C++ Generics - Templates

CSE 333 Winter 2021

Instructor: John Zahorjan

Teaching Assistants:

Matthew Arnold	Nonthakit Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

Lecture Outline

Generic programming https://www.definitions.net/definition/generic+programming

In the simplest definition, generic programming is a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters. This approach, pioneered by ML in 1973, permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication. Such software entities are known as generics in Ada, Delphi, Eiffel, Java, C#, F#, and Visual Basic .NET; parametric polymorphism in ML, Scala and Haskell; templates in C++ and D; and parameterized types in the influential 1994 book Design Patterns. The authors of Design Patterns note that this technique, especially when combined with delegation, is very powerful but that "[dynamic], highly parameterized software is harder to understand than more static software." The term generic programming was originally coined by David Musser and Alexander Stepanov *[principal behind C++ Standard Template Library]* in a more specific sense than the above, to describe a programming paradigm whereby fundamental requirements on types are abstracted from across concrete examples of algorithms and data structures and formalised as concepts, with generic functions implemented in terms of these concepts, typically using language genericity mechanisms as described above.

- **C++ Templates**
  - Essential to the implementation of the C++ standard library
  - Possibly useful to your application code

Generalizing From Examples

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const int &value1, const int &value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const string &value1, const string &value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const Vector3d &value1, const Vector3d &value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}
```

Recognizing and Generalizing

- Recognizing
  - There is one abstract computation
  - The program implementation requires three distinct pieces of code expressing that computation
  - Why?

- Generalizing
  - Could we instead write a single generic expression of a computation that is generalizable across an appropriate set of specific types?
  - Answer: Yes!

Parametric Polymorphism

- The programmer provides a single, generic implementation of a computation that applies to many types
  - We do not require there be any parent/sub-class relationship among the types
- From that, we need to instantiate a concrete version of generic code that is specialized to the types of the arguments actually supplied
  - A highly simplified example: what does "x+y" mean?
- When should this specialization take place?
  - Statically – at compile time
  - Dynamically – during program execution
- Statically results in faster executables
  - Guess which approach C++ takes…!

C++ Templates

- A C++ template is, well, a generic code template from which specific code can be generated
  - The template takes one (or more) arguments that typically are type names
  - When the compiler encounters a use of the generic code, it instantiates the specific concrete realization of it according to the types the generic code is being asked to operate on
- How does the compiler determine the types for a particular use?
  - The programmer tells it, or
  - The compiler can figure it out because it knows the types of the arguments

C++ Function Templates

```
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>   // <...> can also be written <class T>
int compare(const T &value1, const T &value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}

int main(int argc, char **argv) {
  std::string h("hello"), w("world");
  std::cout << compare<int>(10, 20) << std::endl;
  std::cout << compare<std::string>(h, w) << std::endl;
  std::cout << compare<double>(50.5, 50.6) << std::endl;
  std::cout << compare <int>(-5, -20) << std::endl;
  return 0;
}
```

No code is generated for the template

Three programmer explicit instantiations

Compiler Inference

- Sometimes the compiler can figure out the instantiated types itself

```
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>
int compare(const T &value1, const T &value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}

int main(int argc, char **argv) {
  std::string h("hello"), w("world");
  std::cout << compare(10, 20) << std::endl; // ok
  std::cout << compare(h, w) << std::endl;   // ok
  std::cout << compare("Hello", "World") << std::endl;   // hm…
  return 0;
}
```

Yeah! ?

BUG!
(Compiler is right; programmer is wrong.)

Templates Aside #1

- What's going on is more complicated that our simple model of compile to assembler, assemble to machine code, and link
  - The compiler has to pick a specific name for the instantiated function, because the linker needs specific names to match call sites to method entry points
  - The compiler needs to instantiate in each compilation unit (source file), because maybe that's the only file in which that templated function is used with that particular type
    - But maybe not
  - So the linker has to be prepared to find multiple definitions of a function with a single name

Templates Aside #2

- Think about trying to implement “templates” in C using preprocessor macros
  - What part of template functionality could you implement?
  - What part of template functionality couldn’t you implement?

Compile Time Instantiation Creates a Restriction

Template code must be available for expansion where *it is used.*

That means it "must go" in the .h file, not in a separate .cc file.

```
#include <iostream>
#include "compare.h"

using namespace std;

int main(int argc, char **argv)
{
  cout << comp<int>(10, 20);
  cout << endl;
  return 0;
}
```

```
#ifndef _COMPARE_H_
#define _COMPARE_H_

template <typename T>
int comp(const T& a, const T& b)
{
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}

#endif  // _COMPARE_H_
```

Templating Creates Another Restriction

```
template <typename T>
int comp(const T& a, const T& b)
{
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
```

vs.

```
template <typename T>
int comp(const T& a, const T& b)
{
  if (a < b) return -1;
  if (a > b) return 1;
  return 0;
}
```

What's the difference between these two implementations?

C++ Template (Constant) Values (Not Types)

- You can use non-types (constant values) in a template
- When instantiating, the value must be known at compile time

```
#include <iostream>

// return pointer to new N-element heap array filled with val
template <typename T, int N>
T* valarray(const T & val) {
  T* a = new T[N];
  for (int i = 0; i < N; ++i)
    a[i] = val;
  return a;
}

int main(int argc, char **argv) {
  int *ip = valarray<int, 10>(17);
  std::string *sp = valarray<std::string, 12>("hello");
   ...
}
```

- This is not macro expansion
- This is not runtime parameter passing

Class Templates

- Templates are useful for classes as well
  - (In fact, that was one of the main motivations for templates!)
- The standard library (STL) is full of them
  - Compelling example: generic containers

- Example: Imagine we want a class whose instances hold a pair of things such that we can:
  - Set the value of the first thing
  - Set the value of the second thing
  - Get the value of the first thing0
  - Get the value of the second thing
  - Swap the values of the things
  - Print the pair of things

Templated Pair Class Definition

```
#ifndef _PAIR_H_
#define _PAIR_H_

template <typename T> class Pair {
 public:
  Pair() { };

  T get_first() const { return first_; }
  T get_second() const { return second_; }
  Pair<T> & set_first(T &other);
  Pair<T> & set_second(T &other);
  Pair<T> & Swap();

 private:
  T first_;
  T second_;
};

// continued on next slide
```

Templated Pair Function Definitions

```
template <typename T>
Pair<T> & Pair<T>::set_first(T &other)
{
  first_ = other;
  return *this;
}

template <typename T>
Pair<T> & Pair<T>::set_second(T &other)
{
  second_ = other;
  return *this;
}

template <typename T>
Pair<T> & Pair<T>::Swap()
{
  T tmp = first_;
  first_ = second_;
  second_ = tmp;
  return *this;
}

// continued on next slide
```

Note: This weird thing where we put method definitions in the .h file but outside of the class declaration can be done with non-template classes as well.

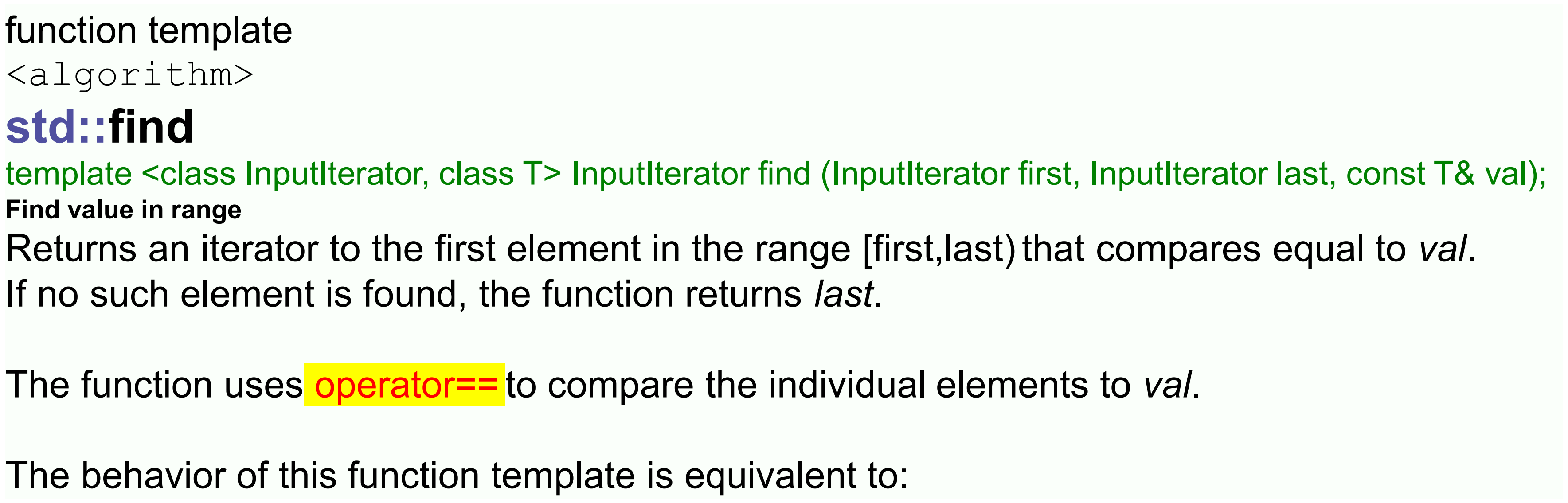
Non-Class Method Using Templated Pair

```
template <typename T>
std::ostream &operator<<(std::ostream &out, const Pair<T>& p) {
  return out << "Pair(" << p.get_first() << ", "
             << p.get_second() << ")";
}

#endif  _PAIR_H_
```

Using Pair in an App

```
#include <iostream>
#include <string>

#include "Pair.h"

int main(int argc, char** argv)
{
  Pair<std::string> ps;
  std::string x("foo"), y("bar");

  ps.set_first(x);
  ps.set_second(y);
  ps.Swap();
  std::cout << ps << std::endl;

  return 0;
}
```

Templated Classes May Have Required Operations



function template
`<algorithm>`

std::find

template <class InputIterator, class T> InputIterator find (InputIterator first, InputIterator last, const T& val);

Find value in range

Returns an iterator to the first element in the range [first,last) that compares equal to *val*.
If no such element is found, the function returns *last*.

The function uses operator== to compare the individual elements to *val*.

The behavior of this function template is equivalent to:

```
template<class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val) {
    while (first!=last)
    {
        if (*first==val)
            return first; ++first;
    }
    return last;
}
```

Our Pair - Warning!

```
void Pair<T>::set_first(T &other) {
  first_ = other;
}

void Pair<T>::set_second(T &other) {
  second_ = other;
}

void Pair<T>::Swap() {
  T tmp = first_;
  first_ = second_;
  second_ = tmp;
}
```

```
std::ostream &operator<<(std::ostream &out, const Pair<T>& p) {
  return out << "Pair(" << p.get_first() << ", "
             << p.get_second() << ")";
}
```

When might this fail?