Designing Types

Cristian Cibils
(ccibils@stanford.edu)

Types in C++

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 and why they exist in the first place
- 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 functions
- To **simplify** the usage of existing code
- To **clarify** the meaning of data

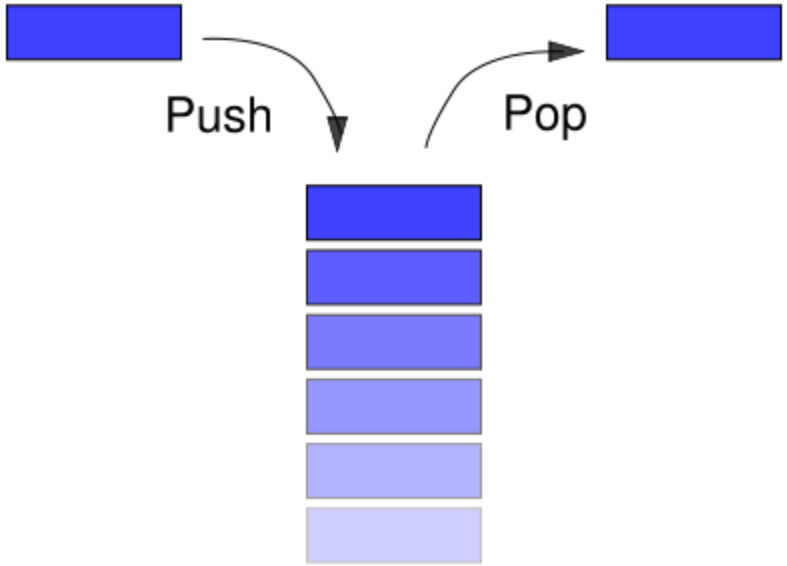
Types in C++

Why do we want to create new types?

- To **implement** new algorithms, data structures, and functions
- To **simplify** the usage of existing code
- To **clarify** the meaning of data

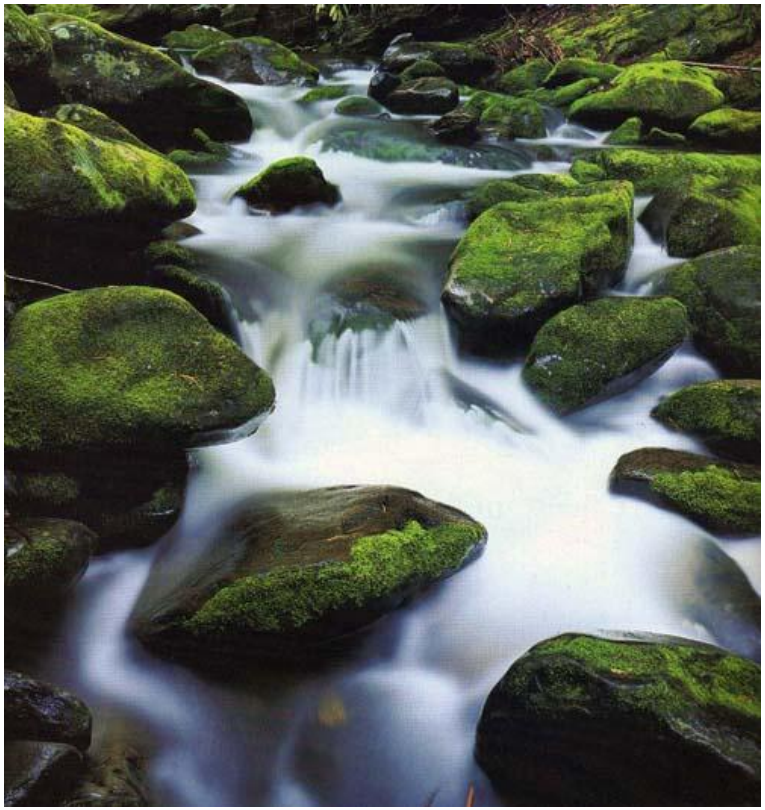
Types in C++

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



Types in C++

The **istream** type **implements** operations on input 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?

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

How about the **Edge** Type?

```
struct Edge {
  size_t start;
  size_t end;
};
```

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

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

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 two vectors of doubles

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
  - Discussed how to use them for basic things
  - Have NOT discussed in depth what they are and how to use them for more complicated things

Types in C++

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

```
struct Student {
  string name;
  int SUID;
};
```

Types in C++

- A class has an instructor and a set of students

```
struct Class {
  string instructor;
  set<Student> 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`
- Saves space and time

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 point

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 types
- 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
  - 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

- The (almost) only technical difference between a **class** and a **struct** in C++ is whether data defaults to private or public
  - In a struct, all data is public unless marked private, in a class, all data is private unless marked public
- The real difference between the two is style

“A struct simply feels like an open pile of bits with very little in the way of encapsulation or functionality. A class feels like a living and responsible member of society with intelligent services, a strong encapsulation barrier, and a well defined interface”

-Bjarne Stroustrup

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

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

Preprocessor

The C++ preprocessor allows for the creation **macro**s that are executed at compile time

```
//Pastes the contents of the file here
#include file
```

Preprocessor

The C++ preprocessor allows for the creation **macro**s that are executed at compile time

```
//Does 2 things: makes TERM “defined”
//replaces all instances of TERM with
//value (value is optional)
#define TERM value
```

Preprocessor

The C++ preprocessor allows for the creation **macro**s that are executed at compile time

```
//Executes the code only if TERM is
//defined
#ifdef TERM
//Code
#endif
```

Preprocessor

The C++ preprocessor allows for the creation **macro**s that are executed at compile time

```
//Executes the code only if TERM is
//not defined
#ifndef TERM
//Code
#endif
```