CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 12
October 8, 2018

Uses of Pointers

Array data member

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

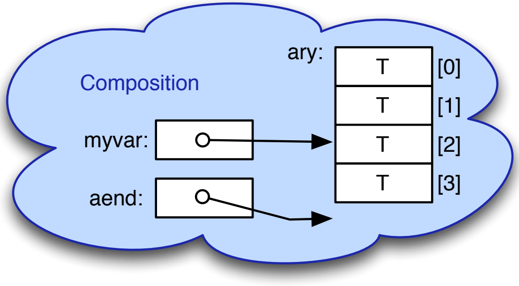
Some ways to represent this relationship.

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.

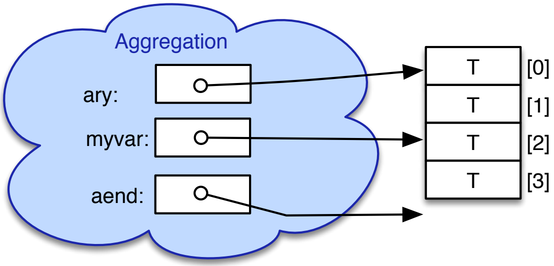
Composition

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



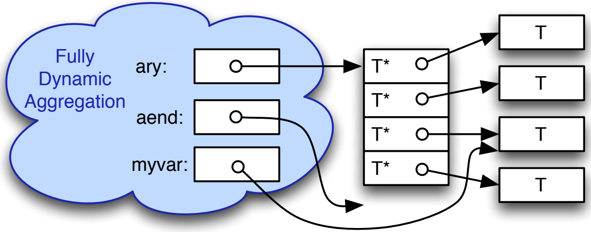
Aggregation

```
T* ary = new T[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.

Feedback on Programming Style

Coding Hints

In the next few slides, I will point out some miscellaneous programming issues that turned up on PS2. Proper C++ style is somewhat different from other languages (include C). Part of professional-level C++ proficiency is learning not just what works but what is simple and efficient.

Zero-tolerance for compiler warnings

Compiler warnings flag things that are not proper C++ usage but may work anyway in some environments. They generally indicate program errors or sloppy style.

You need to learn what the warnings mean and how to avoid them. Don't just ignore warnings because you think they are unimportant. "Unimportant" warnings will mask important ones that result from real bugs in your code.

Example: Comparing an `unsigned int` with an `int` gives such a warning.

Fix: Use appropriate integer types.

Declaration order in classes

There are two schools of thought on the order of declarations within classes:

1. Put the public functions first followed by the private.
   Rationale: The public functions represent the interface and are what clients of the class wnat to see.
2. Put the private data members and functions first followed by the public.
   Rationale: Generally names must be declared before they are used. It's natural to declare data members before functions that might use them, even if C++ provides some flexibility.

In this course, I require the second style: private first, public last.

Construct semantically consistent objects

Constructors should leave objects in a semantically meaningful state.

Avoid the paradigm common in other languages to create uninitialized objects and then initialize data members from member functions.

Use `break`

Instead of

```
bool exit = false;
while (!exit) {
  ...
  if (...) exit = true;
  else {
     ...
  }
}
```

use

```
for (;;) {
  ...
  if (...) break;
   ...
}
```

Use tolower()

Instead of

```
if (input=='Q' || input=='q') ...
```

use

```
#include <cctype>
...
input = tolower(input);
if (input=='q') ...
```

Use `switch`

Instead of

```
if (input=='a' || input=='b' || input=='c') { ... }
else if (input=='p') {
  ...
```

use

```
switch (input) {
case 'a':
case 'b':
case 'c': ...; break;
case 'p': ...; break;
}
```

Use stream input to read data

Instead of

```
int x;
string s;
s.getline(in);
// extract substring
// convert substring to number
 ...
```

use

```
int x;
in >> x;
```

Instead of

```
for (;;) {
   in >> x;
   if ( <error> ) {
      <handle error>
   }
   else {
      <do stuff>
      in >> y;
      if ( <error> ) {
         <handle error>
      }
      else {
        <do stuff>
      }
   }
}
```

Use `continue`

```
for (;;) {
   in >> x;
   if ( <error> ) {
      <handle error>
      continue;
   }
   <do stuff>
   in >> y;
   if ( <error> ) {
      <handle error>
      continue;
   }
   <do stuff>
}
```

Use `new` and `delete`, not `malloc` and `free`

C uses `malloc` and `free` to allocate and free dynamic storage.

C++ uses `new` and `delete`.

What are the differences?

1. `new` and `delete` are type safe; `malloc` and `free` are not.
2. `new` calls the constructor and `delete` calls the destructor. `malloc` and `free` are unaware of C++ classes and just handle uninitialized storage.
3. Array forms `new[]` and `delete[]` call default constructors and destructors of array elements.

Don't use `malloc` and `free` in C++ programs.

End-of-file handling

Don't use

```
while (!in.eof()) {
  in >> x;
  <do stuff with x>
}
```

to read and process a file of numbers. Even if `in.eof()` returns `false`, the next read might fail. Instead, use

```
for (;;) {
  in >> x;
  if (in.fail()) { <handle error/eof condition> }
  <do  stuff with x>
}
```

Include guards

Include guards are a method of using the C++ preprocessor to make sure that the declarations in a header file are not included more than once in a compilation. Here's how they work:

- A preprocessor symbol `GATE_HPP` is associated with a header file `gate.hpp`. Initially, `GATE_HPP` is undefined.
- Before `gate.hpp` is processed, `#ifndef GATE_HPP` is used to test if `GATE_HPP` is already defined.
- If it is, `gate.hpp` has already been processed and is skipped.
- If not, `#define GATE_HPP` defines `GATE_HPP` and the header file `gate.hpp` is processed.

Where do the include guards go?

They could be used to protect either the `#include "gate.hpp"` statement or the body of the header file `gate.hpp`.

Because there may be many `#include "gate.hpp"` statements in the program but there is only one `gate.hpp` file, they are normally placed inside the header file itself, e.g.,

```
// File gate.hpp
#ifndef GATE_HPP
#define GATE_HPP
   <body of header file>
#endif
```