
		string(char *);			// Construtora
		void operator +(char *);	// Operador de classe
		void operator –(char);		// Operador de classe
		void exibe_string(void);
	private:
		char dados[256];
};
```

Figura 2.36 – Criando operadores dentro de classe.

A classe sobrecarrega os operadores de mais e de menos. Ao sobrecarregar um operador, a classe precisará especificar uma função que implemente a operação correspondente do operador. No caso do operador de mais, a definição da função torna-se o seguinte:

```
void string::operator +(char *str)
{
	strcat(dados, str);
}
```

Figura 2.37 – Definição de operador de classe.

O programa SobreOpe.CPP, a seguir, ilustra o uso dos operadores de mais e menos sobrecarregados:

```
#include <iostream.h>
#include <string.h>

class string
{
	public:
		string(char *);			// Construtora
		void operator +(char *);	// Operador de classe
		void operator –(char);		// Operador de classe
		void exibe_string(void);
	private:
		char dados[256];
};
```

```
string::string(char *str)
{
	strcpy(dados, str);
}

void string::operator +(char *str)
{
	return(strcat(dados, str));
}

void string::operator -(char *letra)
{
	char temp[256];
	int i, j;

	for(i = 0, j = 0; data[i]; i++)
	{
		if(dados[i] != letra)
		{
			temp[j++] = dados[i];
		}
	}

	temp[j] = NULL;

return(strcpy(data, temp));
}

void string::exibe_string(void)
{
	cout << data << endl;
}

void main(void)
{
	string titulo(“Aprendendo C++”);
	string exemplo(“compreendendo a sobrecarga de operador”);

	titulo.exibe_string();
	titulo = titulo + “ neste tópico!”;
	titulo.exibe_string();

	lição.exibe_string();
	licao = licao – ‘n’;
	licao.exibe_string();
}
```

Figura 2.38 – Exemplo de programa usando classe com operadores sobrecarregados.

3 – Sfloat

Sfloat simula o armazenamento e o funcionamento de um número de ponto flutuante binário normalizado. Está implementado como uma classe, possuindo

variáveis-membro, métodos de classe e operadores aritméticos sobrecarregados, como será visto mais adiante.

3.1 – Implementação da classe Sfloat

Na classe de Sfloat há um arranjo de variáveis booleanas chamado ArrayBit, representando os *bits* de um número do tipo ponto flutuante no formato padrão. A classe Sfloat foi implementada como mostra o código fonte a seguir.

```
#include <string.h>
#include <iostream.h>
#include <math.h>
#include <stdio.h>
#include "parameters.h"

class Sfloat
{
public:

// Funções construtoras, destrutora e operador atribuição
       Sfloat(bool[TamSfloat]);
       Sfloat(char[TamSfloat]);
       Sfloat(double);
       Sfloat(void);
       ~Sfloat(void);
Sfloat operator=(double);

// Operadores para exibição de valores
       Sfloat operator<<(Sfloat);
       void Sout(int);
       void SoutInfo();

// Operadores logicos binários

       bool operator >(Sfloat);
       bool operator <(Sfloat);
       bool operator >=(Sfloat);
       bool operator <=(Sfloat);
       bool operator ==(Sfloat);
       bool operator !=(Sfloat);

// Operadores logicos binarios sobrecarregados com Sfloat x double

       static bool operator>(Sfloat, double);
       static bool operator<(Sfloat, double);
       static bool operator>=(Sfloat, double);
       static bool operator<=(Sfloat, double);
       static bool operator==(Sfloat, double);
       static bool operator!=(Sfloat, double);
```

```
// Funcoes auxiliares

        Sfloat Abs(void);
        void Std(bool[TamExp], bool[TamEstMant]);
        void DigDecExpPSfloat(int Number, int Expoente);
        bool MaiorAbs(Sfloat,Sfloat);
        double Double(void);

// Operadores aritméticos

        // Unários
        Sfloat operator+(void);
        Sfloat operator-(void);

        // Binarios
        Sfloat operator+(Sfloat);
        Sfloat operator-(Sfloat);
        Sfloat operator*(Sfloat);
        Sfloat operator/(Sfloat);

        // Binarios sobrecarregados com Sfloat x double
        static Sfloat operator+(Sfloat, double);
        static Sfloat operator-(Sfloat, double);
        static Sfloat operator*(Sfloat, double);
        static Sfloat operator/(Sfloat, double);

        // operador << sobrecarregado
        friend ostream& operator<< (ostream& Sout, Sfloat& SfloatA);

private:

        bool ArrayBit[TamSfloat];

};
```

Figura 3.1 – Definição da classe Sfloat.

Cada trecho da implementação acima será explanado a seguir.

Bibliotecas

Antes da definição da classe são indicadas as bibliotecas que conferem funcionalidade a Sfloat, como pode ser visto na Figura 3.2.

```
#include <string.h>
#include <iostream.h>
#include <math.h>
#include <stdio.h>
#include "parameters.h"
```

Figura 3.2 – Bibliotecas de Sfloat.

As bibliotecas permitem funções como processamento de strings, entrada e saída de valores. A última biblioteca “parameters.h” é um arquivo onde estão os

parâmetros e variáveis globais de Sfloat. A seguir segue o corpo de texto do referido arquivo.

```
const int TamSfloat = 64;          // number of bits in Sfloat
const int TamMant = 52;            // number of bits in Sfloat mantissa

// Parametros de sfloat
const int TamExp = TamSfloat - TamMant - 1;
const int InicioExp = 1;
const int FinalExp = TamExp;
const int InicioMant = FinalExp + 1;
const int FinalMant = TamSfloat - 1;

// Parametros auxiliares
const int TamTempMant = TamMant + 1;
const int TamEstMant = TamMant + 2;

bool BoolDifExp[TamExp], BoolTempExp[TamExp+2], Expoente[TamExp];
int IntDifExp, IntBinPDec;
int IntIndex, IntI, IntJ, IntK;
bool MantA[TamMant+1], MantB[TamMant+1], EstMant[TamEstMant];
bool Um[TamSfloat], Dez[TamSfloat];
bool UpDown1, Up1, ZeroA, ZeroB, ZeroC, InfA, InfB, InfC, NaNA, NaNB, NaNC;
```

Figura 3.3 – Arquivo de parâmetros de Sfloat

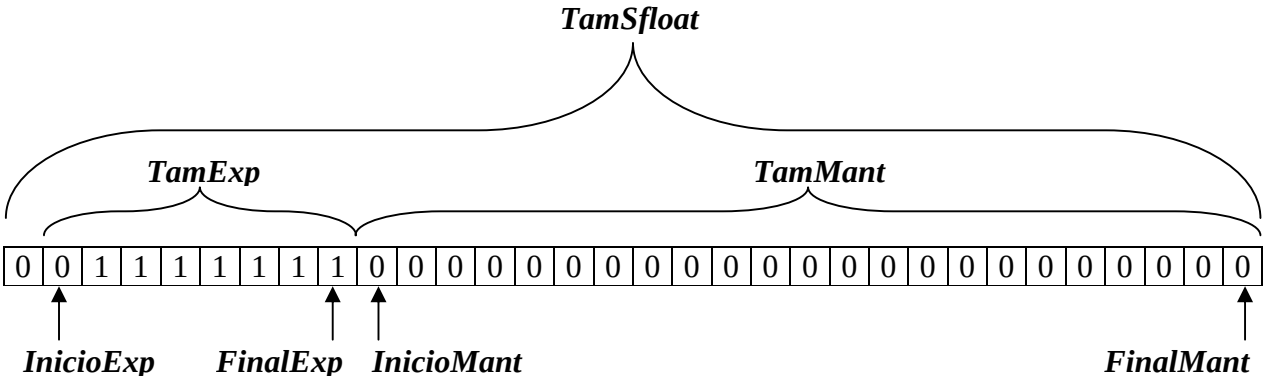


Figura 3.4 – Esquematização dos parâmetros de Sfloat.

Na figura 3.4 estão os parâmetros que definem a classe Sfloat:

TamSfloat: é o parâmetro mais importante, pois define o tamanho do *array*;

TamMant: Igualmente importante, define o tamanho da mantissa, onde estão os *bits* mais importantes. Todos os outros parâmetros são diretamente definidos a partir de *TamSfloat* e *TamMant*;

TamExp: Representa o tamanho do expoente e é definido em função do tamanho total de Sfloat, subtraindo o tamanho da mantissa e um bit do sinal. Assim, *TamExp = TamSfloat – TamMant – 1*;

InicioExp: Marca o início do expoente. Aqui, *InicioExp = 1*, uma vez que em C++ a indexação dos arrays iniciam no zero e a variável ocupa a segunda posição;

FinalExp: Final do expoente. FinalExp = InicioExp + TamExp – 1, mas como InicioExp = 1, tem-se FinalExp = TamExp;

InicioMant: Marca o início da mantissa. Sua posição é imediatamente após o final do expoente. Assim: *InicioMant = FinalExp + 1*:

FinalMant: Marca o final da mantissa e coincide com o final da seqüência de *bits* de Sfloat. Assim: *FinalMant = TamSfloat –1*;

Esses parâmetros definem como são armazenados os dados dos números de ponto flutuante.

Em seqüência dos parâmetros de Sfloat, estão os parâmetros auxiliares, que são constantes e variáveis globais que ajudam a simplificar o código fonte.

Os parâmetros auxiliares são:

TamTemMant: Define o tamanho das mantissas temporárias, que armazenam os valores de mantissa dos números durante operações aritméticas acrescido do dígito escondido. Por isso *TamTempMant = TamMant+ 1*;

TamEstMant: Define o tamanho da mantissa resultado das operações aritméticas. Além de armazenar o dígito escondido, armazena também mais um dígito para arredondamento: *TamEstMant = TamMant + 2*;

BoolDifExp, *BoolTempExp* e *Expoente*: Arranjos utilizados para armazenar o valor de diferença entre expoentes como número binário e valor de expoente temporariamente;

IntDifExp e *IntBinPDec*: utilizados para a conversão valor de expoente decimal para valor binário e vice-versa;

IntIndex, *IntI*, *IntJ*, *IntK*: Usados como variáveis globais de contagem, utilizadas apenas pelos operadores binários aritméticos.

MantA, *MantB* e *EstMant*: Os dois primeiros copiam os valores das mantissas de dois números durante o uso de operadores aritméticos binários. O último armazena a mantissa do resultado de operações aritméticas antes de normalizar e arredondar.

Inicializando e atribuindo valores

Sfloat está implementado com quatro funções construtoras sobrecarregadas e uma função destrutora, como mostra a seguir.

```
// Funções construtoras e destrutora
      Sfloat(bool[TamSfloat]);
      Sfloat(char[TamSfloat]);
      Sfloat(double);
      Sfloat(void);
      ~Sfloat(void);
```

Figura 3.5 – Trecho da definição da classe Sfloat (funções construtoras e destrutora)

Quando um programa cria um objeto do tipo Sfloat, uma função construtora é chamada. Se o programa não passar nenhum parâmetro, a função chamada é *Sfloat(void)*. Algumas das funções construtoras sobrecarregadas permitem inicializar um objeto Sfloat com um valor definido, são elas:

- *Sfloat(bool[TamSfloat])*: Um objeto pode ser inicializado com os valores binários na seqüência normalizada;
- *Sfloat(char[TamSfloat])*: Inicializa um objeto Sfloat a partir de uma seqüência de caracteres;
- *Sfloat(double)*: Inicializa um objeto Sfloat a partir de um número do tipo *double*;

As funções construtoras inicializam um objeto Sfloat. Durante a execução de um programa, os valores de um objeto Sfloat são alterados pelo operador atribuição "=", quando recebe um valor de um objeto que também seja do tipo Sfloat. Se o valor atribuído for um número como no exemplo a seguir, utiliza-se o operador atribuição sobrecarregado *operator=(double)*.

```
// Objetos Sfloat: Criando, inicializando e atribuindo valores

double DoubA = 23;          // Cria e inicializa um double

Sfloat SfA;                 // Sem passar parâmetros
Sfloat SfB(DoubA);          // Passando uma variável do tipo double
Sfloat SfC = 2;             // Passando um número interpretado como double

SfA = SfC;                  // Atribuindo um valor a partir de um objeto Sfloat
SfB = 3;                    // Atribuindo um valor a partir do operador "=" sobrecarregado
```

Figura 3.6 – Inicializando e atribuindo valores a objetos Sfloat.

Na linha cinco, *sfloat SfA*, o programa cria uma variável to tipo Sfloat sem nenhum parâmetro. Neste caso, o programa chama a função construtora *Sfloat(void)*, que cria um Sfloat onde todos os valores do ArrayBit são nulos.

Na linha seis, *Sfloat SfB(DoubA)*, a variável DoubA que foi criada e inicializada na linha três é o parâmetro da função construtora *Sfloat(double)*.

Na linha sete, o objeto Sfloat é criado e imediatamente inicializado através do operador atribuição sobrecarregado *operator=(double)*, que também é invocado na linha dez, ao atribuir um valor interpretado como sendo um *double*.

Na linha nove, o operador atribuição "=" comum. Este operador é definido automaticamente quando a classe é definida, como visto no capítulo anterior.

Saída de valores

Para exibir valores de objetos Sfloat foram implementadas três funções e o operador "<<" foi sobrecarregado. Os valores também podem ser convertidos em números *double* e depois exibidos. Na definição da classe as funções e o operador sobrecarregado aparecem no trecho a seguir.

```
// Operadores para exibição de valores
      void Sout(int);
      void SoutInfo();
      Sfloat operator<<(Sfloat);
```

Figura 3.7 – Funções e operador para exibição de valores.

A função *Sout(int)* converte o número Sflaot em um *double* e o exibe na forma científica com o número de casas definido por um parâmetro inteiro.

A função *SoutInfo()* não converte o número, ela exibe os valores binários separando o sinal do expoente através de um espaço e separando também o expoente da mantissa com um espaço.

O operador sobrecarregado "<<" converte o número Sfloat em um número do tipo *double* e exibe este último. Geralmente, o número será exibido na forma inteira ou decimal, se o número exigir maiores cuidados será exibido na forma científica.

No trecho a seguir, alguns exemplos de valores exibidos.

```
Sout(3):      2.33e002
Sout(4):      2.330e002
SoutInfo():   0 10000000 11000000000000000000000
"<<":         233
"<<":         3e10
```

Figura 3.8 – Exibindo números do tipo Sfloat.

Operadores lógicos binários e lógicos binários relacionais

Os operadores lógicos binários relacionais são, em C++, && e |, que significam *e* e *ou*, respectivamente. Estes não precisam ser implementados pois continuam operando da mesma maneira, ou seja, combinam dois operandos com valores lógicos para retornar um valor lógico.

Já os operadores lógicos binários combinam dois operandos do tipo Sfloat ou uma combinação de operando Sfloat com outro valor. Assim precisam ser sobrecarregados.

A seguir, o trecho que define o protótipo dos operadores sobrecarregados dentro da definição da classe.

```
// Operadores lógicos binários
```

```
        bool operator >(Sfloat);
        bool operator <(Sfloat);
        bool operator >=(Sfloat);
        bool operator <=(Sfloat);
        bool operator ==(Sfloat);
        bool operator !=(Sfloat);

// Operadores lógicos binarios sobrecarregados com Sfloat x double

        static bool operator>(Sfloat, double);
        static bool operator<(Sfloat, double);
        static bool operator>=(Sfloat, double);
        static bool operator<=(Sfloat, double);
        static bool operator==(Sfloat, double);
        static bool operator!=(Sfloat, double);
```

Figura 3.9 – Definição dos protótipos dos operadores lógicos binários.

Como pode ser visto acima os operadores lógicos binários são: <, >, <=, >=, == e !=. Eles operam sobre dois números Sfloat e retornam um valor lógico verdadeiro(um) ou falso(zero) de acordo com a condição estabelecida pelo próprio operador.

Se um operador lógico binário receber um operando Sfloat e depois um operando *double*, é necessário ser sobrecarregado. Neste caso o primeiro operando é um número do tipo Sfloat e o segundo operando é do tipo *double*.

Se o primeiro argumento for do tipo *double* e o segundo do tipo Sfloat, o operador sobrecarregado não é um método de classe e deve ser definido fora da classe, o que o torna um operador global. Estes operadores são definidos como estão no trecho a seguir.

```
// Operador binário > sobrecarregado para double x Sfloat
bool operator >(double DoubA, Sfloat SfloatB)
{
        bool Result = 0;
        Sfloat SfloatA = DoubA;             // Criando e inicializando SfloatA
        Result = SfloatA > SfloatB;         // Chamando o operador >
        return Result;
}

// Operador binário < sobrecarregado para double x Sfloat
bool operator <(double DoubA, Sfloat SfloatB)
{
        bool Result = 0;
        Sfloat SfloatA = DoubA;
```

```
        Result = SfloatA < SfloatB;
        return Result;
}
```

Figura 3.10 – Operadores lógicos binários sobrecarregados com operandos mistos.

Quando um operador deste é chamado pelo programa, ele converte o argumento do tipo *double* para o tipo Sfloat e chamando em seguida o operador que utiliza dois argumentos do tipo Sfloat como está definido no trecho anterior.

Operadores aritméticos

Os operadores aritméticos são divididos em unários e binários. Os unários são "+" e "–" e os binários são "+", "–", "*" e "/". Os operadores "+" e "–" comportan-se como unários ou como binários, dependendo do número de argumentos.

A seguir podem ser vistos exemplos de utilização do operadores aritméticos unários e binários.

```
C = +A;             // Operador unário positivo +
C = -B;             // Operador unário negativo –

C = A + B;          // Operador binário soma +
C = A – B;          // Operador binário subtração –
C = A * B;          // Operador binário multipicação *
C = A / B;          // Operador binário divisão /
```

Figura 3.11 – Operadores aritméticos binários.

Os operadores aritméticos binários serão abordados no tópico a seguir.

3.2 – Operadores aritméticos binários "+", "-", "*" e "/"

O operadores executam as operações aritméticas como sugerido no padrão IEEE 754. Nesta parte será explanado como funcionam e como estão implementados.

Operadores aritméticos binários "+" e "–"

Em uma soma, se dois números são positivos, a operação se processa naturalmente. Por outro lado, se o segundo número for negativo a soma, na realidade, se processa como uma subtração. Se porventura, os dois números forem negativos, a operação se processa novamente como uma soma, retornando como resposta um número negativo.

Diante deste fato, soma e subtração funcionam da mesma forma, ficando a diferença por conta dos sinais dos dois argumentos.

A Tabela 3.1 ajuda a comparar os operadores soma e subtração.

Tabela 3.1 – Entrando, processando e saindo valores de soma e subração.

Operador	Sinal dos argumentos 1º	2º	Operação	Sinal do resultado
+	+	+	Soma	+
+	+	-	Subtração	Sinal do maior
+	-	+	Subtração	Sinal do maior
+	-	-	Soma	-
-	+	+	Subtração	Sinal do maior[#]
-	+	-	Soma	+
-	-	+	Soma	-
-	-	-	Subtração	Sinal do maior[#]

[#] Se o primeiro argumento for maior em módulo, o sinal deste prevalecerá, mas se o segundo argumento for maior em módulo, o sinal oposto ao seu prevalecerá, uma vez que o sinal do segundo argumento inverte-se.

Como pode ser notado, os operadores soma e subtração tem as mesmas características principais de soma e subtração dependentes dos sinais dos argumentos.

A implementação destes é extensa e não será apresentada aqui. Mas o algoritmo principal, tanto do operador soma quanto do operador subtração é o que segue, seguindo os mesmos passos do tópico 1.6 deste trabalho, com pequena alteração para verificação de exceções antes da operação.

(i) Verificação da existência de exceções
(ii) Deslocar a mantissa do número de menor expoente para a direita o número igual a diferença absoluta entre os expoentes dos operandos;
(iii) Fazer o expoente do resultado igual ao expoente do maior;
(iv) Executar a adição ou subtração das mantissas e determinar o sinal do resultado;
(v) Normalizar o valor do resultado;
(vi) Arredondar o valor do resultado;
(vii) Verificar ocorrência de *overflow* ou *underflow*.

Figura 3.12 – Passos executados nos operadores de soma e de subtração.

No caso de um dos argumentos apresentar uma exceção, o resultado sofre uma alteração. Se um deles for considerado infinito, implica em um caso de *overflow*. No caso de um *underflow*, o argumento que possuir esta característica será considerado igual a zero.

Pode ser visto na Tabela 3.2, a seguir, como a existência de exceções modifica o resultado das operações, sendo as exceções INF e NaN, infinito e “Not a Number”, respectivamente. No caso de *underflow*, o número que tiver esta característica será considerado zero e a operação será processada.

Tabela 3.2 – Exceções influenciando resultados de somas e subtrações.

Argumentos		Soma	Subtração
+∞	+∞	+∞	NaN
+∞	-∞	NaN	+∞
-∞	+∞	NaN	-∞
-∞	-∞	-∞	NaN

+∞	Número	+∞	+∞
-∞	Número	-∞	-∞
Número	+∞	+∞	-∞
Número	-∞	-∞	+∞
NaN	Número ou ±∞	NaN	NaN
Número ou ±∞	NaN	NaN	NaN

Operador multiplicação

O operador multiplicação tem implementação mais simples que os dois anteriores, uma vez que a multiplicação processa-se sempre da mesma forma. A particularidade do operador multiplicação fica por conta do jogo de sinais. Sinais iguais acarretam em sinal positivo para o resultado e sinais diferentes acarretam sinal negativo para o resultado.

O algoritmo principal, segue os passos do tópico 1.7 deste trabalho. Como anteriormente, há verificação de exceções antes da operação. A seguir os passos principais do algoritmo.

(i) Verificar a existência de exceções
(ii) Colocar o expoente de resultado igual à soma dos expoentes dos operandos;
(iii) Executar a multiplicação das mantissas e determinar o sinal do resultado;
(iv) Normalizar o valor do resultado, se necessário;
(v) Arredondar o valor do resultado, se necessário e
(vi) Verificar se houve *overflow* ou *underflow*.

Figura 3.13 – Passos executados pelo operador multiplicação.

A Tabela 3.3 mostra como a existência de exceções modifica o resultado da operação de multiplicação.

Tabela 3.3 – Exceções influenciando resultados de multiplicações.

Argumentos		Soma
Primeiro	Segundo	Resultado
+∞	+∞	+∞
+∞	-∞	-∞
-∞	+∞	-∞
-∞	-∞	+∞
+∞	Número positivo	+∞
+∞	Número negativo	-∞
-∞	Número positivo	-∞
-∞	Número negativo	+∞
Número positivo	+∞	+∞
Número negativo	+∞	-∞
Número positivo	-∞	-∞
Número negativo	-∞	+∞
NaN	Número ou ±∞	NaN
Número ou ±∞	NaN	NaN

Operador divisão

Assim como o operador multiplicação, o operador divisão leva os sinais em conta apenas para atribuir o sinal do resultado. As operações principais independem dos sinais dos operandos.

Basicamente, a implementação do operador divisão segue os passos do tópico 1.8, também acrescido de uma verificação de exceções antes de executar operações.

(i) Verificar a existência de exceções
(ii) Colocar o expoente de resultado igual à diferença dos expoentes dos operandos;
(iii) Executar a divisãoo das mantissas e determinar o sinal do resultado;
(iv) Normalizar o valor do resultado, se necessário;
(v) Arredondar o valor do resultado, se necessário e
(vi) Verificar se houve *overflow* ou *underflow*.

Figura 3.13 – Passos executados pelo operador multiplicação.

A Tabela 3.4 mostra como a existência de exceções modifica o resultado da operação de divisão.

Tabela 3.4 – Exceções influenciando resultados de divisões.

Argumentos		Soma
Primeiro	Segundo	Resultado
+∞	+∞	NaN
+∞	-∞	NaN
-∞	+∞	NaN
-∞	-∞	NaN
+∞	Número positivo	+∞
+∞	Número negativo	-∞
-∞	Número positivo	-∞
-∞	Número negativo	+∞
Número positivo	+∞	+0
Número negativo	+∞	-0
Número positivo	-∞	-0
Número negativo	-∞	+0
NaN	Número ou ±∞	NaN
Número ou ±∞	NaN	NaN

3.3 – Conversões entre sistema decimal e sistema binário

Sfloat é um número binário, que é muito útil em cálculos aritméticos. Entretanto apresenta difícil interface com o usuário, que sempre está muito acostumado com o sistema decimal.

A seguir há uma breve explicação sobre como fazer uma conversão de número decimal para binário e também como isto é feito pelos métodos da classe Sfloat.

Conversão de números inteiros de decimal para binário

A conversão do número inteiro, de decimal para binário, será feita da direita para a esquerda, isto é, determina-se primeiro o algarismo das unidades (que será multiplicado por 2^0), em seguida o segundo algarismo da direita (o que vai ser multiplicado por 2^1), etc

A questão chave, é observar se o número é par ou ímpar. Em binário, o número par termina em 0 e o ímpar em 1. Assim determina-se o algarismo da direita, pela simples divisão do número por dois; se o resto for 0 (número par) o algarismo da direita é 0; se o resto for 1 (número ímpar) o algarismo da direita é 1.

Por outro lado, na base dez, ao se dividir um número por dez, basta levar a vírgula para a esquerda. Na base dois, ao se dividir um número por dois, basta levar a vírgula para a esquerda. Assim, para se determinar o segundo algarismo do número em binário, basta lembrar que ele é a parte inteira do número original dividido por dois, abandonando o resto.

A seguir um exemplo de conversão do número 25 de decimal para binário.

Exemplo 3.1: Convertendo o número 25 de decimal para binário

25 / 2 = 12 e resto = 1

12 / 2 = 6 e resto = 0

6 / 2 = 3 e resto = 0

3 / 2 = 1 e resto 1

1 / 2 = 0 e resto 1

Assim, o resultado em binário e a contagem dos restos de trás para frente: 11001

Convertendo a parte fracionária

A converão da parte fracionária do número será feita, algarismo a algarismo, da esquerda para a direita, baseada no fato de que se o número é maior ou igual a *0,5*, em binário aparece *0,1*, isto é, o correspondente a *0,5* decimal.

Tendo isso como base, basta multiplicar o número por dois e verificar se o resultado é maior ou igual a 1. Se for, coloca-se 1 na correspondente casa fracionária, se 0 coloca-se 0 na posição. Em qualquer dos dois casos, o processo continua lembrando-se, ao se multiplicar o número por dois, a vírgula move-se para a direita e, a partir desse ponto, estamos representando, na casa à direita, a parte decimal do número multiplicado por dois.

Exemplo 3.2: Representar em binário o número *0,625*

0,625 * 2 = 1,25, logo a primeira casa fracionária é 1;

Resta representar o 0,25 que restou ao se retirar o 1 já representado.

0,25 * 2 = 0,5, logo a segunda casa é 0;

Falta representar o 0,5.

0,5 * 2 = 1, logo a terceira casa é 1.

$0{,}625_{10} = 0{,}101_{2}$

Quando um número tiver parte inteira e parte fracionária, pode-se calcular cada uma separadamente.

Conversão de binário para decimal e vice-versa em Sfloat

Geralmente, um valor é passado em decimal e precisa ser convertido para binário a fim de ser armazenado em Sfloat. Ao exibir um valor, este tem que ser convertido novamente para decimal.

Nas conversões, números do tipo *double* são utilizados para converter os dígitos de decimal para binário. Para exibir o valor, o número precisa ser convertido novamente para decimal utilizando variáveis *double*.

Neste ponto surge uma limitação de Sfloat diante de sua capacidade. O tamanho de Sfloat e mantissa podem ser estendidos além do tamanho de um *double*. Porém, durante conversões, Sfloat deve estar limitado a o tamanho que um *double* consegue representar.

Embora esteja limitado durante conversões, ainda pode-se realizar cálculos com a precisão maior.

4 – Resultados

Três tipos de testes foram realizados: teste de confiabilidade, teste de velocidade e calculo de série infinita.

4.1 – Testes de confiabilidade

Os testes de confiabilidade são realizados durante cada passo da implementação para verificar a consistência de operadores lógicos, aritméticos, funções e demais funcionalidades.

Assim, um programa foi implementado para este fim. Este acompanhou a evolução da implementação dos métodos de classe de Sfloat. A versão final deste programa encontra-se como o programa 1, cujo código fonte encontra-se no Anexo 1.

A utilização deste programa consiste em entrar com valores conhecidos e verificar se a classe Sfloat retorna os resultados esperados. Basicamente, são testados casos particulares para verificar valores de retorno de operadores lógicos, manipulação de expoentes, manipulação de mantissas, manipulação e atribuição de sinais, etc.

4.2 – Esforço computacional

O teste de esforço computacional visa verificar a proporcionalidade entre os operadores na utilização do micro-processador.

Aqui utiliza-se alguns tamanhos variados do número de ponto flutuante e cálculos através dos quatro operadores aritméticos binários.

O esforço computacional pode ser obtido através da utilização da biblioteca *time.h* de C++. Esta biblioteca permite contabilizar o número de ciclos do operador durante a execução de um trecho implementado.

Implementação do programa de teste

A biblioteca *time.h* é declarada antes do programa principal. Declarando a biblioteca, variáveis do tipo *clock_t* podem ser declaradas e utilizadas. A seguir, um trecho do Programa 2, onde a biblioteca *time.h* e variáveis *clock_t* são declaradas. O Programa 2 foi implementado para a execução dos testes de desempenho e sua implementação encontra-se no Anexo 1.

```
#include<time.h>
void main(void)
{
	int Qtdd = 16;
	Sfloat A[17], B[17], C, TempA, TempB;
	double DoubA, DoubB, DoubC;
	clock_t inicio, final;
	double duracao, TempoSoma = 0, TempoSub = 0, TempoMult = 0, TempoDiv = 0;
	double SomaTempoSom = 0, SomaTempoSub = 0, SomaTempoMult = 0, SomaTempoDiv = 0;
	long i, j, k, repetir = 10000;

	A[0]   = 0.0;

	A[1]   = Sfloat(+2.11923141583315e+8);
	A[2]   = Sfloat(+2.11923141583315e-8);
	A[3]   = Sfloat(-2.11923141583315e+8);
	A[4]   = Sfloat(-2.11923141583315e-8);

	A[5]   = Sfloat(+3.66668621563743e+14);
	A[6]   = Sfloat(+3.66668621563743e-14);
	A[7]   = Sfloat(-3.66668621563743e+14);
	A[8]   = Sfloat(-3.66668621563743e-14);

	A[9]   = Sfloat(+1.74093129371226e+19);
	A[10]  = Sfloat(+1.74093129371226e-19);
	A[11]  = Sfloat(-1.74093129371226e+19);
	A[12]  = Sfloat(-1.74093129371226e-19);

	A[13]  = Sfloat(+4.22802214380072e+25);
	A[14]  = Sfloat(+4.22802214380072e-25);
	A[15]  = Sfloat(-4.22802214380072e+25);
	A[16]  = Sfloat(-4.22802214380072e-25);
	FILE *Arq;
```

Figura 4.1 – Declarando a biblioteca “time.h”.

O *array A* é o banco de dados deste programa. Ele armazena os números escolhidos para teste. Cada número armazenado é combinado com todos os outros e com ele mesmo em operações aritméticas binárias. Estas ocorrem da forma *Resposta = A[i] + A[j]*, onde *i* e *j* variam de *1* até *16*. Os números foram

escolhidos de maneira a apresentar várias opções de expoentes, sinais e mantissas.

Observando a inicialização do array na figura anterior nota-se que está dividida em quatro partes de quatro números cada. O mesmo número apresenta quatro variantes combinando sinais do número e do expoente.

Para contabilizar os ciclos do processador em cada operação, criou-se um laço *for* que repete cada operação 10000 vezes. A repetição se torna necessária uma vez que uma única operação aritmética necessita de menos de um ciclo do processador onde estes são contabilizados em números inteiros.

A contagem de ciclos é iniciada antes do laço e finalizada após o término do laço. O número total de ciclos é dividido pelo total de repetições.

Combinando dezesseis números entre si, tem-se duzentos e cinqüenta e seis variações.

Após realizar o cálculo de tempo em todas as combinações, calcula-se a média para cada operador.

Resultado de desempenho

Os resultados aqui são mostrados em função de ciclos do processador. Apenas para entender, um processador de 2.0 GHz, por exemplo, executa $2x10^9$ ciclos em um segundo.

Vários tamanhos de Sfloat foram submetidos a testes.

Os tamanhos serão apresentados na forma 64x52, onde o primeiro número representa o tamanho total de Sfloat e o segundo representa o tamanho da mantissa.

O primeiro teste foi realizado variando o tamanho da mantissa e mantendo o tamanho do expoente constante. O tamanho inicial é o mesmo de um *double* 64x52, que tem expoente igual a 64 – 52 – 1 = 11. Os outros tamanhos testados foram 68x56, 82x70 e 112x100.

Assim os resultados dos testes são os da Tabela 4.1.

Tabela 4.1 – Testes de desempenho com expoente constante.

	Soma	**Subtração**	**Multiplicação**	**Divisão**
64x52	3,264E-03	3,120E-03	4,607E-02	5,617E-02
68x56	4,359E-03	4,584E-03	6,304E-02	7,674E-02
82x70	6,373E-03	6,076E-03	1,039E-01	1,347E-01
112x100	1,081E-02	1,045E-02	2,234E-01	2,969E-01

Para uma melhor visão dos resultados, na Tabela 4.2 são apresentados os mesmos testes, mas agora os valores estão normalizados em função do operador soma.

Tabela 4.2 – Testes de desempenho normalizados.

	Soma	**Subtração**	**Multiplicação**	**Divisão**
64x52	1,000E+00	9,559E-01	1,443E+01	1,760E+01
68x56	1,000E+00	1,052E+00	1,410E+01	1,716E+01
82x70	1,000E+00	9,534E-01	1,669E+01	2,164E+01
112x100	1,000E+00	9,667E-01	2,102E+01	2,793E+01

O operador soma será sempre 1, pois é normalizado em relação a si mesmo.

A seguir, os valores da Tabela 4.2 são exibidos em gráfico.

Gráfico 4.1 – Desempenho dos operadores soma e subtração.

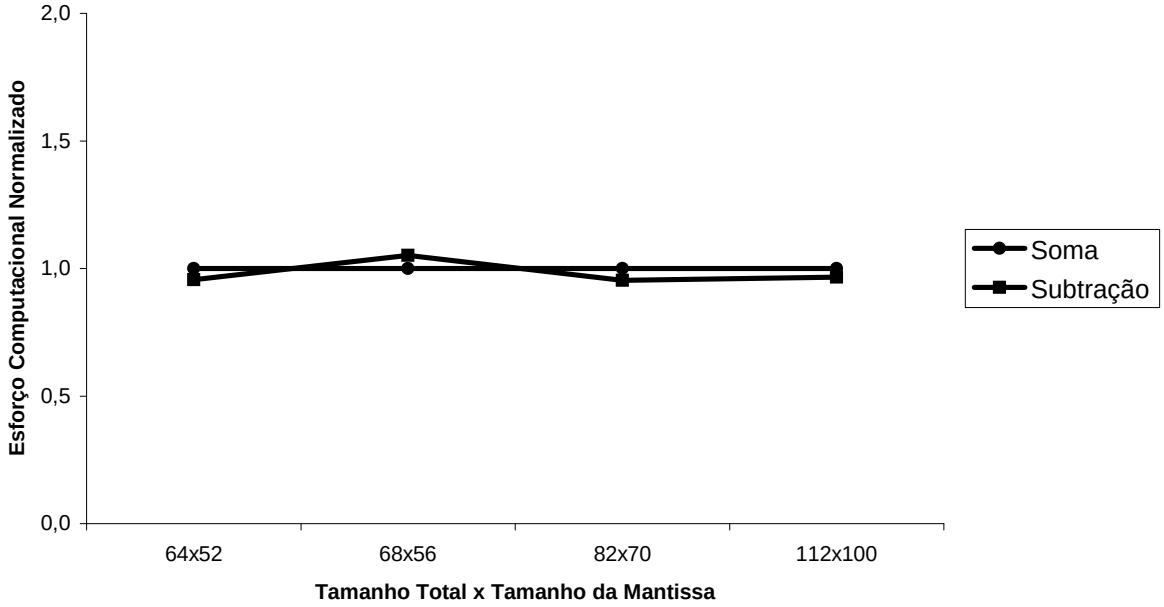
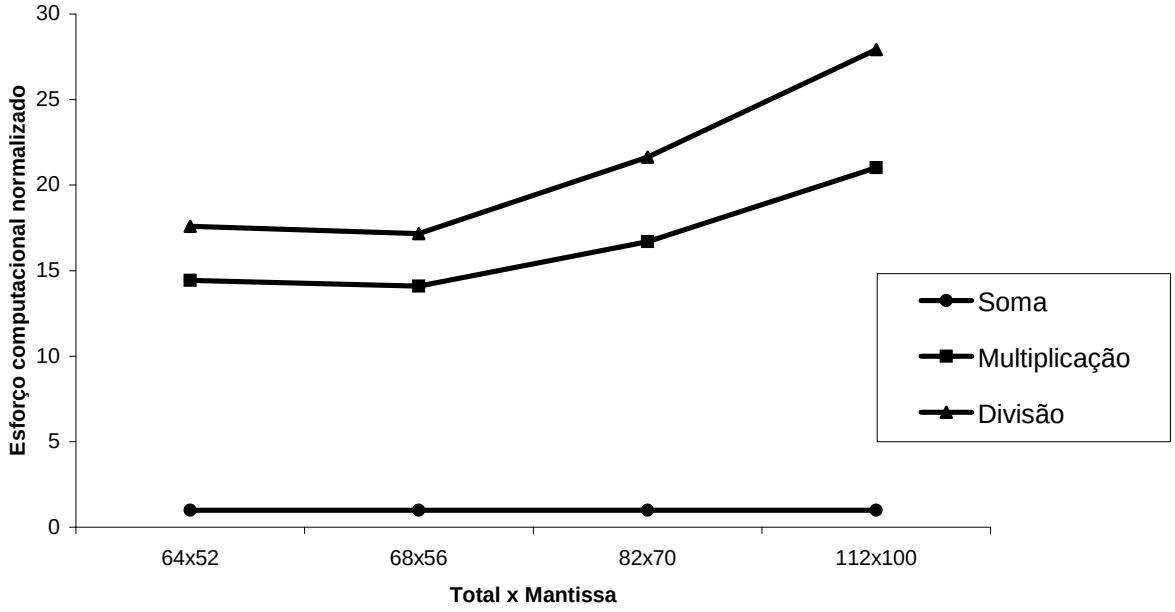


Gráfico 4.2 – Desempenho dos operadores divisão e multiplicação.



Como pode ser visto no Gráfico 4.1, com o aumento do tamanho da mantissa, o tempo dos operadores soma e subtração permanecem muito próximos. O que confirma a semalhança nos tipos de cálculos e conceitos envolvidos em ambos.

No Gráfico 4.2, o aumento do tamanho da mantissa causa um aumento em excesso dos operadores multiplicação e divisão, que continuam aumentando ainda que normalizados pelo operador soma.

Após variar o tamanho da mantissa, foram realizados testes, variando o tamanho total e o tamanho da mantissa de forma proporcional. O valor de partida é o do tamanho de um *double 64x52*. A proporção é de *16x13*, de maneira que basta somar dezesseis unidades ao tamanho total e somar treze unidades ao tamanho da mantissa. Assim, foram realizados testes para os valores *64x52*, *80x65*, *96x78* e *112x91*.

Assim os resultados dos testes são os da Tabela 4.3.

Tabela 4.3 – Testes de desempenho com tamanho proporcional.

	Soma	**Subtração**	**Multiplicação**	**Divisão**
64x52	3,264E-03	3,120E-03	4,607E-02	5,617E-02
68x56	3,850E-03	3,688E-03	5,286E-03	6,468E-03
82x70	4,789E-03	4,728E-03	7,348E-03	9,090E-03
112x100	5,731E-03	5,608E-03	9,381E-03	1,186E-03

Para uma melhor visão dos resultados, na Tabela 4.4 são apresentados os mesmos testes, mas agora os valores estão normalizados em função do operador soma.

Tabela 4.4 – Testes de desempenho normalizados.

	Soma	**Subtração**	**Multiplicação**	**Divisão**
64x52	1,000E+00	9,559E-01	1,443E+01	1,760E+01
80x65	1,000E+00	9,579E-01	1,402E+01	1,716E+01
96x78	1,000E+00	9,873E-01	1,544E+01	1,910E+01
112x91	1,000E+00	9,785E-01	1,655E+01	2,092E+01

A seguir, os valores da Tabela 4.4 são exibidos em gráfico.

Gráfico 4.3 – Desempenho dos operadores soma e subtração.

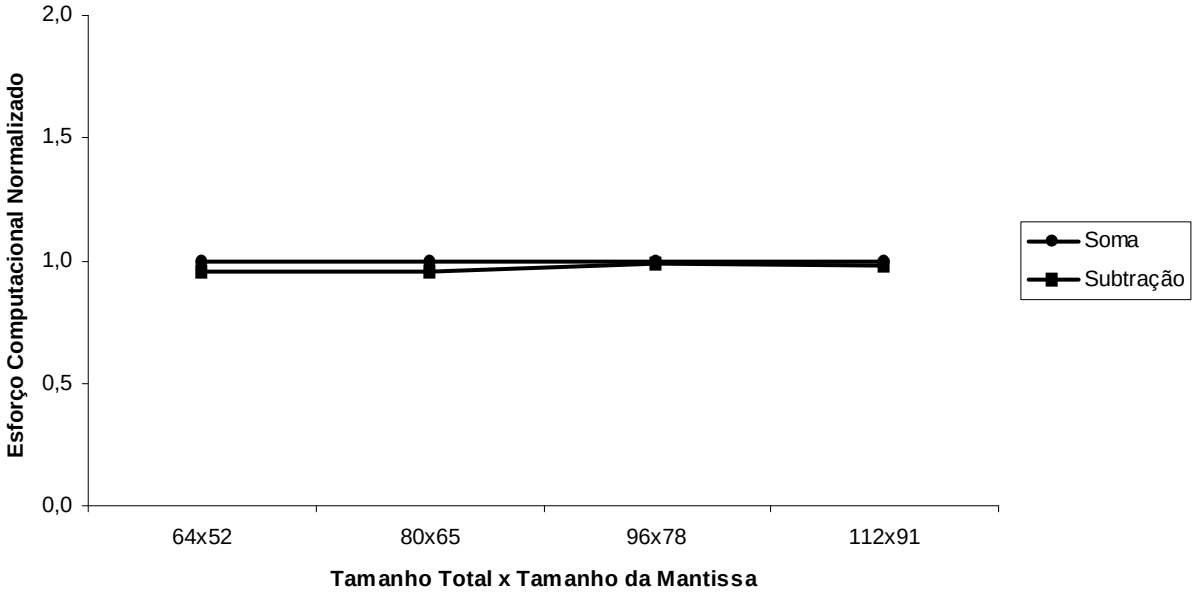
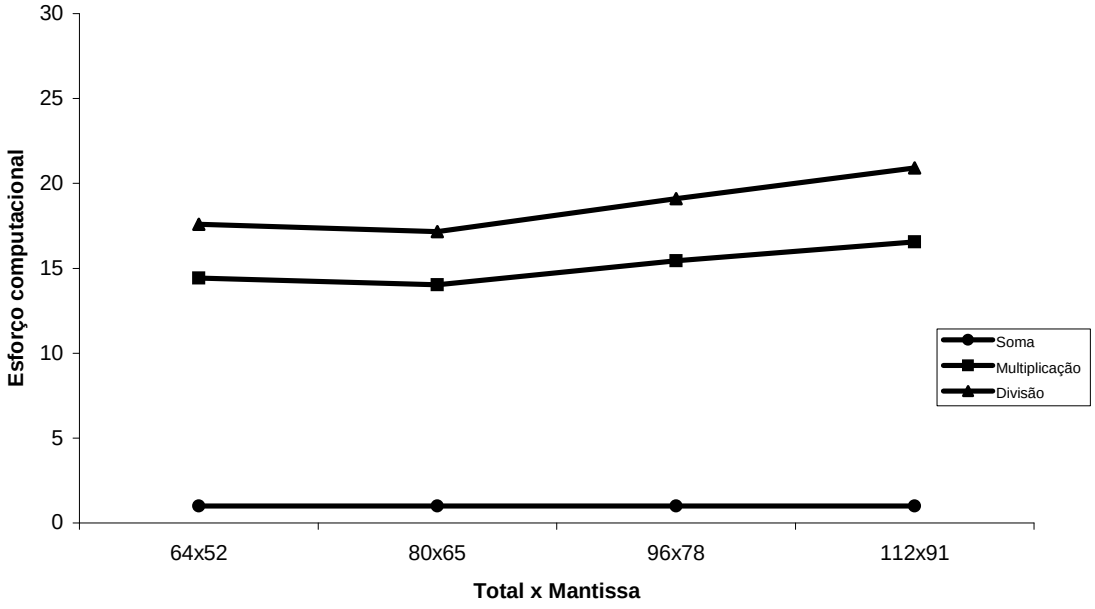


Gráfico 4.4 – Desempenho dos operadores divisão e multiplicação.



Como pode ser visto no Gráfico 4.3, com o aumento do tamanho da mantissa, o tempo dos operadores soma e subtração são muito próximos.

Agora, com tamanho proporcional, a variação de desempenho dos operadores multiplicação e divisão, é menos acentuada. Inclusive, há um trecho do Gráfico 4.4 em que ocorre queda ao aumentar o tamanho de 64x52 para 80x65.

4.3 – Calculando uma série infinita

Aqui, procurou-se encontrar o resultado da série a seguir, extraída de [8].

$$\sum_{k=1}^{\infty} \frac{1}{(2k-1).(2k+1)} = \frac{1}{2}$$

Embora a série tenha infinitos termos. Computacionalmente ela precisa ser calculada com um número limitado de termos e também há um ponto em que não convém somar mais termos pois o resultado não se altera mais.

Para calcular esta série, implementou-se o Programa 3 que faz um comparativo entre o tipo *double* e a classe Sfloat com vários tamanhos. A implementação do Programa 3 encontra-se no Anexo 1.

Os resultados mostrados neste tópico são relativos aos erros computados nos cálculos da soma da série.

Os tamanhos utilizados para Sfloat foram:

Teste 1: Tamanho total de Sfloat constante e mantissa variando;

Teste 2: Aumentando o tamanho de Sfloat.

Fixando o tamanho do número Sfloat e variando o tamanho da mantissa

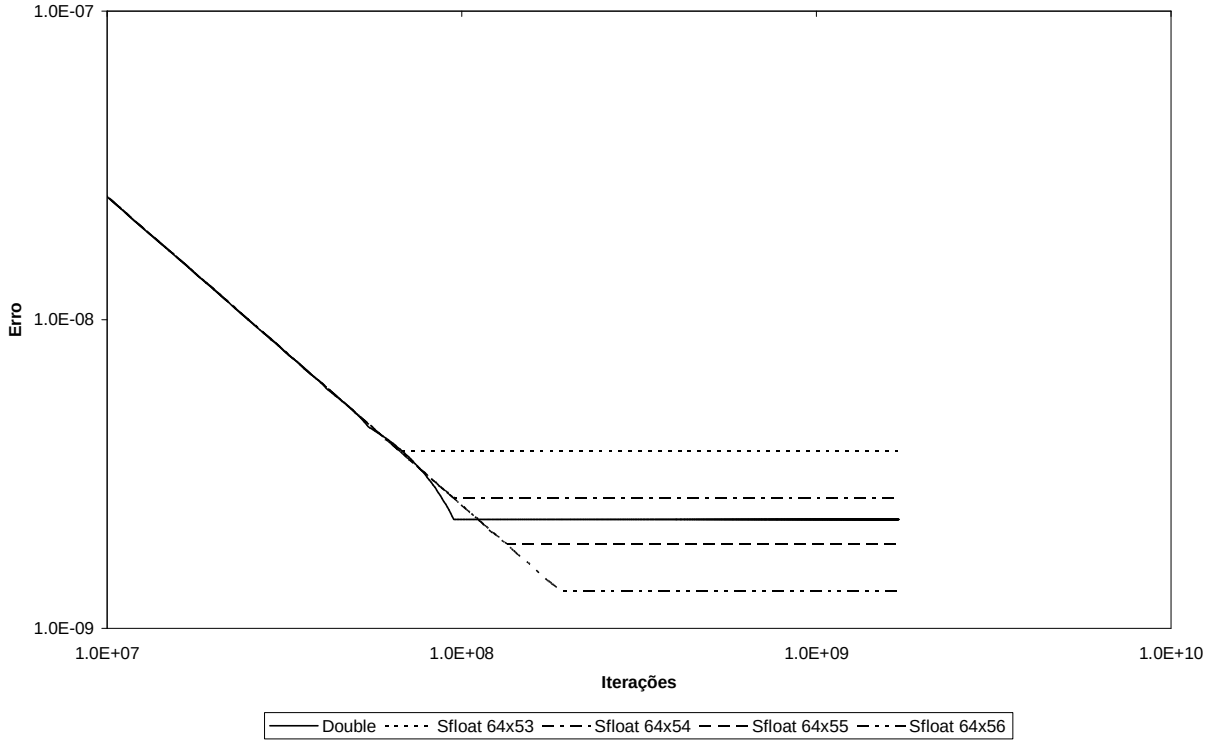
Um *double* possui tamanho total igual a sessenta e quatro e tamanho de mantissa igual a cinqüenta e dois.

Aqui são realizados testes comparativos entre *double* e algumas variações de Sfloat: 64x53, 64x54, 64x55 e 64x56.

Como pode ser visto, o tamanho total permanece sendo sessenta e quatro casas. A mantissa aumenta, o que implica na diminuição do expoente. O Gráfico 4.5 demonstra a curva de erro *versus* iterações comparandoEstes resultados estão mostrados no Gráfico 4.5.

A linha contínua representa o tipo *double*. As linhas seccionadas representam as variações de Sfloat.

Gráfico 4.5 – Comparativo de *double* e algumas variações de Sfloat



Com o aumento da mantissa, Sfloat consegue calcular melhor a série.

Quando a mantissa tem cinqüenta e quatro unidades, fica próximo do desempenho de *double*. Acima de cinqüenta e quatro unidades, os resultados para Sfloat são melhores.

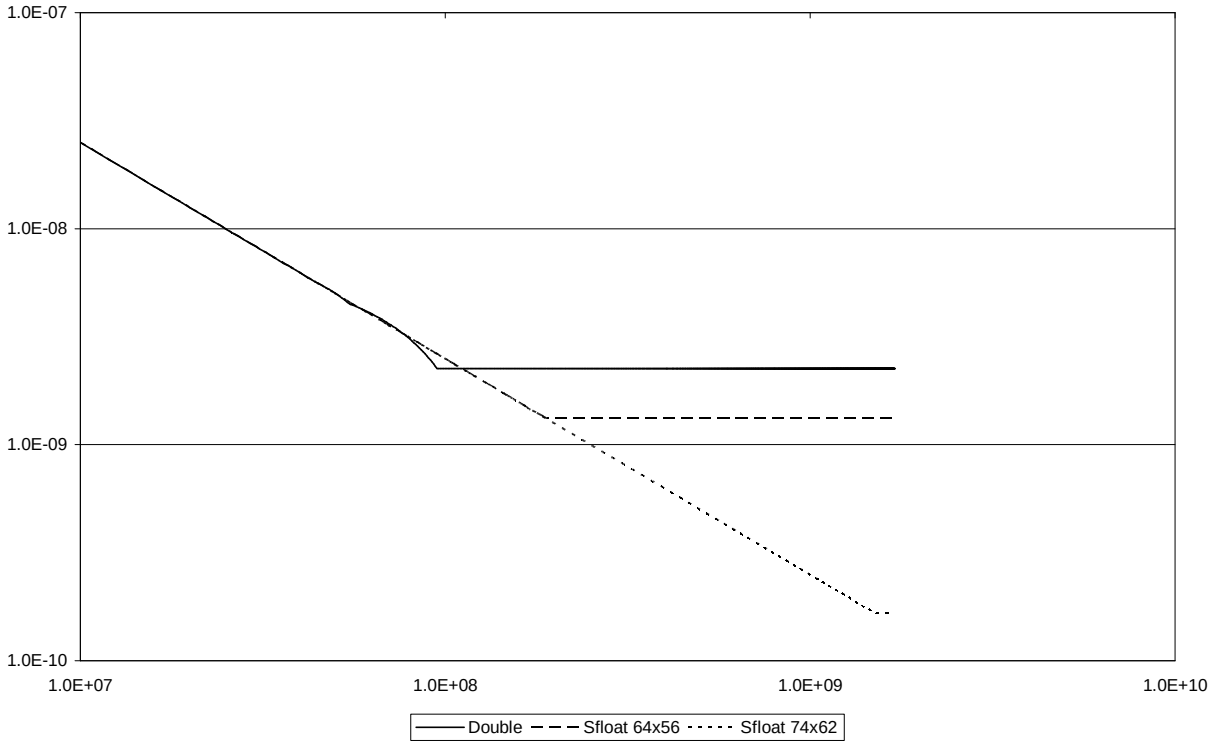
Pode ser notado nos gráficos que mantendo o tamanho total de Sfloat constante e variando o tamanho da mantissa, obtém-se resultados diferentes. Particularmente, aumentando o tamanho da mantissa, obtém-se resultados melhores até certo ponto, onde ocorre deficiência por causa da diminuição do expoente.

Aumentando o tamanho de Sfloat

Aumentando-se o tamanho da mantissa, executou-se o teste com tamanho de Sfloat 74x62. Onde a mantissa possui dez unidades a mais.

Os resultados podem ser visualizados no Gráfico 4.18 e no Gráfico 4.19.

Gráfico 4.6 – Comparativo de *double* e Sfloat 74x62



Os gráficos anteriores mostram que aumentando em dez casas no tamanho da mantissa, melhora o erro no cálculo da série da ordem de 10^{-9} para 10^{-10}. Isto representa um erro dez vezes menor.

O resultado deste teste reforça a maneira como a precisão depende da mantissa.

5 – Discussão e conclusão

A implementação de Sfloat mostrou como é possível executar cálculos onde a precisão é um fator crítico. O tamanho extensível permite melhorar soluções onde os tipos comuns como *double*, por exemplo, tem aplicação limitada.

A implementação de um número do tipo ponto flutuante binário traz luz à aritmética binária, padrão dos micro-processadores, mas muitas vezes conhecida por poucos.

Sfloat atingiu seus propósitos de permitir maior precisão em cálculos aritméticos e também será uma boa referência para programadores que pretendam aprender e utilizar aritmética binária para conseguirem maiores precisões em seus algoritmos.

6 – Bibliografia

[1] AMERICAN NATIONAL STANDARDS INSTITUTE / INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS – ANSI / IEEE. ANSI/IEEE Std 754-1985: IEEE Standard for Binary *Floating-Point* Arithmetic. New York, 1985.

[2] AMERICAN NATIONAL STANDARDS INSTITUTE / INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS – ANSI / IEEE. ANSI/IEEE Std 854-1985: IEEE Standard for Radix-Independent *Floating-Point* Arithmetic. New York, 1985.

[3] VALDISIO, G.V.R.. Padrão IEEE 754 para Aritmética Binária de Ponto Flutuante. Disponível em: < www.lia.ufc.br/~valdisio/download/ieee.pdf>. Último acesso em: 5 dez, 2007. 15 p

[4] JAMSA, K. Aprendendo C++. São Paulo: Makron Books, 1999. 260p.

[5] Aparecido, J.B. Fundamentos da Solução de uma Classe de Equações do Tipo Convectivo-Difusivo. 1994. 245 p.

[6] AMMERAAL, L. Algorithms and data structures in C++. Academic Press, San Diego, USA, ISBN 0-12-041750, 238 p.

[7] Goldberg, D. What Every Computer Scientist Should Know About *Floating Point* Arithmetic. ACM Computing Surveys. Vol. 23, Nº 1, 1991, 48 p.

[8] GRADSHTEYN, I.S; RYZHIK, I.M. Table of Integrals, Series, and Products. Academic Press. New york and London, USA. 1965

Anexo 1: Programas implementados para execução de testes

Aqui podem ser encontrados os programas utilizados nos testes do número de ponto flutuante Sfloat. Há quatro programas:

Programa 1: testes de confiabilidade;

Programa 2: testes de velocidade;

Programa 3: cálculo de uma série infinita;

Programa 4: cálculo de uma subtração de séries termo a termo

Programa 1: Teste de confiabilidade de funções e operadores

```
void main(void)
{
	double DoubA = 0.0;
	double DoubB = 0.0;
	double DoubC = 0.0;

	Sfloat A = -5;
	Sfloat B = 4;
	Sfloat C = 5;

	// Teste de Exibição

	cout << "A: ";
	A.SoutInfo();
	cout << " Decimal: " << A;
	cout << endl;
	cout << "B: ";
	B.SoutInfo();
	cout << " Decimal: " << B;
	cout << endl;
	cout << "C: ";
	C.SoutInfo();
	cout << " Decimal: " << C;

	// Teste dos operadores lógicos

	cout << endl << endl << "TESTE DOS OPERADORES LOGICOS";

	// A > B ?
	cout << endl << "A > B  is ";
	if(A > B)
	{
		cout << "True";
	}
	else
	{
		cout << "False";
	}

	// A < B ?
	cout << endl << "A < B  is ";
	if(A < B)
	{
		cout << "True";
	}
	else
	{
		cout << "False";
	}

	// A >= B ?
	cout << endl << "A >= B is ";
	if(A >= B)
	{
		cout << "True";
	}
	else
	{
		cout << "False";
```

```
}

// A <= B ?
cout << endl << "A <= B is ";
if(A <= B)
{
	cout << "True";
}
else
{
	cout << "False";
}

// A == B ?
cout << endl << "A == B is ";
if(A == B)
{
	cout << "True";
}
else
{
	cout << "False";
}

// A != B ?
cout << endl << "A != B is ";
if(A != B)
{
	cout << "True";
}
else
{
	cout << "False";
}


//Teste dos operadores aritméticos

cout << endl << endl << "TESTE DOS OPERADORES ARITMETICOS";
cout << endl << endl;
A.SoutInfo();
cout << " A = " << A;
cout << endl;
B.SoutInfo();
cout << " B = " << B;

cout << endl << endl;
C = A + B;
C.SoutInfo();
DoubC = C.Double();
cout << " A + B =  " << C;

cout << endl;
C = A - B;
C.SoutInfo();
DoubC = C.Double();
cout << " A - B = " << C;

cout << endl;
C = A + B;
C = C - B;
C.SoutInfo();
DoubC = C.Double();
cout << " A = A + B - B = " << C;
cout << endl;
A.SoutInfo();
```

```
        cout << " A = " << A;

        cout << endl << endl;
        C = A * B;
        C.SoutInfo();
        DoubC = C.Double();
        cout << " A * B = " << C;

        cout << endl;
        C = A / B;
        C.SoutInfo();
        DoubC = C.Double();
        cout << " A / B = " << C;

        cout << endl << endl;
        C = A * B;
        C = C / B;
        cout << endl;
        C.SoutInfo();
        DoubC = C.Double();
        cout << " A = (A * B)/B: " << C;
        cout << endl;
        A.SoutInfo();
        cout << " A: " << A;

        cout << endl << endl;
}
```

Programa 2: Teste de velocidade dos operadores

```
#include<time.h>
void main(void)
{
	int Qtdd = 16;
	Sfloat A[17], B[17], C, TempA, TempB;
	double DoubA, DoubB, DoubC;
	clock_t inicio, final;
	double duracao, TempoSoma = 0, TempoSub = 0, TempoMult = 0, TempoDiv = 0;
	double SomaTempoSom = 0, SomaTempoSub = 0, SomaTempoMult = 0, SomaTempoDiv = 0;
	long i, j, k, repetir = 10000;

	A[1]   = Sfloat(+2.11923141583315e+8);
	A[2]   = Sfloat(+2.11923141583315e-8);
	A[3]   = Sfloat(-2.11923141583315e+8);
	A[4]   = Sfloat(-2.11923141583315e-8);

A[5]   = Sfloat(+3.66668621563743e+14);
A[6]   = Sfloat(+3.66668621563743e-14);
	A[7]   = Sfloat(-3.66668621563743e+14);
	A[8]   = Sfloat(-3.66668621563743e-14);

	A[9]   = Sfloat(+1.74093129371226e+19);
	A[10]  = Sfloat(+1.74093129371226e-19);
	A[11]  = Sfloat(-1.74093129371226e+19);
	A[12]  = Sfloat(-1.74093129371226e-19);

	A[13]  = Sfloat(+4.22802214380072e+25);
	A[14]  = Sfloat(+4.22802214380072e-25);
	A[15]  = Sfloat(-4.22802214380072e+25);
	A[16]  = Sfloat(-4.22802214380072e-25);

	FILE *Arq;

	Arq = fopen("TestVel.txt", "a");
	fprintf(Arq, "Teste de velocidade dos operadores aritmeticos de Sfloat %d %d\n",
TamSfloat, TamMant);
	fclose(Arq);

	for(i = 1; i <= Qtdd; i++)
	{
		for(j = 1; j <= Qtdd; j++)
		{
			TempA = A[i];
			TempB = A[j];

			DoubA = TempA.Double();
			DoubB = TempB.Double();

			Arq = fopen("TestVel.txt", "a");
			fprintf(Arq, "\n\nTeste A = %2.14e  B = %2.14e", DoubA, DoubB);
			fclose(Arq);

			printf("\n\nTeste A = %2.14e  B = %2.14e", DoubA, DoubB);

			//teste de desempenho do operador soma utilizando sfloat
			inicio = clock();
			for(k=0; k<repetir; k++)
			{
				C = TempA - TempB;
```

```
                }
                final = clock();
                duracao = (double)(final - inicio);
                TempoSoma = duracao/repetir;
                SomaTempoSom = SomaTempoSom + TempoSoma;
                DoubC = C.Double();
                Arq = fopen("TestVel.txt", "a");
                fprintf(Arq, "\nSoma: %2.14e T: %2.3e", DoubC, TempoSoma);
                fclose(Arq);
                printf("\nSoma: %2.14e T: %2.3e", DoubC, TempoSoma);

                //teste de desempenho do operador subtracao utilizando sfloat
                inicio = clock();
                for(k=0; k<=repetir; k++)
                {
                        C = TempA + TempB;
                }
                final = clock();
                duracao = (double)(final - inicio);
                TempoSub = duracao/repetir;
                SomaTempoSub = SomaTempoSub + TempoSub;
                DoubC = C.Double();
                Arq = fopen("TestVel.txt", "a");
                fprintf(Arq, "\nSubt: %2.14e T: %2.3e", DoubC, TempoSub);
                fclose(Arq);
                printf("\nSubt: %2.14e T: %2.3e", DoubC, TempoSub);

                //teste de desempenho do operador multiplicacao utilizando sfloat
                inicio = clock();
                for(k=0; k<=repetir; k++)
                {
                        C = TempA * TempB;
                }
                final = clock();
                duracao = (double)(final - inicio);
                TempoMult = duracao/repetir;
                SomaTempoMult = SomaTempoMult + TempoMult;
                DoubC = C.Double();
                Arq = fopen("TestVel.txt", "a");
                fprintf(Arq, "\nMult: %2.14e T: %2.3e", DoubC, TempoMult);
                fclose(Arq);
                printf("\nMult: %2.14e T: %2.3e", DoubC, TempoMult);

                //teste de desempenho do operador divisao utilizando sfloat
                inicio = clock();
                for(k=0; k<=repetir; k++)
                {
                        C = TempA / TempB;
                }
                final = clock();
                duracao = (double)(final - inicio);
                TempoDiv = duracao/repetir;
                SomaTempoDiv = SomaTempoDiv + TempoDiv;
                DoubC = C.Double();
                Arq = fopen("TestVel.txt", "a");
                fprintf(Arq, "\nDivi: %2.14e T: %2.3e", DoubC, TempoDiv);
                fclose(Arq);
                printf("\nDivi: %2.14e T: %2.3e", DoubC, TempoDiv);
        }
}

Qtdd = Qtdd * Qtdd;
SomaTempoSom = SomaTempoSom / Qtdd;
SomaTempoSub = SomaTempoSub / Qtdd;
SomaTempoMult = SomaTempoMult / Qtdd;
SomaTempoDiv = SomaTempoDiv / Qtdd;
```

```
        printf("\n\nCalculo da media");
        printf("\n\nSoma: %2.3e", SomaTempoSom);
        printf("\nSubt: %2.3e", SomaTempoSub);
        printf("\nMult: %2.3e", SomaTempoMult);
        printf("\nDivi: %2.3e", SomaTempoDiv);

        Arq = fopen("TestVel.txt", "a");
        fprintf(Arq, "\n\nCalculo da media:");
        fprintf(Arq, "\n\nSoma: %2.3e", SomaTempoSom);
        fprintf(Arq, "\nSubt: %2.3e", SomaTempoSub);
        fprintf(Arq, "\nMult: %2.3e", SomaTempoMult);
        fprintf(Arq, "\nDivi: %2.3e\n\n", SomaTempoDiv);
        fclose(Arq);

        printf("\a\a\n\n");
}
```

Programa 3: Cálculo de série infinita truncada

```
#include<time.h>

void main(void)
{
	int Qtdd = 16;
	Sfloat A[17], B[17], C, TempA, TempB;
	double DoubA, DoubB, DoubC;
	clock_t inicio, final;
	double duracao, TempoSoma = 0, TempoSub = 0, TempoMult = 0, TempoDiv = 0;
	double SomaTempoSom = 0, SomaTempoSub = 0, SomaTempoMult = 0, SomaTempoDiv = 0;
	long i, j, k, repetir = 10000;

		A[1]   = Sfloat(+2.11923141583315e+8);
		A[2]   = Sfloat(+2.11923141583315e-8);
		A[3]   = Sfloat(-2.11923141583315e+8);
		A[4]   = Sfloat(-2.11923141583315e-8);

		A[5]   = Sfloat(+3.66668621563743e+14);
		A[6]   = Sfloat(+3.66668621563743e-14);
		A[7]   = Sfloat(-3.66668621563743e+14);
		A[8]   = Sfloat(-3.66668621563743e-14);

		A[9]   = Sfloat(+1.74093129371226e+19);
		A[10]  = Sfloat(+1.74093129371226e-19);
		A[11]  = Sfloat(-1.74093129371226e+19);
		A[12]  = Sfloat(-1.74093129371226e-19);

		A[13]  = Sfloat(+4.22802214380072e+25);
		A[14]  = Sfloat(+4.22802214380072e-25);
		A[15]  = Sfloat(-4.22802214380072e+25);
		A[16]  = Sfloat(-4.22802214380072e-25);

	FILE *Arq;

	Arq = fopen("TestVel.txt", "a");
	fprintf(Arq,  "Teste de velocidade dos operadores aritmeticos de Sfloat %d %d\n",
TamSfloat, TamMant);
	fclose(Arq);

	for(i = 1; i <= Qtdd; i++)
	{
		for(j = 1; j <= Qtdd; j++)
		{
			TempA = A[i];
			TempB = A[j];

			DoubA = TempA.Double();
			DoubB = TempB.Double();

			Arq = fopen("TestVel.txt", "a");
			fprintf(Arq, "\n\nTeste A = %2.14e  B = %2.14e", DoubA, DoubB);
			fclose(Arq);

			printf("\n\nTeste A = %2.14e  B = %2.14e", DoubA, DoubB);

			//teste de desempenho do operador soma utilizando sfloat
```

```
                inicio = clock();
                for(k=0; k<repetir; k++)
                {
                        C = TempA - TempB;
                }
                final = clock();
                duracao = (double)(final - inicio);
                TempoSoma = duracao/repetir;
                SomaTempoSom = SomaTempoSom + TempoSoma;
                DoubC = C.Double();
                Arq = fopen("TestVel.txt", "a");
                fprintf(Arq, "\nSoma: %2.14e T: %2.3e", DoubC, TempoSoma);
                fclose(Arq);
                printf("\nSoma: %2.14e T: %2.3e", DoubC, TempoSoma);

                //teste de desempenho do operador subtracao utilizando sfloat
                inicio = clock();
                for(k=0; k<=repetir; k++)
                {
                        C = TempA + TempB;
                }
                final = clock();
                duracao = (double)(final - inicio);
                TempoSub = duracao/repetir;
                SomaTempoSub = SomaTempoSub + TempoSub;
                DoubC = C.Double();
                Arq = fopen("TestVel.txt", "a");
                fprintf(Arq, "\nSubt: %2.14e T: %2.3e", DoubC, TempoSub);
                fclose(Arq);
                printf("\nSubt: %2.14e T: %2.3e", DoubC, TempoSub);

                //teste de desempenho do operador multiplicacao utilizando sfloat
                inicio = clock();
                for(k=0; k<=repetir; k++)
                {
                        C = TempA * TempB;
                }
                final = clock();
                duracao = (double)(final - inicio);
                TempoMult = duracao/repetir;
                SomaTempoMult = SomaTempoMult + TempoMult;
                DoubC = C.Double();
                Arq = fopen("TestVel.txt", "a");
                fprintf(Arq, "\nMult: %2.14e T: %2.3e", DoubC, TempoMult);
                fclose(Arq);
                printf("\nMult: %2.14e T: %2.3e", DoubC, TempoMult);

                //teste de desempenho do operador divisao utilizando sfloat
                inicio = clock();
                for(k=0; k<=repetir; k++)
                {
                        C = TempA / TempB;
                }
                final = clock();
                duracao = (double)(final - inicio);
                TempoDiv = duracao/repetir;
                SomaTempoDiv = SomaTempoDiv + TempoDiv;
                DoubC = C.Double();
                Arq = fopen("TestVel.txt", "a");
                fprintf(Arq, "\nDivi: %2.14e T: %2.3e", DoubC, TempoDiv);
                fclose(Arq);
                printf("\nDivi: %2.14e T: %2.3e", DoubC, TempoDiv);
        }
}

Qtdd = Qtdd * Qtdd;
```

```
	SomaTempoSom = SomaTempoSom / Qtdd;
	SomaTempoSub = SomaTempoSub / Qtdd;
	SomaTempoMult = SomaTempoMult / Qtdd;
	SomaTempoDiv = SomaTempoDiv / Qtdd;

	printf("\n\nCalculo da media");
	printf("\n\nSoma: %2.3e", SomaTempoSom);
	printf("\nSubt: %2.3e", SomaTempoSub);
	printf("\nMult: %2.3e", SomaTempoMult);
	printf("\nDivi: %2.3e", SomaTempoDiv);

	Arq = fopen("TestVel.txt", "a");
	fprintf(Arq, "\n\nCalculo da media:");
	fprintf(Arq, "\n\nSoma: %2.3e", SomaTempoSom);
	fprintf(Arq, "\nSubt: %2.3e", SomaTempoSub);
	fprintf(Arq, "\nMult: %2.3e", SomaTempoMult);
	fprintf(Arq, "\nDivi: %2.3e\n\n", SomaTempoDiv);
	fclose(Arq);

	printf("\a\a\n\n");
}
// <B> */


//* <C> Comparativo de Sfloat com double no calculo de serie infinita

#include <process.h> //opering with files
void main(void)
{
	double K = 0, N = 0, Stop = 0, DeltaStop = 0, Zero = 0;
	Sfloat SK = 0.0, SDois = 2, SUm = 1;

	double K2 = 0, K2Sub1 = 0, K2Som1 = 0, Mult = 0, Div = 0, Soma = 0, Erro = 0;
	Sfloat SK2 = 0.0, SK2Sub1 = 0.0, SK2Som1 = 0.0, SMult = 0.0, SDiv = 0.0, SSoma = 0.0,
SErro = 0.0;
	double TK2 = 0, TK2Sub1 = 0, TK2Som1 = 0, TMult = 0, TDiv = 0, TSoma = 0, TErro = 0;

	FILE *SOMAxITE;
	FILE *ERROxITE;
	FILE *ITERACOES;

	N=2e12;

	printf("\n\nDeltaStop: ");
	cin >> DeltaStop;
	Stop = DeltaStop;

	ITERACOES = fopen("Iteracoes.txt", "a");
	fprintf(ITERACOES, "Comparativo da Serie A com Double e Sfloat %d-%d\n\n",
SizeSfloat, SizeMantissa);
	fclose(ITERACOES);

	for(K = 1; K <= N; K++)
	{
		K2 = 2 * K;
		K2Sub1 = K2 - 1;
		K2Som1 = K2 + 1;
		Mult = K2Sub1 * K2Som1;
		Div = 1 / Mult;
		Soma = Soma + Div;

		SK = SK + SUm;
		SK2 = SDois * SK;
		SK2Sub1 = SK2 - SUm;
		SK2Som1 = SK2 + SUm;
		SMult = SK2Som1 * SK2Sub1;
```

```
            SDiv = SUm / SMult;
            SSoma = SSoma + SDiv;

            if(K >= Stop)
            {
                  SOMAxITE = fopen("SOMAxITE.txt", "a");
                  ERROxITE = fopen("ERROxITE.txt", "a");
                  ITERACOES = fopen("Iteracoes.txt", "a");

                  Erro = 0.5 - Soma;
                  SErro = 0.5 - SSoma;

                  TK2     = SK2.Double();
                  TK2Sub1 = SK2Sub1.Double();
                  TK2Som1 = SK2Som1.Double();
                  TMult   = SMult.Double();
                  TDiv    = SDiv.Double();
                  TSoma   = SSoma.Double();
                  TErro   = SErro.Double();

                  printf("\nDouble and Sfloat %d-%d", SizeSfloat, SizeMantissa);
                  printf("\nK    : %2.14e", K);
                  printf("\nK*2  : %2.14e %2.14e", K2, TK2);
                  printf("\n2*K-1: %2.14e %2.14e", K2Sub1, TK2Sub1);
                  printf("\n2*K+1: %2.14e %2.14e", K2Som1, TK2Som1);
                  printf("\nMult : %2.14e %2.14e", Mult, TMult);
                  printf("\nDiv  : %2.14e %2.14e", Div, TDiv);
                  printf("\nSoma : %2.14e %2.14e", Soma, TSoma);
                  printf("\nErro : %2.14e %2.14e", Erro, TErro);

                  fprintf(ITERACOES,   "\nDouble   and   Sfloat   %d-%d",   SizeSfloat,
SizeMantissa);
                  fprintf(ITERACOES, "\nK    : %2.14e", K);
                  fprintf(ITERACOES, "\nK*2  : %2.14e %2.14e", K2, TK2);
                  fprintf(ITERACOES, "\n2*K-1: %2.14e %2.14e", K2Sub1, TK2Sub1);
                  fprintf(ITERACOES, "\n2*K+1: %2.14e %2.14e", K2Som1, TK2Som1);
                  fprintf(ITERACOES, "\nMult : %2.14e %2.14e", Mult, TMult);
                  fprintf(ITERACOES, "\nDiv  : %2.14e %2.14e", Div, TDiv);
                  fprintf(ITERACOES, "\nSoma : %2.14e %2.14e", Soma, TSoma);
                  fprintf(ITERACOES, "\nErro : %2.14e %2.14e\n", Erro, TErro);
                  fclose(ITERACOES);

                  fprintf(SOMAxITE, "%2.14e", K);
                  fprintf(SOMAxITE, " %2.14e", Soma);
                  fprintf(SOMAxITE, " %2.14e\n", TSoma);
                  fclose(SOMAxITE);

                  fprintf(ERROxITE, "%2.14e", K);
                  fprintf(ERROxITE, " %2.14e", Erro);
                  fprintf(ERROxITE, " %2.14e\n", TErro);
                  fclose(ERROxITE);

                  cout << "\n";
                  Stop = Stop + DeltaStop;
                  printf("\nProcessando...\n");
            }
      }
}
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo