Objects and Namespaces

...

Original from : Mike Precup (mprecup@cs.stanford.edu)
ENJMIN Edition 2016

Namespaces

duration

stack

string

seconds

tuple

pair

vector

ignore

fstream

hours

stringstream

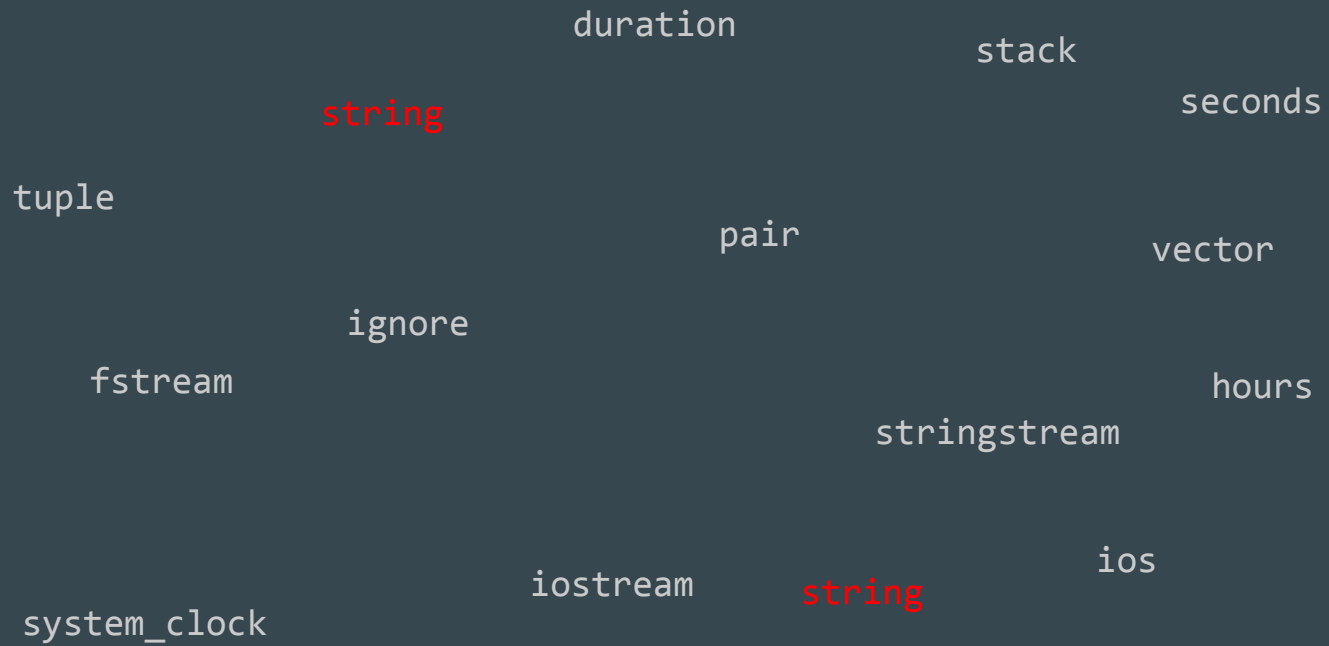
ios

iostream

string

system_clock

Namespaces


duration
stack
string
seconds
tuple
pair
vector
ignore
fstream
hours
stringstream
ios
iostream
string
system_clock

Namespaces

Is there a problem with having the standard library take common names?

Depends on who you ask. It is more convenient sometimes, but if you want your own implementation, it's unfortunate.

Namespaces

Is there a problem with having my library take common names?

Yes. No qualifications on this one. It's way too easy for two people's libraries to conflict on naming! This was a common issue in C, and the fixes for it generally involve abusing the compiler, if a fix was even possible.

Namespaces

Most modern languages fix this problem in similar ways. You may already be familiar with namespaces!

```
# Generate a random number in Python

import random

print random.random()
```

Accessing Namespace Members

Nothing fancy here, just use `namespace::member` (i.e. `std::string`)

Many languages use `.`, but that's the same as for accessing members in an object

In C++, the scope resolution operator is `::`

A Basic Namespace

```
namespace stanford {

  void foo();

}

int main() {

  stanford::foo();

}
```

Splitting Namespaces

```
namespace stanford {

  void foo();

}

namespace stanford {

  void bar();

}
```

Many files can contribute to a namespace!

The `std` Namespace

- `std` is a special namespace used by the standard library
- Technically, it's special in that you're not allowed to add to it
  - Most compilers let you anyways

Being Lazy

You obviously don't need to write `std::` every single time you use string, we hadn't been until today. How can we get around the extra typing?

Being Lazy

You obviously don't need to write `std::` every single time you use string, we hadn't been until today. How can we get around the extra typing?

- using namespace std;
  - Makes entire namespace visible

Being Lazy

You obviously don't need to write `std::` every single time you use string, we hadn't been until today. How can we get around the extra typing?

- using namespace std;
  - Makes entire namespace visible
- using std::string;
  - Makes one member visible

Nested Namespaces

You can nest namespaces like so:

```
namespace A {
    namespace B {
        void foo();
    }
}
```

You can then access `foo` with `A::B::foo`

Nested Namespaces

C++11 adds another syntax for it:

```
namespace A::B {

    void foo();

}
```

You can then access `foo` with `A::B::foo`

`Student::`

`::` doesn't just apply to namespaces, it's the scope resolution operator

If you want to access something scoped to a class, you need to use `::`

:: vs .

- ::
  - Use the scope resolution operator when accessing something that belongs to all instances of a class
- .
  - Use the member operator when accessing something that belongs to an object

“All Instances of a Class”

So what belongs to every instance of a class?

- Implementations for functions
- `static` members
- Nested classes

`static`

- `static` is, as far as I know, the most overloaded keyword in C++
- Today we'll be talking about what it means in a class definition
- There are many other uses of `static`, so if you see it outside of a class, it probably means something different

`static` Members

`static` members are shared by all instances of a class

```
class GlobalCounter {
public:
    int increment() {
        return ++count;
    }
private:
    static int count = 0;
};
```

```
int main() {
    GlobalCounter gc1;
    GlobalCounter gc2;
    // Prints 1
    cout << gc1.increment() << endl;
    // Prints 2
    cout << gc2.increment() << endl;
}
```

Fun Fact

This code doesn't actually compile!

```
class GlobalCounter {
public:
    int increment() {
        return ++count;
    }
private:
    static int count = 0;
};
```

```
int main() {
    GlobalCounter gc1;
    GlobalCounter gc2;
    // Prints 1
    cout << gc1.increment() << endl;
    // Prints 2
    cout << gc2.increment() << endl;
}
```

Fun Fact

This code does actually compile!

```
class GlobalCounter {
public:
    int increment() {
        return ++count;
    }
private:
    static int count;
};

int GlobalCounter::count = 0;
```

```
int main() {
    GlobalCounter gc1;
    GlobalCounter gc2;
    // Prints 1
    cout << gc1.increment() << endl;
    // Prints 2
    cout << gc2.increment() << endl;
}
```

u wot m8

This class also compiles:

```
class Angle {
public:
    // TODO: Implement actual functions
private:
    const static double PI = 3.14159;
    double angle;
};
```

Implementing `static` Members

Most of the time, you need what is essentially a prototype and an implementation for `static` members, like `counter`.

There's an exception made for `const` numeric types.

When you provide an implementation, you do it like so:

```
int GlobalCounter::count = 0;
```

Note the scope resolution operator!

Accessing `static` Members

```
class Day {
public:
    const static int NUM_HOURS = 24;
    // other stuff
private:
    // other stuff
};

int main() {
    cout << Day::NUM_HOURS << endl; // Prints 24
}
```

`static` Functions

```
class Day {
public:
    static int numHours() {
        return 24;
    }
};

int main() {
    cout << Day::numHours() << endl; // Prints 24
}
```

Nested Classes

You can nest classes, and it works pretty much as expected

```
class A {
public:
    class B {
        // Implementation
    };
};

int main() {
    A::B b;
}
```

Nested Classes

- Access specifiers (e.g. `public`, `private`) work as expected
- Gets a bit weird when you're using templates
  - We'll talk about that in a later lecture