Rule of Three

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

- Section 13.1 of the text book C++ Primer.
- Various sections of Effective C++.
- Various sections of More Effective C++.
- Microsoft provides lots of information and examples on copy constructors and copy assignments.

If you don't declare them yourself, C++ compilers will synthesize their own versions of a copy constructor, a copy assignment, a destructor [and a pair of address-of operators - but these are not relevant here]. The Rule of Three claims that if one of these is defined by the programmer, then the other two must also be explicitly defined by the programmer.

Consider a class for representing `String` objects:

```
class String {
public:
  String();
  String(char const*);
  ~String();
  // no copy constructor nor copy assignment
  char const* c_str() const { return data; }
private:
  size_t len;
  char    *data;
};

// non-member, non-friend operator<< function
std::ostream& operator<<(std::ostream&, String const&);
```

The two constructors and destructor are defined like this:

```
String::String() : len{0}, data{new char [len+1]} {
  *data = '\0'; // null-terminated empty string
}

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

String::~String() { delete [] data; }
```

Notice that I'm careful to use `[]` with `new` in both constructors, even though in one of the places only a single `char` value is required. It is essential to employ the same form in corresponding applications of `new` and `delete`, so I was careful to be consistent in my uses of `new`. This is something you do not want to forget. Always make sure that you use `[]` with `delete` if and only if you used `[]` with the corresponding use of `new`.

> ***Always make sure that you use `[]` with `delete` if and only if you used `[]` with the corresponding use of `new`.***

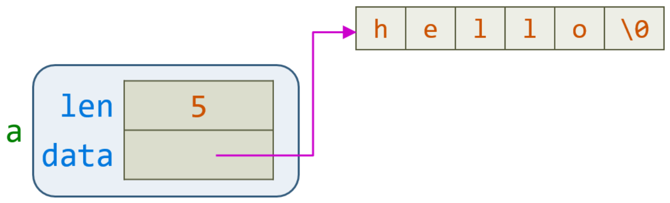
The non-member, non-friend `operator<<` function is defined like this:

```
std::ostream& operator<<(std::ostream& os, String const& rhs) {
  os << '<' << rhs.c_str() << '>';
  return os;
}
```

Note that there is no assignment operator or copy constructor declared in this class. As you'll see, this has some unfortunate consequences. If you make this definition:

```
String a{"hello"}; // or String a("hello") in old C++
```

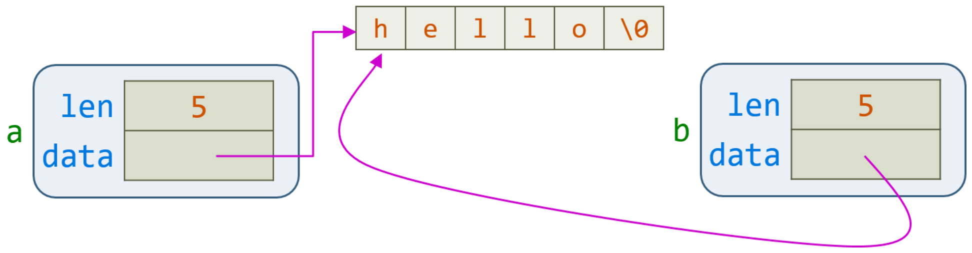
the memory layout associated with object `a` is shown below:



Now, if you define a new object `b` like this:

```
String b{a}; // or String b(a) in old C++
```

the compiler will synthesize a copy constructor that will perform a member-wise initialization of the members of `b` with the members of `a`, which for pointer `b.data` is just an initialization that will make a copy of the bits in `a.data`. This *shallow copy* is shown below:

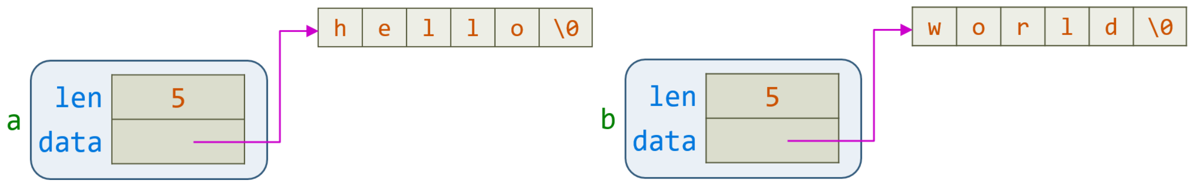


The problem with this shallow copy is that both `a` and `b` now contain pointers to the same character string. When one of them goes out of scope, its destructor will delete the memory still pointed to by the other.

Now, let's consider the assignments. If you make these object definitions:

```
String a{"Hello"}, b{"World"};
```

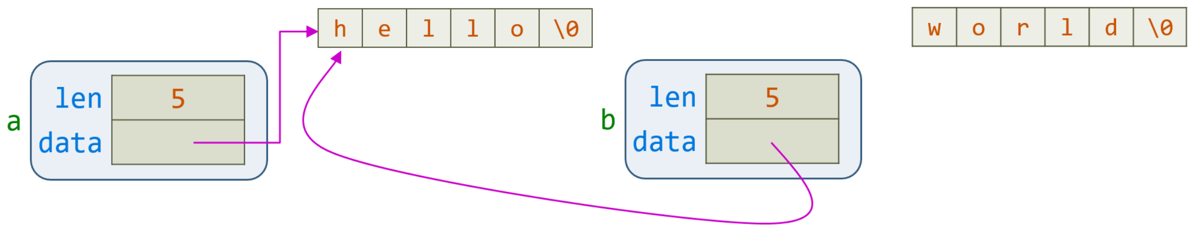
the situation is as shown below:



Inside object `a` there is a pointer to memory containing the character string `"Hello"`. Separate from that is an object `b` containing a pointer to character string `"World"`. If you now perform an assignment,

```cpp
b = a; // shallow copy
```

there is no client-defined copy assignment to call, so C++ generates and calls the default copy assignment instead. This default copy assignment performs member-wise assignment from members of `a` to members of `b`, which for pointers `a.data` and `b.data` is just a bitwise copy. The *shallow copy* resulting from this assignment is shown below:



> ***In shallow copy, we only copy the address of the data but not the data itself.***

There are at least two problems with shallow copies. First, the memory that `b` used to point to was never deleted; it is orphaned and lost forever. This is a classic example of how a memory leak can arise. Second, both `a` and `b` now contain pointers to the same character string. When one of them goes out of scope, its destructor will delete the memory still pointed to by the other. For example:

```cpp
String a{"hello"};     // define and construct a
{                      // open new scope
  String b{"world"};   // define and construct b
  // other unrelated stuff ...
  b = a;               // execute default op=, lose b's memory
}                      // close scope, call b's destructor
String c{a};           // c.data is undefined!
                       // a.data is already deleted
```

The last statement in this example is a call to the copy constructor, which also isn't defined in the class, hence will be generated by C++ in the same manner as the copy assignment and with the same behavior: bitwise copy of the underlying pointers. That leads to the same kind of problem, but without the worry of a memory leak, because the object being initialized can't yet point to any allocated memory. In the case of the code above, for example, there is no memory leak when `c.data` is initialized with the value of `a.data`, because `c.data` doesn't yet point anywhere. However, after `c` is initialized with `a`, both `c.data` and `a.data` point to the same place, so that place will be deleted twice: once when `c` is destroyed, once again when `a` is destroyed.

Because pass-by-value semantics are implemented by the copy constructor, the synthesized copy constructor can bite you more than the synthesized copy assignment. As an example, consider the following pass-by-value function `do_nothing`:

```cpp
void do_nothing(String str) { /* does nothing by design */ }
String s { "The Truth Is Out There" };
do_nothing(s);
```

Everything looks innocuous enough, but because `str` is passed by value, it must be initialized from `s` via the synthesized copy constructor. Hence, `str` has a copy of the *pointer* that is inside `s`. When `do_nothing` finishes executing, `str` goes out of scope, and its destructor is called. The end result is by now familiar: `s` contains a pointer to memory that `str` has already deleted. The result of using `delete` on a pointer that has already been deleted is undefined, so even if `s` is never used again, there could well be a problem when it goes out of scope.

> ***The solution to these kinds of pointer aliasing problems is to write your own versions of the copy constructor and the copy assignment if you have pointers in your class. Inside those functions, you implement a deep copy by copying the pointed-to data structures so that every object has its own copy.***

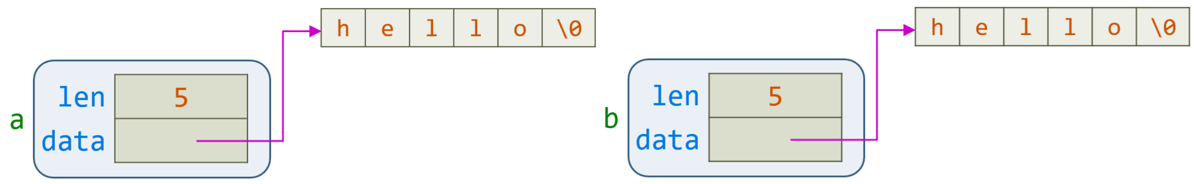
Let's begin by declaring and defining a copy constructor function:

```cpp
 1  class String {
 2  public:
 3    // same as before except that copy ctor is declared
 4    String(String const&);
 5  private:
 6    size_t len;
 7    char   *data;
 8  };
 9
10  String::String(String const& rhs) : len{rhs.len}, data{new char [len+1]} {
11    std::strcpy(data, rhs.data);
12  }
```

Lines 10 and 11 perform the deep copy required for two `String` objects to be considered different. Using the defined copy constructor, the following definitions

```cpp
String a{"hello"}, b {a};
```

will create two objects that have exclusive ownership on their pointed-to data structures:



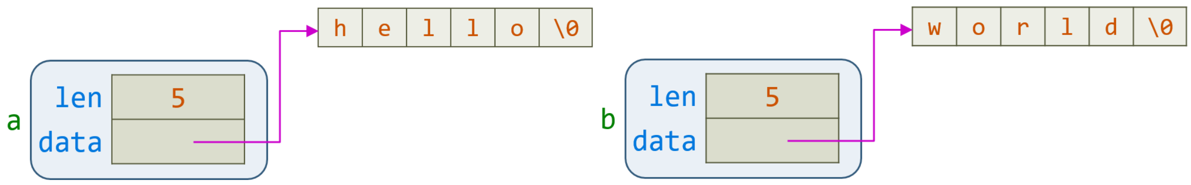
The declaration and definition of the copy assignment will look like this:

```cpp
class String {
public:
  // same as before except that operator= is declared
  String& operator=(String const&);
private:
  size_t len;
  char   *data;
};

String& String::operator=(String const& rhs) {
  // make a copy of rhs first!!!
  size_t tmp_len   { rhs.len };
  char   *tmp_data { new char [tmp_len+1] };
  std::strcpy(tmp_data, rhs.data);
  // then, swap old with new!!!
  len = tmp_len;
  delete [] data;
  data = tmp_data;

  return *this;
}
```

The following definitions

```cpp
String a{"hello"}, b{"world"};
```

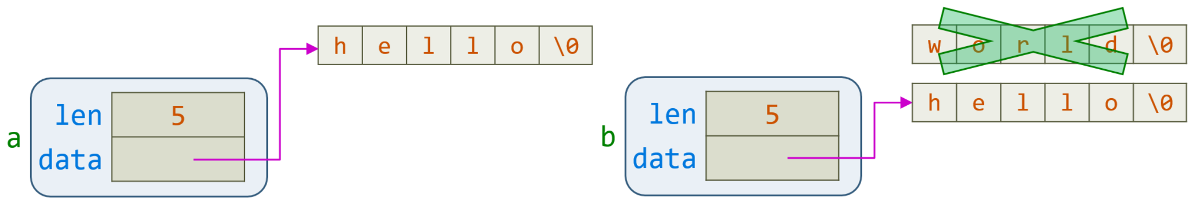
will look like this in memory:



The following assignment expression

```cpp
b = a;
```

will use the programmer-defined assignment operator to perform a deep copy without memory leaks or errors. The memory image will look like this:



What to do if your class should not be copying?

For some classes, it's more trouble than it's worth to implement copy constructors and copy assignments, especially when you have reason to believe that your clients won't make copies or perform assignments. The examples above demonstrate that omitting the corresponding member functions reflects poor design, but what do you do if writing them isn't practical, either? In that case, add `= delete` to the function's declaration:

```cpp
class X {
public:
  X(X const&)             = delete;
  X& operator=(X const&) = delete;
private:
  // ...
}
```