COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/

Today

- Class syntax, Constructors, Destructors
- Static methods
- Inheritance, Abstract methods
- Access modifiers
- Namespaces
- Smart Pointers

Dynamic Memory Allocation in C++

- `new` instead of `malloc`
- `delete` instead of `free`
- `delete[]` for arrays!

Allocating Objects

```
int * p = new int;
*p      = 42;
int * q = p;
delete q;
```

Allocating Arrays

```
int * p = new int[5];
p[4]    = 42;
int * q = &p[1];
delete[] p;
```

C++ structs

```
struct stack {
  element * first;
  void init();
  bool isempty();
  void push(element * value);
  element * pop();
  element * top();
  void free();
};
```

Implementing the Functions

```
void stack::push(element * value) {
  value->next = first;
  first = value;
}
element * stack::top() { return first; }
void stack::init()     { first = NULL; }
```

Using the `struct`

```
struct stack s;
s.init();
s.push(e);
cout << *(s.first); // allowed!
```

`structs` vs. `classes`

- All members are `public` (anyone can freely access)
- No inheritance or subtyping
- Syntactically virtually identical to C++ classes

C++ classess

```
class stack {
  public:
    stack(); // constructor!
    bool isempty();
    void push(element * value);
    element * pop();
    element * top();
    ~stack(); // destructor!
  private:
    element * first;
};
```

Using the `class`

```
stack s;

cout << "Stack is " << (s.isempty()?"":"not ")
     << "empty" << endl;
s.push(e);
cout << *(s.first); // NOT allowed!
```

Implementing the Methods

```
// constructor
stack::stack() {
  first = NULL;
}
bool stack::isempty() {
  return NULL == first;
}
```

Implementing the Destructor

```
// destructor
stack::~stack() {
  element * save;
  while (! isempty()) {
    save = first->next;
    delete first;
    first = save;
  }
}
```

About Destructors

- A class can only have one destructor
- Destructors take no arguments and return `void`
- Destructors must not throw exceptions
- Destructors should never be called directly
- Destructors should always be `virtual`

Who calls Destructors?

- `delete` will call the destructor for the object
- `delete[]` will call the destructor for each element in the array (and, possibly, the array itself)
- For stack-allocated objects, the destructor is called when the scope ends

Constructors and Destructors

```
void f() {
  stack s2;
  stack * h = new stack;
  delete h;
  throw 4;
}
int main(int argc, char ** argv) {
  stack s;
  try { f() }
  catch (...) { }
  return 0;
}
```

About Constructors

- A class can have many constructors (overloading)
- Constructors can be called with `new`
- Constructors must return `void`
- Constructors may throw exceptions
- If no constructor is specified, a default constructor that does nothing is generated (not even zeroing as in Java!)

Constructors with Arguments

```
stack::stack(int max) {
  first = NULL;
  this->max = max;
}
stack * h = new stack(42);
stack s(42);
stack s2 = 42; // works too (coersion constructor)
```

The Copy Constructor

- Copy constructors are implicitly used for assignments:

```
stack s(42);
stack t = s; // here!
```

- Copy constructors for a class of type T must have the form `T::T(const T&)`
- You can also use them explicitly using constructs like:

```
T s(42);
T t(s); // here!
```

Constructors and Arrays

```
stack * sv = new stack[10];
```

- The above allocates an array for 10 "stack" objects
- Each object is initialized using the default constructor (no arguments!)
- "sv" must be freed using "delete[]"

Static Methods

- Use the modifier `static`
- No `this` available in the body
- No access to fields of the class

$\Rightarrow$ They work just like in Java!

Note: you can also have `static` fields.

Inheritance

```
class stack : public vector {
  public:
    stack(int size) : vector(size) { };
    element * pop() { return removeAt(size()-1); }
    element * top() { return get(size()-1); }
    void push(element * e) { add(e, size()); }
}
```

Virtual Functions

- In Java, all methods are by default virtual
- In C++, you must declare methods as `virtual` for the same effect; otherwise the static type of the receiver determines what function is called!

Example

```
class P {
  public: void m() { cout << 1 << endl; }
}
class S : public P {
  public: void m() { cout << 2 << endl; }
}
P * p = new S;  p->m(); // 1
S * s = new S;  s->m(); // 2
```

Example

```
class P {
  public: virtual void m() { cout << 1 << endl; }
}
class S : public P {
  public: virtual void m() { cout << 2 << endl; }
}
P * p = new S;  p->m(); // 2
S * s = new S;  s->m(); // 2
```

Abstract Functions

Abstract functions in C++ work just like in Java, except for the syntax:

```
class AbstractBaseClass {
  public: virtual void absfun(int args) = 0;
}
```

The "= 0" indicates that the function is abstract.

Multiple Inheritance

- In Java, a class can implement multiple interfaces
- In C++, a class can inherit from multiple classes!

```
class Child : public Father, public Mother {
  // ...
}
```

Multiple Inheritance

- The inheritance DAG must not have any cycles
- However, diamonds are allowed:

```
class Human { string name; }
class Father : public Human { }
class Mother : public Human { }
class Child : public Father, public Mother {
  // ...
}
```

Here, `Child` would have the field `name` twice!

Virtual Inheritance

The keyword `virtual`, when used when extending a class, will ensure that inherited fields are not replicated:

```
class Human { string name; }
class Father : public virtual Human { }
class Mother : public virtual Human { }
class Child : public Father, public Mother {
  // ...
}
```

Here, `Child` would have the field `name` only once!

`protected`

- Members that are declared `protected` are only visible to subtypes
- This is different from Java where `protected` members are also visible in the same package!

Namespaces

- C++ namespaces are like Java packages
- Instead of writing "`package`", you write:

```
namespace myspace {
  class Foo { ... };
}
```

- Instead of writing "`import`", you write "`using namespace`"
- Instead of writing "`java.util.List`", you write "`std::list`"

Friends

`friend` allows particular functions to access (all) `private` members:

```
class complex {
  private:
    double re, im;
  public:
    complex(double, double);
  friend double geo::dist(complex, complex);
}
```

Smart Pointers

Smart pointers are constructions that try to address common issues with "normal" pointers:

- Dangling pointers (use-after-free)
- Memory leaks (missing free)
- Failure to initialize (random pointers)
- Aliasing (multiple pointers to same object)

Exceptions and Memory Leaks

```
int f() { throw 52; }
int g() {
  SomeClass * s = new SomeClass;
  f();
  delete s; // not called!
}
int main() {
  try { g(); }
  catch (int i) { }
}
```

A Bad "Solution"...

```
int f() { throw 52; }
int g() {
  SomeClass * s = new SomeClass;
  try { f(); }
  catch(...) { delete s; throw; }
  delete s;
}
int main() {
  try { g(); }
  catch (int i) { }
}
```

The `auto_ptr` solution

```
int f() { throw 52; }
int g() {
  auto_ptr<SomeClass> s = new SomeClass;
  f();
}
int main() {
  try { g(); }
  catch (int i) { }
}
```

How does it work?

- `auto_ptr<SomeClass>` instance "s" acts in most respects just like "`SomeClass*`"
- Except, the destructor of `auto_ptr` calls `delete` on the underlying instance

$\Rightarrow$ `delete` automatically called whenever `s` leaves the scope, including during an exception!

Other properties of `auto_ptr`

- When an alias is created, `auto_ptr` sets the old pointer to `null`

$\Rightarrow$ Only one reference to the object at any time

$\Rightarrow$ Obvious which `auto_ptr` will free

$\Rightarrow$ No unexpected change of object through other alias

Questions

?