COP3503

Templates

Code Reuse Speeds Up Development

- **Inventing** the wheel is hard, but necessary.
- **Reinventing** the wheel is a waste.
- Some systems are nearly identical, but only differ in the **types** that they use.
- We may have to duplicate code and change the types.

```
void DoSomeStuff(int);
void DoSomeStuff(double);
void DoSomeStuff(unsigned char);
```

```
class DynamicArrayInt{};
class DynamicArrayFloat{};
class DynamicArrayString{};
```

We **could** duplicate code like this… or maybe there's a better way!

Imagine a Simple Input Function…

```
int GetNumericValue(int min, int max)
{
    int value;
    /* Assume proper input code here */
    return value;
}
```

What about other numeric types?

```
float GetNumericValue(float min, float max)
{
    float value;
    /* Assume proper input code here */
    return value;
}
```

```
short GetNumericValue(short min, short max) …
```

```
char GetNumericValue(char min, char max) …
```

```
unsigned long GetNumericValue(unsigned long min, unsigned long max) …
```

That's a lot of repetition!
Copy+paste, change data type…
Copy+paste, change data type…
Copy+paste, change data type…

Templates to the Rescue!

```
template <typename T>
T GetNumericValue(T min, T max)
{
    T value;
    /* Assume proper input code here */
    return value;
}
```

All the data types (that we've seen up to now) are gone!

```
float x = GetNumericValue(3.14f, 29.774f);
double dx = GetNumericValue(3.14, 19.9);
int y = GetNumericValue(-5, 2000000);
char z = GetNumericValue('a', 'z');
```

You write one template, and the compiler uses that to create 4 **specializations** – one for each data type.

- The compiler will create a new version, or **specialization**, of this function, for each data type that it finds

Isn't the Function Overloading?

- In this particular example, it's **very similar**, but not the same as overloading a function.
- Overloading a function is passing different types of parameters (possibly a different number of parameters).

```
// Overloading a function – multiple, DIFFERENT versions
int GetNumericValue(int min);
int GetNumericValue(int min, int max);
int GetNumericValue(int min, int max, string error);
SomeObject* obj = new SomeObject;
delete obj;
```

```
// Template – one function (written by you)
template <typename T>
T GetNumericValue(T min);
```

Multiple **specializations** created (if necessary), by the compiler, from your template.

The specializations are invisible to you and only created on an as-needed basis by the compiler.

Template Breakdown

```
template <typename T>
```

`T`
the newly-defined data type
(T is common, but it can be anything—keep it small, simple)

`template`
defines this as a template (genius!)

`typename`
define whatever follows this as a **usable data type**

```
template <typename T>
T GetNumericValue(T min, T max)
{
    T value; // Make a variable of SOME type
    int inputAttempts = 0;
    return b;
}
```

Depends on the specialization
(how you tried calling the function)

Template Specialization

```
template <typename T>
void TemplateFunction(T min);
```

- For functions, specializations are typically deduced by the compiler, based on parameter types:

```
TemplateFunction(2);      // T == int
TemplateFunction(-27.8);  // T == double
TemplateFunction(3.14f);  // T == float (f means float instead of double)
```

```
// You can explicitly indicate a specialization
char someValue = GetNumericValue<char>(2, 15);
```

Tell your compiler to create or use the **char** specialization.

- Specializations of classes are a little different.
- For template classes, you must provide a type.

Templates and Classes

```
// GenericClass.h
template <typename T>
class GenericClass
{
    T singleObject;
    T* pointerToT;
    T lotsOfT[100];
    int otherVariable;
    char otherStuff[12];
public:
    GenericClass(int someValue);
};
```

Replace all instances of T with the specialization type, then compile that.

Create a specialization of the class.

```
GenericClass<int> intSpecialization;
GenericClass<float> floatSpec;
GenericClass<Widget> widgetSpec;
GenericClass<RandomObject> random;
// Etc…
```

You've already used templates with the `std::vector` class:

```
std::vector<int> numbers;
std::vector<string> words;
```

Templates and Classes

```cpp
// GenericClass.h
template <typename T>
class GenericClass
{
    int singleObject;
    int* pointerToT;
    int lotsOfT[100];
    int otherVariable;
    char otherStuff[12];
public:
    GenericClass(int someValue);
};
```

Create a specialization of the class.

```cpp
GenericClass<int> intSpecialization;
GenericClass<float> floatSpec;
GenericClass<Widget> widgetSpec;
GenericClass<RandomObject> random;
// Etc...
```

Templates and Classes

```
// GenericClass.h
template <typename T>
class GenericClass
{
    float singleObject;
    float* pointerToT;
    float lotsOfT[100];
    int otherVariable;
    char otherStuff[12];
public:
    GenericClass(int someValue);
};
```

Create a specialization of the class.

```
GenericClass<int> intSpecialization;
GenericClass<float> floatSpec;
GenericClass<Widget> widgetSpec;
GenericClass<RandomObject> random;
// Etc...
```

Templates and Classes

```
// GenericClass.h
template <typename T>
class GenericClass
{
    Widget singleObject;
    Widget* pointerToT;
    Widget lotsOfT[100];
    int otherVariable;
    char otherStuff[12];
public:
    GenericClass(int someValue);
};
```

Create a specialization of the class.

```
GenericClass<int> intSpecialization;
GenericClass<float> floatSpec;
GenericClass<Widget> widgetSpec;
GenericClass<RandomObject> random;
// Etc...
```

Templates and Classes

We can create as many specializations as we want, and we only have to write the class once!

```
// GenericClass.h
template <typename T>
class GenericClass
{
    RandomObject singleObject;
    RandomObject* pointerToT;
    RandomObject lotsOfT[100];
    int otherVariable;
    char otherStuff[12];
public:
    GenericClass(int someValue);
};
```

Create a specialization of the class.

```
GenericClass<int> intSpecialization;
GenericClass<float> floatSpec;
GenericClass<Widget> widgetSpec;
GenericClass<RandomObject> random;
// Etc...
```

What About Multiple Typenames?

```
template <typename Type1, typename Type2>
void Foo(Type1 obj1, Type2 obj2);
```

```
template <typename A, typename B, typename C>
class SomeClass
{
    A object;
    B* pointer;
    C arrayOfThings[20];
};
```

```
string name = "Batman";
// Foo<std::string, const char*>
Foo(name, "Robin");

// Foo<int, float>
Foo(25, -4.6f);

// Foo<int, int>
Foo(0, 10);
```

```
// Create an object with 1 string, pointer to // an integer and an array of 20 float pointers
SomeClass<string, int, float*> myObject;
```

```
// Create an object with 1 int, a pointer to a SomeClass object,
// and an array of 20 chars
SomeClass<int, SomeClass, char> myObject;
```

Whether one template type or multiple, it’s the same concept—just more of it!

Template Class Functions

```
// Still in GenericClass.h...
template <typename T>
class GenericClass
{
    int numberOfThings;
    T* someArray;
public:
    GenericClass(int someValue);
};

template <typename T>
GenericClass<T>::GenericClass(int someValue)
{
    this->numberOfThings = someValue;

    // Allocate an array of... somethings
    this->someArray = new T[someValue];
}
```

`template <typename T>`

This has to be above **every** definition of a class member function, no exceptions.

`<T>`

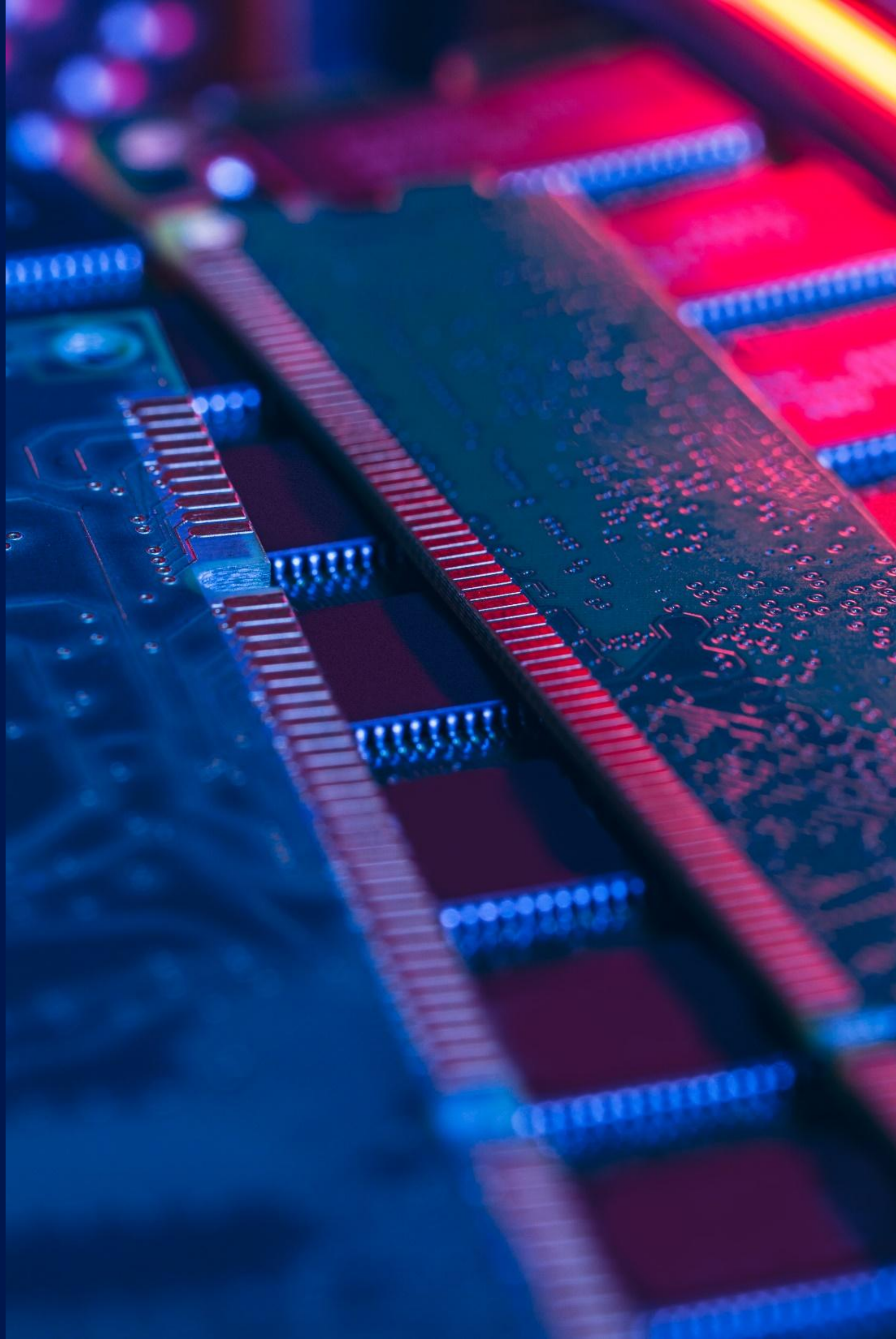
This has to go after the class name, before the scope-resolution operator, in **every** function definition.

Use T, or whatever you called it, as a regular data type, anywhere you need one of those variables.

Template Definition Rule

Only 1 file

- Template classes must be written entirely in a single file.
- Normally it’s “good practice” to split classes into .h/.cpp files.
- Template classes **cannot** be split into two files, **because of the compiler**.
- In order to create specializations, the compiler must know about ALL of the code your class uses.
- To do **that**, the compiler must be able to “see” all of the code in a class, including function definitions.
- Just remember: Templates work differently, and they must exist entirely in a single file.



Still Split Class Declaration and Definition

```
// File: SomeClass.h
template <typename T>
class SomeClass
{
public:
    void Foo(int x);
    int Bar(string s, float xyz);
};
/*====== END CLASS DEFINITION, BEGIN MEMBER FUNCTION DEFINITIONS ======*/

template <typename T>
void SomeClass<T>::Foo(int x)
{

}

template <typename T>
int SomeClass<T>::Bar(string s, float xyz)
{

}
```

(It's good practice.)

Define the class itself at the top of the header file.

You **can** write the functions inside the class—it's just a style choice.

```
template <typename T>
class SomeClass
{
public:
    void Foo(int x)
    {
    }
};
```

What About Nested Classes?

```
template <typename T>
class TemplateClass
{
    T var1, var2;
    T array_Of_T[7];
public:

    class NestedClass
    {
    public:
        T someValue;
        int data1;
        float data2;
        // Etc...
    };
};
```

The nested class will
use
its

```
// Normal instantiation of a nested class
NonTemplateClass::NestedClass nestedObject;
nestedObject.data1 = 50;
```

```
// The T is a float, for the outer AND inner classes
// float is "passed on" to NestedClass
TemplateClass<float>::NestedClass nestedObject;
nestedObject.someValue = 3.14f; // someValue is type float

// The T is a char, for the outer AND inner classes
TemplateClass<char>::NestedClass nested;
nested.someValue = '$'; // someValue is type char
```

Why Write Template Classes?

- One word: **storage**
- A template class is often **like a cardboard box**:
  - Store **something** in it (or remove something from it)
  - Transfer it and its contents from point A to B
- Templates (like boxes) are plain, versatile, **don’t know about their contents**
- They deal with **objects as a whole**, not specific details.
- A template (or a box) is a **bad choice for custom functionality**.

A “template” for storing and shipping things

You Can’t Use Templates for Everything

```
template <typename T>
T GetNumericValue(T min, T max)
{
    T value;
    /* Assume proper input code here */
    return value;
}
```

The compiler will **try** to create a specialization that replaces all instances of `T` with `Dinosaur`.

Given this function, that will likely fail!

There’s no way to stop someone from trying it. Templates are open for all data types

```
Dinosaur trex = GetNumericValue(stegosaurus, triceratops);
```

This probably doesn’t make sense to you (me neither, and I wrote it!)

Keep Templates Generic

- Don’t use unique functionality of objects inside a template.

```
template <typename T>
void GenericClass<T>::Foo()
{
    T someVariable;
    someVariable.Bar();
}
```

Every type you use to create a specialization of this class **must** have a `Bar()` function.

```
GenericClass<int> intSpec;
// Won’t work, compiler error.
// *int* isn’t a class, doesn’t have a
// Bar() function (or any functions)
```

- Templates should deal with generic, reusable functionality.
- Storing some data, not operating on it.

Can You Have a Template That Doesn't Work With Everything?

- You **could** write template code that expects a specific interface.
- May be a style you (or your team) develop
- C++ cannot limit types you use with a template—it's up to you to do things properly.

```
class Good
{
public:
    void DoStuff();
};
```

```
class Bad
{
public:
    void Nope();
};
```

```
// Won't compile, no Bad::DoStuff()
// function exists
GenericClass<Bad> nonConforming;

// In main()…
// No problem, Good::DoStuff() exists
GenericClass<Good> someTemplate;
```

```
template <typename T>
void GenericClass<T>::Foo(T& other)
{
    // As long as T has DoStuff() overloaded, it's okay
    other.DoStuff();
}
```

You **can** assume this function exists as part of your program design…as long as others know this is a choice you've made

What About Templates Within Templates?

- `std::vector<T>` is a general purpose, go-to storage container.
- It can store any type of data, even `std::vector<T>`.

Even in “simple” terms this could be a confusing concept.

```
vector<int> numbers;  // specialization: int
vector<string> words; // specialization: string

vector<vector<int>> numberGroups; // Specialization: vector<int>
```

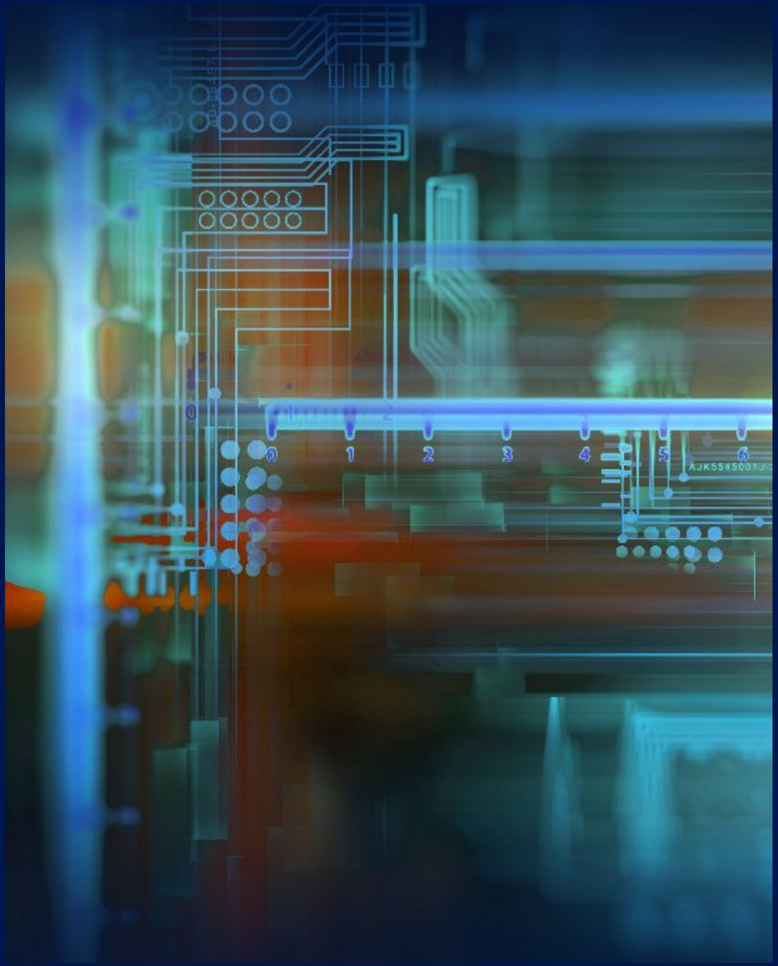
```
// May look scary or intimidating, but works just fine
vector<vector<vector<int>>> this_is_scary;
```

This is: A vector of... vectors of... vectors of integers
Or: A box containing boxes containing boxes of numbers

Recap

- Templates let you reuse **functions** and **classes** with **different data types**.
- Templates define new types, often named **T** or other simple names.
- The compiler creates **specializations** of a template with the type(s) specified.
- Template classes make excellent **storage containers**.
- Template classes must be **completely defined in a single file**—typically a header file.
- Template code can sometimes look confusing, but is very powerful and versatile.

Conclusion



Placeholder for the instructor's welcome message. Video team, please insert the instructor's video here.

Thank you for watching.