É claro que meus filhos terão computadores, mas antes terão livros.

Bill Gates

1 Escrevendo as nossas funções - facilitando a nossa vida

A maioria dos jogos que vemos por ai são compostas de milhares de linhas de programação. Um programa em C++ é uma sequência de instruções. À medida em que um programa cresce, ele pode se tornar complexo e pouco legível. Além disso, certas sequências de comandos podem ser usadas com frequência em diversos pontos do programa, tendo de ser inteiramente reescritas em cada um desses pontos, o que é certamente uma fonte de erros.

As técnicas de programação dizem que, sempre que possível, evite códigos extensos, separando o mesmo em funções, visando um fácil entendimento e uma manutenção facilitada. De acordo com a técnica, devem-se agrupar códigos correlatos em uma função.

Para enfrentar essa situação podemos dividir nosso programa em módulos, separando logicamente as diferentes etapas do programa. A *programação modular* é uma técnica que tem por objetivo simplificar o desenvolvimento de programas através de sua divisão em *partes*. Cada uma dessas *partes* pode ser mais facilmente entendida, programadas, testada e modificada, pois o problema que está sendo resolvido é menor. Além disso, podemos agrupar trechos de código frequentemente usados em um módulo separado, que pode ser ativado a partir de diversos pontos do programa.

A linguagem C++ possibilita criar funções, sendo possível passar parâmetros para elas e retornar valores tanto no nome da função como em algum parâmetro passado.

Poder declarar instruções, estendendo a linguagem, certamente é o recurso mais valioso das linguagens de programação, por proporcionar reusabilidade e portanto produtividade. É conveniente também saber como armazená-las em bibliotecas, para que elas sejam facilmente reutilizáveis. Mas lembre-se: uma instrução só será reutilizável se puder ser usada sem qualquer outra declaração, isto é, é preciso saber apenas o seu nome e seus parâmetros. Também não será conveniente que a instrução mostre resultados na tela, a menos que isso faça parte da sua finalidade. Por exemplo, se a finalidade for *calcular a média*, a instrução retorna o valor calculado mas não mostra na tela. Se a finalidade for *Mostrar uma mensagem na tela*, então ela deve fazer isso e não efetuar cálculos de valores que não estejam relacionados a isso.

Funções são usadas para criar pequenos pedaços de códigos separados do programa principal. Em C++, tudo, na verdade, é uma função. *int main (void)* é uma função, por exemplo. Exceto a função *main*, todas as outras funções são secundárias, o que significa que elas podem existir ou não.

Uma função é um pedaço de código que faz alguma tarefa específica e pode ser chamado de qualquer parte do programa quantas vezes desejarmos. Utilizamos funções para obter:

Clareza do código : separando pedaços de código da função main(), podemos entender mais facilmente o que cada parte do código faz. Além disso, para procurarmos por uma certa ação feita pelo programa, basta buscar a função correspondente. Isso torna muito mais fácil o ato de procurar por erros.

Reutilização : muitas vezes queremos executar uma certa tarefa várias vezes ao longo do programa. Repetir todo o código para essa operação é muito trabalhoso, e torna mais difícil a manutenção do código: se acharmos um erro nesse código, teremos que corrigi-lo em todas as repetições do código. Chamar uma função diversas vezes contorna esses dois problemas.

Independência : uma função é relativamente independente do código que a chamou. Uma função pode modificar variáveis globais ou ponteiros, mas limitando-se aos dados fornecidos pela chamada de função.

2 Não conte para ninguém...

Mas já estamos utilizando funções a muito tempo. Funções da linguagem C++ e funções da nossa biblioteca. Por exemplo, para posicionarmos em algum ponto da tela:

```
1  gotoXY(10,20);
```

Ou quando estamos lendo informações do teclado:

```
1  nome = readString("Nome do jogador:");
```

Na prática, apenas temos que observar algumas coisas ao chamar um função. Observe a figura 1:

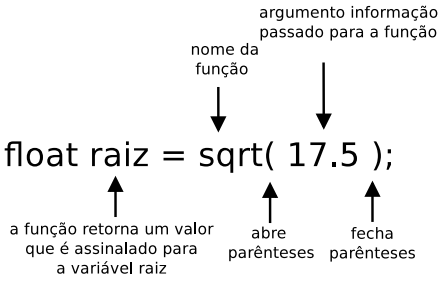

Figura 1: Chamada de uma função.

Podemos observar que toda função tem um nome seguido de um abre e fecha de parênteses *( )*, podendo haver informações entre os parênteses e sendo possível retornar um valor e atribuir para uma variável. Mas o que ocorre quando uma função é chamada? Observe a figura 2:

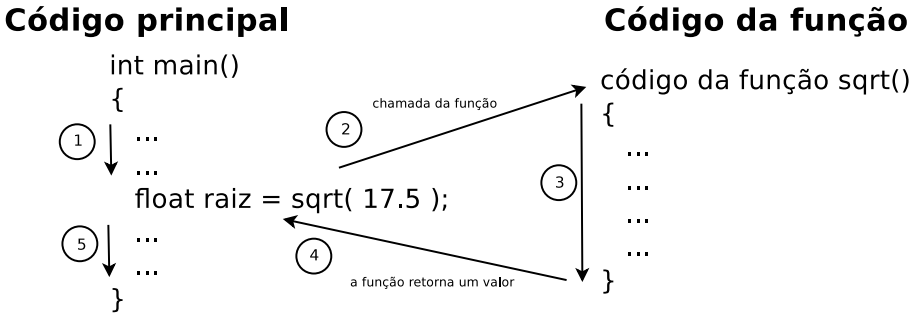

Figura 2: Chamando uma função.

A expressão *sqrt(17.5)* invoca, ou *chama*, a função *sqrt()*. A expressão *sqrt(17.5)* é conhecido como *chamada de função*. A função que realizou a chamada (nosso programa principal *main*) é conhecido como *chamadora da função* e o código da função é conhecido como *função chamada*.

O valor entre parênteses (no nosso exemplo 17.5) é a informação que será passada para a função chamada. Este valor é conhecido como *argumento* ou *parâmetro* de uma função. A função *sqrt()* irá calcular a raiz quadrada do valor informado. Este valor é conhecido como *retorno da função*. O retorno da função então é atribuído para a variável *raiz*.

3 Minha primeira função

Escrever funções não é complicado, basta entender algumas regras necessárias. Vejamos o programa 1.

Programa 1: Minha primeira função.

```
1  // Minha primeira função.
2  // programa_001.cpp
3  #include "biblaureano.h"
4
5  //protótipo da função
6  void fim();
7
8  int main()
9  {
10    cout << "Um pouco mais do mesmo!!" << endl;
11    fim(); //chamada da função
12
13    return 0;
14 }
15
16 void fim()
17 {
18   cout << "Game over!!" << endl;
19   return;
20 }
```

3.1 Entendendo o programa

O uso de funções na Linguagem C++ exige certas regras. Primeiramente a função deve estar definida, isto é, deve-se indicar para o compilador qual o nome da função e quais são os parâmetros esperados. Uma maneira simples de se resolver isto é a definição do protótipo da função no início do programa.

```
1    //protótipo da função
2    void fim();
```

No protótipo de uma função é definido somente o necessário para o compilador não acusar erros. No protótipo somente são informados o nome da função, o seu tipo de retorno e o tipo de cada parâmetro esperado.

Após a definição do protótipo, podemos trabalhar com o nosso programa normalmente.

```
1  int main()
2  {
3     cout << "Um pouco mais do mesmo!!" << endl;
4     fim(); //chamada da função
5     return 0;
6  }
```

E finalmente escrevemos o código da nossa função.

```
1  void fim()
2  {
```

```
3   cout << "Game over!!" << endl;
4   return;
5 }
```

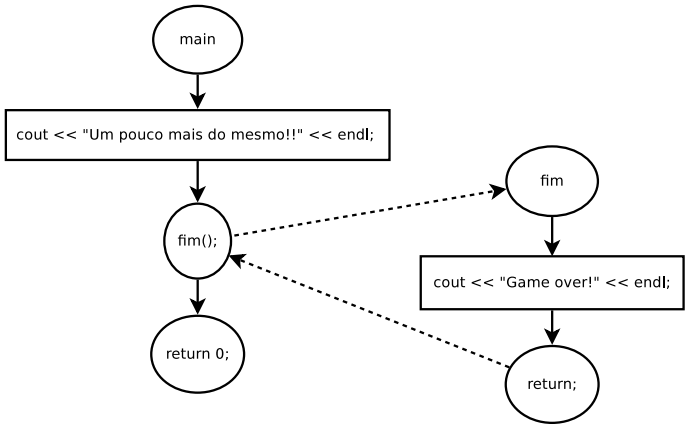

Figura 3: Fluxograma do programa 1.

3.2 Reutilizando a função *fim()*

Como já dito anteriormente, uma das vantagens da utilização de funções é a reutilização do código, ou seja, poder realizar várias chamadas a uma mesma função. Por exemplo, podemos modificar o programa 1 para realizar duas chamadas a função *fim()*:

```
1 int main()
2 {
3    cout << "Um pouco mais do mesmo!!" << endl;
4    fim(); //chamada da função
5   fim(); //outra chamada da mesma função
6    return 0;
7 }
```

Não significa que o código da função *fim()* será escrito duas vezes, apenas será realizado 2 chamadas para o mesmo trecho de código (figura 4):

Como pode ser observado na figura 4, ocorreu 2 chamadas distintas a função *fim()*.

4 Minha segunda função - passando parâmetros

Para se definir uma função deve-se indicar o tipo do retorno da função, seu nome e os parâmetros da mesma. Uma função pode ou não retornar um valor. Se uma função não retorna nenhum valor seu retorno deve ser definido como *void*. Os parâmetros devem ser definidos, um por um, indicando o seu tipo e nome separado por virgula.

O programa 2 implementa três funções: duas que não recebem parâmetros (*void*) e outra que recebe três parâmetros (imagem, coluna, linha) indicando a posição onde a imagem será impressa na tela. Estas funções ainda não retornam valores (logo são definidas com retorno *void*).

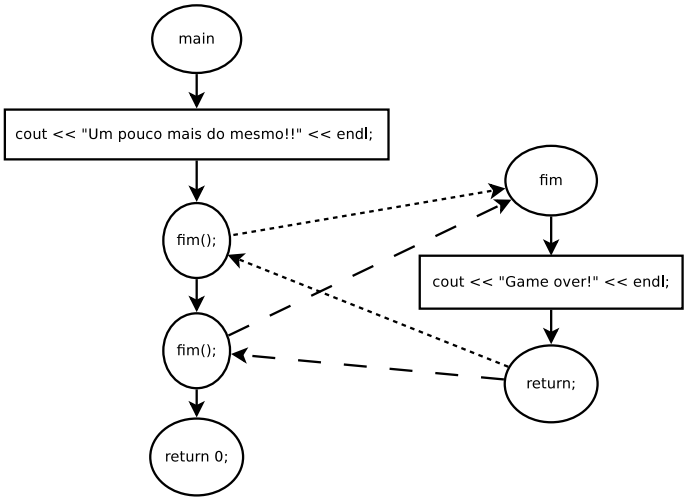

Figura 4: Fluxograma do programa 1 com 2 chamadas a função *fim()*.

Programa 2: Minha segunda função.

```
// Minha segunda função
// programa_002.cpp
#include "biblaureano.h"

void imprimeSprite( string sprite, int x, int y);
void aviao(void);
void navio();

int main()
{
   string tipo;
   do
   {
      tipo = readString("[A]viao ou [N]avio?");
   } while( tipo != "A" && tipo != "N");
   if( tipo == "A")
   {
      aviao();
   }
   else
   {
      navio();
   }
   cout << "Game over!!" << endl;
   return 0;
}

void aviao(void)
{
   string aviao =  "    |\n-=O=-\n";
   int x=1;
   for( int y=1;y<20;++y)
   {
      imprimeSprite(aviao, x, y);
      espera(100);
      limparTela();
      ++x;
   }
   return;
}

void navio(void)
```

```
43 {
44    string navio =  "__/\\__\n\\____/\n";
45    for( int x=1;x<60;++x)
46    {
47       imprimeSprite(navio, x, 10);
48       espera(100);
49       limparTela();
50    }
51    return;
52 }
53
54 void imprimeSprite( string sprite, int x, int y)
55 {
56    int pos;
57    //quebra os \n para reposicionar e não dar erro
58    while( (pos = sprite.find("\n")) != string::npos)
59    {
60       gotoXY(x,y);
61       string sprite_parcial = sprite.substr(0,pos+1);
62       cout << sprite_parcial;
63       ++y;
64       sprite.erase( 0,pos+1);
65    }
66    gotoXY(x,y); //imprime última parte que não tem /n
67    cout << sprite; //<<endl;
68    return;
69 }
```

4.1 Entendendo o programa

Inicialmente declaramos os protótipos das três funções. As funções *aviao* e *navio* não recebem nenhum parâmetro, portanto é informado a palavra reservada[1] *void*. A função *imprimeSprite* recebe uma string contendo a representação da imagem e dois valores inteiros cuja finalidade é informar a coluna e linha onde a imagem será impressa.

```
1 ...
2 void imprimeSprite( string sprite, int x, int y);
3 void aviao(void);
4 void navio(); //se não colocar void entre parênteses a linguagem assume void automaticamente
5 ...
```

A sintaxe correta para a declaração de uma função é:

```
1 tipo_variável nome_função(tipo_variável nome_parâmetro_01,  tipo_variável nome_parâmetro_02, ...)
2 {
3    ...
4    ...
5    bloco de comandos;
6    ...
7    ...
8    return tipo_retorno;
9 }
```

Reparem que os parâmetros informados são separados sempre por vírgulas e sempre deve ser definido o tipo de cada parâmetro. Uma função pode receber qualquer valor (*int*, *float*, *double*, etc) e retornar qualquer valor.
Cabe lembrar que uma função retorna apenas um único valor embora possa receber vários valores por parâmetros.

[1] Palavra chave da linguagem de programação e que não pode ser utilizada para outras finalidades

Na figura 5 pode ser observado o fluxograma exemplificando as chamadas das funções do programa 2. Como pode ser verificado, o programa principal (*main*) faz uma chamada para as funções *aviao* e *navio*. Na sequência, as funções realizam uma chamada a função *imprimeSprite*. Podemos dizer que o programa principal (*main*) está no nível 0, as funções *aviao* e *navio* estão no nível 1 e a função *imprimeSprite* está no nível 2:

Na sequência temos o nosso programa principal (*main*) que pergunta qual imagem deve ser exibida e chama a função correspondente:

```
int main()
{
   string tipo;
   do
   {
      tipo = readString("[A]viao ou [N]avio?");
   } while( tipo != "A" && tipo != "N");
   if( tipo == "A")
   {
      aviao();
   }
   else
   {
      navio();
   }
   cout << "Game over!!" << endl;
   return 0;
}
```

E finalmente chegamos no código das funções. Observe que as funções *aviao* e *navio* realizam uma chamada a função *imprimeSprite*. Logo, podemos afirmar que uma função pode realizar chamadas para outras funções.

```
void aviao(void)
{
   string aviao =  "    |\n-=O=--\n";
   int x=1;
   for( int y=1;y<20;++y)
   {
      imprimeSprite(aviao, x, y);
      espera(100);
      limparTela();
      ++x;
   }
   return;
}

void navio(void)
{
   string navio =  "__/\\__\n\\____/\n";
   for( int x=1;x<60;++x)
   {
      imprimeSprite(navio, x, 10);
      espera(100);
      limparTela();
   }
   return;
}

void imprimeSprite( string sprite, int x, int y)
{
   int pos;
   //quebra os \n para reposicionar e não dar erro
   while( (pos = sprite.find("\n")) != string::npos)
   {
      gotoXY(x,y);
      string sprite_parcial = sprite.substr(0,pos+1);
      cout << sprite_parcial;
      ++y;
      sprite.erase( 0,pos+1);
   }
   gotoXY(x,y); //imprime última parte que não tem /n
   cout << sprite; //<<endl;
```

```
41     return;
42 }
```

Quando uma função é declarada como retorno *void* (colocando antes do nome da função), estamos indicando que a função não irá retornar nenhum valor e portanto o comando *return* é utilizado sem parâmetros.

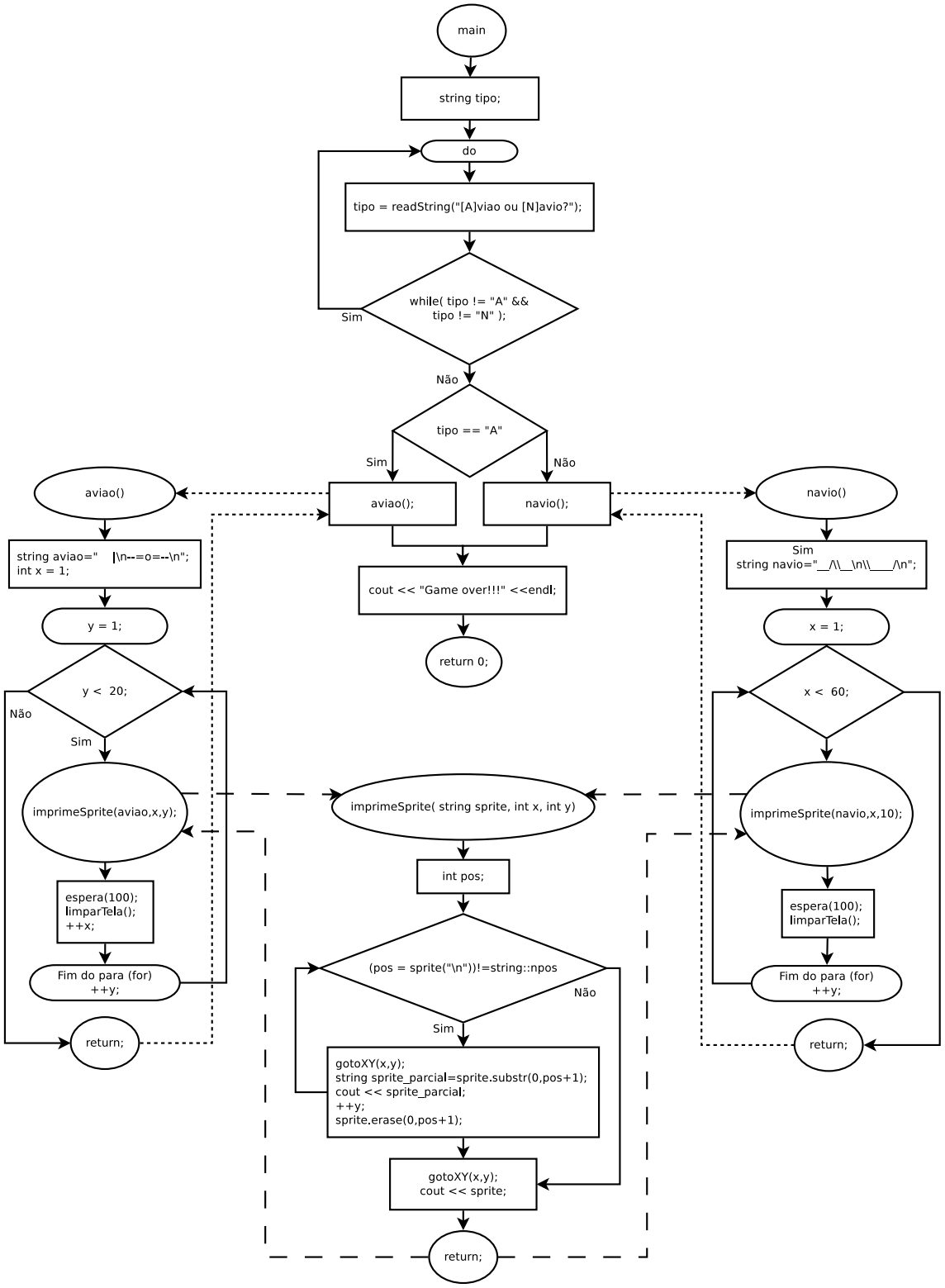

Figura 5: Fluxograma do programa 2.

5 Minha terceira função - passando parâmetros e recebendo valores

Certas funções buscam determinar um resultado específico e único a partir dos dados de entrada que recebe, como por exemplo o determinante de uma equação do 2ºgrau a partir dos valores dos coeficientes de $f(x) = ax^2 + bx + c$. Portanto a maior utilidade de uma função está na capacidade de retornar o resultado do seu processamento.

No programa 3 podemo observar a utilização de funções que retornam um valor.

Programa 3: Minha terceira função.

```
// Minha terceira função
// programa_003.cpp
#include "biblaureano.h"

char perguntaSimNao1(void);
char perguntaSimNao2(string pergunta);

int main()
{
   char resposta;

   resposta = perguntaSimNao1();

   cout << "Sua resposta foi " << resposta << endl;

   do
   {
      cout << "Que jogo excitante..." << endl;
      cout << "...." << endl;
      resposta = perguntaSimNao2("Continua o jogo ?");
   } while( resposta == 's');

   resposta = perguntaSimNao2("Salvo o jogo ?");
   if( resposta == 's' )
   {
      cout << "Salvando o jogo..." << endl;
   }
   cout << "Game over!!" << endl;
   return 0;
}

//retorna um char e não recebe nada
char perguntaSimNao1(void)
{
   char resposta;
   do
   {
      cout << "Entre com [s]im ou [n]ão:";
      cin >> resposta;
   } while( resposta != 's' && resposta != 'n');
   return resposta;
}

//retorna um char e recebe uma pergunta
char perguntaSimNao2(string pergunta)
{
   char resposta;
   do
   {
      cout << pergunta << " [s]im ou [n]ão:";
      cin >> resposta;
   } while( resposta != 's' && resposta != 'n');
   return resposta;
}
```

Uma função pode receber vários valores (parâmetros) mas retorna apenas um *único* valor.

Na figura 6 pode ser observado o fluxograma exemplificando as chamadas das funções:

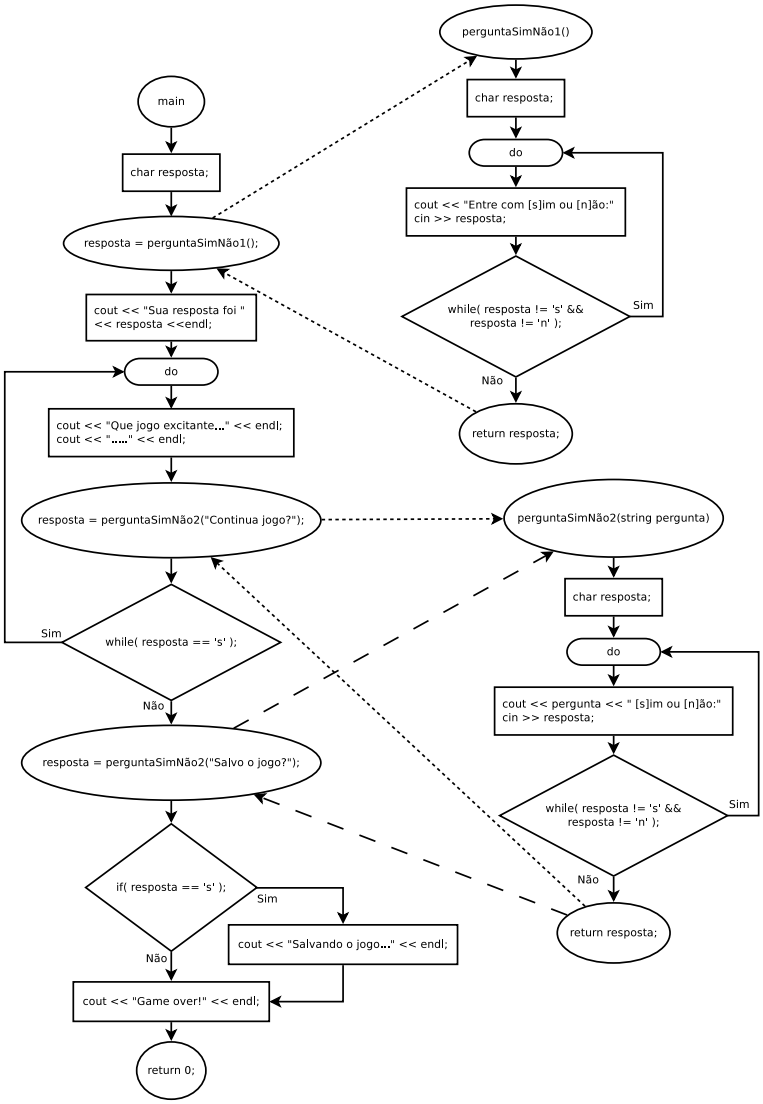

Figura 6: Fluxograma do programa 3.

5.1 Entendendo o programa

A exemplo dos outros exemplos vistos até o momento, primeiros nós declaramos os protótipos das funções. O detalhe está no retorno das funções pois ambas retornam um valor do tipo *char*.

```
1    char perguntaSimNao1(void);
2    char perguntaSimNao2(string pergunta);
```

O tipo de retorno de uma função é indicado pelo tipo informado antes do nome da função. O retorno de uma função ocorre na execução do comando *return* em algum ponto do corpo da função.

Na sequência temos a utilização das funções (chamadas as funções). Reparem que agora efetuamos a chamada da função e o retorno dela é atribuída para uma variável declarada previamente e do mesmo tipo (*char*) que a função retorna.

```
1    ...
2    char reposta;
3    resposta = perguntaSimNao1();
4    cout << "Sua resposta foi " << resposta << endl;
5    ...
```

A próxima função, além de retornar um valor também recebe uma mensagem a ser mostrada pelo usuário.

```
1    ...
2    do
3    {
4        cout << "Que jogo excitante..." << endl;
5        cout << "...." << endl;
6        resposta = perguntaSimNao2(Continua o jogo ?);
7    } while( resposta == 's');
8    ...
```

E a principal vantagem da utilização de funções fica evidente nas linhas sequentes, pois novamente realizamos uma pergunta ao usuário (utilizando a função *perguntaSimNao2*) mas agora com outro questionamento.

```
1    ...
2    resposta = perguntaSimNao2(Salvo o jogo ?);
3    if( resposta == 's' )
4    {
5        cout << "Salvando o jogo..." << endl;
6    }
7    ...
```

Finalmente, temos a declaração das nossas funções.

```
1  //retorna um char e não recebe nada
2  char perguntaSimNao1(void)
3  {
4      char resposta;
5      do
6      {
7          cout << "Entre com [s]im ou [n]ão:";
8          cin >> resposta;
9      } while( resposta != 's' && resposta != 'n');
```

```
10      return resposta;
11  }
12
13  //retorna um char e recebe uma pergunta
14  char perguntaSimNao2(string pergunta)
15  {
16      char resposta;
17      do
18      {
19          cout << pergunta << " [s]im ou [n]ão:";
20          cin >> resposta;
21      } while( resposta != 's' && resposta != 'n');
22      return resposta;
23  }
```

A definição do retorno de uma função ocorre no momento da sua declaração colocando-se o tipo a ser retornado antes do nome da função e que o retorno da função é dado pela utilização do comando *return* seguido do valor a ser retornado.

É possível também criar funções que contenham múltiplos comandos return, cada um dos quais retornando um valor para uma condição específica. Por exemplo, considere a função *comparaValores*, mostrada a seguir:

```
int comparaValores( int primeiro, int segundo)
{
   if( primeiro == segundo )
   {
      return 0;
   }
   else if( primeiro > segundo )
   {
      return 1;
   }
   else if( primeiro < segundo )
   {
      return -1;
   }
}
```

A função *comparaValores* examina dois valores listados na tabela abaixo:

Resultado	Significado
0	Os valores são iguais.
1	O primeiro valor é maior que o segundo.
-1	O segundo valor é maior que o primeiro.

Como regra, deve-se tentar limitar as funções a usar somente um comando *return*. À medida que as funções se tornarem maiores e mais complexas, ter muitos comandos *return* normalmente tornará as funções mais difíceis de compreender. Na maioria dos casos, pode-se reescrever a função para que ela use somente um comando *return*, como pode ser observado no programa a seguir:

```
int comparaValores( int primeiro, int segundo)
{
   int retorno;
   if( primeiro == segundo )
   {
      retorno = 0;
   }
   else if( primeiro > segundo )
   {
      retorno = 1;
   }
   else if( primeiro < segundo )
   {
      retorno = -1;
   }
   return retorno;
}
```

6 Funções com retorno *void*

Como colocado até o momento, uma função retorna um valor. E pode receber parâmetros. O *void* é utilizado da seguinte forma:

```
1 void semUtilidade(void)
2 {
3    ...
4    // codigo
5    ...
6    return;
7 }
```

No exemplo acima, a palavra *void* define que:

- Não vai receber parâmetros;
- Não vai retornar qualquer valor.

Ou melhor, *void* é uma **explicitação** do programador que aquela função não vai receber ou retornar nenhum valor. O valor da função é ignorado, mas a função realmente retorna um valor, por isso para que o resultado não seja interpretado como um erro é bom declarar *void*.

7 *main* como uma função

Todo programa possui uma função principal que contém todos os comandos e chamadas para outras funções presentes no programa. A função *main* funciona como uma função normal: possui um protótipo e uma definição. Geralmente omitimos o protótipo, fazendo apenas a definição da função *main* da seguinte forma:

```
1 int main (void)
2 {
3    //corpo do programa
4    ...
5    return 0;
6 }
```

Note que a função *main* é do tipo *int*, e retorna 0. Entretanto, não existe outra função acima de *main* que a tenha chamado, para que ela possa retornar um valor de resposta. Para que serve este retorno então ? Simples: consideramos que a *função chamadora* de *main* é o próprio sistema operacional. Assim, utilizamos o retorno para indicar o funcionamento do programa. Caso o programa termine e retorne o valor 0 para o sistema operacional, sabemos que tudo correu bem e que o programa terminou normalmente. Um valor retornado diferente de 0 indica que o programa não rodou até o final (ou seja, até o ponto *return 0;*) e que aconteceu algum erro. Muitos sistemas operacionais e programas utilizam esse sistema simples para detectar erros durante a execução de seus aplicativos.

Não se pode utilizar *void* na função principal *main*, apesar de existirem exemplos com *void* em algumas bibliografias. Infelizmente, alguns compiladores aceitam *void main()*. O *main()* é especial e tem de retornar um *int*. Uma execução bem sucedida do programa costuma retornar 0 (zero) e, em caso de erro, retorna 1 (um).

8 Exercícios

1. Escrever uma função que calcula um inteiro elevado a outro inteiro, usando multiplicação.
2. Elabore uma função que calcule a expressão $N = \frac{1!}{(N+1)!} + \frac{2!}{(N+2)!} + \frac{3!}{(N+3)!} + \ldots + \frac{N!}{(N+N)!}$. Sua função recebe o valor N e retorna para o programa principal.
3. Faça um programa onde duas espaçonaves fique se movendo aleatoriamente para direita ou esquerda. Faça uma função para cada espaçonave.