ECS 165B: Database System Implementation
Lecture 2

UC Davis, Spring 2011

Portions of slides based on earlier ones by Raghu Ramakrishnan, Johannes Gehrke, Jennifer Widom, Bertram Ludaescher, and Michael Gertz.

Announcements

No class Friday (NorCal Database Day 2011 @ UCD)

http://dbday.cs.ucdavis.edu

Warmup homework assignment (serialization and memory management in C++) out later **today**, due Sunday @ 11:59pm

Project starts with forming teams on Monday

Today's Agenda

Coding skills: serialization and memory management in C++

File and buffer management in a DBMS

Reading: Chapter 13 of textbook

Coding Skills: Memory Management in C++

Much of DavisDB project work (starting with part 1) will require writing **serialization** and **deserialization** code

Students last year had trouble with this, hence we'll cover the basics up front (and get some practice in the "warmup homework")

Coding Skills: Memory Management in C++

DBMS is a long-running server process, cannot afford to **leak memory**

Many (most) “real” applications are like this: web servers, GUI applications, operating system device drivers, ...

In DavisDB, you’ll be required to clean up memory resources properly

What's Wrong With This Code?

```
class Foo {
   char* str_;
   ...
   void setString(char* str) {
        str_ = strdup(str);
   }
   ...
   ~Foo() {
   }
};
```

What's Wrong With This Code?

```
class Foo {
   char* str_ = 0;
   ...
   void setString(const char* str) {
       if (str_ != 0) {
           free(str_);
       }
       str_ = strdup(str);
   }
   ...
   ~Foo() {
       if (str_ != 0) {
           free(str_);
           str_ = 0;
       }
   }
}
```

Basic Idea in Memory Management

Allocated memory must be freed when no longer needed

Don't have to worry about this (as much) in garbage-collected languages like Java, C#, Python, etc

Definitely **do** have to worry about this in C/C++

To be a good C/C++ programmer, need to become very disciplined about memory management

How to Allocate and Free Memory in C++

Confusingly, there are **three** ways to allocate memory from the heap in C++: `malloc`, `new`, and `new[]`

`malloc`	inherited from C, used by C library routines like strdup
`new`	typed, object-oriented version of malloc; used to allocate a **single** object
`new[]`	used to allocate an **array** of objects

Memory must be freed differently depending on how it was allocated:

`malloc`	<->	`free`
`new`	<->	`delete`
`new[]`	<->	`delete[]`

How to Remember to Free Unused Memory

Whenever you write a piece of code allocating some memory, you already must be thinking about **who** will free it, **where** and **when**

Ownership of memory must be very clearly understood

DESCRIPTION
The strdup() function allocates sufficient memory for a copy of the string s1, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3).

With strdup, it is clear that the caller owns the returned memory

So What's Wrong With This Code?

```
struct Point {
   int x;
   int y;
}

Point* foo(int x, int y) {
   Point p;
   p.x = x;
   p.y = y;
   return &p;
}

void bar() {
   Point* point = foo(1,2);
   delete point;
}
```

So What's Wrong With This Code?

```
struct Point {
   int x;
   int y;
}

Point* foo(int x, int y) {
   Point p;
   p.x = x;
   p.y = y;
   return &p;  // uh-oh, returning a pointer to stack-
               // allocated memory.  big no-no.
}

void bar() {
   Point* point = foo(1,2);
   delete point;
}
```

So What's Wrong With This Code?

```
struct Point {
   int x;
   int y;
}

void foo(int x, int y,
            Point* p) {
   p->x = x;
   p->y = y;
}

void bar() {
   Point point;
   foo(1,2, &point);
}
```

```
struct Point {
   int x;
   int y;
}

Point* foo(int x, int y) {
   Point* p = new Point();
   p->x = x;
   p->y = y;
   return p;
}

void bar() {
   Point p = foo(1,2);
   delete p;
}
```

Finding Memory Leaks

Very useful and easy-to-use tool on Unix-based platforms (including CSIF): **valgrind**

Usage: **valgrind [options] <prog-and-args>**

Will tell you if you've leaked memory, and where it was allocated

We'll use this in the warmup homework, and in grading your project components

Smart Pointers: a Useful C++ Idiom for Memory Management

Basic idea: instead of working with “raw” pointer, like

```
Foo* ptr = new Foo(1);
ptr = new Foo(2); // oops, leaked the first object
```

work with a “wrapped” version that automatically frees the underlying object when leaving scope

```
std::auto_ptr<Foo> ptr = new Foo(1);
ptr = new Foo(2); // first object deleted automatically
                  // on assignment
```

Smart Pointers: Pros and Cons

Pros: automates away much of the grunt work of memory management; great idea so long as used uniformly in a code base and all developers involved understand them

Cons: no free lunch, still need to understand underlying memory management concepts, plus smart pointers introduce their own complexities; bad idea to mix code that uses smart pointers with code that doesn't (very confusing)

Since this is a "DIY" class and you need to understand memory management anyway, we'll forego smart pointers

Serialization/Deserialization

Serialization (aka **marshalling**): conversion of data structure or object into format that can be stored (e.g., in file or memory buffer, or transmitted on network) and **deserialized** (or **unmarshalled**) later to recover data structure/object

As with memory management and smart pointers, some standard facilities (e.g., Boost serialization library) exist to help with this

But in the spirit of DIY (and avoiding going deeply into Boost libraries), we'll do it ourselves in this class

Primitive for Serialization: memcpy

Given a structure Foo, how to serialize Foo into a byte-buffer?

```
struct Foo {
    int x;
    float y;
} f1 = {1, 0.5};
char buffer[128];
memcpy(buffer, &f1, sizeof(f1));
```

How to deserialize? Use memcpy again:

```
struct Foo f2;
memcpy(&f2, buffer, sizeof(f2));
```

Serialization Gotchas

One problem with preceding approach: deserializing Foo on another machine may not work properly

- endianness issues
- size of data types (32-bit vs. 64-bit vs. ...)

We'll ignore this issue for DavisDB (single-machine)

Second problem: **versioning**. Suppose a later version of code adds a third field to Foo. What can go wrong?

We'll ignore this issue too for DavisDB (single-version)

Third problem, this one we **do** need to worry about for DavisDB: **memcpy of the structure only works for "plain old data"...**

What's Wrong Here?

```
struct Bar {
  int x;
  char* str;
};
char buffer[128];
char str[] = new char[64];
sprintf(str, "hello, world!");
struct Bar bar = {1, str};
memcpy(buffer, &bar, sizeof(bar));
delete[] str;
...
memcpy(&bar, buffer, sizeof(bar));
```

What's Wrong Here?

```
struct Bar {
   int x;
   char* str;   // structure is not "plain old data"
};
char buffer[128];
char str[] = new char[64];
sprintf(str, "hello, world!");
struct Bar bar = {1, str};
memcpy(buffer, &bar, sizeof(bar));
delete[] str;
...
memcpy(&bar, buffer, sizeof(bar));  // bar will have
                                    // bad str
                                    // pointer
```

The Fix: Custom Serialize/Deserialize Methods

```
class Bar {
   int x;
   char* str = 0;
   void serialize(char* buffer) {
        memcpy(buffer, &x, sizeof(x));
        int len = strlen(str);
        memcpy(buffer+sizeof(x), &len, sizeof(len));
        memcpy(buffer+sizeof(x)+sizeof(len), str,
            len*sizeof(char));
   }
   static Bar* deserialize(char* buffer) {
        Bar* bar = new Bar();
        memcpy(&(bar->x), buffer, sizeof(bar->x));
        int len;
        memcpy(&len, buffer+sizeof(bar->x), sizeof(len));
        bar->str = new char[len+1];
        memcpy(bar->str, buffer+sizeof(bar->x)+sizeof(len),
                        len*sizeof(char));
   }
   ...
};
```

The Fix: Custom Serialize/Deserialize Methods

Usage:

```
Bar b;
b.x = 1;
b.str = “hello, world!”;
char buffer[128];
b.serialize(buffer);
...
Bar* c = Bar::deserialize(buffer, &c);
...
delete c;
```

The Fix: Custom Serialize/Deserialize Methods

Code can be greatly simplified using buffer that automatically keeps track of read/write position: i.e., behaves like a **stream**

```
void serialize(char* buffer) {
      memcpy(buffer, &x, sizeof(x));
      int len = strlen(str);
      memcpy(buffer+sizeof(x), &len, sizeof(len));
      memcpy(buffer+sizeof(x)+sizeof(len), str,
           len*sizeof(char));
}
```

becomes

```
void serialize(Bytestream* buffer) {
      buffer->write(&x, sizeof(x));
      int len = strlen(str);
      buffer->write(&len, sizeof(len));
      buffer->write(str, len*sizeof(char));
}
```

part of homework will ask you to write such a Bytestream class