# CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 12
October 10, 2016

# Uses of Pointers

# Array data member

A class `A` commonly relates to several instances of class `T`.

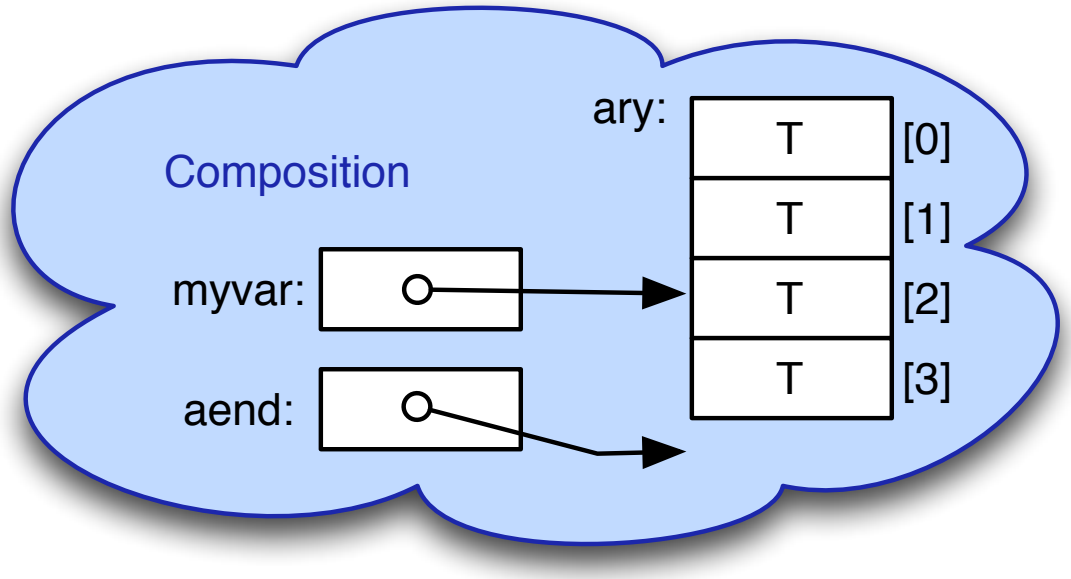
Some ways to represent this relationship.

1. **Composition:** `A` can **compose** an array of instances of `T`. This means that the `T`-instances are inside of each `A`-instance.
2. **Aggregation:** `A` can contain a pointer to a dynamically-allocated array of instances of `T`. `A` **composes** the pointer but **aggregates** the `T`-array to which it points.
3. **Fully dynamic aggregation:** `A` can contain a pointer to a dynamically-allocated array of *pointers* to instances of `T`. The individual `T`-instances can be scattered throughout memory.

Pictures of these three methods are given on the next slides.

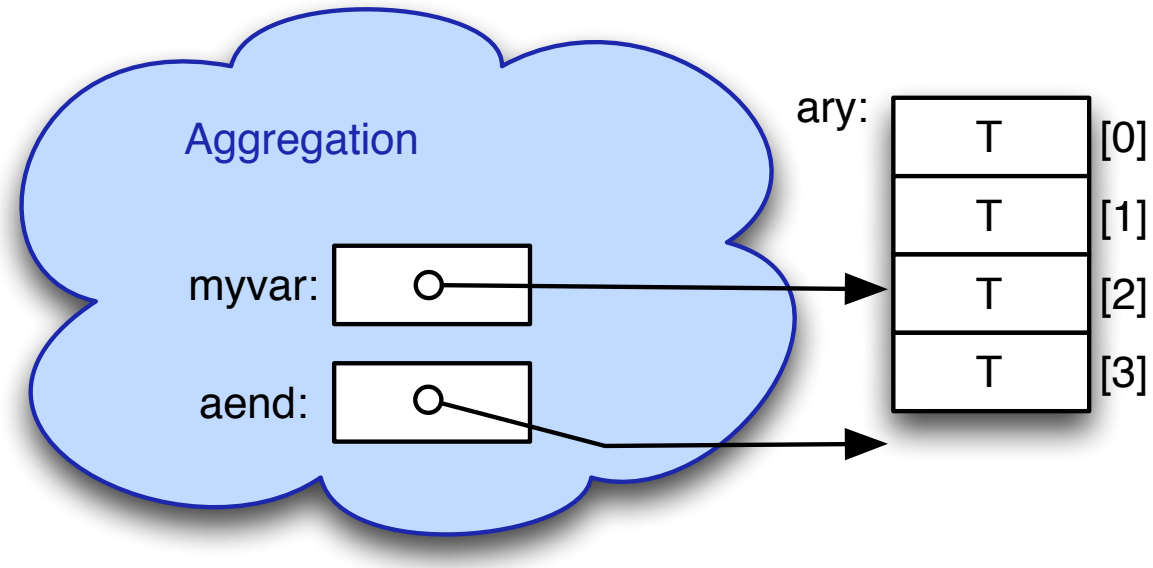
# Composition

```
T   ary[4];
T* aend = ary+4;
T* myvar = &ary[2];
```



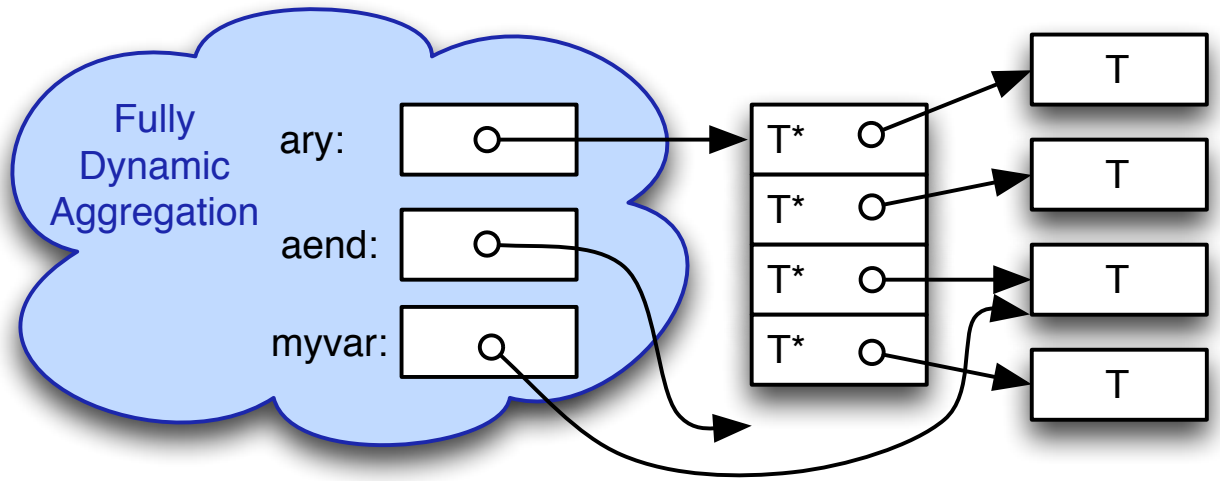
# Aggregation

```
T   ary[4];
T* aend = ary+4;
T* myvar = &ary[2];
```



# Fully dynamic aggregation

```
T** ary = new T*[4];
T** aend = ary+4;
for( k=0; k<4; ++k ) {
    ary[k] = new T;
}
T* myvar = ary[2];
```



# Pointer Arithmetic

Addition and subtraction of a pointer and an integer gives a new pointer.

```
int a[10];
int* p;
int* q;
p = &a[3];
q = &a[5];
// q-p == 2
// p+1 == &a[4];
// q-5 == &a[0];
// What is q-6?
```

# Implementation

Pointers are represented internally by memory addresses.

The meaning of `p+k` is to add `k*sizeof *p` to the address stored in `p`.

Example: Suppose `p` points to a `double` stored at memory location 500, and suppose `sizeof(double) == 8`. Then `p+1` is a pointer to memory location 508.

508 is the memory location of the first byte following the 8 bytes reserved for the double at location 500.

If `p` points to an element of an *array* of `double`, then `p+1` points to the *next* element of that array.

# Custody of Objects

# Copying and Moving

One of the goals of C++ is to make user-defined objects look as much like primitive objects as possible.

In particular, they can reside in static storage, on the stack, or in the heap, they can be passed to and returned from functions, and they can be initialized and assigned to.

With primitive types, initialization, assignment, call-by-value parameters and function return values are all implemented by a simple copy of the primitive value.

The same is done with objects, but shallow copy is used by default.

This can lead to problems with large objects (cost) and with objects having dynamic extensions (double-delete problem) as discussed in lecture 07.

# Custody

We say that a function or class has **custody** of a dynamically-allocated object if it is responsible for eventually deleting the object.

A simple strategy for managing a dynamic extension in a class is for the constructor to create the extension using `new` and for the destructor to free it using `delete`.

In this case, we say that custody remains in the class.

# Transfer of Custody

Sometimes we need to transfer custody of a dynamic object from one place to another.

For example, a function might create an object and return a pointer to it. In this case, custody passes to the caller, since the creating function has given up custody when it returns.

Example:

```
Gate* makeGate(...) {
    return new Gate(...);
}
```

# Custody of dynamic extensions

Similarly, with a shallow copy of an object with a dynamic extensions, there is an implicit transfer of custody of the dynamic extension from the old object to the new.

Problem: How does the old object give up custody? Possibilities:

1. Explicitly the pointer to the extension to `nullptr`.
2. Destroy the object.

The first is cumbersome and error-prone. The second causes a double-delete if the destructor does the `delete`.

# Move versus copy

What we want in these cases is to **move** the object instead of copying it. The move first performs the shallow copy and then transfers custody to the copy.

Move semantics were introduced in c++11 in order to solve this problem of transfer of custody of dynamic extensions.

# Move Semantics

# When to move?

With primitives, move and copy are the same. With large objects and objects with dynamic extensions, the programmer needs to be able to control whether to move or copy.

C++ introduces a new kind of type called an **rvalue reference**.

An rvalue reference to a type `T` is written `T&&`.

Intuitively, an rvalue reference is a reference to a temporary. The actual semantics are more complicated.

# Temporaries

Conceptually, a **pure** value is a disembodied piece of information floating in space.

In reality, values always exist somewhere—in variables or in temporary registers.

Languages such as Java distinguish between **primitive values** like characters and numbers that can live on the stack, and **object values** that live in permanent storage and can only be accessed via pointers.

A goal of C++ is to make primitive values and objects look as much alike as possible. In particular, both can live on the stack, in dynamic memory, or in temporaries.

# Move semantics

An object can be moved instead of copied. The idea is that the data in the source object is removed from the source object and placed in the target object. The source object is then said to be *empty*.

As we will see, what actually happens to the source object depends on the object's type.

For objects with dynamic extensions, the pointer to the extension is copied from source to target, and the source pointer is set to `nullptr`.

Deleting `nullptr` is a no-op and causes no problems.

We say that **custody** has been transferred from source to target.

# Motivation

A big motivation for move semantics comes from containers such as `vector`.

Containers need to be able to move objects around. Old-style containers can't work with dynamic extensions.

C++ containers support moving an object into or out of the container.

While in the container, the container has custody of the object.

Move is like a shallow copy, but it avoids the double-delete problem.

# Implementation in C++

Here are the changes to C++ that enable move semantics.

1. The type system has been extended to include **rvalue references**. These are denoted by double ampersand, e.g., `int&&`.
2. Results in temporaries are marked as having rvalue reference type.
3. A class has now six special member functions: constructor, destructor, copy constructor, copy assignment, move constructor, move assignment. These are special because they are defined automatically if the programmer does not redefine them.

# Move and copy constructors and assignment operators

Copy and move *constructors* are distinguished by their prototypes.

`class T`:

- *Copy constructor:* `T( const T& other ) { ... }`
- *Move constructor:* `T( T&& other ) { ... }`

Similarly, copy and move *assignment operators* have different prototypes.

`class T`:

- *Copy assignment:* `T& operator=( const T& other ) { ... }`
- *Move assignment:* `T& operator=( T&& other ) { ... }`

# Default constructors and assignment operators

Under some conditions, the system will automatically create default move and copy constructors and assignment operators.

The default copy constructors and copy assignment operators do a shallow copy. Object data members are copied using the copy constructor/assignment operator defined for the object's class.

The default move constructors and move assignment operators do a shallow copy. Object data members are moved using the move constructor/assignment operator defined for the object's class.

Default definitions can be specified or inhibited by use of the keywords `=default` or `=delete`.

# Moving from a temporary object

A mutable temporary object always has rvalue reference type.

Thus, the following code *moves* the temporary string created by the on-the-fly constructor `string("cat")` into the vector `v`:

```
#include <string>
#include <vector>
vector<string> v;
v.push_back( string("cat") );
```

# Forcing a move from a non-temporary object

The function `std::move()` in the `utility` library can be used to force a move from a non-temporary object.

The following code *moves* the string in `s` into the vector `v`. After the move, `s` contains the null string.

```
#include <iostream>
#include <string>
#include <utility>
#include <vector>
vector<string> v;
string s;
cin >> s;
v.push_back( move(s) );
```

# The full story

I've covered the most common uses for rvalue references, but there are many subtle points about how defaults work and what happens in unusual cases.

Some good references for further information are:

- *Move semantics and rvalue references in C++11* by Alex Allain.
- *C++ Rvalue References Explained* by Thomas Becker.

# 12-BracketsWithMove demo

Please look at the 12-BracketsWithMove demo for an example of how move semantics could be incorporated into real code in a way to avoid the double-delete problem.

There are several changes from the code in 06-BracketsCpp.

- A dummy dynamic extension has been added to `Token` for the purpose of exposing the double-delete problem in case move was implemented incorrectly.
- `Stack` now takes custody of the objects put onto it.
- `Token` has new move constructors and move assignment definitions.
- Lots of little changes to make this all work.