ECE 150 Characters and strings

Douglas Harder

December 2023

1 Characters

A character is one of four primitive data types in C++, including also integers, floating-point numbers and Boolean values.

Not on the examination

You can use both `signed char` and `unsigned char` to store an integer within the ranges −128 to 127 and 0 to 255, respectively, in one byte. These types allow you to perform arithmetic operations on variables declared as such. It's important to explicitly specify whether a `char` is `signed` or `unsigned` because the C++ standard does not mandate that `char` be either signed or unsigned by default; this depends on the compiler and platform.
In contrast, `signed int` and `int` describe the same data type, which is why we haven't focused on this distinction until now. While using `signed char` and `unsigned char` for storing small integers is common in certain areas, such as embedded systems, this course does not cover those specific use cases.

A character is represented in C++ by the type `char`. This occupies exactly one byte, meaning that it can take on values from `0b00000000` to `0b11111111`, `0x00` to `0xff`, or 0 to 255. Consequently, each character can be represented by two hexadecimal characters.

Each different alphabetic character, digit, or symbol on the keyboard is represented by a different binary value between `0x00` and `0xff`. For example, the character `A` is represented by `0x41`, `B` by `0x42`, etc. You can view all the characters from `0x00` to `0x7f` by looking up any ASCII table.

To include a character explicitly in your source code, you must use the apostrophe; for example, `'a'` or `'3'` or `'%'`.

A variable can store a character if it is declared to be of the type `char`:

```
char identifier{ 'a' };
```

Because the C++ programming language uses the apostrophe `'` to delimit a character, we need to somehow indicate that an actual apostrophe character is not the C++ symbol used to delimit characters. We do this using the backslash:

```
char apostrophe{ '\'' };
```

The backslash is not the backslash character, but rather, it tells the compiler that the next character must be interpreted differently. This is called an escape sequence, so for example, we have the following common escape sequences:

```
char backslash{ '\\' };
char new_line{ '\n' };
char tab{ '\t' };
```

You can test if a character `char ch` is a digit if `('0' <= ch) && (ch <= '9')`, a lower-case letter if `('a' <= ch) && (ch <= 'z')`, and an upper-case letter if `('A' <= ch) && (ch <= 'Z')`.

Not on the examination

Other programming languages have different escape sequences; for example, in XML, the most significant characters are `<`, `>`, and `&`. The first two are used to delimit tags; for example

```
<p>This is a paragraph with <b>bold</b> and
<i>italicized</i> text, together with a
<a href="https://uwaterloo.ca/">hyperlink</a>.
</p>
```

The character that starts an escape sequence is the `&` and ends in a semicolon, so if you want an explicit angled bracket or ampersand symbol, you use `<`, `>` or `&`, respectively. You get Greek characters with `&alpha;`. Alternatively, you can give a number, so to get a ℜ, you use `&real;` or you can use the corresponding number: `&#8476;` or, using hexadecimal digits, `&#x211C;` (as $12 + 16 + 16^2 + 2 \cdot 16^3 = 8476$.

Not on the examination

Given a character `ch`, then `ch + 1` is the next ASCII character. For example, this runs through all printable ASCII characters:

```
for ( char ch{ 32 }; ch <= 126; ++ch ) {
    std::cout << ch;

    // Go to the next line after each 19 characters
    if ( ch == (32 + 47) ) {
        std::cout << std::endl;
    }
}

std::cout << std::endl;
```

when compiles prints out

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Not on the examination

There are $126 - 32 = 94$ printable characters not including the space, so it is therefore possible to represent four bytes using five printable characters, as $2^{4 \cdot 8} = 4294967296 < 85^5 = 4437053125$.

Not on the examination

If a character is a lower-case letter, you can change it to an upper-case letter by setting Bit 5 to 1 (recalling that Bit 0 is the least-significant bit), and you can you therefore change an upper-case letter to a lower case letter by setting bit 5 to 0:

```
char lower{ 'h' };
char bit_5{ 1 << 5 }; // 0b00100000 or see bit shifting operations

// For these, see bitwise operations
char upper{  static_cast<char>( lower & ~bit_5 ) };
char lower2{ static_cast<char>( upper |  bit_5 ) };
std::cout << lower << upper << lower2 << std::endl;
```

should print `hHh`.

2 The null character

One character of significance is the *null character*. This is not a space, but rather the character equal to zero:

```
char null_character{ '\0' };
```

This is the character with all bits set to zero, so `0x00`. This is a special unprintable character and is not the space character, which has an ASCII value of `0b00100000` or `0x20`.

Not on the examination

You can simply assign a character the value `0` to achieve the null character, but if you are using the type `char` to store characters, it is easier for a reader to understand setting a character to `'\0'` instead of just setting a character to `0`.

3 Strings

A "string" is a string of zero or more characters that represents a sequence of characters that can be printed to, for example, the screen.

A literal string can be entered into your source code using double-quotes:

```
std::cout << "Hello world!" << std::endl;
```

A string with no characters in it is called the *empty string*. A literal empty string can be written using two consecutive double-quotes: `""`. Like a character, if you want to include a double-quote character in a literal string, it must be escaped, but it is no longer necessary to escape the apostrophe; thus, both of these are acceptable as characters: `'"'` and `'\"'`, and both of these are acceptable as strings: `"'"` and `"\'"`, but to have a literal apostrophe character, it must be escaped (`'\''`) and to have a literal double-quote in a string, it too, must be escaped (`"The student said \"Hi\"."`).

A string can be stored in one of two ways:

1. A C-style string (a character array).
2. An instance of the `std::string` class.

We will describe both there.

3.1 C-style strings

A C-style string (also called a null-terminated character array) is actually an array of characters. If a string is to have n characters, then the array capacity must be at least $n + 1$ where the character at index `n` is the null-character `'\0'`. In C, you could even initialize a character array with a string:

```
// 'str' is a character array
char *str = "Hello world!";
printf( "%d\n", str[11] == '!' );
printf( "%d\n", str[12] == '\0' );
```

In C++, you could define a character array as follows:

```
char str[20]{
  'H', 'e', 'l', 'l', 'o', ' ',
  'w', 'o', 'r', 'l', 'd', '!'
};
```

There are twelve printable characters occupying entries from index `0` to index `11`. Because the array has a capacity of 20, all entries from index `12` to `19` are `\0`, or the null character. Thus, when interpreted as a C-style string, the string ends at the exclamation point, and when you print a character array, it continues to print characters until it comes across the first null character:

```
std::cout << ">>>" << str << "<<<" << std::endl;
```

prints `>>>Hello world!<<<`. You can set any character to the null character, and so shorten the string:

```
    str[6] = '\0';
    std::cout << ">>>" << str << "<<<" << std::endl;
```

prints `>>>Hello <<<` (note the space at the end, as the character at index `6` was the `'w'`).

To calculate the length of a string (that is, the number of printing characters), you'd have to walk through the array until you find a null character: if the first null character is at index `k`, then the number of printable characters is `k`.

```
// Function declaration
std::size_t length( char *str );

// Function definition
std::size_t length( char *str ) {
    // The argument should not be the null pointer
    assert( str != nullptr );

    std::size_t string_length{ 0 };

    while ( str[string_length] != '\0' ) {
        ++string_length;
    }

    return string_length;
}
```

To concatenate a second string onto the end of a first string, first, you'd have to assume that the first array is large enough to hold all the characters (there is no way of checking this), and then you copy over all the characters in the second string.

```
// Function declaration
std::size_t concat( char *destination, char *source );

// Function definition
std::size_t concat( char *destination, char *source );
    // The argument should not be the null pointer
    assert( destination != nullptr );
    assert( source != nullptr );

    std::size_t id{ 0 };
    std::size_t characters_copied{ 0 };

    // Find the null character in the first string
    while ( destination[id] != '\0' ) {
        ++id;
    }

    std::size_t is{ 0 };

    for ( is = 0; source[is] != '\0'; ++id, ++is ) {
        destination[id] = source[is];
    }
```

```
    destination[id] = '\0';

    // Return the number of characters that were copied
    return is;
}
```

4 The string class

The string class is defined in the string library, so you must include it to use this class:

```
#include <string>
```

The default string is the empty string, but you can also pass the constructor a string that is then stored internally in the object.

```
    std::string empty_str{};
    std::string str{ "Hello world!" };
```

The member function `size()` returns the number of characters in the string, and you can access or assign to the characters in the string using the indexing operator:

```
    std::cout << "\"";

    for ( std::size_t k{ 0 }; k < str.size(); ++k ) {
        std::cout << str[k];
    }

    std::cout << "\"" << std::endl;
```

This would print `"Hello world!"`. There are many more member functions that allow you to access and manipulate the characters in a string.

Not on the examination

The binary operator `+` can be used to concatenate two `std::string`s: this operator generates a new string that is the second operand concatenated onto the end of the first.
The auto-assignment operator `+=` concatenates the string on the right-hand side of the operator to the end of the string on the left-hand side.
The comparison operators compare two strings based on a *lexicographical* ordering (or dictionary ordering): the first two letters are compared, and we only go to the next letter if the first two letters are equal. All six operators `<`, `<=`, `==`, `!=`, `>=` and `>` are overloaded. The following would be a crude implementation of a comparison operator:

```
bool std::string::operator<(
    std::string const &lhs,
    std::string const &rhs
) {
  for ( std::size_t k{ 0 };
        k < std::min( lhs.size(), rhs.size() );
        ++k
  ) {
    if ( lhs[k] < rhs[k] ) {
      return true;
    } else if ( lhs[k] > rhs[k] ) {
      return false;
    }

    assert( lhs[k] == rhs[k] );
  }

  // At this point, all the common letters are
  // equal, so at this point, whichever string
  // is shorter is first (as \verb|base| appears
  // in the dictionary before \verb|baseball|).
  return lhs.size() < rhs.size();
}
```

Not on the examination

You can use an iterator to walk through the entries of the string:

```
#include <iostream>
#include <string>

int main() {
  std::string str{ "Hi there, how are you?" };

  for ( auto itr{ str.begin() }; itr != str.end(); ++itr ) {
    if ( *itr == ' ' ) {
      std::cout << std::endl;
    } else {
      std::cout << *itr;
    }
  }

  std::cout << std::endl;

  return 0;
}
```

The output of this is each word printed on a separate line:

```
Hi
there,
how
are
you?
```

5 Exercises

1. Write a function that returns the length of a C-style string `char *s_str`.
2. Write a function that how often a character `char ch` appears in a C-style string `char *s_str`.
3. Write a function that determines if two C-style strings are the same (that is, having the same characters), returning the first character where they differ, or the index of both null characters if they are equal.
4. Write a function that copies a C-style string `char *s_source` to an address `char *s_dest` under the assumption there is sufficient memory at the second location to store the first string.
5. Write a function that concatenates one C-style string `char *s_str_2` onto the end of another `char *s_str_1` under the assumption that there is sufficient memory at address `s_str_1`.
6. Write a function that determines if the C-style string `char *s_sub` is a sub-string (a string contained with a string) of the argument `char *s_str`.
7. Write a function that reverses the characters of a string `char *s_str`.
8. Write a function that assumes a C-style string `s_integer` is a sequence of digits representing an integer with the first character possibly being a `+` or `-`, and calculate and return the storing `int`. You may assume the number falls between -2^{31} and $2^{31} - 1$.

9. Write a function that turns a string into *title case* (as opposed to *sentence case*) where each word is capitalized (that is, any character immediately following a space is capitalized).

10. Write a function that calculates how many "words" appear in a string. White space is defined as one or more spaces, tabs, or end-of-line characters (`' '`, `'\t'` or `'\n'`). A word is defined as sequence of one or more non-white-space characters (any character other than `'\0'`) in a row either at the start of the string, at the end, or surrounded by white space.

11. Write a function that takes an argument `unsigned int n` and repeatedly calculates `'!' + n%94` and then performs `n %= 94` a total of five times, saving the characters to the array `char tmp[5]` in order `0` through `4`, but then prints them to the screen in the order `4` down to `0`. This is one way of representing any four bytes as five characters.