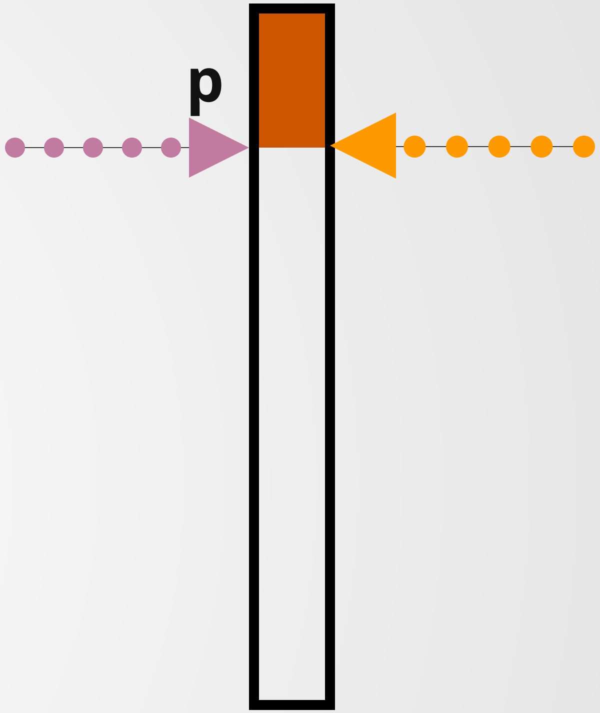
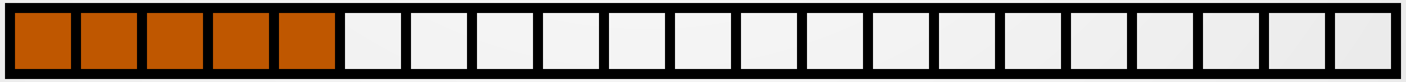
# CS 105C: Lecture 6

# Last Time…



The Stack and Heap

What are the properties of these two memory stores?



# RAII

A technique for managing resource lifetimes (e.g. memory, files, locks, objects) by tying the lifetime of the resource to the lifetime of a stack-allocated class.

When the stack variable goes out of scope, its destructor is called and the resource is released. Since stack variables go out of scope eventually, the resource is eventually released.

# Questions!

# Q: When should we use malloc() vs new?

**A: In C++, use new**

# Q: How does C++ know that a variable has gone out of scope?

## A: The compiler can read the source code and insert the appropriate statements at end of scope.

# CS 105C: Lecture 6

## Templates

With much love to UW and UPenn for providing inspiration for this lecture

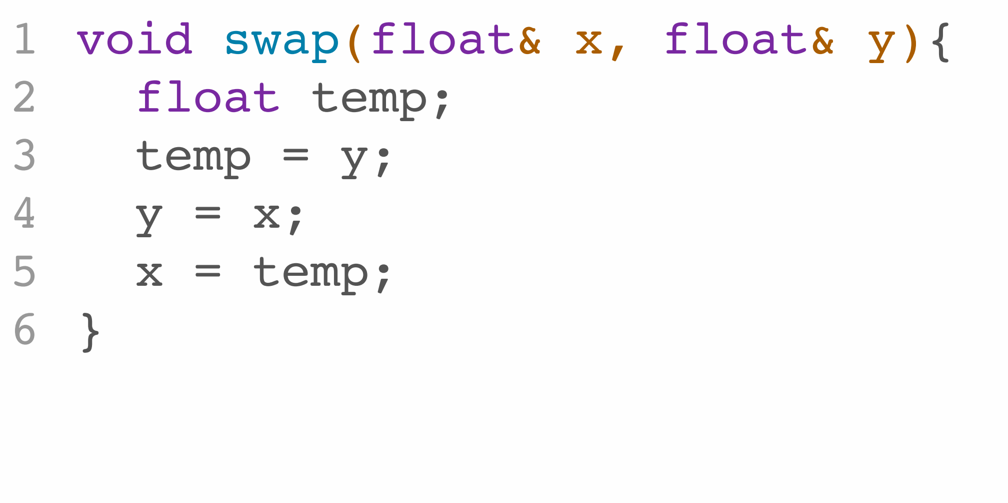
# Why Templates?

I want to swap two ints

```cpp
void swap(int& x, int& y){
  int temp;
  temp = y;
  y = x;
  x = temp;
}
```

# Why Templates?

I want to swap two ~~ints~~ floats



```cpp
void swap(float& x, float& y){
  float temp;
  temp = y;
  y = x;
  x = temp;
}
```

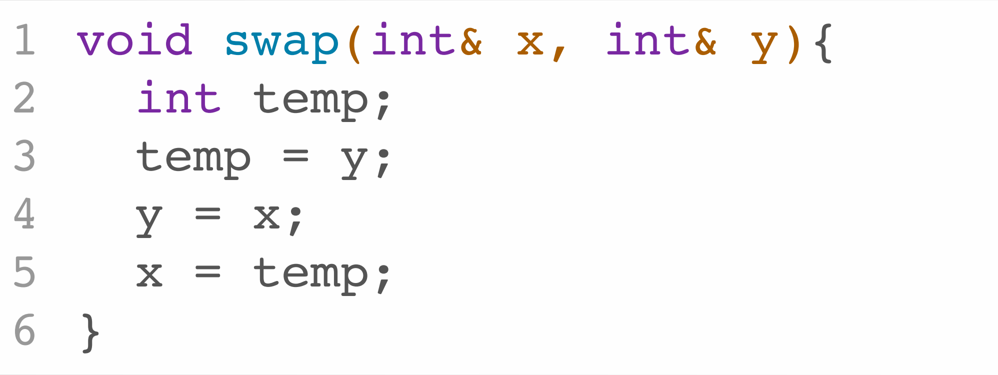
Question: do I need to rename swap to something like swap_float?

# Why Templates?

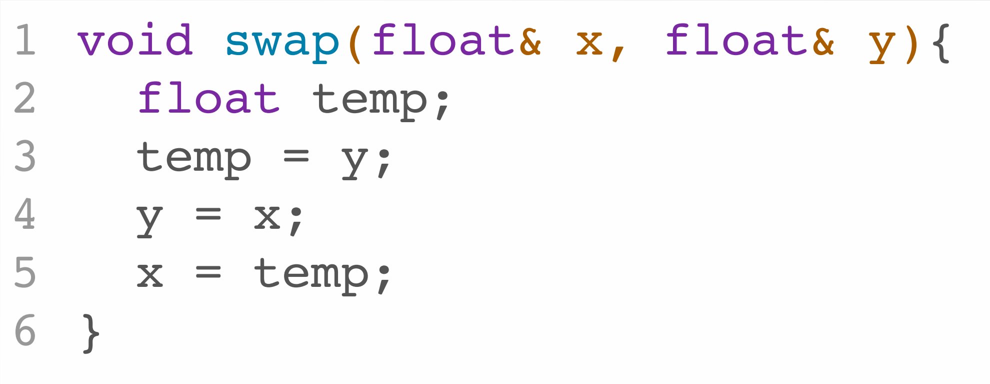
I want to swap two ~~ints floats~~ strings

```
void swap(std::string& x, std::string& y){
  std::string temp;
  temp = y;
  y = x;
  x = temp;
}
```

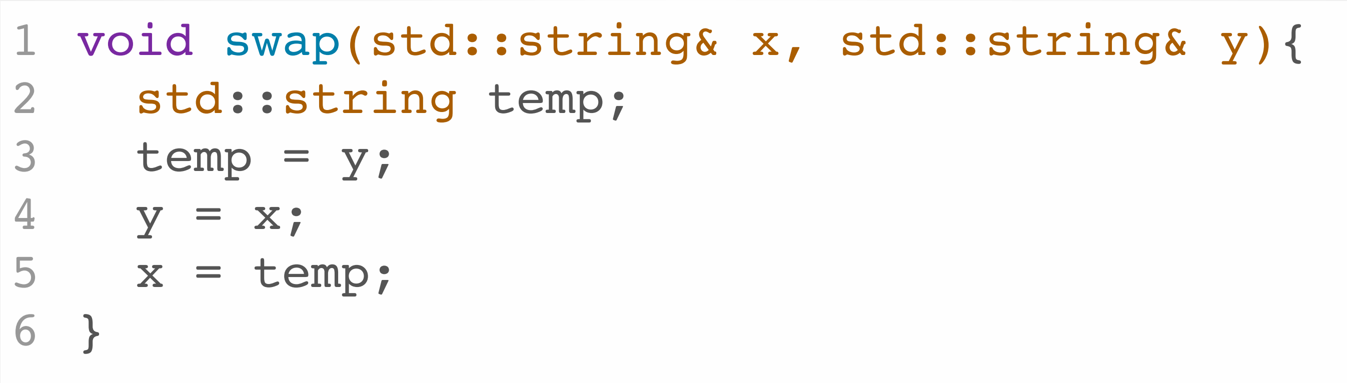
Thanks to overloading, we don't have to name our functions differently...



```cpp
void swap(int& x, int& y){
  int temp;
  temp = y;
  y = x;
  x = temp;
}
```



```cpp
void swap(float& x, float& y){
  float temp;
  temp = y;
  y = x;
  x = temp;
}
```



```cpp
void swap(std::string& x, std::string& y){
  std::string temp;
  temp = y;
  y = x;
  x = temp;
}
```

...but we still have to write the same code over and over, which is **A Bad Thing™**

# Solution: Parametric Polymorphism!

## Polymorphism

In programming: the ability to present the same interface for many different underlying datatypes (shapes).

We have already seen distinctions between types of polymorphism:

- **Compile-time** polymorphism
- **Runtime** polymorphism

# Solution: Parametric Polymorphism!

Another distinction:

If the code does the same thing for all underlying types, it is known as

**parametric polymorphism**

If the code does different things for different underlying types, we call it

**ad-hoc polymorphism**

Which type is inheritance?

```cpp
void swap(std::string& x, std::string& y){
  std::string temp;
  temp = y;
  y = x;
  x = temp;
}
```

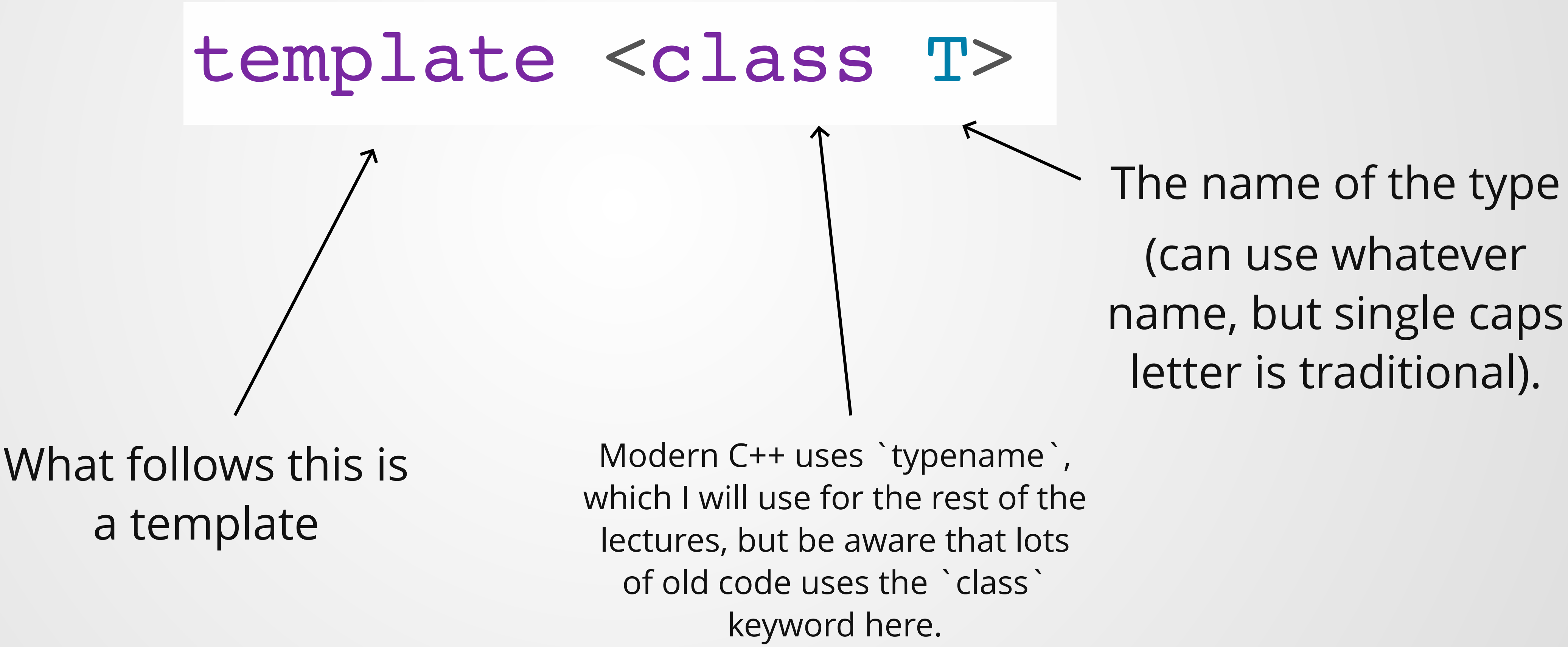
C++ handles **parametric polymorphism** using templates

```cpp
template <class T>
void swap(T& x, T& y){
  T temp;
  temp = y;
  y = x;
  x = temp;
}
```

Code *must* do same thing for all T, since there is no mechanism to figure out what T is

# Using Templates

# To indicate that the following C++ construct is template code, use the following:



```
template <class T>
```

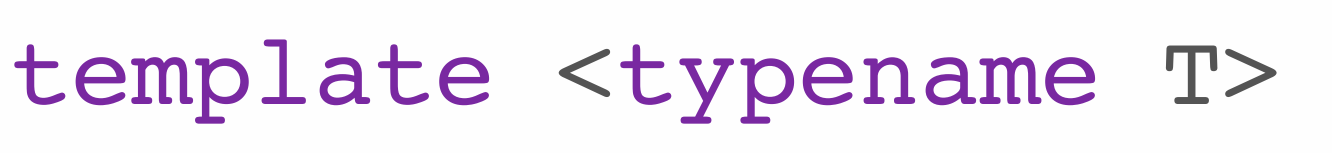
What follows this is a template

Modern C++ uses `typename`, which I will use for the rest of the lectures, but be aware that lots of old code uses the `class` keyword here.

The name of the type

(can use whatever name, but single caps letter is traditional).

# To indicate that the following C++ construct is template code, use the following:



```
template <typename T>
```

The function/class that follows the template prefix will resolve T to some type (e.g. int, char, Ball).

# Which calls on the right are valid?

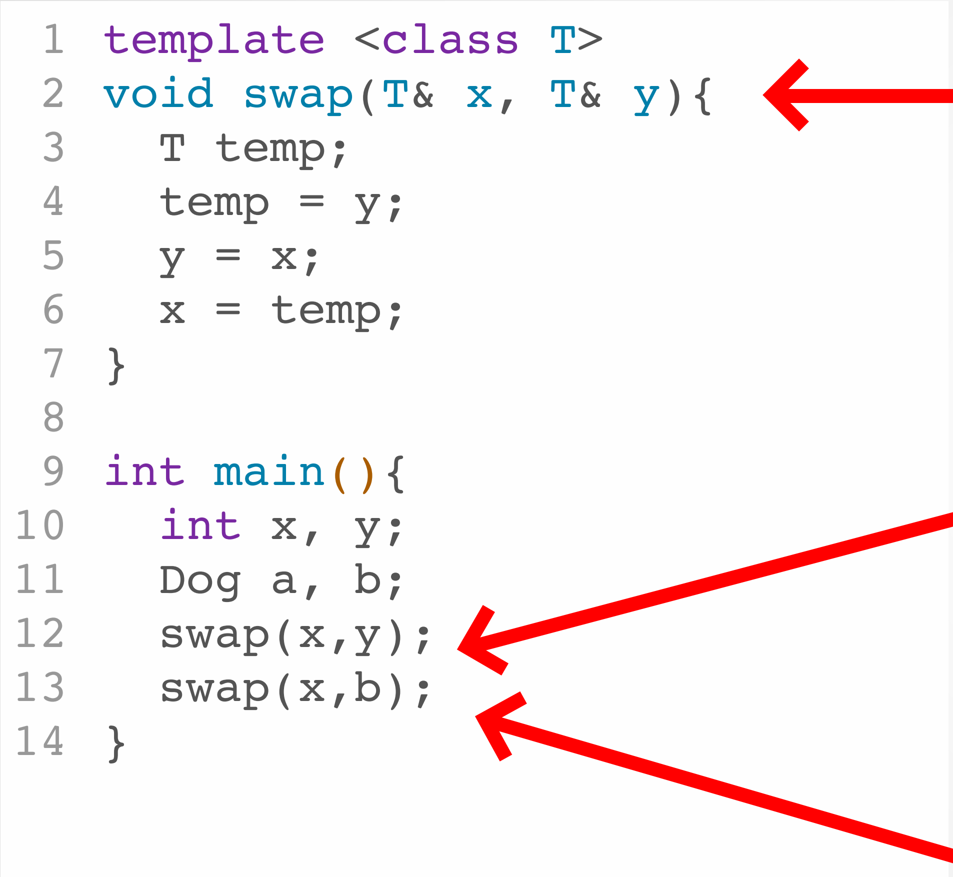
```
template <class T>
void swap(T& x, T& y){
  T temp;
  temp = y;
  y = x;
  x = temp;
}
```

```
swap(int, int);
swap(char, char);
swap(arkanoid::Ball, arkanoid::Ball);
~~swap(int, long);~~
swap(std::string, std::string);
~~swap(float, double);~~
```

# Now that we have declared template code, the compiler can generate code to compute any calls to swap()

How and when does the compiler know to create template code?

# The compiler doesn't generate code until it sees the first usage of the template!



```
 1 template <class T>
 2 void swap(T& x, T& y){
 3   T temp;
 4   temp = y;
 5   y = x;
 6   x = temp;
 7 }
 8
 9 int main(){
10   int x, y;
11   Dog a, b;
12   swap(x,y);
13   swap(x,b);
14 }
```

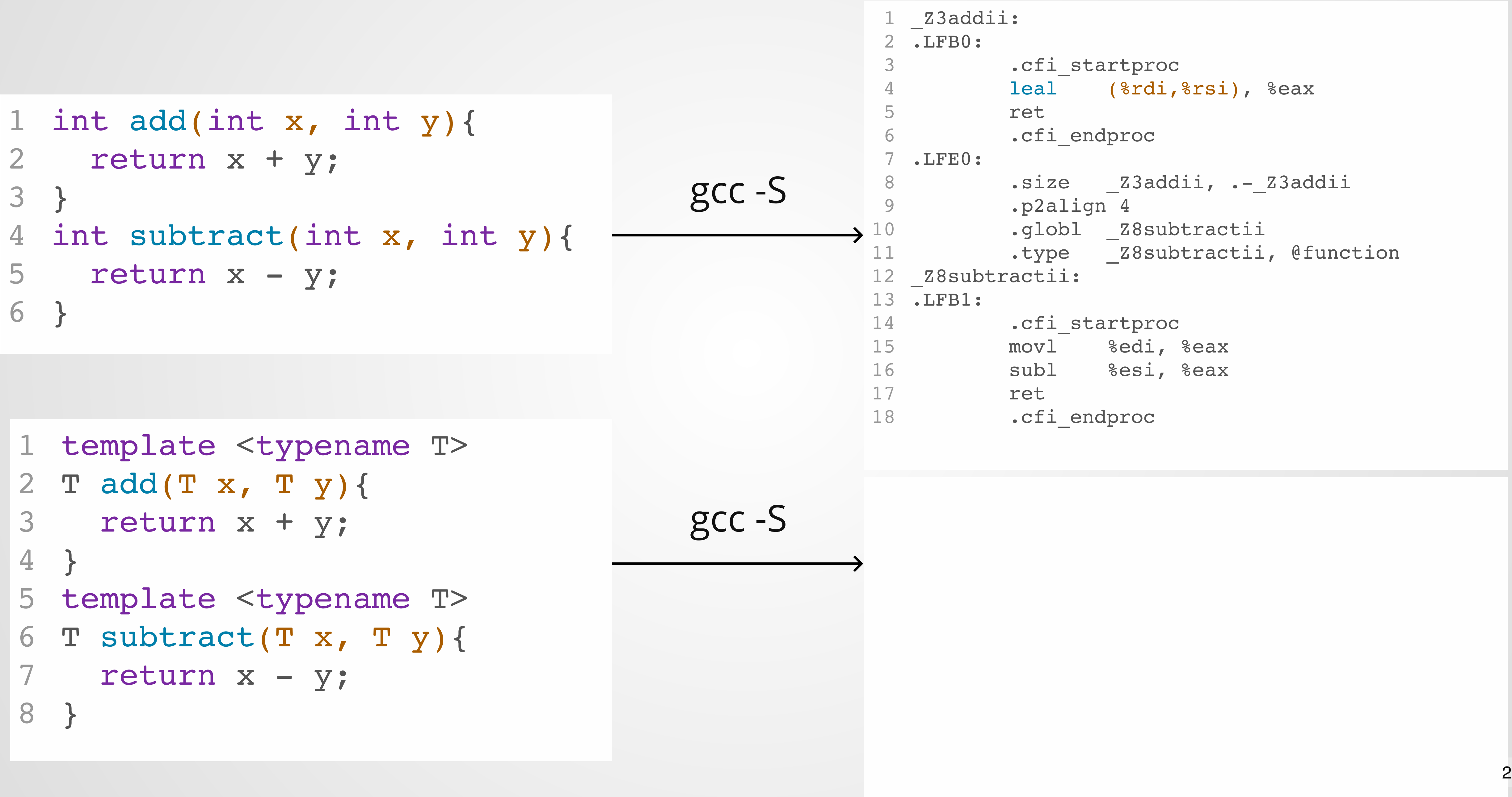
(1) Compiler reads template definition.

It now knows that swap() is a template, but **it does not generate any code yet!!**

(2) Compiler sees usage of swap. It looks up the template and creates an <int> **specialization** or **instantiation**.

(3) Compiler sees usage of swap. It looks up the template and fails to create an instantiation. This causes an error on line 13.

# Template Instantiation



```
int add(int x, int y){
  return x + y;
}
int subtract(int x, int y){
  return x - y;
}
```

gcc -S

```
_Z3addii:
.LFB0:
        .cfi_startproc
        leal    (%rdi,%rsi), %eax
        ret
        .cfi_endproc
.LFE0:
        .size   _Z3addii, .-_Z3addii
        .p2align 4
        .globl  _Z8subtractii
        .type   _Z8subtractii, @function
_Z8subtractii:
.LFB1:
        .cfi_startproc
        movl    %edi, %eax
        subl    %esi, %eax
        ret
        .cfi_endproc
```

```
template <typename T>
T add(T x, T y){
  return x + y;
}
template <typename T>
T subtract(T x, T y){
  return x - y;
}
```

gcc -S

# Using typical C++ code organization techniques with templates will cause **catastrophic failure**

```cpp
// File main.cpp
#include "swap.h"

int main(){
  int a = 3, b = 8;
  swap(a,b);
}
```

```cpp
// File swap.h
template <typename T>
void swap(T& v1, T& v2);
```

```cpp
// File swap.cpp
#include "swap.h"

template <typename T>
void swap(T& a, T& b){
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

# First we compile main.cpp

```cpp
// File swap.h
template <typename T>
void swap(T& v1, T& v2);
```

```cpp
// File main.cpp
#include "swap.h"

int main(){
  int a = 3, b = 8;
  swap(a,b);
}
```

```
_start:
   blah blah blah

main:
   push 8
   push 3
   call swap
```

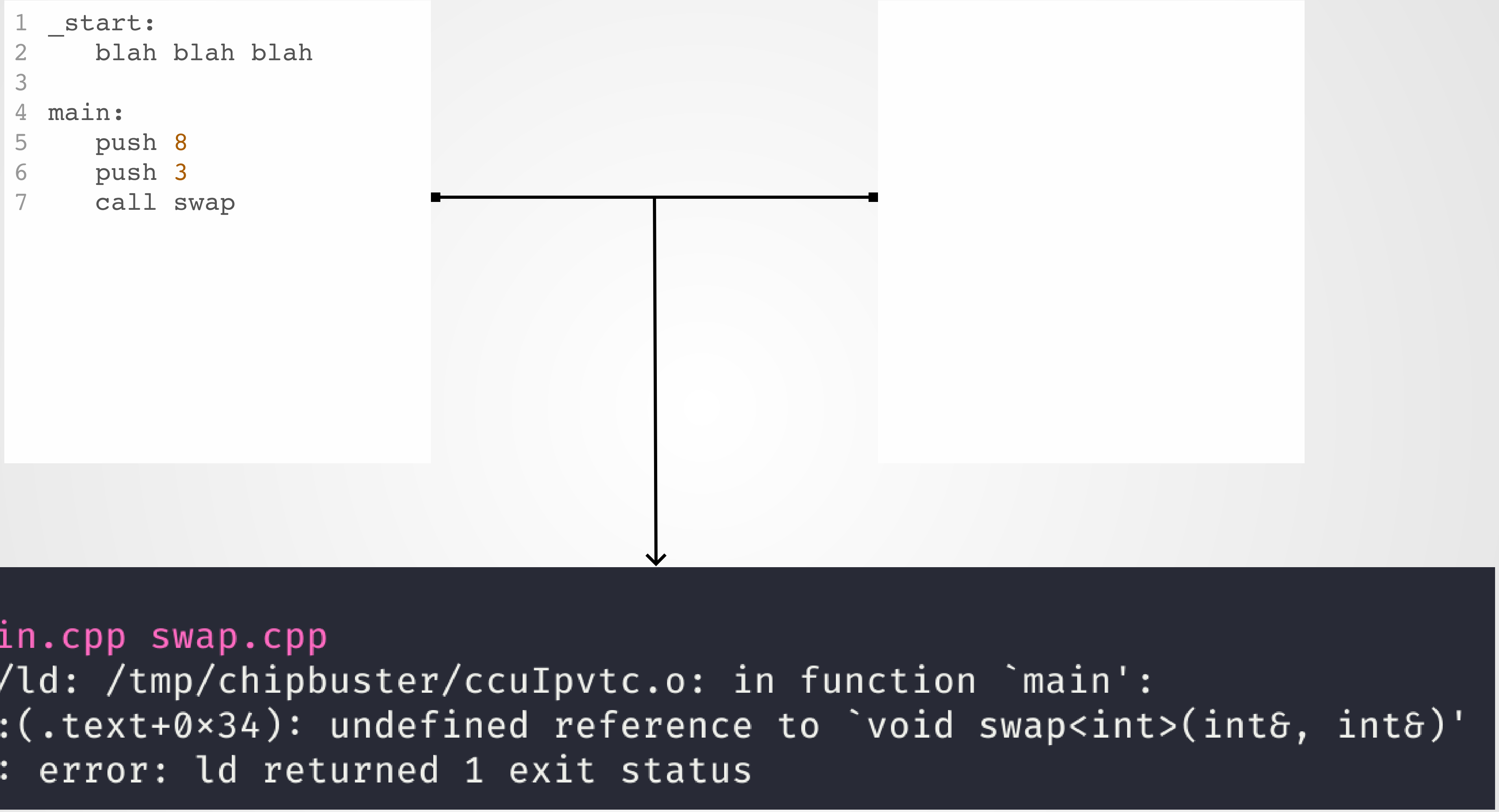
# Next we compile swap.cpp

```cpp
// File swap.h
template <typename T>
void swap(T& v1, T& v2);
```

```cpp
// File swap.cpp
#include "swap.h"

template <typename T>
void swap(T& a, T& b){
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

No usage of template:
no instantiation!

# Now we try to link these...

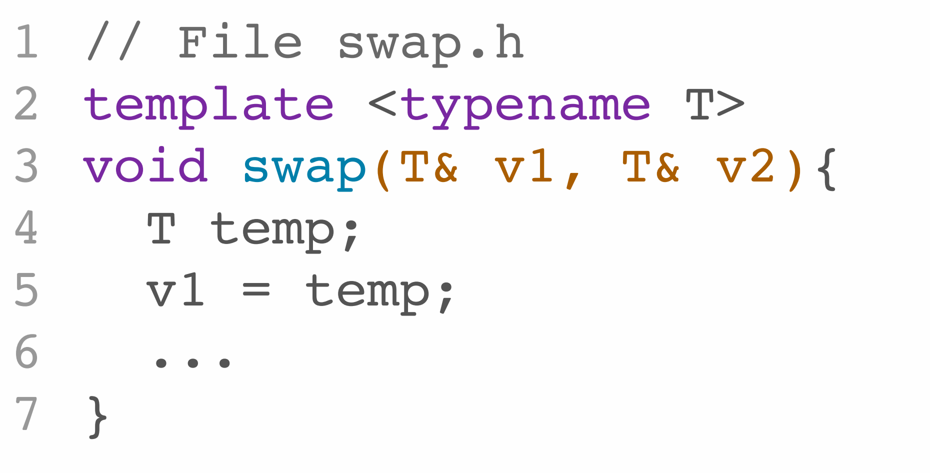


```
_start:
   blah blah blah

main:
   push 8
   push 3
   call swap
```

```
~/t/C++
❯ g++ main.cpp swap.cpp
/usr/bin/ld: /tmp/chipbuster/ccuIpvtc.o: in function `main':
main.cpp:(.text+0x34): undefined reference to `void swap<int>(int&, int&)'
collect2: error: ld returned 1 exit status
```

# Solution?

There are several solutions. The easiest and most common is to place template **definitions** in the header file.

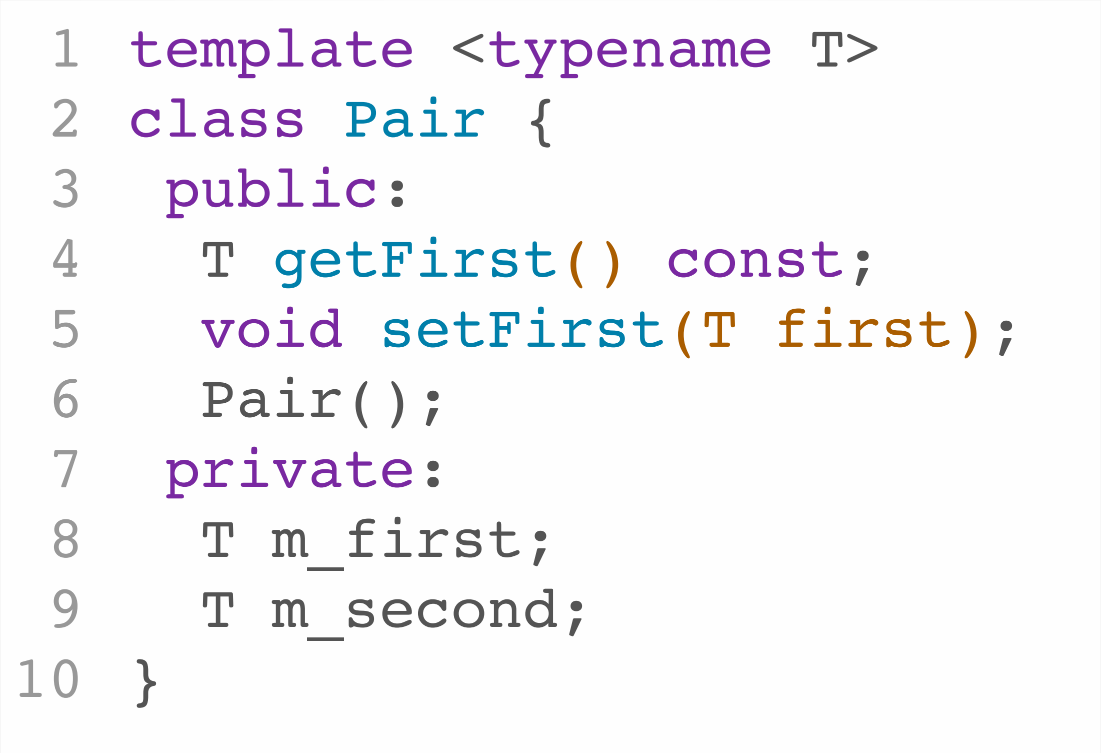


```
// File swap.h
template <typename T>
void swap(T& v1, T& v2){
  T temp;
  v1 = temp;
  ...
}
```

This is the **<u>exact opposite</u>** of what we normally do with header files!!

# Template Classes

Templates for classes work pretty much the exact same way for template functions:



```cpp
template <typename T>
class Pair {
 public:
  T getFirst() const;
  void setFirst(T first);
  Pair();
 private:
  T m_first;
  T m_second;
}
```

# Caveat

If you choose to implement a method outside of the declaration (which usually isn't done), you need to add the template prefix and scope resolution operation:

```
// Broken
T Pair::getFirst() const{
  return this->m_first;
}

// Works
template <typename T>
T Pair<T>::getFirst() const{
  return this->m_first;
}
```

# Intermediate Templates

Templates are written without any way to control what types can be implemented

This sometimes leads to interesting problems.

```
template <typename T, typename U>
??? add(T x, U y){
  return x + y;
}
```

What type should add return?

T = float, U = int?

T = int, U = float?

T = char, U = long?

```
template <typename T, typename U>
??? add(T x, U y){
  return x + y;
}
```

Note: once we know what T, U are, we can decide! But not before then.

Solution: decltype

```
template <typename T, typename U>
decltype(x+y) add(T x, U y){
  return x + y;
}
```

Note: decltype is a function which returns *types*.

# Other type-level programming

None of this will be tested, though declval() will show up in project 3.

# Other type-level programming

Sometimes, you want to use the *reference* of a type instead of the type, (e.g. for late-binding polymorphism), but you only have access to the type.

```
int main(){
  /* Will use compile-time polymorphism
  because MySubClass returns a value */
  decltype(MySubClass().foo()) a1;

  /* Use declval to use late-binding */
  decltype(std::declval<NonDefault>().foo()) a1;
}
```

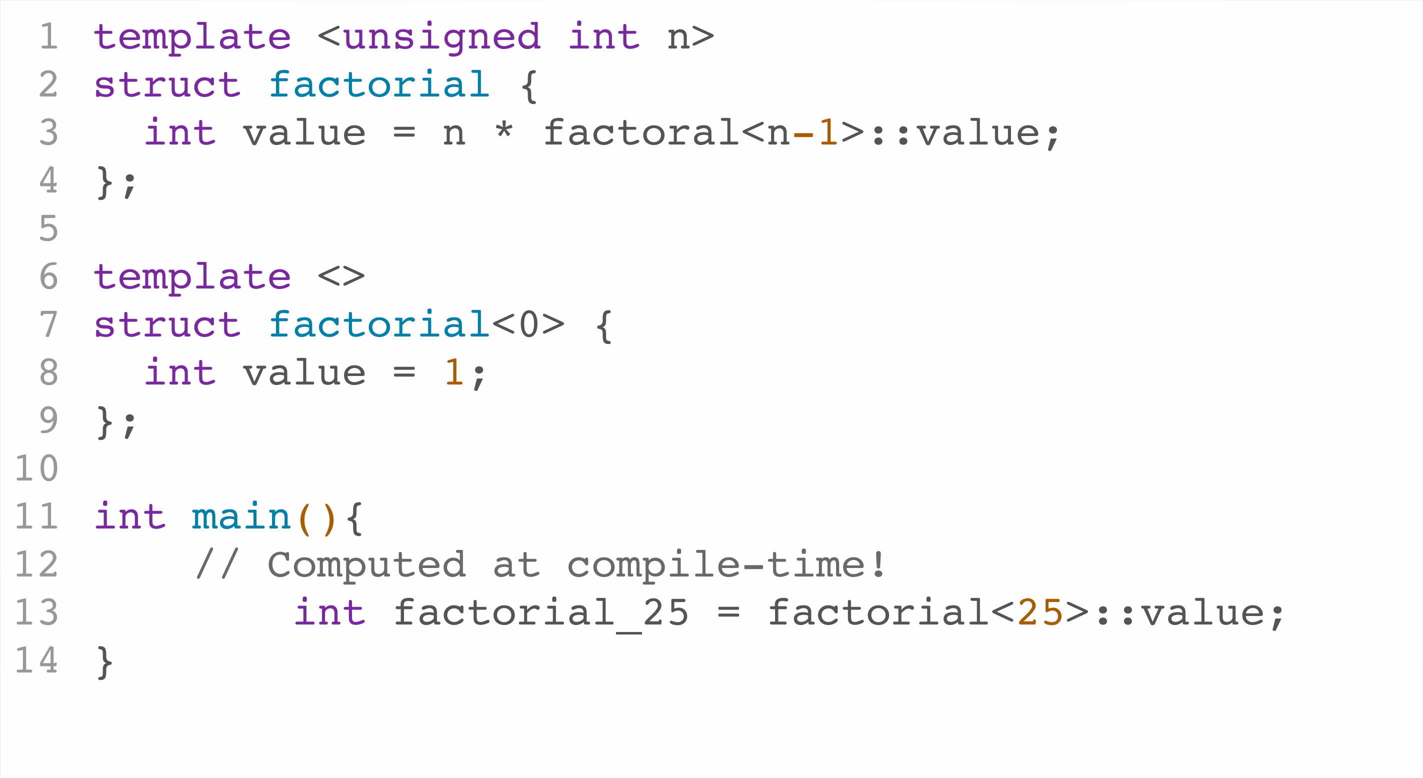
# Other type-level programming

Sometimes, you want to use the *reference* of a type instead of the type, (e.g. for late-binding polymorphism), but you only have access to the type.

```
int main(){
  /* Will use compile-time polymorphism
  because MySubClass returns a value */
  decltype(MySubClass().foo()) a1;

  /* Use declval to use late-binding */
  decltype(std::declval<NonDefault>().foo()) a1;
}
```

# Other type-level programming

Templates can also take values!

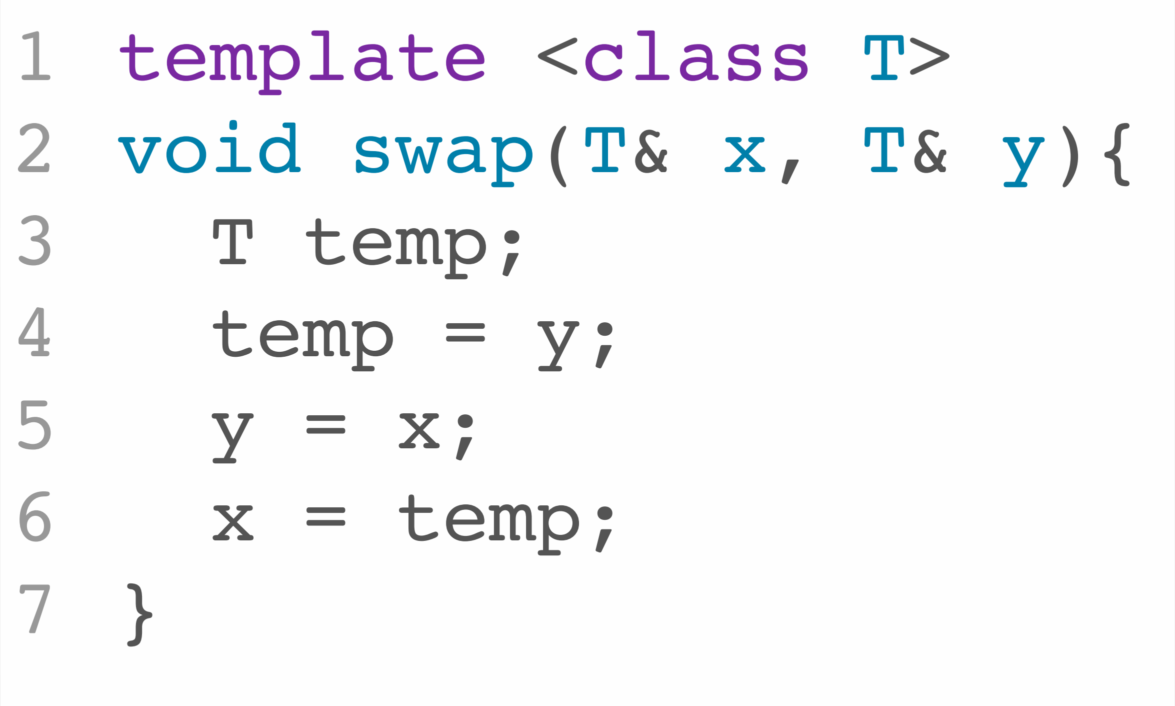


```
template <unsigned int n>
struct factorial {
  int value = n * factoral<n-1>::value;
};

template <>
struct factorial<0> {
  int value = 1;
};

int main(){
    // Computed at compile-time!
        int factorial_25 = factorial<25>::value;
}
```

# Summary

# Templates implement parametric polymorphism in C++

The *same* code for all types.



```cpp
template <class T>
void swap(T& x, T& y){
  T temp;
  temp = y;
  y = x;
  x = temp;
}
```

# Templates need to be instantiated--for this, the definition needs to be known

This file structure will fail to compile, since template definition is not known on line 6 of main()

```cpp
1  // File swap.h
2  template <typename T>
3  void swap(T& v1, T& v2);
```

```cpp
1  // File main.cpp
2  #include "swap.h"
3
4  int main(){
5    int a = 3, b = 8;
6    swap(a,b);
7  }
```

```cpp
1  // File swap.cpp
2  #include "swap.h"
3
4  template <typename T>
5  void swap(T& a, T& b){
6    T temp;
7    temp = a;
8    a = b;
9    b = temp;
10 }
```

# Templates need to be instantiated--for this, the definition needs to be known

Solution: template definitions go in the header file!

This is exactly the opposite of how non-template code should be structured!

# *decltype* can be used to determine the type of an expression

Useful when determining type is difficult or impossible

```cpp
template <typename T, typename U>
decltype(x+y) add(T x, U y){
  return x + y;
}
```

# Feedback

## Quiz Formats:

- **Out of class quizzes**

  I'd so love to do this, but I can't.

- **Paper quizzes**

  Formal vote on Piazza before next quiz

- **More frequent quizzes**

  Given our other constraints, this would cut too much into lecture time...but maybe.

# Feedback

## Quiz Formats:

- **Out of class quizzes**

  I'd so love to do this, but I can't.

- **Paper quizzes**

  Formal vote on Piazza before next quiz

- **More frequent quizzes**

  Given our other constraints, this would cut too much into lecture time...but maybe.

# Feedback

**Resources:**

- **More office hours**

  Email me to schedule some. I'd like to have more regular ones, but nobody shows up to the regular ones as it is, which makes it hard to justify.

- **More lecture time/3 hour class**

  Leave that on the end-of-semester course eva--that can be read by the department.

- **More outside resources (readings, links)**

  Now this I can easily do :)

# External Resources

- An Idiot's Guide to C++ Templates

  A blog post explaining a lot of C++ template stuff.
- Explaining Decltype and Declval

  Good, solid explanations for these two features which may prove helpful for project 3
- C++ Primer, 5e. Chapter 17, sections:
  - 16.1.*
  - 16.2.1, 16.2.2, 16.2.3
  - 16.3

# Feedback

**Lectures:**

- **Slow down in lectures**

  I'm trying. Is there a feedback mechanism you can think of to let me know if a lecture is going/went too fast?

- **Pair activities aren't helpful**

  Should we do solo activities? Better planned/more advanced activities?

- **Other**

  If you don't see your feedback up here, there's a decent chance I didn't understand it. Feel free to drop me a line!

# Notecards

- Name and EID
- One thing you learned today (can be "nothing")
- One question you have about the material. **If you leave this blank, you will be docked points.**

  If you do not want your question to be put on Piazza, please write the letters **NPZ** and circle them.