Classes: Terminology

The material in this handout is collected from the following references:

- Chapters 2 and 7 of the text book C++ Primer.
- Various sections of Effective C++.
- Various sections of More Effective C++.

There is a small C++ vocabulary that everybody should understand. It is critical for all serious C++ programmers to completely and thoroughly understand the following terms: declaration, definition, initialization, assignment, default constructer, copy constructor, and copy assignment operator.

A *declaration* tells compilers about the name and type of an object, function, class, or template, but it omits certain details. These are declarations:

```
extern int x;                           // object declaration
int num_digits(int number);             // function declaration
class StopWatch;                        // class declaration
template<typename T> class my_vector; // template declaration
```

A *definition*, on the other hand, provides compilers with the details. For an object, the definition is where compilers allocate memory for the object. For a function or a function template, the definition provides the body of the code:

```
int x;                          // object definition

// function definition: function returns number of digits in its parameter
int num_digits(int number) {
  int digits_so_far = 1;
  if (number < 0) {
    number = -number;
    ++digits_so_far;
  }
  while (number /= 10) ++digits_so_far;
  return digits_so_far;
}
```

For a class or a class template, the definition lists the members of the class or template:

```
class StopWatch { // definition of class StopWatch
public:
  StopWatch(); // default constructor
  StopWatch(int seconds); // conversion constructor
  StopWatch(int hours, int minutes, int seconds); // non-default ctor
// other member functions
private:
  int seconds = 0;
};

template<typename T>
```

```
class my_vector {                    // template definition
public:
  my_vector(T *p = 0);
  ~my_vector();
  T const* operator() const;
  // other stuff
};
```

A *default constructor* is one that can be called without any arguments. Such a constructor either has no parameters or has a default value for every parameter.

> ***You can explicitly initialize a single object or a small array of objects using a constructor. However, it is not practical to explicitly initialize a large number of dynamically allocated objects on the free store. Classes for such objects must define a default constructor.***

```
class A {
public:
  A() = default;     // default constructor
};
A arrayA[10];        // 10 constructors called

class B {
public:
  B(int x = 0);      // also default constructor
};
B arrayB[10]; // 10 constructors called, each with an arg of 0

class C {
public:
  C(int x);          // not a default constructor
};
C arrayC[10];        // error!!!
```

A *copy constructor* is used to initialize an object with a different object of the same type:

```
class String {
public:
  String();                  // default constructor
  String(const String& rhs); // copy constructor
  // other functions ...
private:
  size_t len;
  char *data;
};

String s1;          // call default constructor
String s2 {s1};     // call copy constructor
String s3 = s2;     // call copy constructor
```

Here's how the copy constructor might be implemented:

```
String::String(String const& rhs) : len{rhs.len}, data{new char[len+1]} {
  std::strcpy(data, rhs.data);
}
```

The most important use of the copy constructor is to define what it means to pass and return objects by value. As an example, consider the following [inefficient] way of writing a non-member function to concatenate two `String` objects:

```
String operator+(String lhs, String rhs) {
  String temp{lhs};
  temp += rhs; // assume member function op+= exists// assume member function op+= exists
  return temp;
}

String a {"Hello"};
String b {" world"};
String c {a + b};   // c = op+=(s1, s2) and c will be assigned "Hello World"
```

This `operator+` takes two `String` objects as parameters and returns a `String` object as a result that is the string obtained by concatenating the strings encapsulated by the arguments. Both the arguments are passed by value and the result is returned by value. So there will be one copy constructor called to initialize parameter `lhs` with argument `a`, one to initialize parameter `rhs` with argument `b`, and one to initialize variable `c` with the function's return value `temp`. In fact, there might even be some additional calls to the copy constructor if a compiler decides to generate intermediate temporary objects. The important point here is that pass-by-value *means* "call the copy constructor."

> ***Pass-by-value means call the copy constructor.***

The next two terms we need to grapple with are *initialization* and *assignment*. An object's initialization occurs when it is given a value for the very first time. For objects of classes or structs with constructors, initialization is *always* accomplished by calling a constructor. This is quite different from object assignment, which occurs when an object that is already initialized is given a new value:

```
std::string s1;          // initialization
std::string s2("Hello"); // initialization
std::string s3 = s2;     // initialization
s1 = s3;                 // assignment
```

> ***The difference between initialization and assignment is that the former is performed by a constructor while the latter is performed by `operator=`. In other words, the two processes correspond to different function calls.***

The reason for the distinction is that the two kinds of functions must worry about different things. Constructors usually have to check their arguments for validity, whereas most copy assignments can take it for granted that their argument is legitimate [because it has already been constructed]. On the other hand, the target of an assignment, unlike an object undergoing construction, may already have resources allocated to it. These resources typically must be released before the copy assignment function returns. Frequently, one of these resources is memory. Before a copy assignment returns, it can must deallocate the memory that was allocated for the old value.

Here is how a `String` constructor with a pointer to character string as parameter and the assignment operator could be implemented:

```
String::String(char const *value) { // a possible String constructor
  if (value) { // if value ptr isn't null
    len = std::strlen(value);
    data = new char[len + 1];
    std::strcpy(data, value);
  } else {     // handle null value ptr
    len = 0;
    data = new char[len + 1];
    *data = '\0';  // add trailing null char
  }
}

// a possible String copy assignment
String& String::operator=(String const& rhs) {
  size_t tmp_len { rhs.len };
  char *tmp_data { new char [tmp_len + 1] }; // allocate new memory
  std::strcpy(tmp_data, rhs.data);
  len = tmp_len;
  delete [] data;  // delete old memory
  data = tmp_data;
  return *this;
}
```

Notice how the constructor must check its parameter for validity and how it must ensure that data member `data` is properly initialized, i.e., it points to a `char*` that is properly null-terminated. On the other hand, the assignment operator takes for granted that its parameter is legitimate. Instead, it concentrates on detecting pathological conditions, such as assignment to itself and on deallocating old memory before returning. The differences between these two functions typify the differences between object initialization and object assignment.

Client is another commonly used term. A client is a programmer who will use the code you write. This programmer will be looking at your code trying to figure out what it does; reading your interface [class definitions], attempting to determine whether she wants to inherit from your classes; or examining your design decisions, hoping to glean insights into their rationale.