Allocators

As seen on the previous slide, `new` combines memory allocation with object construction, and `delete` combines destruction with deallocation. This is true also for arrays of objects: `new Point[10]` default-constructs 10 objects.

Now consider implementing the library class `vector`. When `push_back(p)` is executed on a full vector and a new chunk of memory must be allocated, no objects should be created. Instead, `p` should be copied (or moved) into raw memory.

The library class `allocator` allocates and deallocates raw memory:

```
allocator<Point> alloc;
auto p = alloc.allocate(10); // allocates space for 10
                             // unconstructed Points,
                             // returns a pointer
...
alloc.deallocate(p, 10);     // deallocate
```

Constructing and Destroying Objects

When raw memory has been allocated, objects can be constructed in that area. Before memory is deallocated, the objects must be destroyed.

```
allocator<Point> alloc;
auto p = alloc.allocate(10);

auto q = p; // where to start constructing objects
alloc.construct(q++, 10, 20); // Point(10, 20)
alloc.construct(q++, 30, 40); // Point(30, 40)
...
while (q != p) {
    alloc.destroy(--q); // execute ~Point() for each object
}

alloc.deallocate(p, 10);
```

Reference Semantics

In Java, all objects are manipulated via reference variables (pointers in C++). You may assign pointers to pointer variables but assignments never touch the "inside" of the objects.

```
Car* myCar = new Car("ABC123");
Car* yourCar = new Car("XYZ789");
...
delete myCar;       // my car to the scrapheap
myCar = yourCar;    // I buy your car
yourCar = nullptr; // you don't own a car now
```

This is called *reference semantics*, and it is often the preferred way to manipulate objecs.

Value Semantics

Reference semantics is not always what you want. Consider manipulating strings via pointers:

```
string* s1 = new string("asdfg");
string* s2 = s1;      // two pointers to the same object

s1->erase(0, 1);      // erase the first character of s1
cout << *s1 << endl; // prints sdfg
cout << *s2 << endl; // also prints sdfg, maybe unexpectedly
                      // since you haven't touched s2
```

In C++ you may copy objects, not only pointers, to get the desired results. This is called *value semantics*.

```
string s1 = "asdfg";
string s2 = s1;       // copy the object
s1.erase(0, 1);       // only s1 is modified
```

Which Semantics is Best?

Value semantics is preferred for “small”, “anonymous”, objects (numbers, strings, dates, colors, . . .). When object *identity* is important, reference semantics is better. Example:

```
Car myCar("ABC123");
Car yourCar("XYZ789");

myCar = yourCar; // copy the object => two cars that are
                 // identical. Not reasonable!
```

Cars have identity, so they should be manipulated using reference semantics. However, in C++ value semantics is sometimes used also for such classes.

Copying Objects

Objects are copied in several cases:

- Initialization

```
Person p1("Bob");
Person p2(p1);
Person p3 = p1;
```

- Assignment

```
Person p4("Alice");
p4 = p1;
```

- Parameter by value

```
void f(Person p) { ... }
f(p1);
```

- Function value return

```
Person g() {
    Person p5("Joe");
    ...
    return p5;
}
```

Initialization and Assignment

The following statements may look similar, but they are handled by different C++ language constructs:

```
Person p1("Bob");
Person p3 = p1; // initialization

Person p4("Alice");
p4 = p1;        // assignment
```

Initialization A new, uninitialized object is initialized with a copy of another object. This is handled by the *copy constructor* of the class, `Classname(const Classname&)`. It also handles value parameters and function value return.

Assignment An existing object is overwritten with a copy of another object. This is handled by the *assignment operator* of the class, `Classname& operator=(const Classname&)`.

The Synthesized Copy Functions

In your own classes, you can implement the copy constructor and assignment operator to get the desired copy behavior. If you don't do this, the compiler synthesizes a copy constructor and an assignment operator which perform memberwise ("shallow") copying.

You should *not* write your own copy constructor or assignment operator in a class that doesn't use dynamic resources. Example:

```
class Person {
public:
    // this is the copy constructor that the compiler
    // creates for you, and you cannot write a better one
    Person(const Person& p)
        : name(p.name), age(p.age) {}
private:
    string name;
    unsigned int age;
};
```

Preventing Copying C++11

Some classes shouldn't allow objects of the class to be copied. This is achieved by declaring the copy constructor and the assignment operator as `delete`-d:

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

In C++98, the same effect was achieved by making the copy constructor and assignment operator `private`.

A String Class

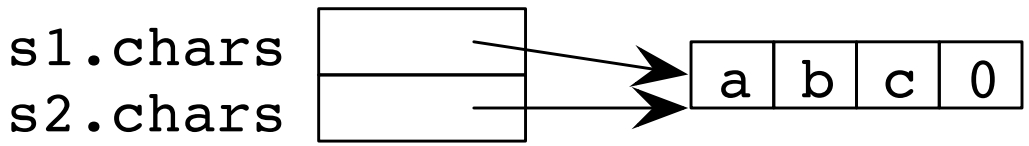
As an example of a class that allocates and deallocates dynamic memory we will use a simplified string class. The class should have value semantics. The characters are stored in a C-style string (a null-terminated `char` array). The `<cstring>` function `strlen` computes the number of characters in a C-style string, the `function` strcpy copies the characters.

```
class String {
public:
    String(const char* s) : chars(new char[strlen(s) + 1]) {
        strcpy(chars, s); // copy s to chars
    }
    ~String() { delete[] chars; }
private:
    char *chars;
};
```

Without a Copy Constructor, I

We haven't written a copy constructor for `String`, so the constructor generated by the compiler is used. It performs a memberwise copy.

```
void f() {
    String s1("abc");
    String s2 = s1; // see figure for memory state after copy
}
```



When the function exits:

1. The destructor for `s2` is called, `s2.chars` is deleted.
2. The destructor for `s1` is called, `s1.chars` is deleted.
3. This is disaster — you must not delete the same object twice.

Without a Copy Constructor, II

Value parameter example:

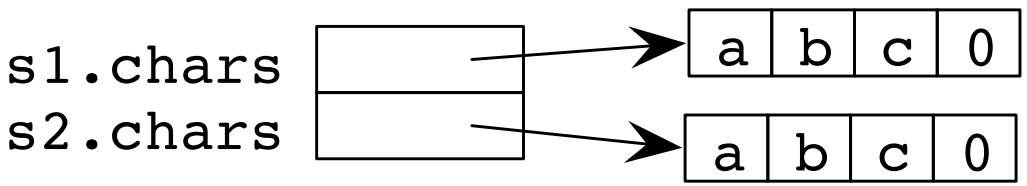
```
void f(String s) {
    ...
}

void g() {
    String s1("abc");
    f(s1);
}
```

1. In the call `f(s1)`, `s1` is copied to `s`. Now, `s.chars` and `s1.chars` point to the same array.
2. When `f` exits, the destructor for `s` is called, `s.chars` is deleted. Now, `s1.chars` points to deallocated memory and `s1` cannot be used.
3. When `g` exits, the destructor for `s1` is called and `s1.chars` is deleted ...

Defining a Copy Constructor

Class `String` must have a copy constructor that performs a *deep copy*, i.e., copies not pointers but what the pointers point to.

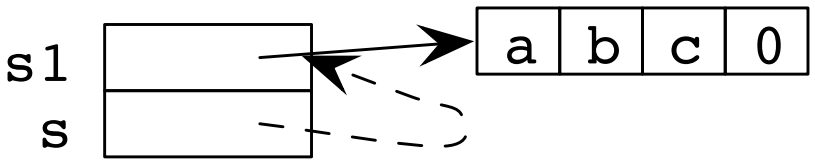
```
String(const String& rhs)
        : chars(new char[strlen(rhs.chars) + 1]) {
    strcpy(chars, rhs.chars);
}

void f() {
    String s1("abc");
    String s2 = s1; // see figure for memory state after copy
}
```



Avoiding Copy Problems

The copy constructor is never invoked if objects always are manipulated via pointers or references. For strings, this is not possible, since we need value semantics. But parameter copying can be avoided, and it doesn't happen if the parameter is passed by reference.

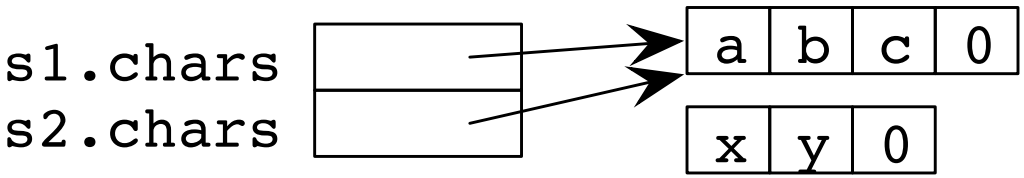
```
void f(const String& s) { ... }

void g() {
    String s1("abc");
    f(s1);
}
```



Assignment

Copy problems, with an extra complication, also occur when objects are assigned.

```
void f() {
    String s1("abc");
    String s2("xy");
    s2 = s1; // see figure for memory state after copy
}
```



The extra complication is that the array "xy" cannot be reached after the assignment, which leads to a memory leak. And as before, the destructor is called twice for the same object.

Overloading Assignment

To solve the assignment copying problems, you must define your own version of the assignment operator. This is done by defining an *operator function* `operator=`.

```
class String {
public:
    String& operator=(const String&);
    ...
};
```

With this operator function, the statement

```
s1 = s2;
```

is converted by the compiler into

```
s1.operator=(s2);
```

Implementing operator=

Implementation of `String::operator=`:

```
String& String::operator=(const String& rhs) {
    if (&rhs == this) {
        return *this;
    }
    delete[] chars;
    chars = new char[strlen(rhs.chars) + 1];
    strcpy(chars, rhs.chars);
    return *this;
}
```

Notice the similarities and dissimilarities to the copy constructor:

- Both functions perform a deep copy of the object's state.
- In addition, `operator=` must 1) delete the old state; 2) return the object being assigned to; 3) check for self-assignment. Explanations on next slide.

Details, operator=

1. Delete the old state (`delete[] chars`): this is necessary to prevent memory leaks.
2. Return the object (`return *this`): this is necessary to make it possible to chain assignments.

```
s1 = s2 = s3; // compiled into s1.operator=(s2.operator=(s3))
```

`operator=` returns a *reference* to the assigned object. This is safe, because the object certainly exists.

3. Check for self-assignment (`if (&rhs == this)`):

```
s1 = s1;      // compiled into s1.operator=(s1)
```

Here, `*this` and `rhs` are the same object. Without the check, you would first delete `s1.chars`, then allocate a new array, then copy from the uninitialized array.

Swapping Values

The library defines a function `swap` which swaps two values (as a template function that may be called with parameters of arbitrary types):

```
template <typename T>
inline void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

For the `String` class it is expensive to perform one copy and two assignments. See next slide for a better alternative.

Efficient Swapping

All you need to do when swapping two `String` objects is to swap the pointers to the character arrays:

```
class String {
    friend void swap(String& s1, String& s2);
public:
    ...
};

inline void swap(String& s1, String& s2) {
    using std::swap;
    swap(s1.chars, s2.chars);
}
```

Moving Objects

In the `String` class, the default copy constructor didn't work correctly:

```
String(const String& rhs) : chars(rhs.chars) {}
```

The copy constructor must perform a deep copy:

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

If we are certain that the object that we're copying *from* will not be used after the copy, we could write the copy constructor like this:

```
String(String&& rhs) : chars(rhs.chars) {
    rhs.chars = nullptr;
}
```

This is called a *move constructor*.

When Moving is Possible

But how can we (or rather the compiler) be "certain that the object that we're copying *from* will not be used after the copy?"

The answer is that *temporary values* can be moved from, since these will be destroyed after use. And the compiler can recognize temporary values:

```
String s1("abc");
String s2("def");
String s3 = s1 + s2; // the result of '+' is a temporary value

void f(String s);
f("abcd"); // => f(String("abcd")), the argument is a temporary

String g() {
    ...
    return ...; // the return value is a temporary
}
```

Lvalues and Rvalues, Rvalue References C++11

An lvalue is persistent (e.g. a variable). An rvalue is not persistent (e.g. a literal or a temporary). A regular non-const reference can bind only to an lvalue. In the new standard *rvalue references* have been introduced. An rvalue reference can bind only to an rvalue. Examples:

```
String s1("abc");
String s2("def");

String& sref = s1; // reference bound to a variable

String&& srr = s1 + s2; // rvalue reference bound to a temporary
```

You obtain an rvalue reference with &&.

Move Constructors

Now that we have rvalue references, the move constructor can be written. The copy constructor is still necessary for regular copying.

```
String(String&& rhs) noexcept : chars(rhs.chars) {
    rhs.chars = nullptr;
}

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

`noexcept` C++11 informs the compiler that the constructor will not throw any exceptions. This is only important for move operations.

Move Assignment Operator

A class whose objects can be moved should also have a move assignment operator:

```
String& operator=(String&& rhs) noexcept {
    if (&rhs == this) {
        return *this;
    }
    delete[] chars;
    chars = rhs.chars;
    rhs.chars = nullptr;
    return *this;
}
```

Explicit Moving

Sometimes the programmer, but not the compiler, is certain that it is safe to move an object rather than copying it. The library function `std::move` returns an rvalue reference to its argument:

```
template <typename T>
inline void swap(T& a, T& b) {
    T temp = std::move(a); // a is moved to temp, a is empty

    a = std::move(b);      // but a isn't used, instead b is
                           // moved to a, b is empty

    b = std::move(temp);   // but b isn't used, instead temp is
                           // moved to b, temp is empty

}                          // but temp isn't used, instead
                           // destroyed
```

Canonical Construction Idiom

1. When a class manages a resource, it needs its own destructor, copy and move constructor, and copy and move assignment operator ("rule of five").
2. The constructor should initialize data members and allocate required resources.
3. The copy constructor should copy every element (deep copy). The move constructor should move instead, making the right-hand side empty.
4. The copy assignment operator should release resources that are owned by the object on the left-hand side of the assignment and then copy elements from the right side to the left side (deep copy). The move assignment operator should move instead, making the right-hand side empty.
5. The destructor should release all resources.