Entradas e Saídas

Gonzalo Travieso

2018

1 No terminal

Aqui falamos de entrada e saída do terminal, mas na verdade é da entrada e saída padrâo do programa, que normalmente é o terminal. O sistema operacional pode redirecionar para outros locais.

1.1 Saída de dados

A forma mais simples e mais usada de realizar saida de dados no terminal é através do uso do objeto `std::cout` e do *operador de inserção* `«`. O código já se adapta automaticamente para os diversos tipos de dados.

```
std::cout << "Oi."; // Mostra uma cadeia de caracteres
std::cout << 10; // Mostra um int
std::cout << 'a'; // Mostra um char
std::cout << 2.1; // Mostra um double
std::cout << 2.1f; // Mostra um float
auto x = 2 * 3 + 4 / 1.5;
std::cout << x; // Mostra um double
```

É permitido encadear diversos operadores de inserção consecutivos.

```
std::cout << "Dois mais dois vale " << 2 + 2 << "!" << std::endl;
```

O `std::endl` usado acima insere uma mudança de linha. Deve-se tomar cuidado com precedência de operadores. Apesar do operador `«` ter precedência baixa, ele tem precedência mais alta do que alguns operadores, portanto em alguma situações é necessário usar parêntesis (veja a tabela de precedência na internet).

```
std::cout << 13 | 3; // Isto e um erro
std::cout << (13 | 3); // Assim funciona
```

O objeto `std::cout` deve ser utilizado para saidas normais do programa. Para mensagens que indicam erro de execução deve-se usar `std::cerr`, e para mensagens usadas para **log**, isto é, registro do funcionamento do programa, deve-se usar `std::clog`, que funcionam de forma similar ao `cout`.

1.2 Entrada de dados

A entrada de dados do terminal pode ser feita pelo objeto `std::cin` usando, geralmente, o operador `»`, neste contexto denominado *operador de extração*. O tipo dos dados a serem lidos é deduzido automaticamente do tipo da variável usada para guardá-los.

```
int i;
double d;
char c;
std::string s;
std::cin >> i; // Le um inteiro
std::cin >> d; // Le um double
std::cin >> c; // Le um caracter
std::cin >> s; // Le uma cadeia de caracteres
```

Quando lidas com » as cadeias terminam assim que um caracter branco (espaço em branco, tabulação ou mudança de linha) for encontrado. Isso significa que não conseguimos ler dessa forma cadeias que possuem espaços em branco. A forma mais simples de fazer leitura de cadeias com caracteres em branco é usando a função `std::getline`.

```
std::string cadeia_com_espaco;
std::getline(std::cin, cadeia_com_espaco);
```

As entradas de dados como apresentadas até agora não avisam ativamente se houve erro na leitura dos dados. O programador tem que, depois de tentar a leitura, verificar se ocorreu erro ou não. Para códigos simples, isso é mais trabalho do que o necessário, e queremos apenas evitar que um erro de leitura se propague pelo código, então o recomendado é, antes de fazer qualquer operação de leitura, executar o seguinte código:

```
std::cin.exceptions(std::ios::failbit |
                    std::ios::badbit);
```

O efeito desse código é que, se ocorrer algum problema na leitura, o programa será terminado com uma mensagem de erro (na verdade, com o lançamento de uma exceção; estudaremos mais tarde o que são exceções e como lidar com elas).

2 Em arquivos

Leituras e escritas em arquivos são realizadas da mesma forma que em `std::cin` e `std::cout`, respectivamente, com a diferença que usamos um objeto associado ao arquivo desejado.

Por exemplo, vamos supor que queremos escrever num arquivo de nome `resultados.txt`. Neste caso, precisamos criar um objeto associado a um arquivo com esse nome:

```
std::ofstream saida("resultados.txt");
```

O tipo `std::ofstream` é definido em `<fstream>`. Após feito isso, basta usar `saida` da mesma forma que usaríamos `std::cout`:

```
saida << "Valor inicial " << 0 << std::endl;
```

Para leitura o procedimento é similar, mas usamos `std::ifstream` (também definido em `<fstream>`):

```
std::ifstream entrada("entrada.txt");
entrada.exceptions(std::ios::failbit |
                   std::ios::badbit);
```

```
std::string linha_inicial;
int N;
std::getline(entrada, linha_inicial);
entrada >> N;
```

Uma diferença na entrada de terminal e de arquivo é que na entrada de terminal normalmente vamos solicitando um dado por vez, em geral com a opção do usuário fornecer um sinal de que nenhum novo dado vai ser fornecido. Na entrada de arquivos, devemos saber quando parar de fazer leitura.

A forma mais simples de lidar com isso, possível em diversas situações, é simplesmente **saber** o número de dados que precisam ser lidos. Isso pode ser porque o número é fixo, ou porque ele é fornecido separadamente, seja em outro arquivo de entrada ou no início do próprio arquivo que está sendo lido. Por exemplo, se temos um arquivo `medidas.dat` com o seguinte formato: primeiro tem um número inteiro N seguido de N valores de ponto flutuante. Então podemos fazer a leitura desta forma:

```
std::ifstream dados("medidas.dat");
dados.exceptions(std::ios::failbit |
                 std::ios::badbit);

int N;
dados >> N;
for (int i = 0; i < N; ++i) {
    double x;
    dados >> x;
    // Faz algo com o valor de x...
}
```

Se isso não é possível, então precisamos alguma forma de detectar que não há mais nada a ser lido do arquivo. A forma mais simples é usar o fato de que o valor retornado pelo operador de extração, caso ele não consiga realizar a leitura, converte para o booleando `false`. Por exemplo, se temos um arquivo `outras-medidas.dat` que consiste de N valores de ponto flutuante, mas não sabemos de antemão o valor de N, podemos fazer a leitura da seguinte forma:

```
std::ifstream dados("outras-medidas.dat");
dados.exceptions(std::ios::failbit |
                 std::ios::badbit);

double x;
while (dados >> x) {
    // Faz algo com o valor de x...
}
```

No código acima, quando a leitura é bem sucedida, a condição da repetição será `true`, e o código do bloco será executado com o valor lido na variável `x`; se a leitura falhar, por exemplo pelo final do arquivo, a condição será `false`, e a repetição será terminada.

Uma outra forma de realizar isso é usar o método `eof` de `std::ifstream`, que retorna `false` se ainda não foi encontrado o final do arquivo, e `true` se já foi encontrado. A forma de fazer a chamada é `dados.eof()`; o valor retornado por

essa chamada será `true` se já foi feita uma tentativa de leitura que encontrou o final do arquivo.