Scanf() & printf(): COMO UTILIZAR

Emanuel P. Barroso Neto

CAPÍTULO 1

Scanf() – Principais Usos

1.1 – Uma breve introdução ao comando *scanf()*

Considere o programa abaixo:

```
#include <stdio.h>
int main(void) {
  int num;
  printf (“Entre com um numero inteiro: “);
  scanf (“%d”,&num);
  printf (“Seu numero eh: %d\n”,num);
  return (0);
}
```

O funcionamento do programa é bem simples: ele pede ao usuário um número inteiro, depois imprime o mesmo na tela. Note o uso do comando *scanf()* – ele é o comando o qual a linguagem C utiliza para captura de dados do teclado. Existem outros comandos de leitura de dados em C, porém o *scanf()* tem a distinção de ser o mais completo deles, uma vez que permite vários modos de leitura de dados, alguns deles não suportados por outros comandos largamente utilizados, como o *fgets()*. Entre os tipos de formatação que o scanf() pode realizar com os dados lidos, incluem-se números inteiros, números reais com precisão simples ou dupla (*floats* e *doubles*, respectivamente), caracteres e *strings*. A tabela a seguir mostra quais especificadores de formato são utilizados pelo *scanf()*. Logo após, é mostrado um programa que utiliza os principais especificadores descritos acima:

Especificador	**Tipo de dado**
%d	*int em decimal*
%ld	*long int em decimal*
%f	*float*
%lf	*double*
%c	*char*
%s	*string* (sem espaços)
%[ ]	Conjunto de caracteres definido por expressão entre

colchetes

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

2- int main(void) {
3-  int inteiro;
4-  float real;
5-  double real2;
6-  char letra;
7-  char nome[50];
8-  char no_space[50];

9-  printf (“Digite um valor inteiro: “);
10- scanf (“%d”,&inteiro);
11- printf (“Digite um valor real, precisao simples: “);
12- scanf (“%f”,&real);
13- printf (“Digite um valor real, precisao dupla: “);
14- scanf (“%lf”,&real2);
15- printf (“Digite um caractere: “);
16- scanf (“%c”,&letra);
17- printf (“Digite uma string sem espacos: “);
18- scanf (“%s”,nome);
19- printf (“Digite uma string com espacos: “);
20- scanf (“%[^\n]”,no_space);
```

```
21- printf (“Inteiro: %d\n”,inteiro);
22- printf (“Float: %f \tDouble: %lf\n”,real,real2);
23- printf (“Caractere: %c\n”,letra);
24- printf (“String sem espaco: %s\n”,nome);
25- printf (“String com espaco: %s\n”,no_space);

26- return (0);
27-}
```

Veja que, normalmente, o nome da variável deve estar precedido de um *‘&’* no *scanf()*, *salvo se a variável for uma string.* A razão disso é muito simples: o *scanf()* trabalha com os *endereços* das variáveis, de forma a efetivamente alterar seu valor (chamamos isso de passagem de parâmetros por referência – conteúdo visto em subalgoritmos – não irei me ater a isso agora). Porém, no caso das *strings*, que são um *conjunto* (*array*, ou, como será visto, *vetor*) de variáveis do tipo caractere, o nome da string já é seu endereço e portanto o *‘&’* não deve ser usado em sua leitura.

Note a linha número 20, *scanf(“%[^\n]”,no_space);*. Esta linha mostra uma outra forma de usar o scanf() ara ler uma cadeia de caracteres (*string*). Ao contrário do uso do *%s* como na linha 18, o especificador *%[^\n]* permite a leitura de *strings* com espaços. Esse último especificador é um dos recursos mais poderosos do *scanf()* – os *scansets*.

1.2 – Os *scansets* (especificador de formato *%[ ]*)

O grande problema do especificador *%s* num comando *scanf()* é que ele interrompe a leitura da *string* assim que encontra um espaço ou outro caracter separador (como uma tabulação, por exemplo). Isso se torna inconveniente quando queremos ler, por exemplo, o nome completo de uma pessoa. Para contornar tal situação, utilizaremos um dos recursos mais poderosos do *scanf()*: o *scanset*.

Basicamente, um *scanset* é dado pelo especificador de formato *%[ ]*. O que varia de *scanset* a *scanset* é o conteúdo dentro dos colchetes. Por exemplo, considere o programa abaixo:

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


2- int main(void) {
3-    char string[30];


4-    printf (“String com letras A,B e C: “);
5-    scanf (“%[ABC]”,string);
6-    printf (“Sua string: %s\n”,string);


7-    return (0);
8- }
```

Experimente digitar uma *string* com uma letra ‘D’ (*“ABCDA”*, por exemplo). Note que a saída terá tudo o que você digitou até o primeiro caractere que não é ‘A’, ‘B’ ou ‘C’. O que o especificador *%[ABC]* faz é o seguinte: ele lê apenas os caracteres ‘A’, ‘B’ e ‘C’ do teclado, parando a leitura caso haja um caractere diferente. O mesmo especificador poderia ser reescrito como *%[A-C]*, em que o hífen denota um intervalo de caracteres (veja a tabela ASCII para maiores detalhes sobre as faixas de valores relacionados a caracteres).

Agora, considere o programa abaixo:

```
1- #include <stdio.h>
2- int main(void) {
3- char string[30];
4-    printf (“Digite seu nome: “);
5-   scanf(“%[^\n]”,string);
6-  printf (“Seu nome: %s\n”,string);
7-   return (0);
8- }
```

Observe o especificador *%[^\n]* dentro do *scanf()* na linha 5. Ele faz com que o *scanf()* leia caracteres até encontrar um salto de linha (ou seja, ‘\n’, que indica que o

usuário apertou *ENTER*). Note a presença do '^' – ele diz ao *scanf()* quais caracteres não serão lidos. Por exemplo:

```
1- #include <stdio.h>
2- int main(void) {
3-    char string[30];
4-    printf ("Digite uma string sem vogais: ");
5-    scanf ("%[^AaEeIiOoUu]",string);
6-    printf ("Sua string: %s\n",string);
7-    return (0);
8- }
```

Esse programa lê do teclado uma *string* qualquer, mas há uma ressalva: ele só irá finalizar a leitura ao encontrar uma vogal entre os caracteres digitados. Caso tal situação seja um inconveniente, mude o especificador de formato, adicionando um '\n' à lista de caracteres que serão ignorados.

Há um detalhe no *scanset* que deve ser levado em conta: *não é necessário adicionar um 's' ao final do especificador* (*%[^\n]s*, por exemplo), uma vez que o C já reconhece o *%[]* como especificador de leitura de *strings* (cadeias de caracteres).

Scansets são, sem dúvida, um meio muito bom de se controlar a leitura de caracteres do teclado, mas na prática, o *scanset* mais utilizado é o *%[^\n]*. Note que o *scanset não limita* a quantidade de caracteres que o *scanf()* lê – isso é um problema sério, uma vez que *strings* possuem um limite de tamanho. Para isso, veremos um pequeno detalhe que irá nos ajudar a contornar tal problema.

1.3 – Limitando a quantidade de caracteres lidos - *%y[ ]*

Considere o programa abaixo:

```
1- #include <stdio.h>
2- int main(void) {
3-    char string[6];
4-   printf ("Entre com uma string de 5 caracteres: ");
5-   scanf ("%[^\n]",string);
6-    printf ("Sua string: %s\n");
```

```
7-    return (0);
8- }
```

O programa acima possui um problema – ele não limita a quantidade de caracteres que o usuário pode digitar. No caso, o usuário deveria poder digitar apenas 5 caracteres, uma vez que toda *string* possui o caractere '\0' (nulo ou *NULL*) no final. Para isso, o *scanf()* deve *limitar* a quantidade de caracteres que serão lidos. Seu especificador de formato, assumindo que estejamos utilizando *scansets*, é dado por *%**y**[ ]*, em que *y* é um número inteiro que equivale ao número de caracteres que serão efetivamente lidos – os outros serão descartados. Para ilustrar tal comportamento de maneira mais clara, seja o seguinte programa:

```
1- #include <stdio.h>
2- int main(void) {
3-   char string[10];
4-    printf ("Entre com uma string: ");
5-    scanf ("%3[^\n]",string);
6-    printf ("Digitou: %s\n",string);
7-    return (0);
8- }
```

Ao executar esse programa, experimente digitar qualquer *string* com mais de 3 caracteres: a saída conterá apenas os 3 primeiros caracteres digitados. O especificador *%3[^\n]* diz ao *scanf()* que serão lidos no máximo 3 caracteres – quaisquer caracteres adicionais serão ignorados. Note que, se forem digitados menos de 3 caracteres, por exemplo, prevalece a interpretação do *scanset*, ou seja, serão lidos caracteres até que se encontre um '\n' (*ENTER*).

Até agora, já dispomos de especificadores para ler todos os dados básicos com um bom grau de confiabilidade, mas ainda existem outros problemas na entrada de dados que podem resolvidos. Por exemplo, considere o programa abaixo:

```
1- #include <stdio.h>
2- int main(void) {
3-    int num;
4-    do {
```

```
5-      printf (“Digite 0 para encerrar: “);
6-      scanf (“%d”,&num);
7-      if (num!=0) {
8-        printf (“Tente Novamente!\n”);
9-      }
10-    } while (num!=0);
11-    printf (“Digitou %d\n”,num);

12-    return (0);
13- }
```

Experimente digitar um caractere qualquer (‘X’, por exemplo). O programa entrará em um *loop* infinito, ou seja, ele vai repetir a execução do laço *do-while* indefinidamente. Isso ocorre em razão da existência de uma “memória” do teclado – o *buffer*, que retém dados não lidos. É necessário, portanto, algum método para lidar com o *buffer*.

1.4 – Métodos de tratamento do *buffer* para evitar o problema do *loop* infinito

A linguagem C possui uma memória temporária para armazenar dados vindos do teclado, chamada *buffer*. O grande inconveniente do *buffer* é que ele pode vir a armazenar caracteres indesejados para o programa. No último exemplo dado nesse texto, por exemplo, o problema ocorre quando digitamos um caractere que não é numérico (uma letra, por exemplo). Como o caractere não é aproveitado pelo *scanf()*, ele permanece no *buffer*. Na próxima iteração do laço *do-while*, o *scanf()* retira o mesmo caractere do *buffer*; como não é um caractere válido, repete-se o laço. Como pode-se facilmente perceber, esse processo não terá fim. Para diminuir tal problema, usaremos um outro comando de leitura de caracteres para “descartar” o conteúdo do buffer. Começaremos com *getchar()*. Portanto, oprograma anterior é mostrado abaixo acrescido de um *getchar()*:

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

2- int main(void) {
```

```
3-   int num;


4-    do {
5-      printf (“Digite 0 para encerrar: “);
6-      scanf (“%d”,&num);
7-      getchar();


8-      if (num!=0) {
9-        printf (“Tente Novamente!\n”);
10-      }
11-    } while (num!=0);


12-    printf (“Digitou %d\n”,num);


13-    return (0);
14- }
```

Agora, o que acontece se digitarmos um caractere? O programa irá repetir o laço, mas o *getchar()* irá “destruir” o caractere inválido. Isso esvazia o *buffer*, de forma a evitar o *loop* infinito. Existe também, como alternativa ao *getchar()*, o especificador de formato *%*c*, que diz ao *scanf()* para desprezar um caractere (assim como *%*d* para ignorar um inteiro, etc.). Experimente, no programa acima, retirar o *getchar()* na linha 7 e trocar o *scanf(“%d”,&num);* da linha 6 por *scanf(“%d%*c”,&num);*. Note que não é necessário associar o *%*c* a uma variável.

Esse método de limpeza é simples e funciona, porém há um ligeiro problema. Tente entrar com uma *string* de três caracteres, por exemplo. O programa, muito provavelmente, entrará em um *loop*, mas um *loop* finito (3 iterações, no caso). Embora isso não inutilize por completo o programa, ainda é um inconveniente. Portanto, iremos ver um “upgrade” do *getchar() – while(getchar()!=’\n’)*.

Agora, seja o programa do último exemplo, mas com uma modificação substancial no método de limpeza do *buffer*:

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

```
2- int main(void) {
3-    int num;

4-    do {
5-      printf (“Digite 0 para encerrar: “);
6-      scanf (“%d”,&num);
7-      while(getchar()!=’\n’);

8-      if (num!=0) {
9-        printf (“Tente Novamente!\n”);
10-      }
11-    } while (num!=0);

12-    printf (“Digitou %d\n”,num);

13-    return (0);
14- }
```

A principal mudança no funcionamento do programa pode ser observadaquando se digita uma *string* de vários caracteres não numéricos na entrada, quando um inteiro é esperado. Ao passar pelo *scanf()*, os caracteres não serão aproveitados e serão armazenados no *buffer*. Porém, o *while(getchar()!=’\n’)* começa a “descartar” *todos* os caracteres do *buffer*, até que um *ENTER* seja encontrado. Em linhas gerais, esse *loop* esvazia o *buffer* de tal forma que o laço *do-while* não experimenta *loops* inesperados. Claro que o *while(getchar()!=’\n’)* possui seus próprios problemas estruturais, mas, para limpeza de *buffer* nos programas implementados em Computação Básica, esse método, é um dos mais simples . Ainda há mais dois métodos de limpeza de *buffer*, mas eles são específicos dos Sistemas Windows e Linux:

- (*rewind(stdin)* para Windows, e

- *__fpurge(stdin)* para Linux (para usar a função *__fpurge()*, utilize um *#include <stdio_ext.h>* ao invés de *#include <stdio.h>*).

Os dois métodos – *getchar()* e *while(getchar()!='\n')* – ainda possuem uma vantagem a mais: em C, é comum que, ao finalizar o programa, o término seja feito de modo automático. Isso pode ser inadequado, principalmente no Windows. Para forçar o programa a esperar um ENTER para encerrar, utilize ou um *getchar()* ou um *while(getchar()!='\n')* antes do comando *return (0)*.

Note que o que foi implementado até agora considera que somente um dado foi digitado. A questão é: como fazer para o *scanf()* reconhecer mais de um dado e como saber quantos dados *foram realmente lidos* pelo *scanf()*? Para isso utilizamos o valor de retorno do *scanf()*.

1.5 – *int scanf(char *format,...)*: O valor de retorno do *scanf()*

Considere o programa abaixo:

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


2- int main(void) {
3-    char str[51] = ”\0”;


4-    do {
5-      printf (“ENTER para encerrar: “);
6-      scanf (“%50[^\n]”,str);
7-    while(getchar()!=’\n’);


8-     if (str[0]!=’\0’) {
9-         printf (“Digitou %s\n”,str);
10-       }
11-    } while (str[0]!=’\0’);


12-    return (0);
13- }
```

Em teoria, o programa acima funciona da seguinte forma: se o usuário digitar apenas *ENTER*, o programa encerra. Se for digitada uma *string* qualquer, ela será mostrada na tela e o programa solicitará uma nova *string*. Porém, o que ocorre na prática é o seguinte: se for digitado primeiro uma *string* (exemplo: “ABC”), o programa a imprimirá na tela e mostrará a mensagem “ENTER para encerrar”. Ao se digitar *ENTER*, entretanto, o programa *não* encerrará, mas mostrará novamente a última string. Como o teste do *do-while* exige uma *string* nula para encerramento do *loop*, o programa entrará em *loop* infinito. Porém, há um detalhe que irá mudar esse programa para a execução esperada: o *valor de retorno* do *scanf()*.

O *scanf()*, como toda função de C, é composto por um nome, parâmetros e o valor de retorno. Esses três componentes são reunidos no que é chamado de *assinatura* da função. Por exemplo: a assinatura do *scanf()* é mostrada abaixo:

```
int scanf(const char *format,...);
```

Analisando a assinatura acima, vemos que a função de nome *scanf()* possui como parâmetros uma *string* e um número variável de argumentos (o uso de funções com número variável de argumentos não faz parte do conteúdo visto em Computação Básica). Observe que o scanf() possui um valor de retorno do tipo inteiro. Isso quer dizer que o *scanf()*, ao ser chamado, devolve um valor. Mais especificamente, esse valor é o *número de variáveis efetivamente lidas pelo scanf()*. Isso é muito importante, pois especificadores como o *%*c*, que ignoram variáveis, e quaisquer outros caracteres dentro da *string* passada ao *scanf()* não serão considerados no valor de retorno. O mais interessante, e talvez essencial, para nossa análise do retorno: você pode declarar uma variável *int* (*int a*, por exemplo) e escrever *a = scanf(...)*: o valor dessa variável será útil para avaliar que valores o scanf() efetivamente leu. Para melhor compreensão, seja o programa abaixo, que consiste em uma versão melhorada do último exemplo:

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


2- int main(void) {

3-     char str[51] = ”\0”;

4 -    int var_lida;


5-     do {

6-       printf (“ENTER para encerrar: “);

7-       var_lida = scanf (“%50[^\n]”,str);

8 -      while(getchar()!=’\n’);
```

```
9-      if (var_lida==1) {
10-        printf (“Digitou %s\n”,str);
11-      }
12-    } while (var_lida==1);


13-    return (0);
14- }
```

Note, na linha 7, *var_lida = scanf(“%50[^\n]”,str);*. A variável *var_lida* receberá o número de variáveis que o *scanf()* leu. Neste caso, receberá 1 se o usuário digitou uma *string* qualquer, ou 0 se o usuário apenas digitou *ENTER*. Note que esse valor *não depende do conteúdo* da *string*. Ou seja, mesmo que refaçamos os testes que evidenciaram os problemas do exemplo anterior no presente exemplo, *var_lida* conterá 0 se nada for digitado pelo usuário, não importando o conteúdo anterior da *string*. Portanto, o *loop* acima sairá normalmente, como o esperado.

O valor de retorno de *scanf()* é, sem dúvida, uma ferramenta muito útil para a crítica de dados de *strings*, já que é possível controlar o tamanho das mesmas, e a validade delas enquanto *strings* não nulas. Porém, um dos melhores usos dessa ferramenta é sua associação com a leitura de *várias variáveis*. A seguir, será mostrado como o *scanf()* consegue ler várias variáveis em um único comando.

1.6 – *scanf(“%s,%d,%d”,str,&a,&b);* - Lendo múltiplas variáveis com *scanf()*

Considere o seguinte problema:

“Dado um círculo de centro (x_0,y_0) e raio R, escreva um programa que, dado um ponto (x,y) qualquer, verifique se o ponto está sobre a circunferência.”

Uma solução para o problema acima seria o programa abaixo (com as devidas críticas de dados possíveis até o momento):

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


2 - int main(void) {
```

```
3 -   double r,x0,y0,x,y;
4 -   int lido;


5 -  do {
6 -     printf (“Digite a coordenada x0 do centro do circulo: “);
7 -    lido = scanf(“%lf”,&x0);
8 -    while (getchar()!=’\n’);


9 -     if ((x0<0) || (lido!=1)) {
10 -      printf (“Tente de novo.\n”);
11 -     }
12 -  } while ((x0<0) || (lido!=1));


13 -   do {
14 -      printf (“Digite a coordenada y0 do centro do circulo: “);
15 -    lido = scanf(“%lf”,&y0);
16 -     while (getchar()!=’\n’);


17 -    if ((y0<0) || (lido!=1)) {
18-       printf (“Tente de novo.\n”);
19 -    }
20 -  } while ((y0<0) || (lido!=1));


21 -  do {
22 -    printf (“Digite o valor do raio do circulo: “);
23 -     lido = scanf(“%lf”,&r);
24 -    while (getchar()!=’\n’);
```

```
25 -    if ((r<0) || (lido!=1)) {
26 -       printf (“Tente de novo.\n”);
27 -     }
28 -  } while ((r<0) || (lido!=1));

29 -  do {
30 -    printf (“Digite a coordenada x do ponto P: “);
31 -    lido = scanf(“%lf”,&x);
32 -     while (getchar()!=’\n’);

33 -    if ((x<0) || (lido!=1)) {
34 -      printf (“Tente de novo.\n”);
35 -    }
36 -  } while ((x<0) || (lido!=1));

37 -  do {
38 -     printf (“Digite a coordenada y do ponto P: “);
39 -   lido = scanf(“%lf”,&y);
40     while (getchar()!=’\n’);

41-    if ((y<0) || (lido!=1)) {
42 -      printf (“Tente de novo.\n”);
43 -    }
44 -  } while ((y<0) || (lido!=1));

45 -  if ( ((x-x0)*(x-x0)+(y-y0)*(y-y0)) == (r*r) ) {
46 -    printf (“O ponto P pertence a circunferência.\n”);
```

```
47 -   }
48 -   else {
49 -     printf (“O ponto P nao pertence a circunferência.\n”);
50 -  }

51 -  return (0);
52 -}
```

O programa acima, sem dúvida, resolve o problema proposto. Porém, ele ficou muito grande, e muito cansativo de se codificar, uma vez que há um *scanf()* para cada valor a ser lido. Vamos então *compactar* o programa onde for possível. Para o nosso problema, por exemplo, as coordenadas dos pontos podem ser lidas na forma de par ordenado. Seja então o programa modificado mostrado abaixo:

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

2. int main(void) {
3.   double r,x0,y0,x,y;
4.   int lido;

5.   do {
6.      printf (“Digite as coordenadas (x0,y0) do centro do
   circulo: “);
7.     lido = scanf(“%lf,%lf”,&x0,&y0);
8.     while (getchar()!=’\n’);

9.     if ((x0<0) || (y0<0) || (lido!=2)) {
10.      printf (“Tente de novo.\n”);
11.    }
12.  } while ((x0<0) || (y0<0) || (lido!=2));
```

```
13.  do {
14.    printf (“Digite o valor do raio do circulo: “);
15.    lido = scanf(“%lf”,&r);
16.    while (getchar()!=’\n’);

17.    if ((r<0) || (lido!=1)) {
18.      printf (“Tente de novo.\n”);
19.    }
20.  } while ((r<0) || (lido!=1));

21.  do {
22.    printf (“Digite as coordenadas (x,y) do ponto P: “);
23.    lido = scanf(“%lf,%lf”,&x,&y);
24.    while (getchar()!=’\n’);

25.    if ((x<0) || (y<0) || (lido!=2)) {
26.      printf (“Tente de novo.\n”);
27.    }
28.  } while ((x<0) || (y<0) || (lido!=2));

29.  if ( ((x-x0)*(x-x0)+(y-y0)*(y-y0)) == (r*r) ) {
30.    printf (“O ponto P pertence a circunferência.\n”);
31.  }
32.  else {
33.    printf (“O ponto P nao pertence a circunferência.\n”);
34.  }
```

```
35.  return (0);
36.}
```

Perceba que este programa ficou menor do que o outro, e a entrada dos dados foi feita de forma mais “natural”, para o usuário, em face do problema pedido. Note como o *scanf()* é utilizado na linha 7 *lido = scanf(“%lf,%lf”,&x0,&y0);*. Há dois especificadores no *scanf()*; portanto, duas variáveis devem ser lidas. E mais: o *scanf()* irá ignorar uma vírgula, pois ela está presente entre os especificadores. Você pode digitar uma entrada como *1.3,2.1* que o programa irá aceitar normalmente. Note que o *scanf()* teve seu valor atribuído à variável *lido* para controlar o número de variáveis *realmente* lidas – esse método será bem explorado na próxima seção. Apresento então um método bem simples, porém poderoso, de crítica de dados usando *scanf()* – O uso de uma variável *lixo*.

1.7 – Método do *lixo* – Aplicação dos conceitos envolvendo *scanf()*

Agora, apresento uma aplicação de todos os conceitos do *scanf()* vistos até aqui: um método simples, porém eficiente, de efetivamente criticar dados recebidos pelo *scanf()*. Vamos começar com um programinha bem simples, que lê o primeiro nome de uma pessoa. Porém, o nome não pode conter mais do que 10 letras. Vejamos:

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


2. int main(void) {
3.     char nome[11];
4.     int lido;


5.     do {
6.       printf (“Digite seu nome: “);
7.       lido = scanf(“%10[^\n]”,nome);
8.       while (getchar()!=’\n’);


9.       if (lido!=1) {
10.        printf (“Tente novamente.\n”);
```

```
11.       }
12.     } while (lido!=1);


13.     printf (“Seu nome eh: %s\n”,nome);
14.     return (0);
15.}
```

O programa acima, ao ser executado, comporta-se normalmente. Se o usuário não digitar um nome, a variável *lido* será diferente de 1, disparando a mensagem de erro. Porém, há um detalhe: Experimente digitar uma *string* muito grande. Ele irá aceitar os 10 primeiros caracteres e imprimi-los na tela, suprimindo o resto. Isso pode parecer um erro, portanto devemos pensar numa forma de evitar tal incidente. Para isso, vamos considerar o seguinte raciocínio: se o usuário digita mais caracteres do que o *scanf()* deve ler, todos o excesso ficará no *buffer* até que o *while(getchar()!=’\n’)* o destrua. Porém, como há caracteres diferentes do *ENTER* no *buffer*, vamos utilizar mais uma variável dentro do mesmo *scanf()*. Será uma variável *string* de apenas *dois* caracteres (um deles será o ‘\0’ – nulo), cuja aplicação é a de *detector de lixo no buffer*, ou *variável lixo*. Seja o mesmo programa acima, após a modificação necessária:

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


2. int main(void) {
3.     char nome[11],lixo[2];
4.     int lido;


5.     do {
6.       printf (“Digite seu nome: “);
7.       lido = scanf(“%10[^\n]%1[^\n]”,nome,lixo);
8.       while (getchar()!=’\n’);


9.       if (lido!=1) {
10.        if (lido==2) {
```

```
11.           printf (“Nome muito grande!\n”);
12.         }


13.         printf (“Tente novamente.\n”);
14.       }
15.     } while (lido!=1);


16.     printf (“Seu nome eh: %s\n”,nome);
17.     return (0);
18.}
```

Veja que uma leitura para a string *lixo* foi adicionada. Em linhas gerais, essa variável só precisa pegar um caractere do *buffer* que não seja o ‘*n*’ para que a variável *lido* na linha 7 receba o valor 2 – disparando a mensagem de erro do programa. Note que, se nada for digitado pelo usuário (quando ele digita apenas o *ENTER*), os *scansets* não pegarão o ‘*n*’, e a variável *lido* conterá 0, disparando o erro. Em resumo, o programa acima limita a leitura de uma *string* ao que é esperado, para que não ocorram erros na posterior manipulação da variável de interesse. Esse é o resultado do uso do método do *lixo*.

O interessante é que o método do lixo não funciona apenas para leitura de *strings* – ele pode ser aplicado em leituras de variáveis quaisquer. Por exemplo, suponha que queiramos montar um programa em C que calcula a média de 2 notas de prova de um aluno. Obviamente, essas notas devem estar entre 0.0 e 10.0. O programa se encontra abaixo (note que vou usar um *scanf()* para ler *ambas* as notas):

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


2. int main(void) {
3.   double n1,n2,media;´
4.   char lixo[2];
5.   int lido;
```

```
6.   do {
7.       printf (“Digite as notas do aluno no formato n1,n2: “);
8.     lido = scanf (“%lf $ %lf%1[^\n]”,&n1,&n2,lixo);
9.     while (getchar()!=’\n’);

10.    if ((lido!=2)||(n1<0)||(n1>10)||(n2<0)||(n2>10)) {
11.      printf (“Dados incorretos. Tente novamente.\n”);
12.    }

13.  } while ((lido!=2)||(n1<0)||(n1>10)||(n2<0)||(n2>10));
14.
15.  media = (n1+n2)/2.0;

16.  printf (“Media final: %.2lf\n”,media);
17.  printf (“Digite ENTER para encerrar...”);
18.  while (getchar()!=’\n’);

19.  return (0);
20.}
```

A tabela abaixo mostra os prováveis resultados com alguns testes:

Entrada do usuário	**Valor de *lido***	**Saída**
2 $ 4	2	3.00
3.5 $ 3	2	3.25
A $ A	0	Erro
-1 $ 10	2	Erro
11 $ 2	2	Erro
1 $ 1b	3	Erro

É interessante salientar que o método do lixo nada mais é do que uma aplicação de todos os conceitos a respeito do *scanf()* vistos até aqui - esse método exige conhecimento de leitura de várias variáveis, valor de retorno do *scanf()*, *scansets* e tratamento de *buffer*.

Agora, para concluirmos este tutorial a respeito de *scanf()*, introduzirei aspectos de duas variações dessa função: *fscanf()* e *sscanf()*. Não se preocupe com o *fscanf()* antes de compreender arquivos (onde ele será essencial) – porém é interessante conhecer o *sscanf()*, já que ele pode ser extremamente útil.

1.8 – Variações do *scanf()*: *fscanf()* e *sscanf()*

Em aspectos essenciais, as funções *fscanf()* e *sscanf()* possuem o mesmo comportamento do *scanf()* – ambas são funções da *stdio.h* que são especializadas na captura de dados e no seu armazenamento em variáveis. A diferença reside no meio em que essas funções trabalham na captura de dados.

Comecemos com uma breve introdução ao comando *fscanf()*, que será visto durante o assunto que envolve tratamento de arquivos. Esse comando captura dados vindos de um *arquivo-texto*. Em resumo, a assinatura do *fscanf()* é a seguinte:

```
int fscanf(FILE *stream, const char *format,...);
```

Na assinatura acima, *format* continua sendo a *string* de onde o *fscanf()* retirará as variáveis. A diferença do *fscanf()* pro *scanf()* reside no 1º parâmetro. *FILE *stream* é o ponteiro para o arquivo de onde o *fscanf()* tentará retirar os dados. Se esse parâmetro for variável *FILE* de nome *stdin* (a entrada padrão de dados do teclado), *fscanf()* se comportará exatamente como *scanf()*. É necessário salientar que praticamente tudo o que foi utilizado com o *scanf()* se aplica ao *fscanf()* – com a exceção, talvez, do tratamento de *buffer* e o método do *lixo* (o *buffer* de arquivo é tratado de forma diferente – e os caracteres de final de linha variam de acordo com o Sistema Operacional).

Uma outra variação do *scanf()* é o *sscanf(), que* é muito similar ao *fscanf()*, mas com uma diferença: os dados serão retirados de uma *string*, ao invés de um arquivo. A assinatura do *sscanf()* é a seguinte:

```
int sscanf(char *input, const char *format,...);
```

Nessa assinatura, *input* é a *string* de onde o *sscanf()* retirará os dados. Os outros parâmetros comportam-se da mesma forma que no *scanf()*. Veja que todos os conceitos

de *scanf()* vistos até aqui aplicam-se ao *sscanf()*, com a exceção do tratamento de *buffer*. Como o *sscanf()* retira dados de uma memória "fixa" (no caso, uma *string*), não é necessário um método para limpar o restante. Porém, vale salientar que o método do *lixo* funciona para o *sscanf()*. Para exemplificar o poder do *sscanf()*, seja o programa abaixo:

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


3. int main(void) {
4.   char par[15],lixo[2];
5.   int lido,lido_string;
6.   double mediag,x,y;


7.   do {
8.     printf ("Entre com um par (x,y) de doubles: ");
9.     lido = scanf("%14[^\n]%1[^\n]",par,lixo);
10.    while (getchar()!='\n');
11.             lido_string   =   sscanf(par,"(%lf,%lf)
  %1[^\n]",&x,&y,lixo);


12.    if ((lido!=1) || (lido_string!=2)) {
13.               printf   ("Dados   incorretos.   Tente
  Novamente.\n");
14.    }
15.  } while ((lido!=1) || (lido_string!=2));


16.  if ((x*y)<0) {
17.    printf ("Nao ha media geometrica.\n");
18.  }
```

```
19.  else {
20.    mediag = sqrt(x*y);
21.    printf (“MG[%.2lf,%.2lf] = %.2lf\n”,x,y,mediag);
22.  }


23.  printf (“Pressione ENTER para encerrar...”);
24.  while(getchar()!=’\n’);
25.  return (0);
26.}
```

Teste o programa acima e veja que o método do *lixo* funciona muito bem para o *sscanf()* – de fato, a crítica de dados com o *sscanf()* chega a ser tão boa quanto a do *scanf()*, mas *sscanf()* possui uma vantagem: quando utilizado junto com o *fscanf()* na leitura de dados de arquivos, desde que o *fscanf()* seja um mero leitor de *strings* – muito mais fáceis de se tratar com os métodos já mostrados.

Foram passados apenas conceitos básicos a respeito da função *scanf()* e suas variações nesse capítulo. Tais conceitos são importantes para domínio da principal ferramenta de leitura de dados em C. Porém, há outras formas de se ler dados do teclado (funções *fgets()*, *gets()*, o próprio *getchar()*), mas nenhuma é tão geral quanto o *scanf()*. Todos os conceitos e métodos apresentados até aqui tiveram como objetivo a compreensão e o domínio de ferramentas do *scanf()* e do C para a leitura correta de dados e seu devido tratamento.