

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 constructor, 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:

```
1  extern int x;                                // object declaration
2  int num_digits(int number);                  // function declaration
3  class Stopwatch;                             // class declaration
4  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:

```
1  int x;                                        // object definition
2
3  int num_digits(int number) { // function definition
4  // function returns the number of digits in its parameter
5      int digits_so_far = 1;
6      if (number < 0) {
7          number = -number;
8          ++digits_so_far;
9      }
10     while (number /= 10) ++digits_so_far;
11     return digits_so_far;
12 }
```

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

```
1  class Stopwatch { // definition of class Stopwatch
2  public:
3      Stopwatch(); // default constructor
4      Stopwatch(int seconds); // conversion constructor
5      Stopwatch(int hours, int minutes, int seconds); // non-default ctor
6      // other member functions
7  private:
8      int seconds = 0;
9  };
10
11  template<typename T>
12  class my_vector { // template definition
13  public:
```

```

14 my_vector(T *p = 0);
15 ~my_vector();
16 T const* operator() const;
17 // other stuff
18 };

```

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.

```

1  class A {
2  public:
3      A() = default;      // default constructor
4  };
5  A arrayA[10];           // 10 constructors called
6
7  class B {
8  public:
9      B(int x = 0);       // also default constructor
10 };
11 B arrayB[10]; // 10 constructors called, each with an arg of 0
12
13 class C {
14 public:
15     C(int x);           // not a default constructor
16 };
17 C arrayC[10];          // error!

```

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

```

1  class String {
2  public:
3      String();           // default constructor
4      String(const String& rhs); // copy constructor
5      // other functions ...
6  private:
7      size_t len;
8      char *data;
9  };
10
11 String s1;              // call default constructor
12 String s2 {s1};         // call copy constructor
13 String s3 = s2;         // call copy constructor

```

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 function to concatenate two `String` objects:

```

1 String operator+(String s1, String s2) {
2     String temp;
3     temp.len = std::strlen(s1.data) + std::strlen(s2.data);
4     delete [] temp.data;
5     temp.data = new char[temp.len + 1];
6     std::strcpy(temp.data, s1.data);
7     std::strcat(temp.data, s2.data);
8     return temp;
9 }
10
11 String a("Hello");
12 String b(" world");
13 String c = a + b;    // c = String("Hello world")

```

This `operator+` takes two `String` objects as parameters and returns one `String` object as a result. Both the parameters and the result will be passed by value, so there will be one copy constructor called to initialize `s1` with `a`, one to initialize `s2` with `b`, and one to initialize `c` with `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:

```

1 std::string s1;           // initialization
2 std::string s2("Hello"); // initialization
3 std::string s3 = s2;     // initialization
4 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 assignment operators 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 new resources can be assigned. Frequently, one of these resources is memory. Before an assignment operator can allocate memory for a new value, it must first deallocate the memory that was allocated for the old value.

Here is how a `String` constructor and assignment operator could be implemented:

```

1 // a possible String constructor
2 String::String(char const *value) {
3     if (value) { // if value ptr isn't null
4         len = std::strlen(value);
5         data = new char[len + 1];
6         std::strcpy(data, value);

```

```

7   } else {      // handle null value ptr
8       len = 0;
9       data = new char[len + 1];
10      *data = '\0'; // add trailing null char
11  }
12  }
13
14  // a possible String assignment operator
15  String& String::operator=(String const& rhs) {
16      size_t tmp_len { rhs.len };
17      char *tmp_data { new char [tmp_len + 1] }; // allocate new memory
18      std::strcpy(tmp_data, rhs.data);
19      len = tmp_len;
20      delete [] data; // delete old memory
21      data = tmp_data;
22      return *this;
23  }

```

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 allocating new memory. 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.