C++ Outline

Notes taken from:
- Drake, Caleb. EECS 370 Course Notes, University of Illinois Chicago, Spring ’97. Chapters 9, 10, 11, 13.1 & 13.2
- Horstman, Cay S. Mastering Object-Oriented Design in C++. Wiley & Sons, 1995. Chapters 1-5, 8, 11, 13, 14. The outline follows the organization of this text.

1. C++ as "A Better C"
   1.1. As a superset of C, C++ is largely the same in the areas of variables, operators and control flow. There are a few new C++ reserved words [Overhead from Deitel, p. 570]

1.2. Header Files
   1.2.1. The usual:
      ```
      #include <iostream.h>
      #include "chisetup.h" // Defines Horstmann’s classes
      ```
   1.2.2. Individual C++ libraries have their own header files. E.g. the Horstmann book includes a disk with classes defined such as Array, String, Args, Text, Date, Queue, List, Stack. Our system includes the RogueWave library.

1.3. Comments
   1.3.1. Single-line comments: Any text from // to the end of the line is ignored. E.g.
      ```
      // This is a comment
      ```
   1.3.2. Comments longer than one line must instead use the construct:
      ```
      /* Comment
       on more than one line
      */
      ```
   1.3.3. The /* */ comments can be used to block comment out lines already containing in-line (// ...) comments.
   1.3.4. To comment out code already containing /* comments */, you must use:
      ```
      #if 0
      code to be commented out
      #endif
      ```

1.4. C++ input & output use the " >> " and " << " symbols, respectively. These are also used as bit-wise left shift & right shift. The context in which they are used determines their interpretation.

1.5. Variables may be declared anywhere in a block prior to their first use, not just at the beginning of the enclosing block (as in C). E.g.
   ```
   cout << "Enter two integers: ";
   int x, y;
   cin >> x >> y;
   cout << "The sum is: " << x + y << '\n';
   ```
   or again as often used in a for loop:
   ```
   for (int i = 0; i < limit; i++)
     cout << i << '\n';
   ```
   These variables remain active until the end of the block.
1.6. Creating new Data types in C++

New types can be created without `typedef` simply by using the keywords `enum`, `struct`, `union`, and the new `class` keyword. E.g.

```cpp
enum Boolean {FALSE, TRUE};
struct Name {
    char first[10];
    char last[10];
};
union Number {
    int i;
    float f;
};
```

which can then be directly used as:

```cpp
Boolean done = FALSE;
Name student;
Number x;
```

1.7. Output

```cpp
#include <iostream.h>

int main()
{
    cout << "Hello, World" << endl;
    return 0;    // 0 indicates normal termination
}
```

Output can be integers, floating-point numbers, characters, and strings and can be chained.

Output may be sent to a file rather than to standard output (cout) and similarly read from a file.

```cpp
#include <fstream.h>

ofstream outStream ("output.dat");
ifstream inStream ("input.dat");
outStream << "Pi is: " << 3.1415 << endl;
int n;
while ( inStream >> n)
    outStream << n << endl;
```

cout of `\n` gives a new line of output, whereas "endl" also flushes the output buffer. This is an important distinction for debugging.
1.8. Input

The >> operator is used for reading data:

```cpp
double x;
String s;
cin >> s >> x;
```

Leading white space is skipped. To read input explicitly character by character use

```cpp
char ch = cin.get();
```

The Horstmann String class includes a function to read an entire line into a string:

```cpp
String s;
s.read_line(cin);
```

You can manipulate input characters explicitly using

```cpp
char c;
cin.get(c); // c is any character, whitespace is not skipped
```

You can "look ahead" to next input character using peek as follows:

```cpp
double x;
String s;
char ch = cin.peek();
if (isdigit(ch) || ch == '+' || ch == '-')
    cin >> x;
else
    cin >> s;
```

If input does not match the data type (e.g. trying to read a number where the input is not numeric), the stream state is set to "fail" and all subsequent operations fail. Input also fails at the end of a fail, for instance:

```cpp
double t = 0;
int n = 0;
while ( !cin.fail() ) {
    double x;
    cin >> x;
    if ( !cin.fail() )
        t += x; // Accumulate sum
        n++; // Increment count
}
if (n > 0)
    cout << "average: " << t/n << endl;
```

You can also read from a file instead of from stdin:

```cpp
#include <fstream.h>
ifstream inputStream("input.dat");
inputStream >> s >> x;
if ( inputStream.fail() )
    return FALSE;
```

Why use streams instead of scanf & printf? Two reasons:
A. It is type-safe: no mismatch between format & data values
B. It is extensible for any user-defined types.
1.9. Arrays

A template could be defined for "smart" dynamic arrays. For instance,

```
Array<X> a;
```

could define a "smart" array of type X which is initially empty and must be "grown" using:

```
Array<double> a;
```
```
a.grow(1,10);
```

which defines an array of 10 floating-point numbers a[1]..a[10]. Array bounds can be determined using:

```
for (int i = a.low(); i <= a.high(); i++)
    do something with a[i];
```
```
a.length() gives the number of elements in array a. Arrays of same type may be copied (1st class):
```
Array<double> b;
```
```
b = a;
```

which discards anything in b and copies corresponding elements of a into b.

1.10. Strings

A String class has been implemented. It is possible to initialize strings as:

```
String s = "Hello";
```
```
String t;
```

and to copy (=) and concatenate strings (+) as follows:

```
String u = s + ", World";
```

Relational operators (==, !=, <=, >=, <, >) represent lexicographical comparisons

```
if (s <= t) cout<< s << " is less than or equal to " << t;
```

The length operation returns the length of the string, and the substr operation extracts a substring with given starting index & length as follows:

```
String s = "Hello, World";
```
```
int n = s.length(); //n = 12
```
```
char ch = s[0]; //Starting character 'H' stored in ch
```
```
cout<< s.substr(2, 3);//Displays "llo"
```

Whenever a C-style string is expected, apply the c_str operation:

```
String filename = "input.dat";
```
```
ifstream inputStream( filename.c_str() );
```

1.11. The const qualifier

1.11.1. We can have "constant variables", as in:

```
const float PI = 3.14159;
```

where PI is a constant, but first it is initialized. These are also called named constants. These could be used rather than a #define as in:

```
const int arraySize = 100;
```
```
int array[ arraySize ];
```

1.11.2. const reference parameters can be used to get the efficiency of reference parameters with the semantics of value parameters

1.11.3. named constants can be used to limit the scope of a const-qualified object, which is not possible to do with #define. E.g.

```
void makeList()
{
    const int count = 10; // Local scope
    int list[count];
    for( int i=0; i < count; i++)
        list[i] = i;
}
```
1.12. Functions

1.12.1. In ANSI C, function prototypes are optional (by default a function is `extern`). In C++ they are required.

1.12.2. An empty parameter list may either have nothing or have `void`, e.g. `int foo(void)
Functions with no arguments are called with an empty argument list
e.g. `x = rand();`;

1.12.3. Function Definitions

Function arguments can be of any type, e.g. class and array templates. For example:

```c++
double findAverage(Array<double> a)
{
    double sum = 0;
    for (int i = a.low(); i <= a.high(); i++)
        sum += a[i];
    int n = a.length();
    if (n == 0)
        return 0;
    else
        return sum / n;
}
```

1.12.4. Functions may be declared as `inline`, which advises the compiler to generate a copy of the functions code in place to avoid the overhead of a function call.

1.12.4.1. The compiler may ignore this, and will for all but the smallest functions
1.12.4.2. Advantage: a debugger in C++ will trace through this code, whereas in C it
would not when using `#define`. The value from the `#define` would show up in
the debugger, but the user would not know which macro was used.

1.12.4.3. `inline` also reduces the risk of incorrect parenthesization of `#define`. E.g.

```c++
#include <iostream.h>
#define VALIDSQUARE(x)    (x) * (x)
#define INVALIDSQUARE(x)  x * x
inline int square(int x) { return x * x; }

main()
{
    cout << "  VALIDSQUARE(2 + 3) = " << VALIDSQUARE(2 + 3)
<< "\nINVALIDSQUARE(2 + 3) = " << INVALIDSQUARE(2 + 3)
<< "\n       square(2 + 3) = " << square(2 + 3) << '\n';

    return 0;
}
```

which gives as output:

```
VALIDSQUARE(2 + 3) = 25
INVALIDSQUARE(2 + 3) = 11
       square(2 + 3) = 25
```

1.12.5. Reference Arguments
We could write a function swap that swaps two integers as:

```cpp
void swap( int x, int y) //Incorrect function
{
    int temp = x;
    x = y;
    y = temp;
}
```

```cpp
main()
{
    cout<< a << b;    //Output is 1 2
    int a = 1;
    int b = 2;
    swap( a, b);
    cout<< a << b;    //Output is still 1 2
}
```

To actually get the changed values to be reflected back to the caller, we make parameters into reference parameters by adding the ampersand:

```cpp
void swap( int& x, int& y) //Corrected function
{
    int temp = x;
    x = y;
    y = temp;
}
```

1.12.6. Default arguments
Defaults may be supplied to formal parameters as follows:

```cpp
// Using default arguments
#include <iostream.h>

// Calculate the volume of a box
inline int boxVolume(int length=1, int width=1, int height=1)
{ return length * width * height; }
```

```cpp
main()
{
    cout << "The default box volume is: " << boxVolume() << "\n\nThe volume of a box with length 10, \n" << "width 1 and height 1 is: " << boxVolume(10) << "\n\nThe volume of a box with length 10, \n" << "width 5 and height 1 is: " << boxVolume(10, 5) << "\n\nThe volume of a box with length 10, \n" << "width 5 and height 2 is: " << boxVolume(10, 5, 2) << "\n"
    return 0;
}
```

1.12.7. Return Values
1.12.7.1. Return values may be any type except built-in (C-style) arrays. E.g.

```cpp
Employee find( Array<Employee> staff, String name);
Array<Employee> findAll( Array<Employee> staff, String s);
```

1.12.7.2. Functions can return pointers, but this can be dangerous when returning a local variable, since it will be deallocated when its stack frame is released. In this case make sure the variable is static
1.12.8. Name Overloading

The same name can be used for different functions as long as the function argument types are different. E.g. `find` can be applied to a string or to an array:

```cpp
int find( Strings, char charToFind);
Employee find( Array<Employee> staff, String nameToFind);
```

and then we could call

```cpp
r = find(a, x);
```

which will determine which `find` function to execute depending on argument types

1.12.9. The `main` Function

Many libraries have an `Args` class (e.g. Horstmann’s book) to parse the command line. In the Horstmann example, the class is initialized by calling:

```cpp
int main( int argc, char* argv[])
{
    Args args( argc, argv);
    ...
}
```

So given a command line to invoke a sort program:

```bash
sort -v -s10 input.dat
```

then we get:

- `args.prog_name()` is the string "sort"
- `args.bool_option('v', b)` sets `b` to TRUE
- `args.int_option('s', n)` sets `n` to 10
- `args.arg(1)` is the string "input.dat"

`main` is declared as int so we can return 0 for success, nonzero otherwise.

1.12.10. To link a C compiled function in with C++ code, write the C prototypes as follows:

```cpp
extern "C" function prototype;
```

1.13. Unary scope resolution operator

To access a global variable when a local variable of the same name is in scope, use `::` e.g:

```cpp
// Using the unary scope resolution operator
#include <iostream.h>

float value = 1.2345;

main()
{
    int value = 7;

    cout << "Local value = " << value
         << "Global value = " << ::value << 'n';

    return 0;
}
```

1.14. Assertions

Allow you to insert code verifying state of your program at a particular point. E.g.

```cpp
#include <assert.h>
#include <math.h>
y = f(x);
assert(y >=0);
z = sqrt(y);
```