C++ Templates

CSE 1320

Alex Dillhoff

University of Texas at Arlington

Templates

Variables are useful because it allows for the creation of general solutions that do not depend on hard-coded values.

What happens when we must implement the same functionality for multiple types?

Templates

A solution would require multiple functions to be created for each unique type.

If the functions perform the same task, we are essentially hard-coding the types.

Templates

To understand this better, let's look at a motivating example: dynamic arrays.

```
class DoubleArray {
    double *arr_;
    int sz_;

    public:
        DoubleArray(int s);
        ~DoubleArray() { delete[] arr_; }
};
```

Templates

If we want to create a dynamic array for a different type, we'll need to create another class.

Essentially, we are hard-coding the type. Luckily, C++ offers a solution: **templates**.

Templates

Templates in C++ allow us to parameterize a class or function with a type.

```
template<typename T>
void print_any(T& obj) {
    cout << obj << endl;
}
```

Templates

We can use a template type to parameterize the type for our dynamic array class.

This is exactly what the `vector` class does!

Dynamic Array Class

```
template<typename T>
class DynArray {
    T* arr_;
    int sz_;

    public:
        DynArray(int s);
        ~DynArray() { delete[] arr_; }
};
```

Dynamic Array Class

Whenever we want to use a type parameter, we add the `template` decorator.

```
template<typename T>
DynArray<T>::DynArray(int s) {
    sz_ = s;
    arr_ = new T[sz_];
}
```

Dynamic Array Class

Full Example: `dynamic_array_template.cpp`