

Templates

Ch 17



POOR DESIGN TEMPLATE

This is an example.

Highlights

- Variable types

```
template <typename T>  
void dostuff(T x)  
{  
    cout << x << endl;  
}
```

```
template <typename T>  
class holdinStuff {  
public:  
    T stuff;  
};
```

Review: Types

A type is a container for a specific value

data
value



data
type



Review: Types

It is normally not good to mix these up...



Review: Types

C++ is fairly picky about most types, only certain values can be stored in a type

42



int

'x'



char

“hello world”



string

You can convert between types easily
(such as from int to double)

But not others (hard to go int to string)

Templates

While C++ has no “I can hold any data” type, it does have a “I can be any type” variables

That is there is no “magic” type that can be both int and string simultaneously

Instead, you can specify that “magic” will be some type... you just don't know what yet

In C++ we call this a template

Templates

You can think of this not as “a type to hold all values” but as “a box to hold any type”

42

int

"hello world"

string



Templates

You have actually seen templates before, namely `static_cast` (a function)

You provide the type you wish to convert the data into, but outside of the normal parenthesis

```
cout << static_cast<char>(100) << endl;
```

You can put any type you want here!

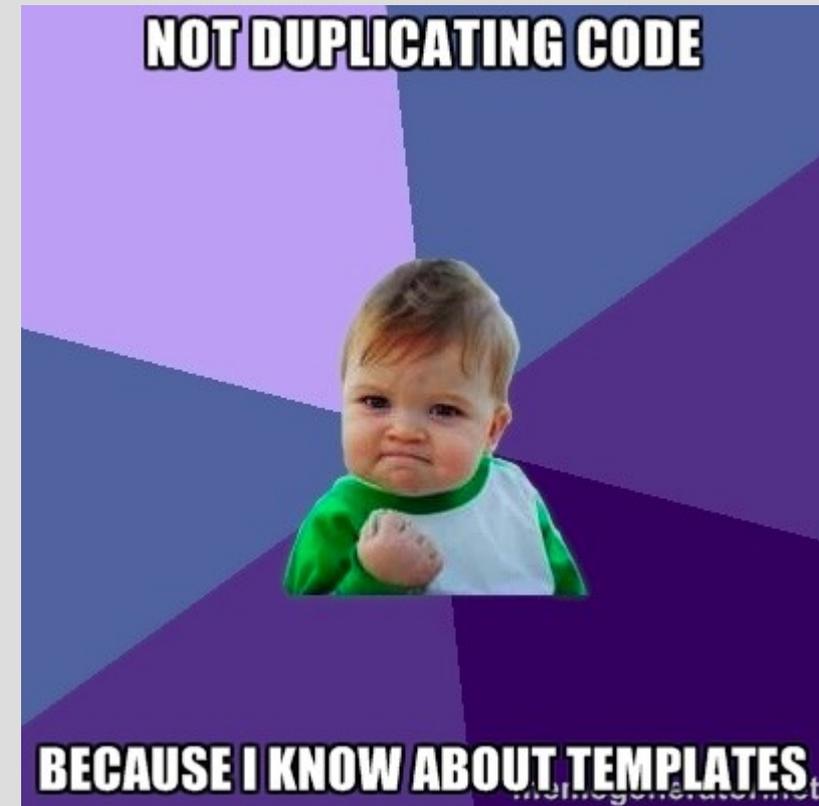
Templates

You can use a variable-type for both:

1. Functions
2. Classes

This allows you to make more general functions (thus less code)

However, this function should be generalizable (for example, factorial only works for ints...)

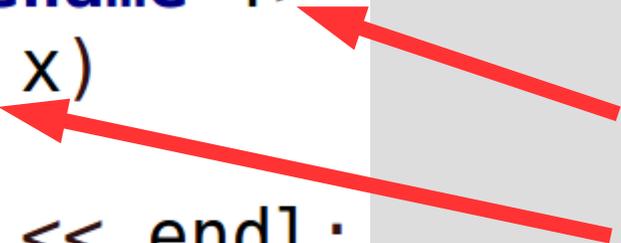


Function: templates

To use a variable-type, you put template before the function and specify the type variable

```
template <typename T>  
void coutMe(T x)  
{  
    cout << x << endl;  
}
```

T is a variable
for the type



This lets you use “T” as a type anywhere in the function

(see: coutMe.cpp) (see: goodSwap.cpp)

Function: templates

You can also use multiple types variables, just separate them with a comma:

```
template <typename T , typename T2>  
void mswap(T& a, T2& b);
```

You can have as many different (or similar) types of input as you want

(Although this does not work well for swap)
(see: multipleTypes.cpp)

Function: templates

You can check to see if the input types are the same by doing this:

```
template <typename T1, typename T2>
void foo(T1 x, T2 y)
{
    if( is_same<T1,T2>::value) {
        cout << "Same types... doing something\n";
    }
    else {
        cout << "Different types\n";
    }
}
```

(see: checkSameType.cpp)

Function: notation

As C++ is rather old, there are a lot of ways to say the same thing (same with templates)

These both mean the same thing (mostly):

```
template <typename T>  
template <class T>
```

Some compilers also see these as the same:

```
coutMe<int>(2);  
coutMe(2);
```

Bad templates!

Templates are not magic that allow you to do anything with any type!

If an operation does not exist between types and you try and use it, computer will get angry

You also cannot ignore types completely by making everything a template (main must have real types at the very least)
(see: badTemplates.cpp)

Classes: templates

Templates for classes are very similar:

```
template <typename T>
class holdinStuff {
public:
    T stuff;
};
```

After using template, you can use “T” as a type inside the class anywhere

(see: classTemplate.cpp)