Chapter

5

Characters and Strings

Individual characters in C++ are represented by the data type named `char` (usually pronounced "tchar", not "kar"). In memory, these values are represented by assigning each character an 8-bit integer code called an ASCII code.

You write **character literals** by enclosing each character **in single quotes**. Thus, the literal `'A'` represents the internal code of the uppercase letter `A`. In addition, C++ allows you to write **special characters** in a multi-character form beginning with a back-slash (\). This form is called **an escape sequence**.

Examples include the **newline** (`\n`), the **tab** (`\t`), and a double-quote inside a string literal (`\"`). Here are the escape sequences that C++ supports.

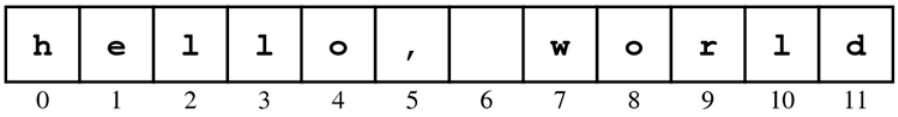
Character Functions

It is useful to have tools for working with individual characters. The `<cctype>` header contains a variety of functions that do that. There are two kinds of functions.

- **Predicate classification functions** test whether a character belongs to a particular category. Calling `isdigit(ch)` returns `true` if `ch` is one of the digit characters in the range between `'0'` and `'9'`. Similarly, `isspace(ch)` returns `true` if `ch` is any of the characters that appear as white space on a display screen, such as spaces and tabs.
- **Conversion macros** make it easy to convert between uppercase and lowercase letters. Calling `toupper('a')`, for example, returns the character `'A'`. If the argument is **not** a letter, the function returns it unchanged, so that `tolower('7')` returns `'7'`.

Selecting Characters

Positions in a `string` are **numbered** (or indexed) starting at `0`. The characters in the `string "hello, world"` are numbered like:



The numbers called the **index** or **subscript**; they must be positive (unlike Python where subscripts can be negative). Indexes start at `0` because it represents how many places you need to travel from the beginning if the string to get to the element you are interested in. To retrieve the `'e'`, you have to travel one character from the beginning.

The `<string>` library has four ways to select characters from a non-empty `string`:

- Use the **subscript operator** like this: `cout << str[0];`
- Use the member function `at()` like this: `cout << str.at(0);`
- Use the members `front()` and `back()` in C++ 11s: `cout << str.front();`

If the `string` variable `str` contains `"hello, world"`, all of these expressions refer to the character `'h'` at the beginning of the string.

The `at()` member function makes sure the index is **in range**; the subscript operator does not. What happens when a subscript is out of range **is undefined**. You should generally use `at()` (similar to `charAt()` in Java).

Modifying Characters

Selecting an individual character in a `string` returns a **reference** to the character in the `string` instead of a copy of that character. This is different than Java's `charAt()`. You may **assign a new value** to that reference. For example:

```
str[0] = 'H';
str.at(0) = 'H';
```

Both lines change the value from `"hello, world"` to `"Hello, world"`.

Substrings

To create a new `string`, initialized with only a portion of an existing `string` (called **a substring**), use the member function named `substr()` which takes two parameters:

- the index of the **first character** you want to select
- the desired **number of characters**.

Calling `str.substr(start, n)` creates a new `string` by extracting `n` characters from `str` starting at the index position specified by `start`. For example, if `str` contains the string `"hello, world"`, then this prints the three-character substring `"ell"`.

```
string str{"hello, world"};
```

```
cout << str.substr(1, 3) << endl;
```

C++ begin at **0**, so the character at index position **1** is the character **'e'**.

Be careful with the **substr()** function, when switching between Java and C++. In Java, the second parameter to its **substring()** method is the ending index (like the **end()** iterator in C++). In C++, though, it is the number of characters in the returned substring. This can lead to hard-to-find bugs (and crashes).

The second argument in the **substr()** member function is **optional**; if missing, **substr()** returns the substring that starts at the **index** and continues to the end:

```
string str{"hello, world"};
cout << str.substr(7) << endl;
```

This returns the **string "world"**. The fragment below uses **substr()** to print the second half of **str**, which includes the middle character if the size of **str** is odd:

```
string str{"hello, world"};
cout << str.substr(str.size() / 2) << endl;
```

When using the **substr(start, end)** version of **substr()**, if **n** is supplied but fewer than **n** characters follow the specified starting position, **substr()** returns characters only up to the end of the original **string**, instead of causing a runtime error. If, however, **start** is beyond the length of the **string**, **you will** get an error.

Searching a string

To search for both characters and substrings, the **string** class contains a member function **find()**, which comes in several forms. The simplest form looks like this:

```
str.find(T target);
```

The argument **target** is the content you're looking for. **T** may be a **string** **or** a **char** or a **string** **literal**. The function searches through **str** looking for the **first occurrence** of **target**. If it **is found**, **find()** returns the index at which the match begins. If you want to find the **last occurrence** of target, use **rfind()** instead.

If **target** is not found, then **find()**returns the constant named **string::npos**. This is defined as part of the **string** class and therefore requires the **string::** qualifier. This is a good candidate for a **named constant** in your code:

```
const auto NOT_FOUND = string::npos;
```

The **find()** member function takes an optional second argument to indicate the index at which to start. Both styles of the **find()** member function are illustrated here:

```
string str{"hello, world"};
auto a = str.find('o');         // char, 4
auto b = str.rfind("o");        // string, 8
auto c = str.find('l', 4);      // 10
auto d = str.find("waldo");     // string::npos
```

The **find()** member functions consider uppercase and lowercase characters to be different. Unlike Java, there is no built-in **toUpperCase()** or **toLowerCase()** member function in the **string** class.

Other Forms

In addition to **find()** and **rfind()**, you can find the position of the first (or last) occurrence of a character that **appears in a set** or that **doesn't** appear in a set. Here are some examples:

```
string s{"\"Hooray\", the crowd cheered!"};
auto a = s.find_first_of("aeiou"); // first lc vowel
auto b = s.find_last_of("\",.!:;"); // last punctuation
auto c = s.find_first_not_of(" \t\n"); // non-whitespace
```

Reference Types

Library types, like string, and the built-in primitive types, like int and double, are called **value types**. In C++ such variables are "boxes" that **contain** data.

C++ also has several **derived types**:

- **pointers**, which contain the address of a variable,
- **arrays**, which contain a sequence of variables
- **references**, which provide an **alias** or alternate name for an existing variable

A reference name is an alternate name for an existing object. An **alias** if you like. Here's an example of a variable **n** and its alias **r**.

```
int n = 3;
int& r = n; // r is an alias for n
r = 42;     // n is also now 42
```

Here, **r** is simply an alternate name for **n**. It **is not** a new variable. Under the hood, the compiler implements references using pointers. However, even if you understand how pointers work, you should try not to get the two concepts confused.

No Conversions

Unlike value-type variables, references have **no implicit conversions**. For instance, the following code will not compile, because **x** is an **int**, but **rx** is a reference to a **double**. If **rx** were a **double** instead of a **double&** then **x** would be promoted and stored in **rx**.

```
int x = 3;
double& rx = x;  // ILLEGAL; x is not a double
```

Constant References

References must refer to an ***lvalue*** of exactly the same type, but **constant references** may refer to **literals** or **temporary** values. Here are some examples:

```
int& lit = 3;               // ILLEGAL
const int& lit2 = 3;        // OK
string& str = "OK";         // ILLEGAL
const string& str2 = "OK";  // OK
```

Reference Parameters

When you pass a variable to a function, the function receives **a copy** of the calling value or **argument**. Assigning to a parameter variable changes the parameter but has no effect on the argument. Consider this function to set a variable to zero:

```
void toZero(int var)
{
    var = 0;
}
```

If you call the procedure like this:

```
setToZero(x);
```

the parameter variable named **var** is initialized **with a copy** of whatever value is stored in **x**. Making a copy of arguments when calling a function, is known as **pass by value** or **call by value**, and the parameter **x** is known as a **value parameter**.

The assignment statement ***var = 0;*** inside the function sets **the parameter variable var** to **0** but leaves **x** unchanged in the calling program.

If you want to change the value of the calling argument, you can change the parameter from a value parameter into a **reference parameter** by adding an ampersand between the type and the name in the function header:

```
void toZero(int& var)
{
    var = 0;
}
```

Unlike value parameters, **reference parameters are not copied.** Instead, the function treats `var` as **a reference to the original variable,** which means that the memory used for that variable is shared between the function and its caller.

String Parameters

Reference parameters don't make a copy, so they are **much more efficient** when copying a class-type argument such as the `string`. Whenever you pass a `string` as an argument to a function, use `const string&` for the parameter if the function will not modify the calling argument, and simply `string&` if it will.

You can add these C++11 **type alias declarations** to your programs if you like:

```
using stringIn = const string&; // input string
using stringRef = string&;      // output string
```

C++17 added the `string_view` type which may be even more efficient for some input string operations. However, it also has several pitfalls, so I'd suggest sticking to these rules until you are more experienced.

Finish Up

- Complete the **reading exercises** (REX) for this chapter.
- Complete the **homework** using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using `make submit`.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.