Designing Types

Reid Watson
(rawatson@stanford.edu)

Administrivia

- We’re grading assignment one
- Assignment two is out!
- If you didn’t do assignment one you **must** complete this assignment to receive credit for this class

Types in C++

Goals for today:

Why and **how** can we design types in C++?

Types in C++

- We've talked about some extremely useful types
  - **string**, **vector**, **map**, **iostream**, and many more
- We haven't really talked about how these types came into existence. Are they special because they're part of the standard?
- How do we create our own types?

Types in C++

Wait, why do we care about creating our own types anyway?

Types in C++

Why do we want to create new types?

- To **implement** new algorithms, data structures, and procedures
- To **simplify** the usage of existing tools
- To **clarify** the meaning of a piece of data

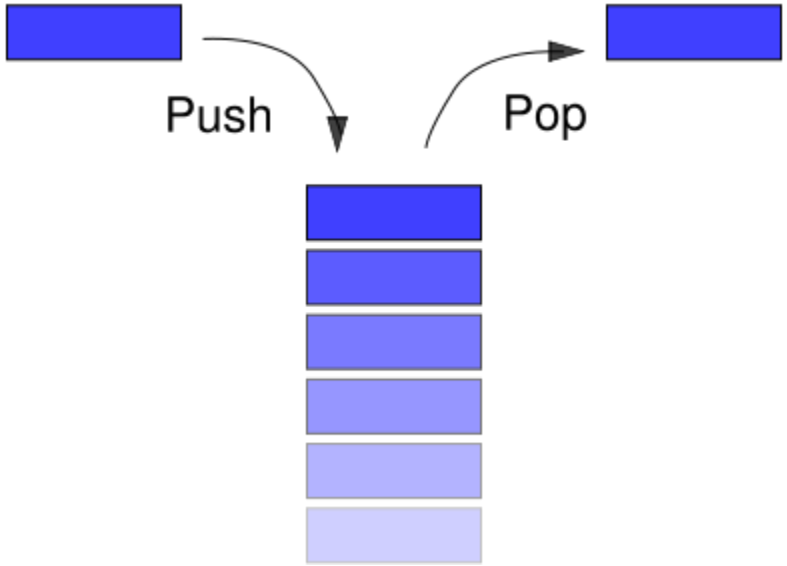
Types in C++

Why do we want to create new types?

- To **implement** new algorithms, data structures, and procedures
- To **simplify** the usage of existing tools
- To **clarify** the meaning of a piece of data

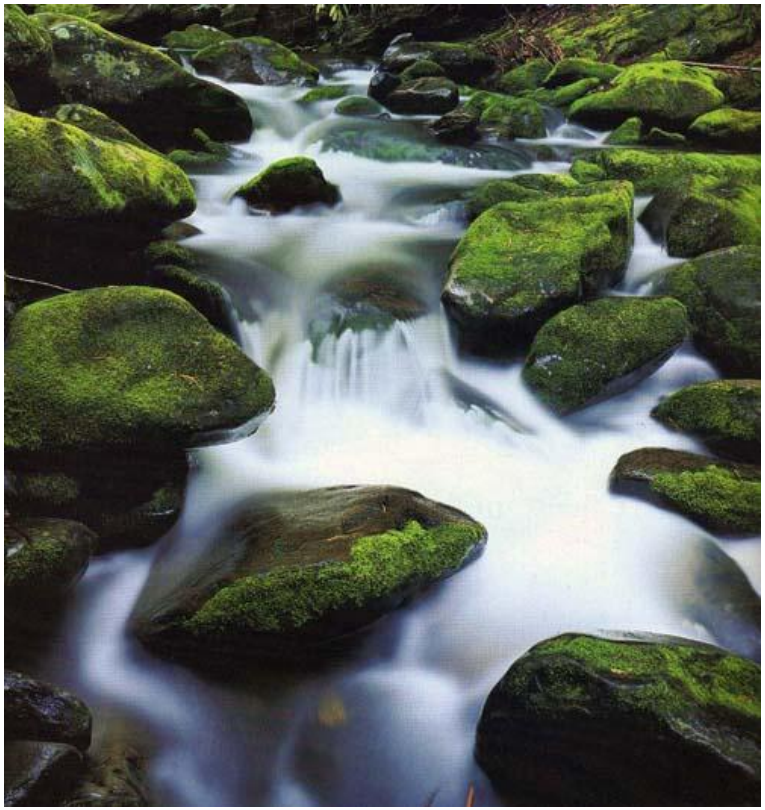
Types in C++

The **stack** type in C++ is an **implementation** of the stack data structure



Types in C++

The **istream** and **ostream** types **implement** operations on output buffers



This has nothing to do with iostreams, I just wanted a picture.

Types in C++

Why do we want to create new types?

- To **implement** new algorithms, data structures, and procedures
- To **simplify** the usage of existing tools
- To **clarify** the meaning of a piece of data

Types in C++

Remember the **Node** type in GraphViz?

How about the **Edge** Type?

These types don't add anything, but they do **simplify** our code

```
struct Node {
  double x;
  double y;
};
```

```
struct Edge {
  int start, end;
};
```

Types in C++

Why do we want to create new types?

- To **implement** new algorithms, data structures, and procedures
- To **simplify** the usage of existing tools
- To **clarify** the meaning of a piece of data

Types in C++

- Using reasonable types can help clarify what code does
- For example, **typedef** does nothing but give another name to an existing type
- This can be very useful though
- Say we wanted to create a mapping between the name of a person and their address book (a mapping from names to phone numbers):

Types in C++

```
// Do you prefer this:
typedef map<string, vector<int>> AddressBook;
typedef pair<string, string> Name;
map<Name, AddressBook> contactsFor;

// Or this:
map<pair<string, string>, map<string,
vector<int>>> contactsFor;
```

Types in C++

```
// Some students created a "force" type in graphviz
struct Force {
  double x, y;
};

// This meant they could keep a single vector of forces instead of a vector for x forces and a separate vector for y forces
```

Types in C++

- We've talked about **primitive types**, like **int**, **char**, and **bool**
- I've briefly mentioned **typedef**, a means to refer to a type by another name
- We've seen **struct**s in GraphViz, and they deserve a bit more attention

Types in C++

- Many concepts we work with can be described as a single entity defined by multiple components.
- A person has a name and an age

```
struct Person {
  string name;
  int age;
};
```

Types in C++

- A class has an instructor and a set of students

```
struct Class {
  Person instructor;
  vector<Person> students;
}
```

Designing our own Type

- We're going to design a type which represents a two dimensional point.
- We have **eight** versions of this type, which we're going to go through in order

Designing Our Own Type

1. First definition
2. Functions on our type
3. Member Functions
4. Using helper member functions
5. Using static member variables
6. Modifying data with helper functions
7. Using **class** instead of **struct**
8. Organizing our code for reuse

Designing Our Own Type

1. First definition
2. Functions on our type
3. Member Functions
4. Using helper member functions
5. Using static member variables
6. Modifying data with helper functions
7. Using **class** instead of **struct**
8. Organizing our code for reuse

Designing Our Own Type

- In its most basic form, a two dimensional point is an x coordinate and a y coordinate.
- We can get and set these values

Designing Our Own Type

See code in point-1.cpp

Designing Our Own Type

1. First definition
2. Functions on our type
3. Member Functions
4. Using helper member functions
5. Using static member variables
6. Modifying data with helper functions
7. Using **class** instead of **struct**
8. Organizing our code for reuse

Designing Our Own Type

- We might want to define a function which operates on our type
- For example, what if we wanted to move the code for printing a point
- Let's look at an implementation of that

Designing Our Own Type

See code in point-2.cpp

Designing Our Own Type

1. First definition
2. Functions on our type
3. Member Functions
4. Using helper member functions
5. Using static member variables
6. Modifying data with helper functions
7. Using **class** instead of **struct**
8. Organizing our code for reuse

Designing Our Own Type

- The name of those functions was kind of awkward
  - Why do we have to say `printPoint(a);`
  - We already know that a is a point
- What we want to be able to do is say `a.print();`
- We can do this by defining the print **member function** of the type Point.

Designing Our Own Type

See code in point-3.cpp

Designing Our Own Type

1. First definition
2. Functions on our type
3. Member Functions
4. Using helper member functions
5. Using static member variables
6. Modifying data with helper functions
7. Using **class** instead of **struct**
8. Organizing our code for reuse

Designing Our Own Type

- The output of our printPolar function looked a bit funny
  - Wouldn't it be nicer if we could see our output in terms of degrees rather than radians?
- Let's try defining another member function for converting between radians and degrees
- Notice that this function will only be used inside the Point type.

Designing Our Own Type

See code in point-4.cpp

Designing Our Own Type

1. First definition
2. Functions on our type
3. Member Functions
4. Using helper member functions
5. Using static member variables
6. Modifying data with helper functions
7. Using **class** instead of **struct**
8. Organizing our code for reuse

Designing Our Own Type

- Notice how we had to define a `kPi` variable to write the `degToRad` function
- We don't want to make this a global variable that everyone has to know about if it's only ever used inside of the Point type
- Let's create a **static member variable**

Designing Our Own Type

- A **static member variable** will only be created once for each type
- Different points may have different `x` and `y` values, but all points will have the same value for `kPi`

Designing Our Own Type

See code in point-5.cpp

Designing Our Own Type

1. First definition
2. Functions on our type
3. Member Functions
4. Using helper member functions
5. Using static member variables
6. Modifying data with helper functions
7. Using **class** instead of **struct**
8. Organizing our code for reuse

Designing Our Own Type

- We can also write member functions to modify the data inside the class
- Let's write a quick one to normalize the magnitude of our vector

Designing Our Own Type

See code in point-6.cpp

Designing Our Own Type

1. First definition
2. Functions on our type
3. Member Functions
4. Using helper member functions
5. Using static member variables
6. Modifying data with helper functions
7. Using **class** instead of **struct**
8. Organizing our code for reuse

Designing Our Own Type

- The idea of **encapsulation** comes up a lot when designing classes
- The print functions should work regardless of whether we used `x` and `y` value or `r` and `theta` values
- **Private** member variables and functions can't be used outside of member functions

Designing Our Own Type

- This means that if we wanted to rewrite our point type to store data in polar form, users of our point type wouldn't be affected

Designing Our Own Type

See code in point-7.cpp

Designing Our Own Type

1. First definition
2. Functions on our type
3. Member Functions
4. Using helper member functions
5. Using static member variables
6. Modifying data with helper functions
7. Using **class** instead of **struct**
8. Organizing our code for reuse

Designing Our Own Type

- The last change we make is to separate our code into separate files
- This makes it easier for other people to use our code, and lets them use our tools just like any other

Designing Our Own Type

- The last change we make is to separate our code into separate files
- We'll put the **interface** of our type in a **header** (.h) file
- We'll put the **implementation** of our type in an **implementation file** (.cpp)
- We'll put our program in a separate implementation file

Designing Our Own Type

See code in point.cpp, point.h, and main.cpp