Basics

Maybe worth looking through these - Wikipedia: Index of C++ Idioms

See Bjarne Strostrup's C++11 FAQ for several examples

ACCU Recommended Reading

C++ Core Guidelines GitHub repository

You can get offline versions of cppreference, for me the most notable option is the offline cppreference manpages available on GitHub. Installation instructions are provided in the github repository README. They're nice when you need to take a quick look, but maybe the full HTML page is better if you're exploring / browsing.

In a C++ program you can check which version is being ran with the following code.

```
if (__cplusplus == 201703L) std::cout << "C++17\n";
else if (__cplusplus == 201402L) std::cout << "C++14\n";
else if (__cplusplus == 201103L) std::cout << "C++11\n";
else if (__cplusplus == 199711L) std::cout << "C++98\n";
else if (__cplusplus == 202002L) std::cout << "C++20\n";
else std::cout << "pre-standard C++ (__cplusplus ==" << __cplusplus << ")\n";
```

When writing `#include "lib-custom.h"`, the compiler checks the CWD first, then the includes directory, and will check system includes last.

When we write `#include <iostream>`, the compiler checks the includes directory first, then system includes.

An **lvalue** is any value that has a location in memory. These can also be viewed as any value that is accessible in more than one place anywhere within your code. These could be named objects, pointers, or references. A general rule of thumb: if you can take it's address, it is an lvalue.

An **rvalue** refers to objects that are only accessible at one exact location within your code. These could be temporary objects like *by-value* function return values, a collection of operations wrapped in parenthesis that is substituted as the value of a new assignment, literal constants like `1`, `10`, `'c'`, or a `"string-literal"`. A general rule of thumb is if it is not an lvalue, it's an rvalue

The definition of these terms provide context for legal and illegal operations in C++. For example, the following statements are legal

```
int i = 0;
++i = 55 + 5;
```

But the following statement is not legal, since in this context `i++` is *not* an **lvalue**. That is, `i++` doesn't have a location in memory until *after* the increment is applied, which makes this assignment invalid.

```
i++ = 55;
```

Some examples of **lvalues** and **rvalues** in C++

```
int x;
x = 10; // x is an lvalue; 10 is an rvalue


int sizeDiff(const int &a, const int &b); // sizeDiff, a, and b are all lvalues; The int that is returned by sizeDiff, is an rvalue
```

Combining the two statements described above, we can better understand an assignment operation

```
int v = 0; // v and s are both lvalues
int s = 5; // 0 and 5 are both rvalues


int *px = sizeDiff(v, s); // *px is an lvalue; sizeDiff(v, s) is an rvalue
```

An **expression** is a mechanism for generating new values. May or may not contain operators, constants, variables;

Literal constant is a value that is stated literally, without representation through a variable.

Constants are variables defined as `const` given a type, name, and value

Const qualifiers are easiest when read right-to-left. For example, consider the following declarations, where we look at differences in const values or pointers. If a value is const, it cannot be changed. If a pointer is const, the location in memory that stores the data cannot be changed. A const reference is considered undefined behavior, but a *reference to a const value* is permitted, and often used to avoid the unnecessary copying of data.

We should notice in the examples below that we cannot assign a const value to a reference or pointer to non-const data

```
int const x = 5; // A constant integer x
const int x = 5; // Also a constant integer x

int const & a = x; // Valid, a is a reference to the const value stored at the memory location of x
const int & b = x; // Valid, b is a reference to the const value stored at the memory location of x (Same as above)
int & c = x; // Error! Cannot assign a reference of const data(x) to reference to non-const data(c)

int const * d = &x; // Valid, d is a non-const pointer to the const data stored at the memory location of x
const int * e = &x; // Valid, e is a non-const pointer to the const data stored at the memory location of x (Same as above)
int * f = &x; // Error! Cannot assign pointer with non-const data to a reference with const data
int const * const g = &x; // Valid, g is a const pointer to const data stored at the memory location of x
const int * const h = &x; // Valid, h is a const pointer to const data stored at the memory location of x (Same as above)
```

When dealing with non-const data, the rules are slightly different. We should notice in the examples below that we can assign a non-const value to a reference or pointer to const data

```
int y = 10; // A non-const integer y

int * i = &y; // Valid, i is a non-const pointer to non-const data stored at the memory location of y
int * const j = &y; // Valid, j is a const pointer to non-const data stored at the memory location of y
int & k = y; // Valid, k is a reference to non-const data stored at the memory location of y

// With the below declarations, we add the const qualifier to previously non-const data
// This makes the value const when we attempt to access it through d, but y is still non-const to those who are within it's scope and able to access it.
int const & l = y; // Valid, l is a reference to const data stored at the memory location of y
const int & m = y; // Valid, m is a reference to const data stored at the memory location of y

// Any declaration with a const reference like those seen below is considered to be unspecified
// If you can find a compiler that lets this happen, the results can vary wildly
int & const n = y; // Error! Unspecified behavior when applying const to reference
int const & const o = y; // Error! Unspecified behavior when applying const to reference
const int & const p = y; // Error! Unspecified behavior when applying const to reference
```

Array initialization can be done using any one of the examples below

```
int array[10]; // All values in array are initialized to an undetermined (arbitrary) value
int arr[10] = {1, 2, 3, 4}; // arr[0] = 1, arr[1] = 2, arr[2] = 3, arr[3] = 4, arr[4] = 0, arr[5] = 0...
int a[10] = { }; // a[0] = 0, a[1] = 0, a[2] = 0, ... , a[9] = 0
int a[5] = {2}; // a[0] = 2, a[1] = 0, ... , a[4] = 0
for (auto &e : a) e = 1; // a[0] = 1, ... , a[4] = 1
// Pay close attention to prefix and postfix decrement and increment below
// + As well as the use (or lack of) of referencing
for (auto e : a) std::cout << --e << std::endl; // 0 0 0 0 0
for (auto &e : a) std::cout << e++ << std::endl; // 1 1 1 1 1
for (auto e : a) std::cout << e << std::endl; // 2 2 2 2 2
```

Pretty Printing

C++ I/O Manipulators

These can simply be used inline with `cout` statements, as in the example below

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

int main() {
    double A; cin >> A;
    double B; cin >> B;
    double C; cin >> C;

    std::cout << std::showbase << std::hex << std::left << std::nouppercase
              << (long long) A << std::endl
              << std::right << std::showpos << std::setprecision(2)
              << std::setw(15) << setfill('_') << std::fixed
              << B << std::endl
              << std::setprecision(9) << std::scientific << std::uppercase << std::noshowpos
              << C << std::endl;
}
```

Exceptions

cppreference: std::exception

cppreference: try-block

We can define a custom exception with the class below

```
#include <iostream>
#include <string>
#include <sstream>
#include <exception>
using namespace std

class BadLengthException : public std::exception {
  public:
    std::string err;
    BadLengthException(int n) : err(std::to_string(n)) { }
    virtual inline const char * what() const throw() { return err.c_str(); }
};
```

And we can then treat this class as a normal exception, since we inherit from the `std::exception` interface

```
throw BadLengthException(n);


try {
  // ...
} catch (BadLengthException e) {
  cout << e.what() << '\n';
  // ...
}
```

Unknown exceptions can be caught using `...` for the `catch`. Below, we provide a condition for the `std::bad_alloc` exception, a condition for general `std::exceptions` (any exception that inherits from `std::exception`), and a final condition for any other exceptions that may occur.

```
    try {
      std::cout << Server::compute(A, B) << std::endl;
    }
    catch (std::bad_alloc &e) {
      std::cout << "Not enough memory" << std::endl;
    }
    catch (std::exception &e) {
      std::cout << "Exception: " << e.what() << std::endl;
    }
    catch (...) {
```

```
    std::cout << "Other Exception" << std::endl;
  }
```

Time Parsing

cppreference: std::tm

cppreference: std::get_time

cppreference: std::put_time

C++ has the above I/O helpers for formatting date and time output, and parsing input. Unfortunately, while working on the `Time Conversion` problem on HackerRank, I stumbled into a bug with parsing the `AM` / `PM` porition of the time string. The bug caused the infomation to be lost, and thus all time strings were defaulting to AM when I used the `std::get_time` function to initialize a `std::tm` struct.

My final solution is below. Here is a link to the StackOverflow question that led me to this solution. In the code below, I use `strptime` and `strftime` from the C header `time.h`. I'm sure there's a C++ way to do this, but currently this is the only method I'm familiar with.

```
#include <time.h>


// Example: s == "7:30:15PM"
// Returns: "19:30:15"
string timeConversion(string s) {
  char result[100];
  std::tm t;
  strptime(s.c_str(), "%I:%M:%S%p", &t);
  strftime(result, sizeof(result), "%H:%M:%S", &t);
  return std::string(result);
}
```

Locales

Another HackerRank question that gave an opportunity to play with managing time in C++ was Day of the Programmer. This question required we use a custom locale, and consider dates several hundred years in the past. The locale for this question was RU, and we had to factor in a calendar change that occured in 1917 for the Russian calendar. This is the reason for the `if` statement and second call to `strptime` in the example below.

```
#include <time.h>


string dayOfProgrammer(int year) {
  std::tm t;
  char result[25];
  setlocale(LC_TIME, "ru_RU.UTF-8");
  strptime(std::string("256" + to_string(year)).c_str(), "%j%Y", &t);
  // The t.tm_year value represents number of years since 1900
  if (t.tm_year <= 17 && t.tm_year % 4 == 0) {
    strptime(std::string("255" + to_string(year)).c_str(), "%j%Y", &t);
  }
  strftime(result, sizeof(result), "%d.%m.%Y", &t);
  return std::string(result);
}
```

It's worth noting that using the above method, `strptime` accounts for leap year and outputs correct dates back to year 1900. See the official documentation for more information.

Revision #21
Created 5 February 2021 18:44:59 by Shaun Reed
Updated 3 April 2022 14:56:57 by Shaun Reed