Function Templates

A Simple Function

```
void printVal(int x);


void printVal(int x) {
   cout << “The value: ” << x << endl;
}
```

Another Simple Function

```
void printVal(int x);
void printVal(string x);

void printVal(int x) {
  cout << “The value: ” << x << endl;
}

void printVal(string x) {
  cout << “The value: ” << x << endl;
}
```

More Simple Functions

```
void printVal(int x);
void printVal(string x);
void printVal(float x);
void printVal(double x);
void printVal(size_t x);
void printVal(Cow x);
void printVal(Barn x);
void printVal(Car x);
void printVal(Train x);
```

Function Template

```
template <typename T>
void printVal(T x);

template <typename T>
void printVal(T x) {
  cout << “The value: ” << x << endl;
}

printVal<int>(5);
printVal<string>(“hi”);
```

Function Template

```
template <typename T>
void printVal(T x);

template <typename T>
void printVal(T x) {
  cout << “The value: ” << x << endl;
}

printVal(5); // Also okay (compiler infers type)
printVal(“hi”);
```

Function Template

```
template <typename T>
void printVal(T x);
```

What's the trouble?
What if T is a class?

```
template <typename T>
void printVal(T x) {
  cout << "The value: " << x << endl;
}

printVal(5); // Also okay (compiler infers type)
printVal("hi");
```

Pass by value makes an unnecessary copy!

Even Better Function Template

```
template <typename T>
void printVal(const T& x);
```

Reference to prevent unnecessary copy
const so we can't alter what was passed in

```
template <typename T>
void printVal(const T& x) {
  cout << "The value: " << x << endl;
}

printVal(5); // Also okay (compiler infers type)
printVal("hi");
```

Summary

- Templates let you write code that works for many types
  - Avoids code duplication
- General rule of thumb: pass and return by reference rather than by value
  - Avoids unnecessary copies
  - Especially important when the given type has an expensive copy operation

Coming Up…

A weird issue when compiling templates.

Compiling Templates

Templates ARE NOT CODE

```
template <typename T>
void printVal(const T& x) {
  cout << “The value: ” << x << endl;
}
```

What if we try to compile this?

- **We can’t!**
- Different types → different instructions!

Templates are *Recipes* for Code

```
template <typename T>
void printVal(const T& x) {
  cout << “The value: ” << x << endl;
}

printVal(“hi”);
```

- Infer that type T is string
- Generate code for printVal<string>
- Compile printVal<string> (with type checks!)

Multifile Compilation – How About This?

```
// util.hpp
template <typename T, typename U>
void printVals(const T& x, const U& Y);
```

```
// util.cpp
#include “util.hpp”
template <typename T, typename U>
void printVals(const T& x, const U& Y) {
  cout << “values: ” << x;
  cout << “, ” << y << endl;
}
```

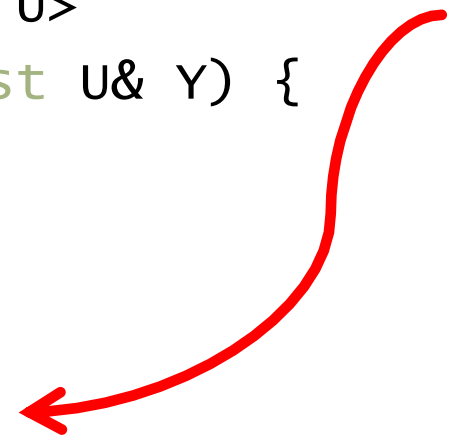
Can’t compile a template on its own!

```
// main.cpp
#include “util.hpp”
int main() {
  printVals(4.2, true);
}
```

Multifile Compilation – How About This?

```
// util.hpp
template <typename T, typename U>
void printVals(const T& x, const U& Y) {
  cout << "Values: " << x;
  cout << ", " << y << endl;
}
```

```
// main.cpp
#include "util.hpp"
int main() {
  printVals(4.2, true);
}
```



Template is compiled along with code that uses it.

This works!

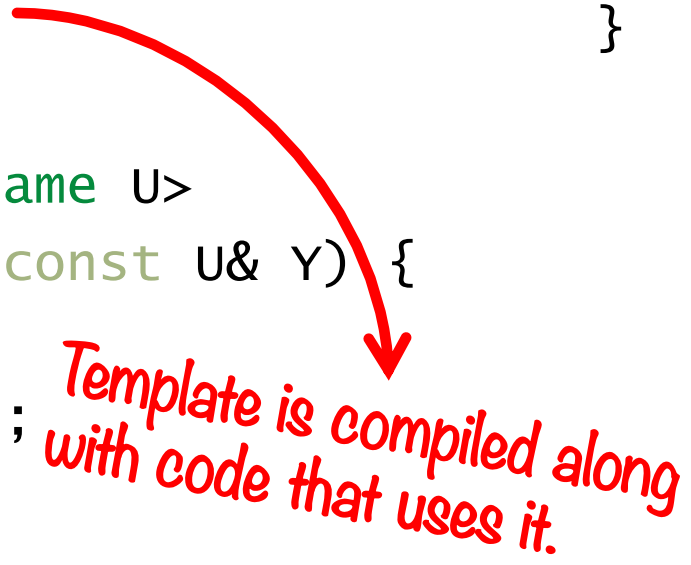
(but we lose the header – definition separation)

Multifile Compilation – A Minor Improvement

```
// util.hpp
template <typename T, typename U>
void printVals(const T& x, const U& Y);
#include “util-private.hpp”
```

```
// util-private.hpp
template <typename T, typename U>
void printVals(const T& x, const U& Y) {
  cout << “Values: ” << x;
  cout << “, ” << y << endl;
}
```

```
// main.cpp
#include “util.hpp”
int main() {
  printVals(4.2, true);
}
```



Summary

- Templates ARE NOT CODE
  - They are *recipes* for code!
- When code uses a template with concrete types
  - The compiler generates and compiles code with those types
- Templates can’t be compiled separately
  - They *must* be compiled with the code that uses them
- Convention:
  - Code that uses the template #includes…
  - mytemplate.hpp, which contains the template header and #includes…
  - mytemplate-private.hpp, which contains the template definition

Coming Up…

Class templates!

Class Templates

A Simple Class

```
// intpair.hpp

class IntPair {
 public:
  IntPair(int f, int s);
  int getFirst() const;
  int getSecond() const;
  void setFirst(int f);
  void setSecond(int s);
 private:
  int first_;
  int second_;
};
```

A Simple Class Template

```
// pair.hpp
template <typename first_t, typename second_t>
class Pair {
 public:
  Pair(const first_t& f, const second_t& s);
  const first_t& getFirst() const;
  const second_t& getSecond() const;
  void setFirst(const first_t& f);
  void setSecond(const second_t& s);
 private:
  first_t first_;
  second_t second_;
};
#include “pair-private.hpp”
```

Defining the Member Functions

A member function of a class template is a function template

```
// pair-private.hpp
template <typename first_t, typename second_t>
Pair<first_t, second_t>::Pair(const first_t& f, const second_t& s) :
 first_(f), second_(s)
{}

template <typename first_t, typename second_t>
const first_t& Pair<first_t, second_t>::getFirst() const {
  return first_;
}

template <typename first_t, typename second_t>
void Pair<first_t, second_t>::setFirst(const first_t& f) {
  first_ = f;
}
```

The name of the class includes the types.
(Pair with no types is NOT A CLASS)

Using a Class Template

```
// main.hpp
#include “pair.hpp”
int main() {
  Pair<float, bool> p1(4.2, true);
  float x = p1.getFirst();

  Pair<int, int*>* p2 = new Pair<int, int*>(6, nullptr);
  int* y = p2->getSecond();

  Pair<Pair<float, bool>, Pair<int, int*>* > p3(p1, p2);

  delete p2;
};
```

Common Pitfalls (Watch Out!)

```
const first_t& Pair<first_t, second_t>::getFirst() const {
  return first_;
}

template <typename first_t, typename second_t>
const first_t& Pair::getFirst() const {
  return first_;
}

int main() {
  Pair p1(4.2, true);
}
```

Missing template declaration
(first_t, second_t undefined!)

Pair is not a class!
You have to specify types!

Summary

- Class templates let you write classes that work with many types
  - e.g. a container that can hold different types
- Don’t forget!
  - Member functions are templates too!
  - You must supply template parameters when referring to the class