10. INPUT/OUTPUT (I/O)

O sistema de I/O do C++ é muito extenso para ser completamente explorado aqui. Por isto, nesta seção será fornecida uma introdução às classes de I/O, bem como às importantes operações como formatação da saída, detecção de erros e acesso a arquivos. Em outras palavras, é descrito nesta seção como o C++ gerencia as entradas e saídas e como esse gerenciamento segue o paradigma da orientação a objetos.

Uma das principais características da linguagem C é o fato da manipulação de dados (entradas e saídas) não fazer parte das palavras reservadas da linguagem. Todo o tratamento de dados requer a inclusão de uma biblioteca de funções de manipulação de entradas e saídas. A linguagem de programação C++ segue este mesmo princípio. Para manipular os dados, deve-se utilizar bibliotecas com as funções adequadas. A linguagem C++ aceita a biblioteca padrão C (stdio.h), mas define novas bibliotecas, mais adequadas ao paradigma da orientação a objetos. Essas bibliotecas constituem a chamada biblioteca *stream* do C++. O termo *stream* significa "fluxo" e é conseqüência do fato de que, a exemplo do C, C++ trata todas operações de I/O como uma seqüência de caracteres [9].

O arquivo *header* "iostream.h" declara as quatro classes básicas de entrada e saída: *ios*, *istream*, *ostream* e *iostream*. Como mostra a figura 12, *ios* é a superclasse para *istream* e *ostream*, que são, por sua vez, superclasses para *iostream*. A classe *ios* fornece algumas facilidades básicas que são usadas por todas as outras classes de I/O, declara muitas variáveis de estado cujos valores gerenciam as operações de I/O, e define um número de constantes que ajudam a especificar o estado dos objetos *stream*. A classe *istream* fornece facilidades para entrada formatada (operador >>, por exemplo), e a *ostream* para a saída formatada (operador <<, por exemplo). A classe *iostream*, consequentemente, fornece facilidades para ambos, entrada e saída [3].

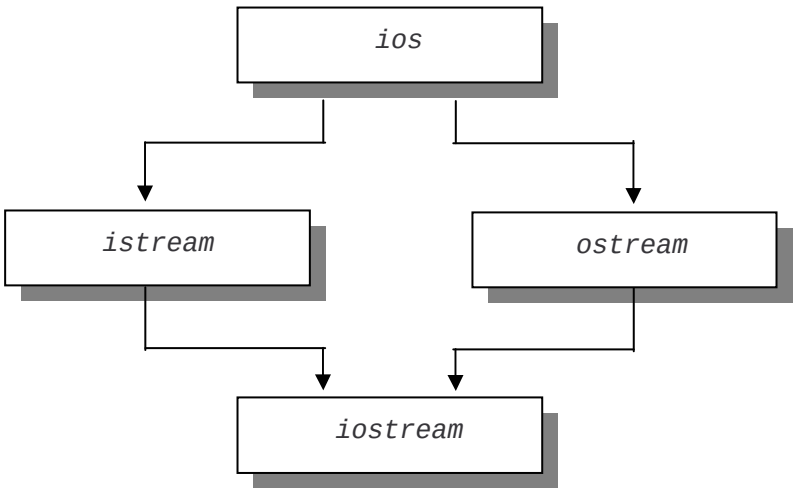

Figura 3 - Classes de I/O

Na biblioteca *stream* encontram-se os três objetos ***cin***, ***cout***, ***cerr*** e ***clog*** destinados, respectivamente, à entrada de dados padrão, à saída de dados via terminal ou via arquivos, e à saída de erros. A diferença entre *cerr* e *clog* é que *cerr* não tem *buffer*, de modo que os dados enviados para ele são imediatamente mostrados. Alternativamente, *clog* tem *buffer*, e a saída só é escrita quando um *buffer* fica cheio. Os únicos arquivos que podem ser acessados com *iostream.h* são os relacionados com os *streams* padrão. Esses objetos possuem a vantagem da sobrecarga (próxima seção), ou seja, o programador pode definir novos códigos de entrada e saída sem perder as definições originais [3, 9, 10].

Para realizar entradas de caracteres ou de seqüências de caracteres (*strings*) em C++, utiliza-se a família de funções *get()*, da classe *istream*. As diferentes formas da função *get()* são:

*get (char *string, int tamanho, char terminator);*
get (streambuf& buffer, char terminator);
get (char &ch);

onde, *string* é um ponteiro para *char* que guarda uma cadeia de caracteres; tamanho é o comprimento máximo da cadeia de caracteres; *terminator* indica um caracter especial cujo valor *default* é "\n" e que, ao ser digitado, encerra a operação de leitura de *string*; *buffer* é um objeto da classe *streambuf*; e *ch* é uma referência a uma variável caracter.

A função *get()*, em suas diferentes formas, está no objeto *cin*. Portanto, para ser acessada, deverá referenciar o objeto a que pertence. Por exemplo: "*cin.get(string, 20, 'Q');*". As demais funções para gerenciamento de entrada de dados são mostradas na tabela a seguir.

Função	Descrição
int gcount()	Retorna o número de caracteres extraídos na última vez
Istream& getline (signed char *s, int, char='\n') Istream& getline (unsigned char *s, int, char='\n')	Realiza as mesmas operações que *get()*, mas extrai inclusive o delimitador,embora não o coloque na *string* s
Istream& ignore(int n=1, int delim=EOF)	Ignora os próximos *n* caracteres ou até que *delim* seja encontrado; este caracter delimitador permanece no fluxo retornado
int peek()	Retorna o próximo caracter do fluxo sem fazer extração, retornando EOF no final
Istream& putback(char)	Devolve o caracter ao fluxo de entrada, retornando o fluxo
Istream& read(char *s, int n) Istream& read(unsigned char *s, int n)	Retiram *n* caracteres do vetor s; é aconselhável usar em conjunto com *gcount()* para descobrir o número de caracteres efetivamente extraídos
long tellg()	Retorna a posição atual do fluxo

Em relação à saída formatada do C++, a forma *default* pela qual os dados são impressos é definida pelo "estado de formato". Este, consiste em um controlador dos detalhes das operações de formatação executadas implicitamente pela linguagem (quando não se usa um *cast*). A modificação do estado de formato pode ser efetivada pelos "manipuladores" ou pelos "sinalizadores de formato" do C++.

No caso dos "sinalizadores", ou *flags*, o programador pode manipular o estado de formato (definindo base e precisão) através da utilização das funções *setf()* e *unsetf()*. Esta formatação é feita através de 16 *flags* de um *bit*. Cada *flag* pode ser "setado" (valor do *bit* é um) ou não (valor do *bit* é zero). Todos os *flags* são armazenados como *bits* em uma variável *long*. Para evitar a preocupação com as posições dos *bits* individuais, cada *flag* é distinguido através de uma constante *ios*. A tabela abaixo lista e descreve sucintamente as constantes *ios* para todos os *flags* e campos de *bit*.

Sinalizador (significado)	Descrição
ios::skipws (Ignora o espaço em branco na entrada)	Caracteres de espaço em branco iniciais (espaços, tabulações, novas linhas) são descartados ao efetuar a entrada
ios::left (Saída ajustada à esquerda) ios::right (Saída ajustada à direita) ios::internal (Preenchimento após indicador de sinal ou base)	Determina como um valor impresso é posicionado dentro do seu campo (área reservada para o valor na página impressa).
ios::dec (Conversão em decimal) ios::oct (Conversão em octal) ios::hex (Conversão em hexadecimal)	Determina quando uma notação decimal, octal ou hexadecimal será usada para imprimir e ler os valores.

ios::showbase (Mostra indicador de base na saída) ios::showpos (Mostra o sinal "+" em inteiros positivos) ios::uppercase (Saída hexadecimal maiúscula) ios::showpoint (Mostra ponto decimal - saída em ponto flutuante)	Determina a forma como valores numéricos são impressos.
ios::fixed (Usa notação normal para exibir valores numéricos em ponto flutuante - 123.45) ios::scientific (Valores em ponto flutuante são exibidos usando-se notação científica - 1.23E2)	Determina como os valores de ponto flutuante serão impressos.
ios::unitbuf (Libera - *flush* - todas as *streams* depois da inserção) ios::stdio (Libera - *flush* - *stdout*, *stderr* depois da inserção) ios::boolalpha (Valores booleanos podem ser recebidos ou mostrados)	Controla o fluxo dos *buffers*.

A biblioteca *iostream* fornece as funções *fill()*, *flags()*, *precision()* e *width()* para acessar componentes de formatação. Estas funções são declaradas em *ios*, herdadas por todas as outras classes *stream*, tais como *istream* e *ostream*, e são usadas de maneiras muito similares. Se elas são chamadas sem argumentos, elas retornam os componentes do formato corrente, sem alterá-lo. Assim, os seguintes comandos salvam os componentes de formato em variáveis:

```
char c = cout.fill()        long f = cout.flags()        int p = cout.precision()        int w = cout.width()
```

Se um argumento é fornecido, a função retorna o valor corrente e altera o componente para o novo valor especificado como argumento.

A sintaxe para usar esses sinalizadores, exemplificados no próximo programa, é a seguinte:

```
cout.setf (ios::sinalizador);     // para ligar
cout.unsetf (ios::sinalizador);  // para desligar
```

```
#include <iostream>
#include <conio.h>
using namespace std;
void main () {
 char  letra = 'A';
 int   inteiro   = 128;
 float real = 4555.57877889898 ;
 clrscr();
 cout.setf(ios::fixed);
 cout.precision(2);
 // Imprime uma Letra
 cout << "\nChar como caracter      : " << letra;
 // Imprime Codigo ASCII da Letra
 cout << "\nCodigo ASCII da Letra   : " << (int) letra;
 // Imprime A Letra cujo ASCII e' 90
 cout << "\nASCII 90 como uma Letra : " << (char) 90;
 // Imprime Um Inteiro de Varias Formas
 cout << "\n\n\nComo Decimal     :  "<< inteiro;
 cout << "\nComo Char        :  "<< (char)  inteiro;
 cout.setf(ios::hex);
 cout << "\nComo Hexadecimal :  " <<  inteiro;
 cout.unsetf(ios::hex);
 cout.setf(ios::oct);
 cout << "\nComo Octal       :  " <<  inteiro;
 cout << "\nComo Float       :  " << (float) inteiro;
 // Imprime Um Float de Varias Formas
 cout << "\n\n\nFloat como Decimal      :  "<< real;
 cout.unsetf(ios::fixed);
 cout << "\nFloat como Cientifico   :  " << real << "\n\n\n";
 cout.setf(ios::fixed);
 // tamanho minimo 20, ajuste a direita
 cout.width(20);
 cout << real << '\n';
 // ajuste a direita com zeros anteriores
 cout.width(20);
 cout.fill('0');
 cout << real << '\n';
 cout.fill (' ');
 // tamanho minimo 20, ajuste a esquerda
 cout.width(20);
 cout.setf(ios::left);
 cout << real << '\n';
 // ajuste a esquerda, com zeros anteriores
 cout.width(20);
 cout.fill('*');
 cout << real;
 cout.unsetf(ios::left);
 cout.fill(' ');
 }
```

O segundo modo de alterar os parâmetros de formato de um canal em C++ pressupõe a utilização de "manipuladores". Alguns são homônimos dos "sinalizadores". A diferença está na forma mais compacta e na inclusão de outra biblioteca de classes, *iomanip.h*. O arquivo *header iomanip.h* fornece manipuladores parametrizados que correspondem a *fill()*, *precision()*, *width()* na forma de argumentos de *setf()* e *unsetf()*. Assim, os seguintes comandos modificam o formato de *cout* determinando e limpando *flags* e trocando os valores dos três parâmetros.

```
cout << setiosflags (ios::showbase);
cout << resetiosflags (ios::left | ios::showpos);
cout << setfill ('*');
```

```
cout << setprecision (5);
cout << setw (10);
```

Os manipuladores *setfill()*, *setprecision()* e *setw()* determinam, respectivamente, o caracter de preenchimento, precisão e largura. Da mesma forma, *setf()* e *setiosflags()* determinam os *flags* especificados e não alteram os outros [3, 10].

A tabela a seguir expõe estes manipuladores (alguns já apresentados anteriormente) e os seus significados, e o próximo programa exemplifica sua utilização [9].

Manipulador	Significado
dec	Entrada/Saída de dados em decimal
oct	Entrada/Saída de dados em octal
hex	Entrada/Saída de dados em hexadecimal
ws	Ignora o espaço em branco de entrada preliminar
endl	Insere nova linha e esvazia o canal (*stream*)
ends	Insere término nulo em *string* ('\0')
flush	Esvazia um canal ostream descarregando fluxo
setbase (int)	Ajusta o formato de conversão para a base *n* (0, 8, 10 ou 16); valor 0 é *default* e dá decimal na saída
resetiosflags (long)	Limpa os *bits* de de formato em *ins* ou *outs* especificadas
setiosflags (long)	Ajusta os *bits* de de formato em *ins* ou *outs* especificadas
setfill (int n)	Define o caracter de preenchimento como *n*
setprecision (int n)	Ajusta a precisão do ponto flutuante para *n*
setw (int n)	Define a largura do campo para *n*

```
#include <iostream>
#include <iomanip>
using namespace std;

void main () {
  char  letra = 'A';
  int   inteiro   = 128;
  float real = 4555.57877889898 ;
  clrscr();
  cout.setf(ios::fixed);
  cout << setprecision(2) ;
  // Imprime uma Letra
  cout << "\nChar como caracter      : " << letra;
  // Imprime Codigo ASCII da Letra
  cout << "\nCodigo ASCII da Letra   : " << (int) letra;
  // Imprime A Letra cujo ASCII e' 90
  cout << "\nASCII 90 como uma Letra : " << (char) 90;
  // Imprime Um Inteiro de Varias Formas
  cout << "\n\n\nComo Decimal     :  "<< inteiro;
  cout << "\nComo Char        :  "<< (char)  inteiro;
  cout << "\nComo Hexadecimal :  "<< hex <<  inteiro;
  cout << "\nComo Octal       :  "<< oct <<  inteiro;
  cout << "\nComo Float       :  "<< (float) inteiro;
  // Imprime Um Float de Varias Formas
  cout << "\n\n\nFloat como Decimal      :  "<< real;
  cout.unsetf(ios::fixed);
  cout << "\nFloat como Cientifico   :  " << real;
  cout.setf(ios::fixed);
  // tamanho minimo 20, ajuste a direita
  cout << "\n\n\n" << setw(20) << real << "\n";
  // ajuste a direita com zeros anteriores
  cout << setw(20) << setfill('0') << real << setfill(' ');
  // tamanho minimo 20, ajuste a esquerda
  cout.setf(ios::left);
  cout << "\n" << setw(20) << real << "\n";
  // ajuste a esquerda, com zeros anteriores
  cout << setw(20) << setfill('*') << real << setfill(' ');
  cout.unsetf(ios::left);

}
```

Ainda com relação à saída de dados, é interessante comentar que a classe *ostream* apresenta uma série de funções associadas ao fluxo de saída. A próxima tabela mostra essas funções e as suas utilidades.

Função	Descrição
ostream& flush()	Descarrega o fluxo
ostream& put(char c)	Insere o caracter *c*; corresponde ao *putchar()* da linguagem C
long tellg()	Retorna a posição atual do fluxo
ostream& write(const signed char *s, int n) ostream& write(const unsigned char *s, int n)	Insere n caracteres da *string s*, incluindo o nulo

É importante citar que a classe *ios* também fornece as seguintes funções, que retornam não zero (verdadeiro) ou zero (falso), para testar erros de um *stream*:

```
int good();        int eof();        int bad();        int fail();
```

A função *good()* retorna verdadeiro se não ocorreu erro, isto é, se as outras três funções retornaram falso. Por exemplo, se *cin.good()* retorna verdadeiro, significa que está tudo correto com a entrada de dados. A função *eof()* retorna verdadeiro se o sistema encontrou o fim do arquivo (*end-of-file*) durante uma operação de entrada. A função *fail()* retorna verdadeiro depois que uma operação de entrada ou saída falhou. Se *bad()* retorna falso, entretanto, pode ser possível se recuperar a partir do erro e continuar usando o *stream*. O código abaixo exemplifica a utilização destas funções.

```
while ( !cin.get(ch).fail() ) {
        // processa o valor de ch...
}
if ( cin.eof() ) {
        // termina o programa normalmente...
}
else if ( cin.bad() ) {
        // aviso de erro fatal...
}
else {
        cin.clear();        // limpa estado de erro
        // tenta se recuperar do erro...
}
```

Para acessar arquivos designados através de nomes, tais como arquivos armazenados em disco, utilizam-se as classes declaradas em *fstream.h*. Estas classes *ifstream*, *ofstream* e *fstream* correspondem, respectivamente, às classes *istream*, *ostream* e *iostream*. Para abrir um arquivo deve-se especificar um nome e um modo que determina como o arquivo será aberto. O modo precisa ser um (ou mais) destes valores:

- *ios::in* - especifica que o arquivo só pode ser lido;
- *ios::out* - especifica que o arquivo pode ser gravado, sendo que qualquer dado existente no arquivo é apagado e substituído pelos dados escritos no *stream*;
- *ios::app* - só pode ser usado com arquivos que podem ser gravados, pois faz toda saída para o arquivo ser anexada no final;
- *ios::ate* - faz ocorrer um posicionamento no final do arquivo quando este for aberto, mantendo assim os dados existentes;
- *ios::binary* - faz um arquivo ser aberto para as operações binárias, o que significa que nenhuma tradução de caracteres ocorrerá;
- *ios::nocreate* - faz a função *open* falhar se o arquivo ainda não existir;
- *ios::noreplace* - faz a função *open* falhar se o arquivo já existir e *ios::ate* ou *ios::app* não forem também especificados;
- *ios::trunc* - elimina o arquivo, se já existe, e o recria.

Torna-se importante comentar que criar um arquivo usando *ifstream* implica leitura, e criar um arquivo usando *ofstream* implica gravação, de modo que nesses casos não é necessário fornecer estes valores. Também deve-se atentar para o fato de que estas constantes podem ser combinadas com o operador OR (|).

Quando se declara um *stream*, pode-se abrir o arquivo correspondente fornecendo o nome e o modo como argumentos do construtor. Assim:

```
ifstream input ("intexto.doc", ios::in);
```

declara *input* como um *ifstream*, relaciona-o com o arquivo *intexto.doc* e estabelece o modo *ios::in*. Da mesma forma, os seguintes comandos declaram um *ofstream* e um *fstream*:

```
ofstream output ("outtexto.doc", ios::out);
fstream io ("iotexto.doc", ios::in | ios::out);
```

Após a criação de um canal, também pode-se associá-lo com um arquivo usando a função *open()*. Essa função é um membro de cada uma das três classes de canais. O exemplo a seguir mostra como utilizar esta função.

```
fstream canal;
canal.open ("teste.txt", ios::in | ios::out);
```

Para fechar um arquivo deve-se usar a função *close()*. Por exemplo, para fechar o arquivo associado com o canal chamado "canal", utiliza-se o comando "*canal.close();*". A função *close()* não usa nenhum parâmetro e não retorna nenhum valor.

O exemplo a seguir ilustra a criação e escrita em arquivos.

```
# include <fstream>
# include <stdlib.h>
using namespace std;
main()  {
        ofstream out ("texto.doc");
        if ( !out ) {           // se ocorreu erro
                cerr << "Nao foi possivel abrir o arquivo! \n";  exit (1);
        }
        out << "Este pequeno texto \n";
        out << "sera escrito no arquivo. \n";
        out.close();
}
```

O exemplo a seguir ilustra a abertura e leitura de arquivos.

```
# include <fstream>
# include <stdlib.h>
using namespace std;
main()  {
        ifstream in ("texto.doc");
        if ( !in ) {  // se ocorreu erro
                cerr << "Nao foi possivel abrir o arquivo! \n";
                exit (1);
        }
        const int SIZE = 81;
        char line[SIZE];
        in.getline( line, SIZE );
        cout << line << endl;
        in.getline( line, SIZE );
        cout << line << endl;
        in.close();
}
```

Também pode-se ler dados binários para/de um arquivo. Uma maneira consiste em escrever um *byte* usando usando a função *put()* e ler um *byte* usando a função *get()*. A função *get()* tem muitas formas, mas a versão mais comumente usada é mostrada aqui, juntamente com *put()*:

```
istream &get(char &ch);    istream &put(char &ch);
```

Neste caso a função *get()* lê um único caracter do canal associado, coloca esse valor em "ch" e retorna uma referência ao canal. A função *put()* escreve "ch" no canal e retorna uma referência ao canal

O próximo programa exibe o conteúdo de qualquer arquivo na tela usando a função *get()*.

```
#include <iostream>
#include <fstream>
using namespace std;
int main (int argc, char *argv[])  {
        char ch;
        if(argc!=2) {
                cout << "Uso: PR <nomearq>\n";  return 1;
        }

        ifstream ent(argv[1], ios::in | ios::binary);

        if(!ent) {
                cout << "Nao foi possivel abrir o arquivo.\n";  return 1;
        }

        while(ent) {  // ent será nulo quando eof for alcancado
                ent.get(ch);  cout << ch;
        }

        ent.close();
}
```

O próximo programa, por sua vez, usa *put()* para escrever uma string que inclui caracteres não-ASCII em um arquivo:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
        char *p = "ola' pessoal \n \r";
        ofstream saida ("teste", ios::out  | ios::binary);
        if(!saida) {
                cout << "Nao foi possivel abrir o arquivo.\n";
                return 1;
        }
        while(*p) saida.put(*p++);
        saida.close();
        return 0;
}
```

Uma segunda maneira de ler e gravar dados binários usa as funções *read()* e *write()*. Os protótipos para duas das formas mais comumente usadas são:

```
istream &read (unsigned char *buf, int num);
ostream &write (const unsigned char *buf, int num);
```

Neste caso, a função *read()* lê "num" *bytes* do canal associado e coloca-os no *buffer* apontado por "buf". A função *write()* escreve "num" *bytes* no canal associado a partir do *buffer* apontado por "buf". O programa abaixo exemplifica o uso destas funções escrevendo e depois lendo uma matriz de inteiros.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()  {
        int n[5] = {1, 2, 3, 4, 5};
        register int i;
        ofstream saida("teste", ios::out | ios::binary);
        if(!saida)
        {
                cout << "Nao foi possivel abrir o arquivo.\n";
                return 1;
        }
        saida.write((unsigned char *) &n, sizeof n);
        saida.close();
        for(i=0; i<5; i++)   // Limpa a matriz
                n[i] = 0;

        ifstream ent("teste", ios::in | ios::binary);
        ent.read((unsigned char *) &n, sizeof n);
        for(i=0; i<5; i++)   // Mostra os valores lidos do arquivo
                cout << n[i] << " ";
        ent.close();
        return 0;
}
```

É importante ressaltar que apesar dos exemplos apresentados até agora acessarem o arquivo seqüencialmente, C++ também permite que os arquivos sejam acessados diretamente ou randomicamente, de maneira que os itens possam ser lidos ou escritos em qualquer posição especificada dentro de um arquivo. Para cada arquivo o sistema mantém uma posição corrente e a posição na qual o próximo item será lido ou escrito. Pode-se acessar a posição corrente com a função *tellg()*. A posição é armazenada como um valor do tipo *streampos*, então:

```
streampos p = direct.tellg();
```

atribui a p a posição corrente do arquivo relacionado com o *stream direct*. Se mais tarde for necessário retornar para esta posição, pode-se usar

```
direct.seekg(p);
```

para determinar a posição corrente para o valor de p. O programa a seguir fornece um exemplo introdutório ao acesso randômico.

```
# include <fstream>
# include <iomanip>
# include <stdlib.h>
using namespace std;
main() {
        fstream io ("test.io", ios::in | ios::out);
        if ( !io ) {   // se ocorreu erro
                cerr << "Nao foi possivel abrir o arquivo! \n";
                exit (1);
        }
        streampos p1 = io.tellg();
        io << "The quick ";
        streampos p2 = io.tellg();
        io << "brown fox ";
        streampos p3 = io.tellg();
        io << "jump over the ";
        streampos p4 = io.tellg();
        io << "lazy dog. \n ";
        char word[6];
        io.seekg (p4);
        io >> setw(6) >> word;
        cout << word << endl;
        io.seekg (p3);
        io >> setw(6) >> word;
        cout << word << endl;
        io.seekg (p2);
        io >> setw(6) >> word;
        cout << word << endl;
        io.seekg (p1);
        io >> setw(6) >> word;
        cout << word << endl;
}
```

Torna-se interessante destacar, como já visto em alguns exemplos desta seção, que muitos sistemas operacionais permitem que parâmetros sejam especificados na linha de comando. Neste caso, para acessar os parâmetros da linha de comando em C++, deve-se declarar a *main()* como uma função com dois argumentos:

```
main ( int argc, char **argv )  { ... }
```

Quando o programa é executado, *argc* é "setado" para o número de parâmetros da linha de comando e *argv* é "setado" para um vetor de *strings*, onde cada elemento é um parâmetro da linha de comando. Por exemplo, para a linha de comando "*iostr5 test1.doc test2.doc*", *argc* é igual a 3 e os elementos de *argv* são: *argv*[0] = "iostr5", *argv*[1] = "test1.doc", *argv*[2] = "test2.doc" [3, 10].