Aula 2

Programação II

Prof. Sandino Jardim

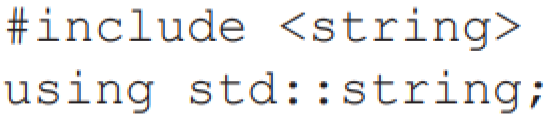
UFMT-CUA

Agenda

- Strings
- Tipos definidos pelo usuário
- Ponteiros
- Funções
    - Passagem por cópia
    - Passagem por referência

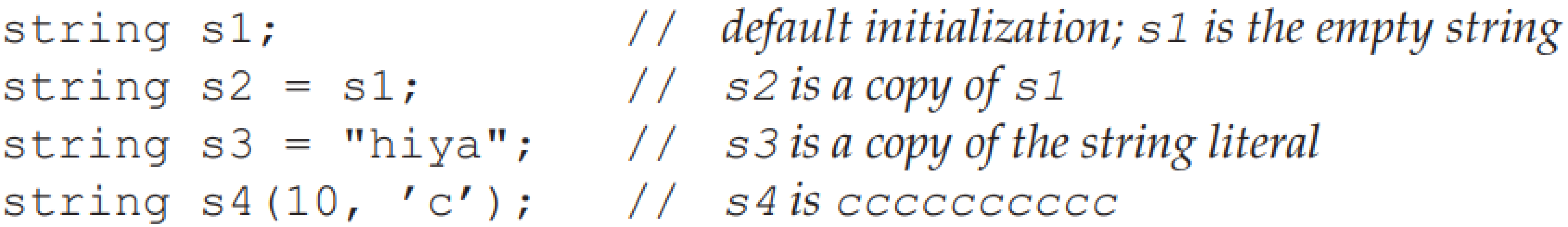
Strings

- Em C++, cadeias de caracteres são gerenciadas pela classe *string*, acessível pela biblioteca <string>



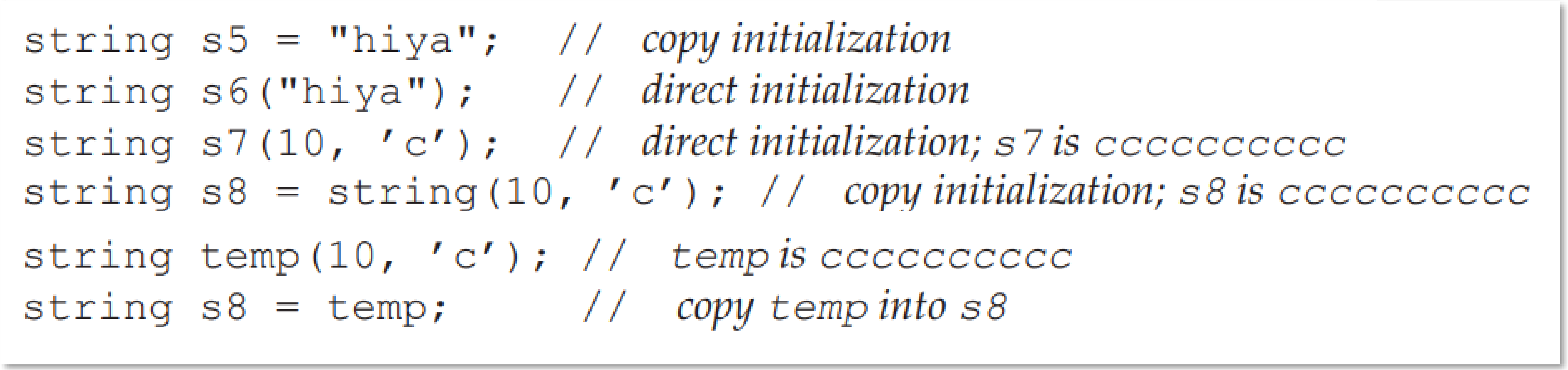
```
#include <string>
using std::string;
```

- Permite múltiplas formas de inicialização



```
string s1;              // default initialization; s1 is the empty string
string s2 = s1;         // s2 is a copy of s1
string s3 = "hiya";     // s3 is a copy of the string literal
string s4(10, ’c’);     // s4 is cccccccccc
```

Strings – outras formas de inicialização



```
string s5 = "hiya";    //  copy initialization
string s6("hiya");     //  direct initialization
string s7(10, ’c’);    //  direct initialization; s7 is cccccccccc
string s8 = string(10, ’c’); //  copy initialization; s8 is cccccccccc
string temp(10, ’c’); //  temp is cccccccccc
string s8 = temp;      //  copy temp into s8
```

Strings - inicialização

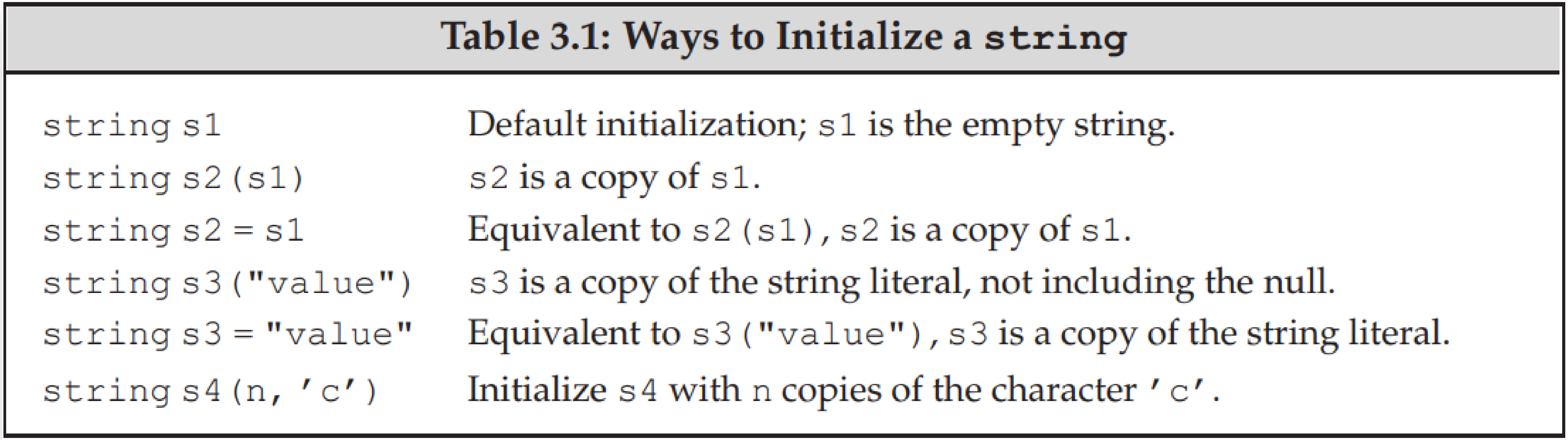
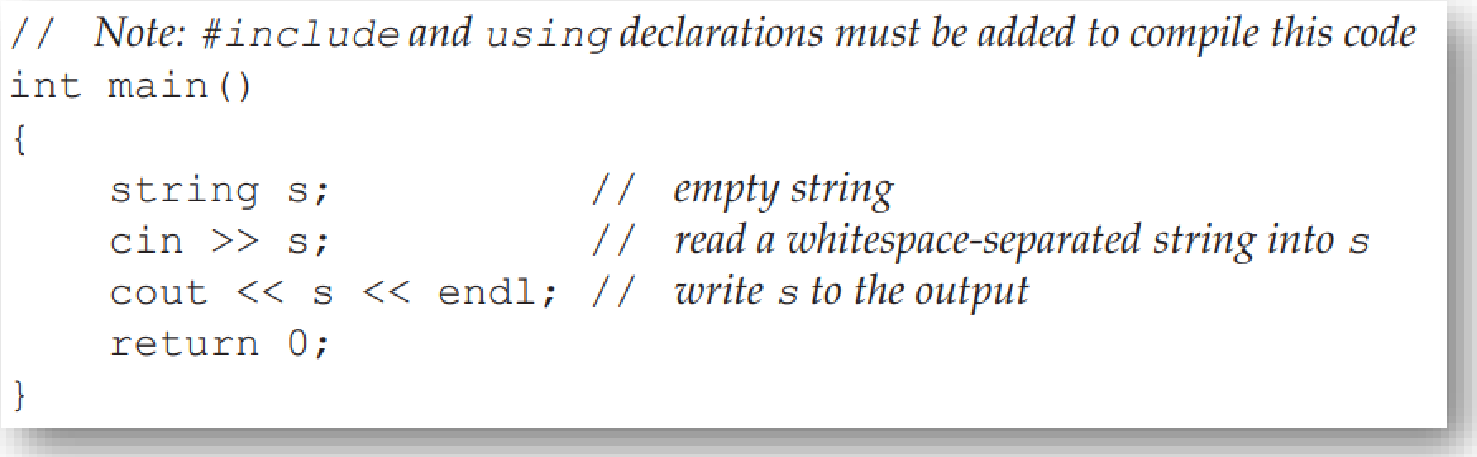


Table 3.1: Ways to Initialize a `string`

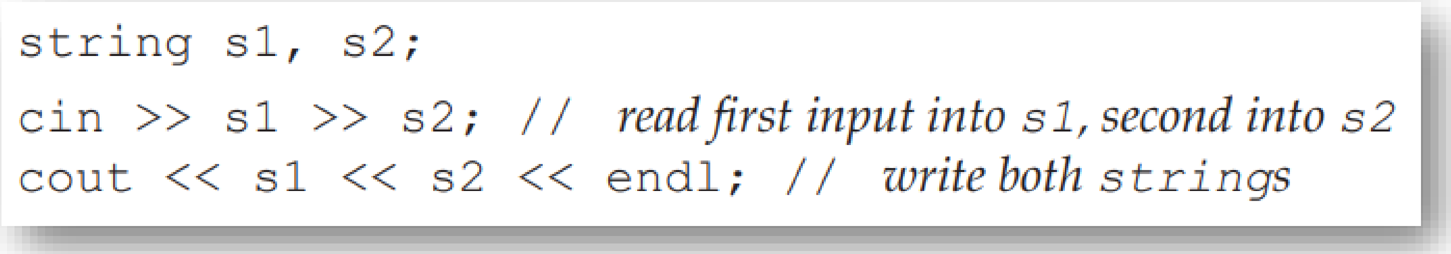
`string s1`	Default initialization; `s1` is the empty string.
`string s2(s1)`	`s2` is a copy of `s1`.
`string s2 = s1`	Equivalent to `s2(s1)`, `s2` is a copy of `s1`.
`string s3("value")`	`s3` is a copy of the string literal, not including the null.
`string s3 = "value"`	Equivalent to `s3("value")`, `s3` is a copy of the string literal.
`string s4(n, ’c’)`	Initialize `s4` with `n` copies of the character `’c’`.

Strings – Leitura e escrita

Utiliza os mesmos operadores de entrada e saída <iostream>



```
//  Note: #include and using declarations must be added to compile this code
int main()
{
    string s;            //  empty string
    cin >> s;            //  read a whitespace-separated string into s
    cout << s << endl;   //  write s to the output
    return 0;
}
```



```
string s1, s2;
cin >> s1 >> s2;  //  read first input into s1, second into s2
cout << s1 << s2 << endl;  //  write both strings
```

String – operações

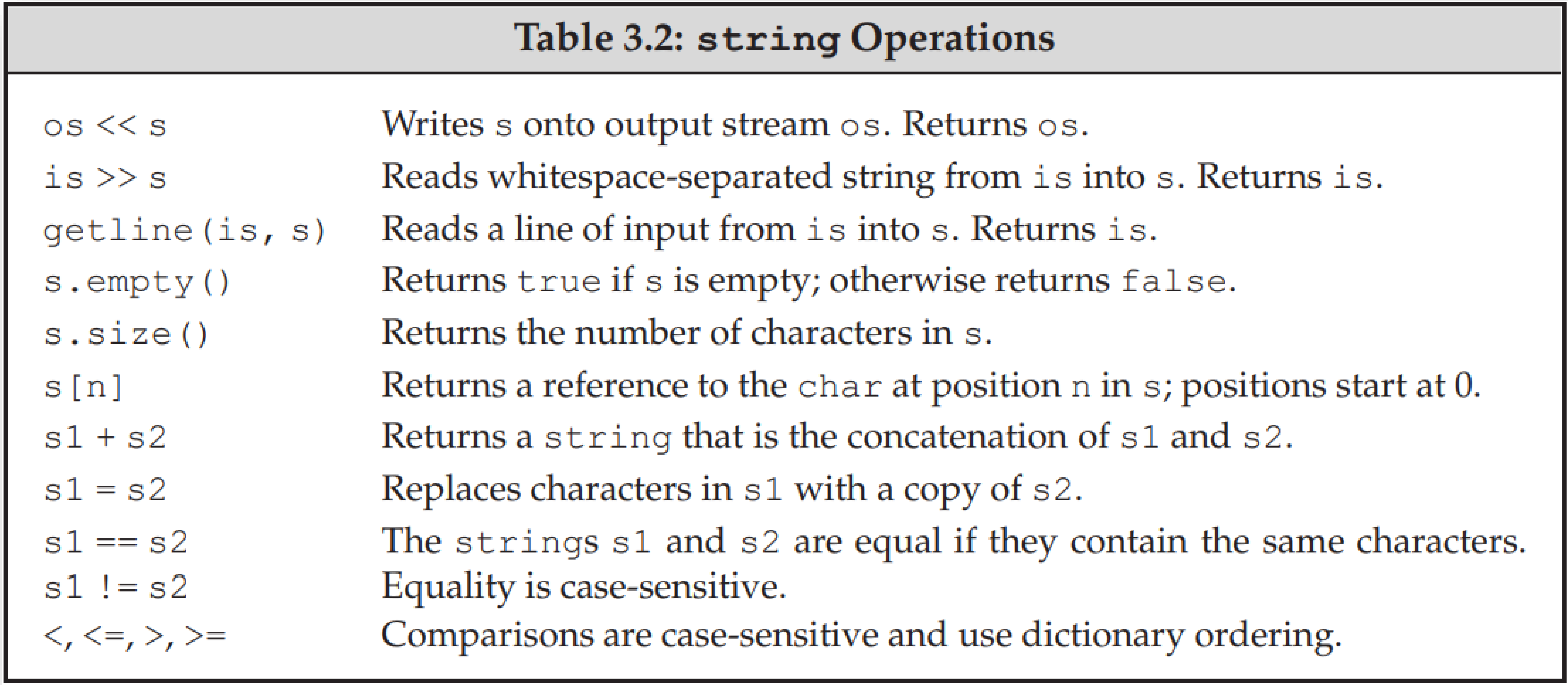
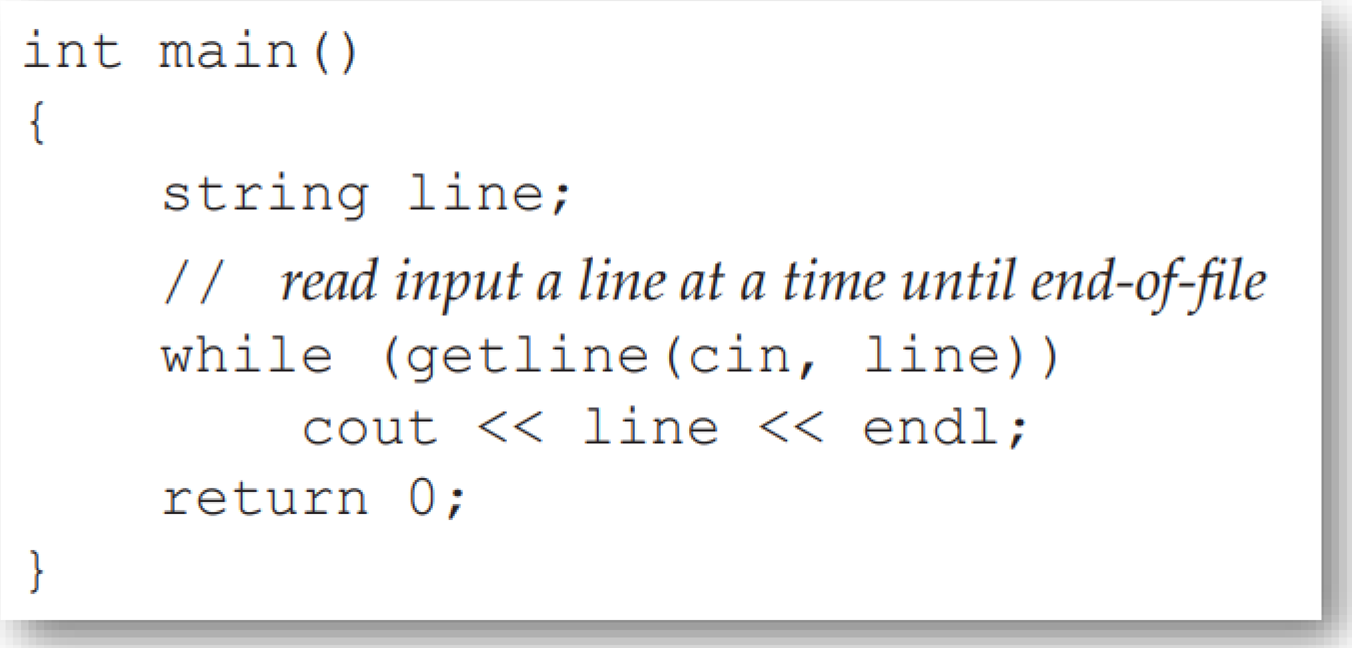


Table 3.2: `string` Operations

`os << s`	Writes `s` onto output stream `os`. Returns `os`.
`is >> s`	Reads whitespace-separated string from `is` into `s`. Returns `is`.
`getline(is, s)`	Reads a line of input from `is` into `s`. Returns `is`.
`s.empty()`	Returns `true` if `s` is empty; otherwise returns `false`.
`s.size()`	Returns the number of characters in `s`.
`s[n]`	Returns a reference to the `char` at position `n` in `s`; positions start at 0.
`s1 + s2`	Returns a `string` that is the concatenation of `s1` and `s2`.
`s1 = s2`	Replaces characters in `s1` with a copy of `s2`.
`s1 == s2`	The `strings` `s1` and `s2` are equal if they contain the same characters.
`s1 != s2`	Equality is case-sensitive.
`<, <=, >, >=`	Comparisons are case-sensitive and use dictionary ordering.

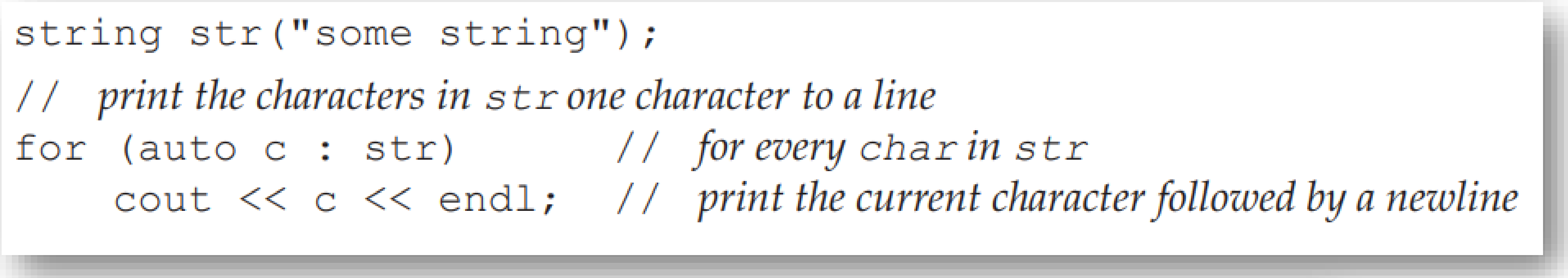
String – leitura/escrita de linhas

- Para leitura de *strings* separadas por espaço, descartando \n

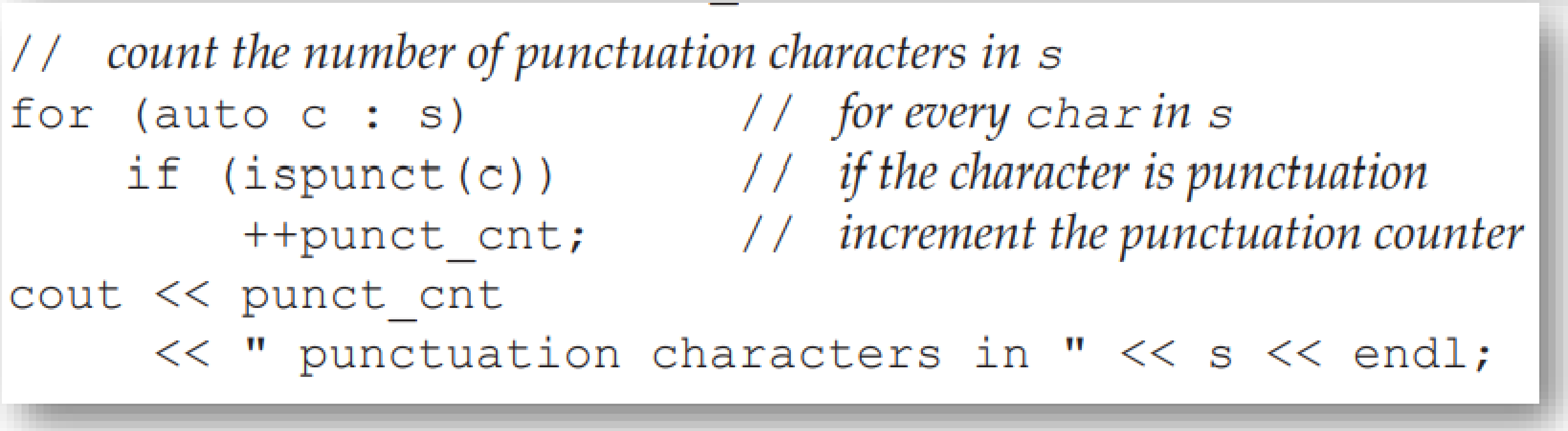


```
int main()
{
    string line;
    //  read input a line at a time until end-of-file
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
```

Strings – acessando caracteres



```
string str("some string");
// print the characters in str one character to a line
for (auto c : str)        // for every char in str
    cout << c << endl;    // print the current character followed by a newline
```



```
// count the number of punctuation characters in s
for (auto c : s)            // for every char in s
    if (ispunct(c))         // if the character is punctuation
        ++punct_cnt;        // increment the punctuation counter
cout << punct_cnt
     << " punctuation characters in " << s << endl;
```

Strings – funções para caracteres

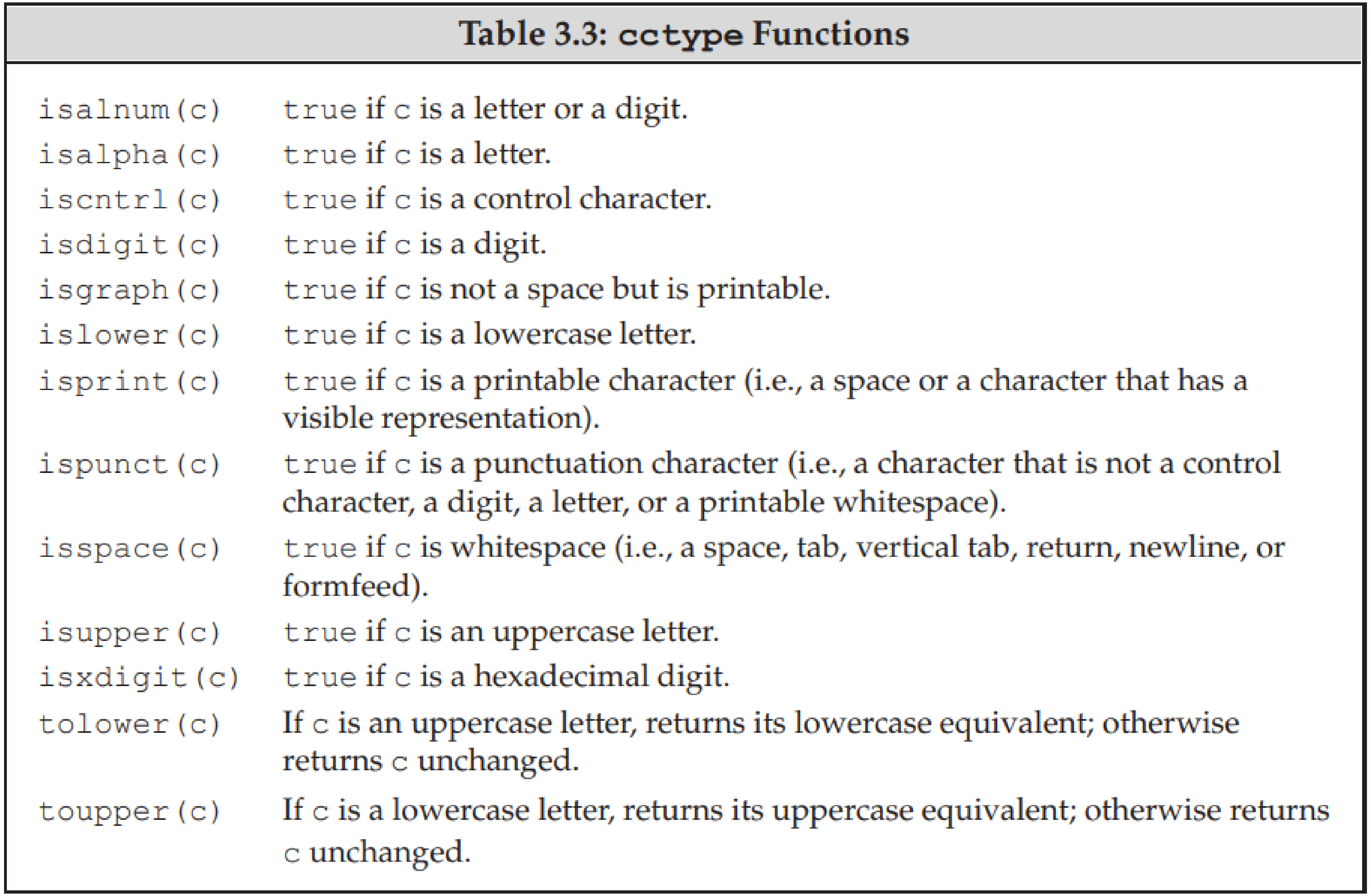
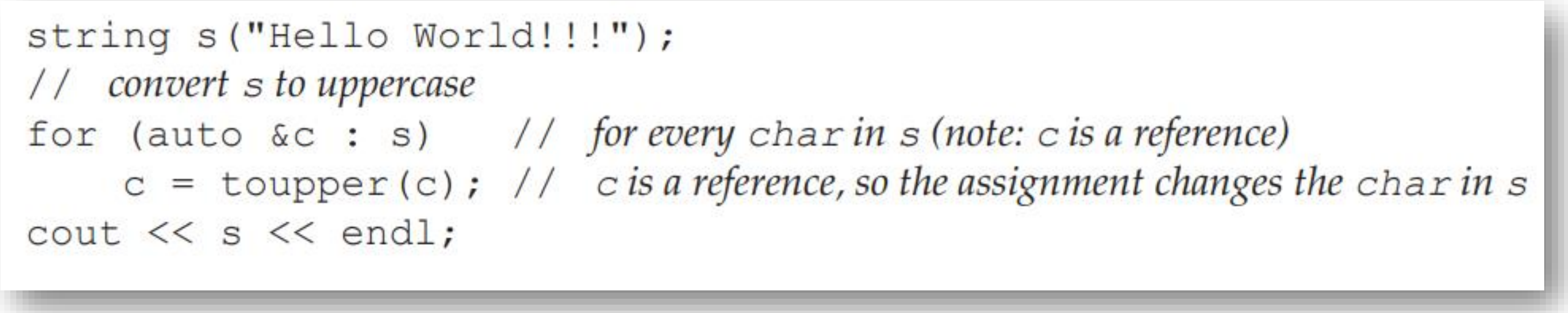


Table 3.3: cctype Functions

isalnum(c)	true if c is a letter or a digit.
isalpha(c)	true if c is a letter.
iscntrl(c)	true if c is a control character.
isdigit(c)	true if c is a digit.
isgraph(c)	true if c is not a space but is printable.
islower(c)	true if c is a lowercase letter.
isprint(c)	true if c is a printable character (i.e., a space or a character that has a visible representation).
ispunct(c)	true if c is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace).
isspace(c)	true if c is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed).
isupper(c)	true if c is an uppercase letter.
isxdigit(c)	true if c is a hexadecimal digit.
tolower(c)	If c is an uppercase letter, returns its lowercase equivalent; otherwise returns c unchanged.
toupper(c)	If c is a lowercase letter, returns its uppercase equivalent; otherwise returns c unchanged.

Strings – alterando caracteres com *range-for*

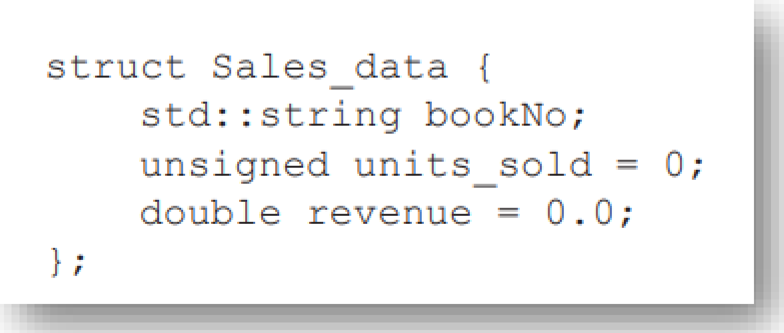
- Usando operador de referência & é possível alterar string original



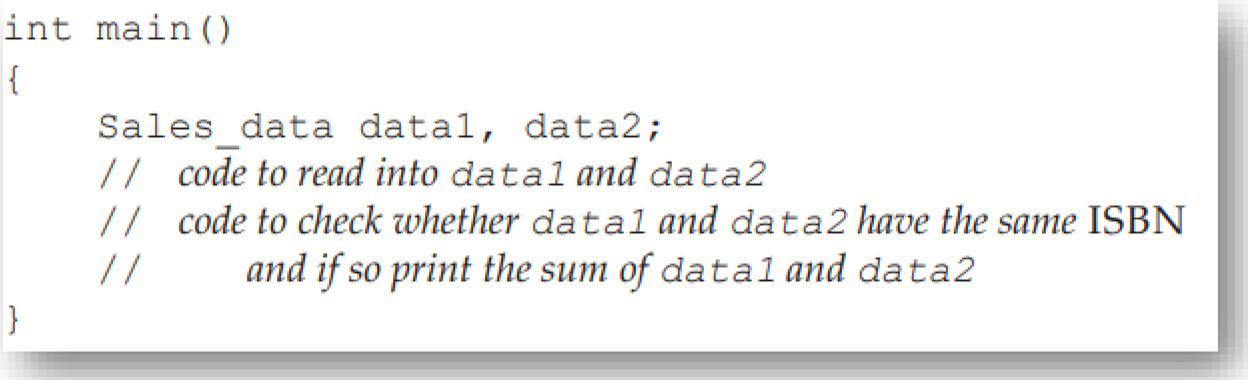
```
string s("Hello World!!!");
// convert s to uppercase
for (auto &c : s)    // for every char in s (note: c is a reference)
    c = toupper(c); // c is a reference, so the assignment changes the char in s
cout << s << endl;
```

Tipos definidos pelo usuário

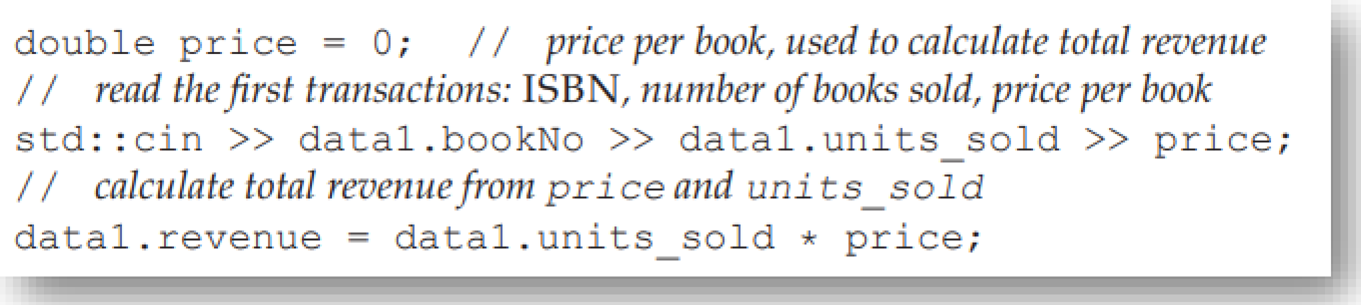
- Assim como C, C++ permite a definição de estruturas de dados que permitem agrupar elementos relacionados



```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```



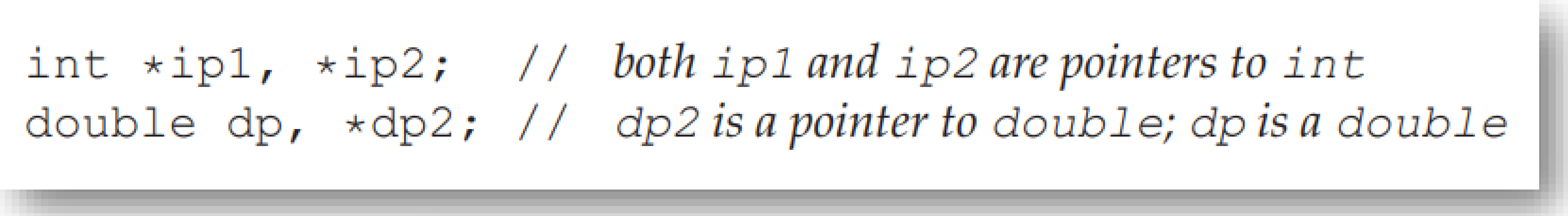
```
int main()
{
    Sales_data data1, data2;
    // code to read into data1 and data2
    // code to check whether data1 and data2 have the same ISBN
    //      and if so print the sum of data1 and data2
}
```



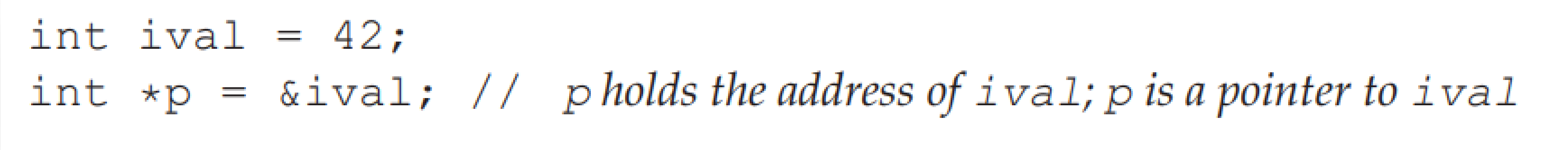
```
double price = 0;   // price per book, used to calculate total revenue
// read the first transactions: ISBN, number of books sold, price per book
std::cin >> data1.bookNo >> data1.units_sold >> price;
// calculate total revenue from price and units_sold
data1.revenue = data1.units_sold * price;
```

Ponteiros

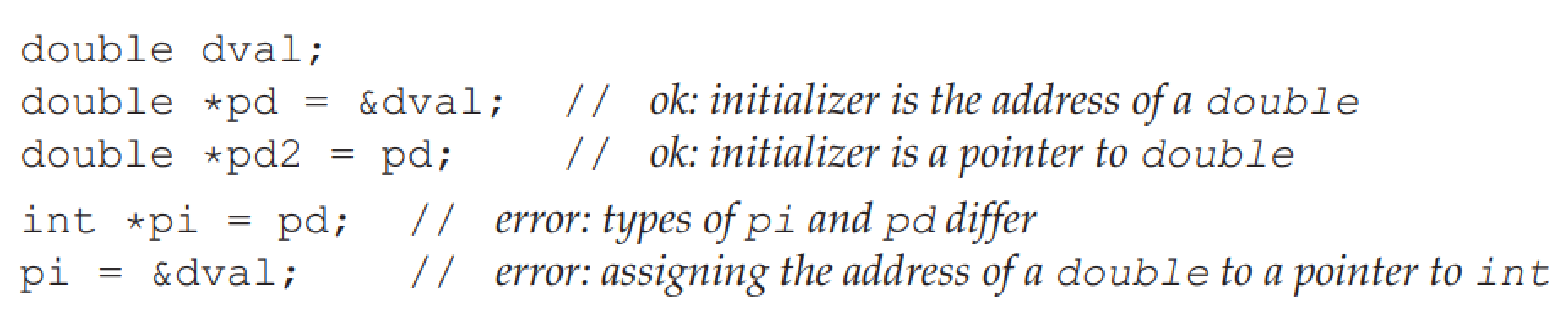
- Tipos de dados que permitem acesso indireto a outras variáveis



```
int *ip1, *ip2;   //  both ip1 and ip2 are pointers to int
double dp, *dp2;  //  dp2 is a pointer to double; dp is a double
```

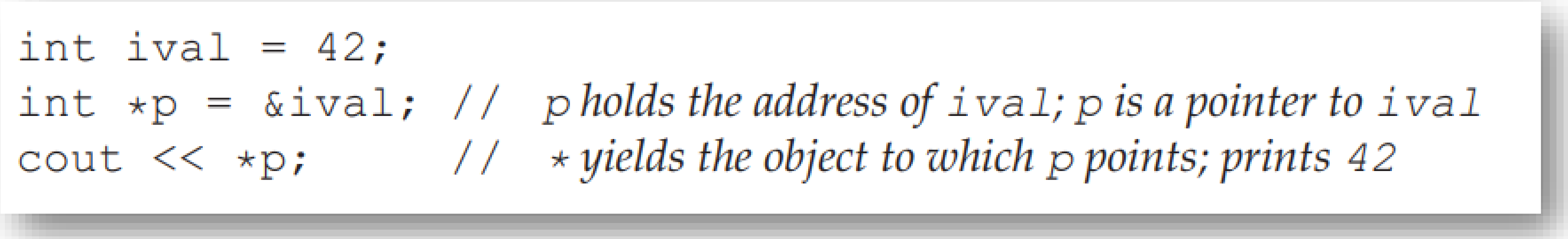


```
int ival = 42;
int *p = &ival; //  p holds the address of ival; p is a pointer to ival
```

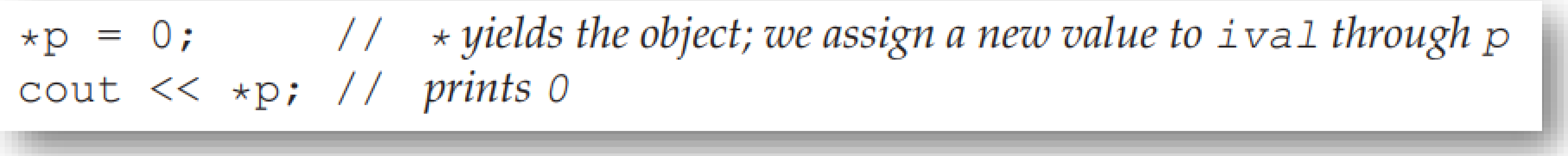


```
double dval;
double *pd = &dval;    //  ok: initializer is the address of a double
double *pd2 = pd;      //  ok: initializer is a pointer to double
int *pi = pd;     //  error: types of pi and pd differ
pi = &dval;       //  error: assigning the address of a double to a pointer to int
```

Ponteiros – acessando objetos

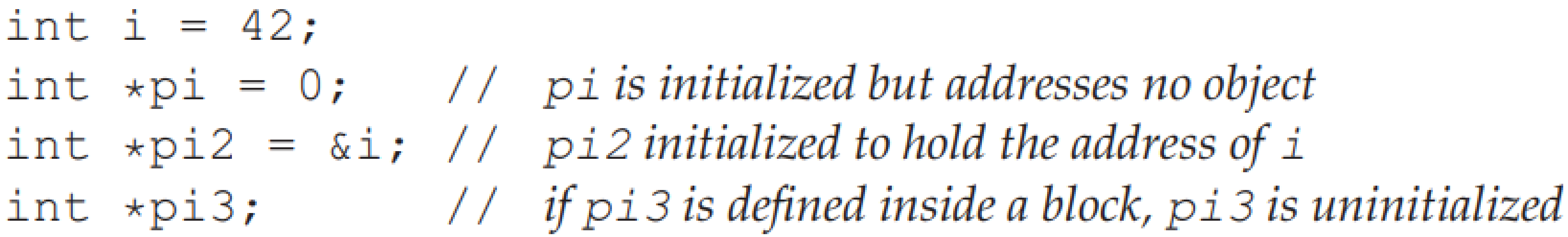


```
int ival = 42;
int *p = &ival; //  p holds the address of ival; p is a pointer to ival
cout << *p;     //  * yields the object to which p points; prints 42
```

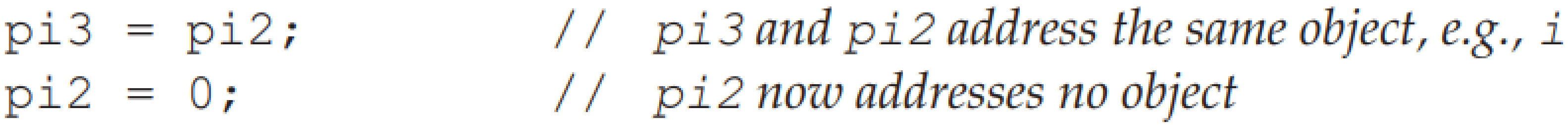


```
*p = 0;         //  * yields the object; we assign a new value to ival through p
cout << *p; //  prints 0
```

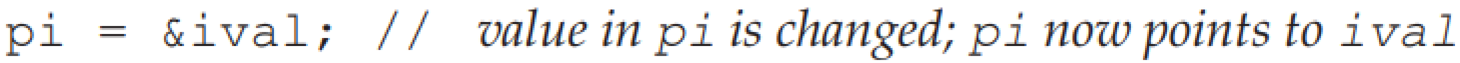
Ponteiros - atribuições



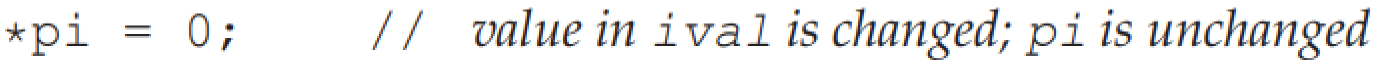
```
int i = 42;
int *pi = 0;     //  pi is initialized but addresses no object
int *pi2 = &i;  //  pi2 initialized to hold the address of i
int *pi3;        //  if pi3 is defined inside a block, pi3 is uninitialized
```



```
pi3 = pi2;          //  pi3 and pi2 address the same object, e.g., i
pi2 = 0;            //  pi2 now addresses no object
```



```
pi = &ival;  //  value in pi is changed; pi now points to ival
```

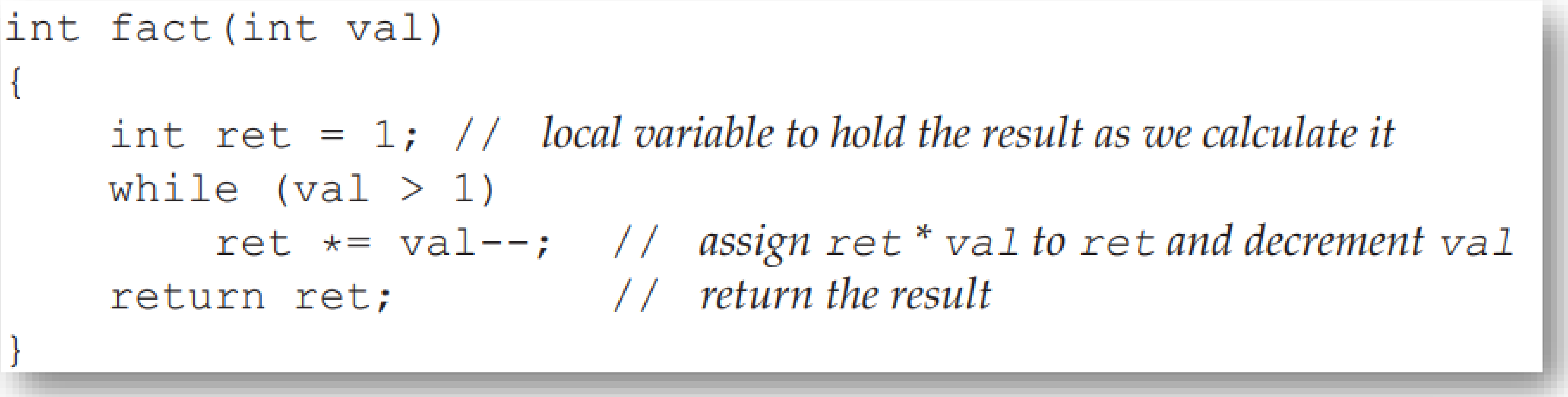


```
*pi = 0;       //  value in ival is changed; pi is unchanged
```

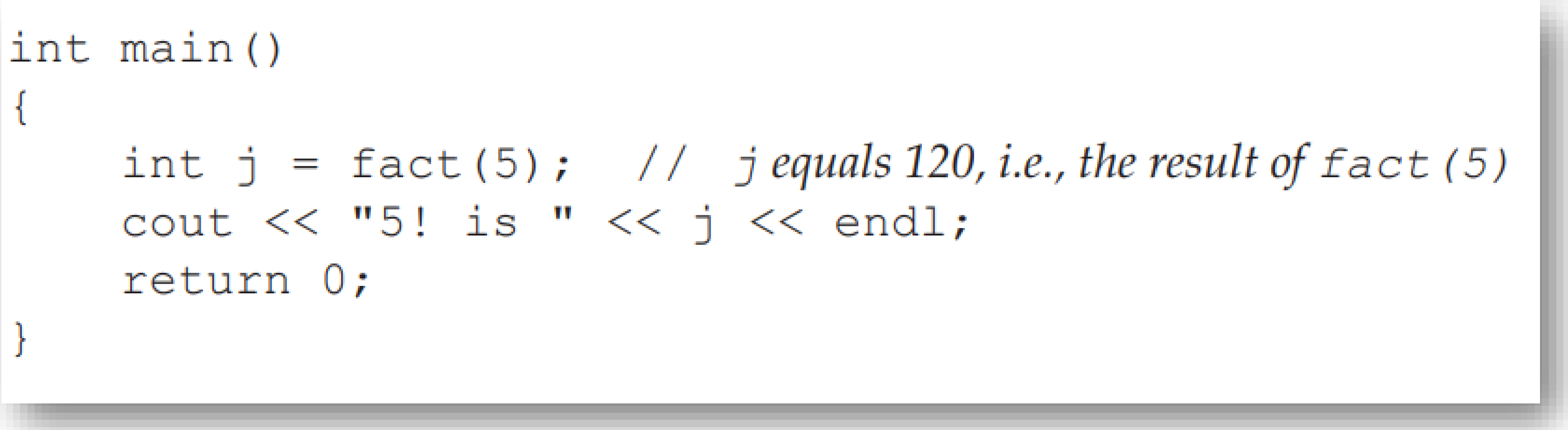
Funções – passagem de parâmetros

- Cada vez que uma função é invocada, seus parâmetros são criados e inicializados pelos argumentos passados na chamada
  - Quando um parâmetro é uma referência, dizemos que o argumento é **passado por referência**
  - Quando o valor do argumento é copiado, parâmetro e argumento são objetos independentes, configurando uma **passagem por valor**

Funções – passagem por valor



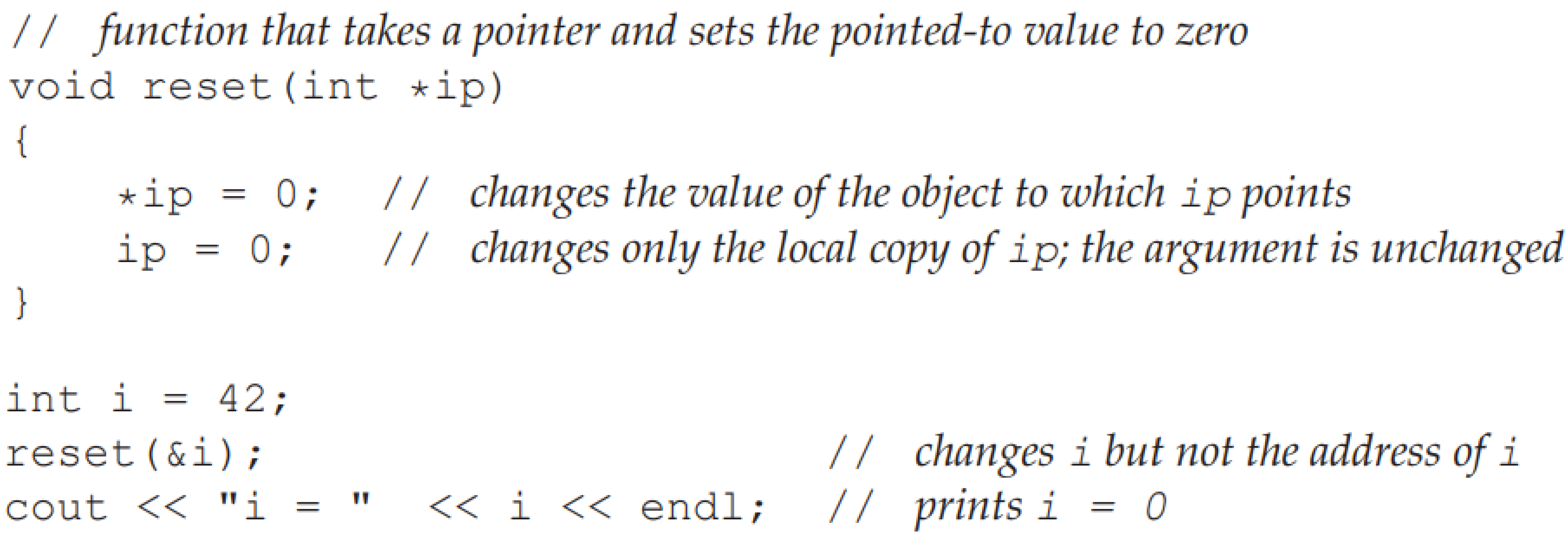
```
int fact(int val)
{
    int ret = 1; //  local variable to hold the result as we calculate it
    while (val > 1)
        ret *= val--;   //  assign ret * val to ret and decrement val
    return ret;         //  return the result
}
```



```
int main()
{
    int j = fact(5);   //  j equals 120, i.e., the result of fact(5)
    cout << "5! is " << j << endl;
    return 0;
}
```

Funções – ponteiros como parâmetro

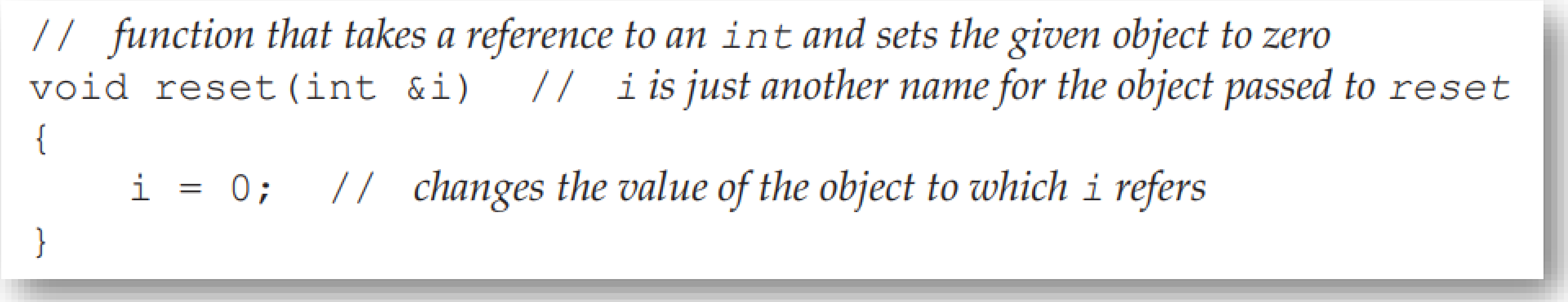
- Ponteiros se comportam como qualquer outro tipo não-referenciável
- Entretanto, um ponteiro também dará acesso indireto ao objeto apontado



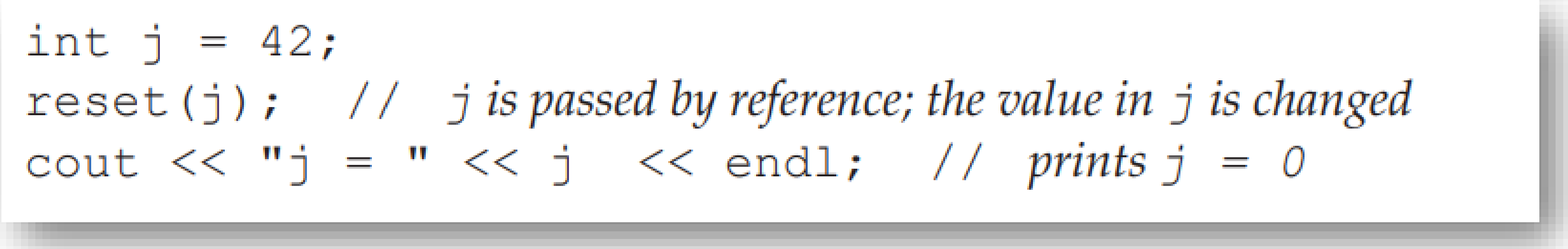
```
// function that takes a pointer and sets the pointed-to value to zero
void reset(int *ip)
{
    *ip = 0;  // changes the value of the object to which ip points
    ip = 0;   // changes only the local copy of ip; the argument is unchanged
}

int i = 42;
reset(&i);                    // changes i but not the address of i
cout << "i = "  << i << endl; // prints i = 0
```

Funções – passagem por referência



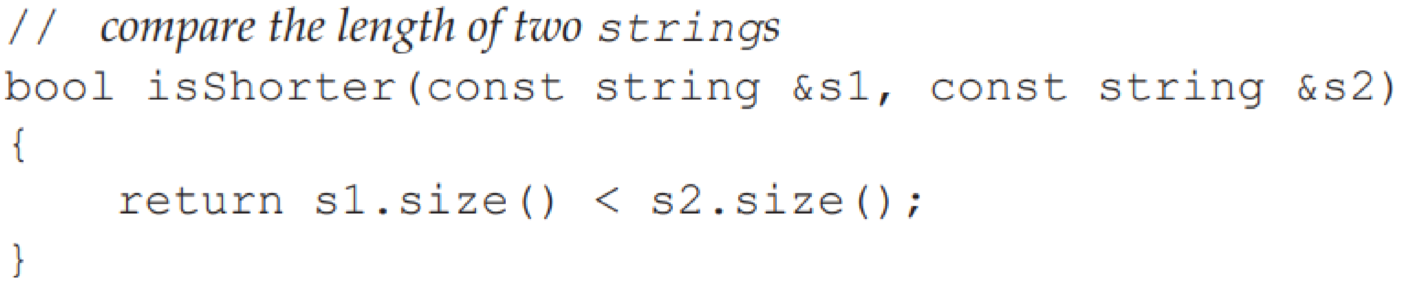
```
// function that takes a reference to an int and sets the given object to zero
void reset(int &i)   // i is just another name for the object passed to reset
{
    i = 0;   // changes the value of the object to which i refers
}
```



```
int j = 42;
reset(j);   // j is passed by reference; the value in j is changed
cout << "j = " << j  << endl;   // prints j = 0
```

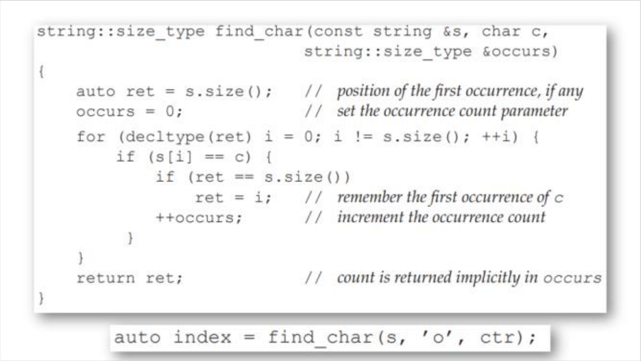
Funções – passagem por referência

- Utilidades:
  - Evitar cópia de dados potencialmente grandes



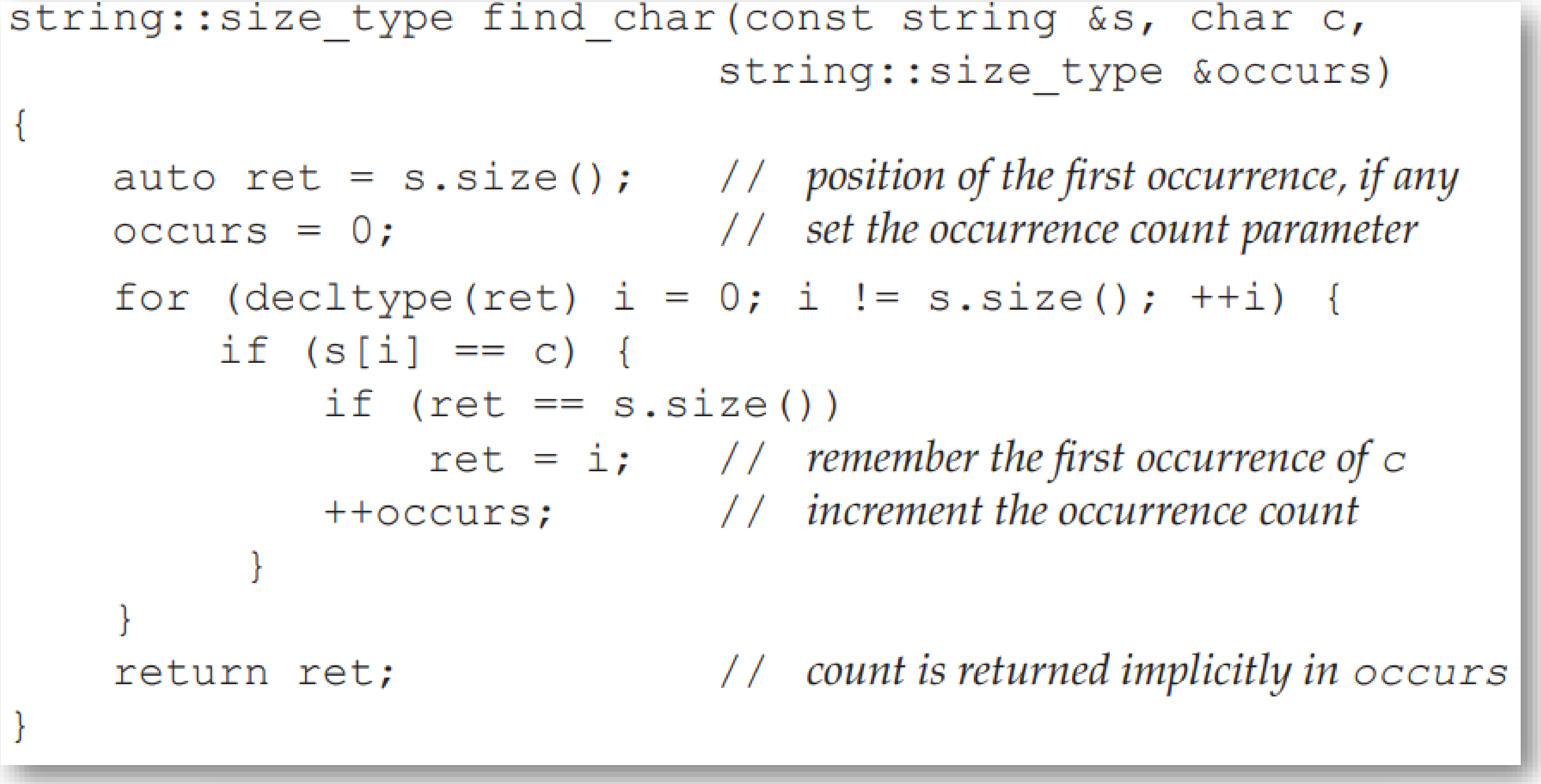
```
//  compare the length of two strings
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

  - “Retornar” mais de um parâmetro

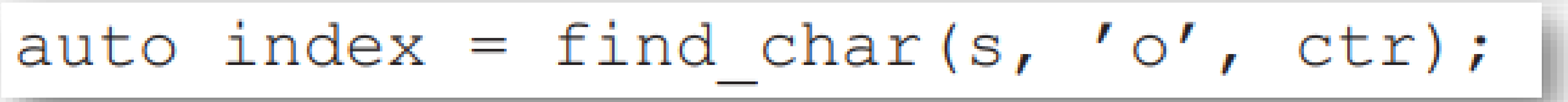


```
string::size_type find_char(const string &s, char c,
                            string::size_type &occurs)
{
    auto ret = s.size();    //  position of the first occurrence, if any
    occurs = 0;             //  set the occurrence count parameter
    for (decltype(ret) i = 0; i != s.size(); ++i) {
        if (s[i] == c) {
            if (ret == s.size())
                ret = i;    //  remember the first occurrence of c
            ++occurs;       //  increment the occurrence count
        }
    }
    return ret;             //  count is returned implicitly in occurs
}
```

```
auto index = find_char(s, 'o', ctr);
```



```
string::size_type find_char(const string &s, char c,
                           string::size_type &occurs)
{
    auto ret = s.size();    // position of the first occurrence, if any
    occurs = 0;             // set the occurrence count parameter
    for (decltype(ret) i = 0; i != s.size(); ++i) {
        if (s[i] == c) {
            if (ret == s.size())
                ret = i;    // remember the first occurrence of c
            ++occurs;       // increment the occurrence count
         }
    }
    return ret;             // count is returned implicitly in occurs
}
```



```
auto index = find_char(s, 'o', ctr);
```