High Level Programming

Templates and Generic Programming

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

License Information

Templates

- In problem solving, data structures often include different types T of data
- In C++ we can write code whose functionalities are independent from the specific type T
  - We have already seen this strategy for containers, generic algorithms, smart pointers, etc.

```
vector<int> vi;
vector<string> vs;
vector<vector<<int>> mi;

map<string,int> word_count;

shared_ptr<string> p1;
```

<int>, <string>, etc.
specify the object type **T**

Templates

- The problem is how to do a similar thing for our functions, classes, etc.

you want to write a generic function compare that can compare two values of any type T. Instead of writing separate versions of compare for each type, you want to write a single version of compare that works for all types T.

- For example
  - Let us suppose we want to write the function

```
int compare (T &a, T &b);
```

  - Which returns
    - -1 if v1<v2, 0 if v1==v2, +1 if v1>v2
  - And it is independent of the type **T**
    - Thus, we can avoid writing several versions of the function **compare** to sharing the structure but working on different types

Examples

```
int compare (const int &v1, const int &v2) {
  if (v1 < v2)
    return -1;
  if (v2 < v1)
    return 1;
  return 0;
}

int compare (const string &v1, const string &v2) {
  if (v1 < v2)
    return -1;
  if (v2 < v1)
    return 1;
  return 0;
}
```

Those functions are nearly identical as the only difference is the type of the parameters

Can we avoid two (N) versions of **compare**?
How can we avoid massive code duplication?
How can we account for user-defined types?

Templates

- Rather than defining a new function for each type we can define a function template
  - Templates are the foundation of generic programming in C++
- A function templates is a formula from which we can generate type-specific versions of that function
  - We write functions taking arguments of **arbitrary types**
  - We start with functions, then we see how to write **class templates**

Function templates

- The template version of the compare function looks like the following
  - T is a template parameter, which must be a type

Template parameter list (comma-separated list of one or more parameters)

T a label which will be substituted by the correct object type

```
template <typename T>
int compare (const T &v1, const T &v2) {
  if (v1 < v2)
    return -1;
  if (v2 < v1)
    return 1;
  return 0;
}
```

Parameters received by reference !

The **compiler** will generate a new version of the function (using polymorphism) when this is required

Instantiation

The template is **never** processed at run-time

- When you call a template function, the compiler
  - Deduces what types to use instead of the template parameters
  - **Instantiates** ("generates") a function with the correct types from the templates

```
cout << compare(1, 0) << endl;

string s1 = "hello";
string s2 = "world";
cout << compare(s1, s2) << endl;

vector<int> v1{1, 2, 3}, v2{4, 5, 6};
cout << compare(v1, v2) << endl;
```

T is an int.
The compiler instantiates
int compare(const int &, const int &)
The first version on page 5

T is a string:
int compare(const string &, const string &)

T is a vector of int.

Personal types

- A template can also be used with personal types
  - Templates usually put some requirements on the argument types
    - If these requirements are not met the instantiation will fail
    - Compilation errors are usually reported during instantiation

T is a Rectangle.
The compiler instantiates
int compare(const Rectangle &, const Rectangle &)

```
Rectangle r1(2,3), r2(3,4.5);
cout << compare(r1, r2) << endl;
```

If the class Rectangle implements operator <, everything is ok, otherwise we have an error at compilation time

Explicit types

- The compiler must always be able to understand to which type T is referring to
  - It is possible to set the type during the call

Type specification
Useless in this case

```
cout << compare<int>(1, 0) << endl;

string s1 = "hello";
string s2 = "world";
cout << compare<string>(s1, s2) << endl;
```

Type specification
Useless in this case

Templates with different types

- A function template can receive more than one type
  - The differentiation **must** make sense
  - The operation **must** be feasible

```
template <typename T1, typename T2>
int compare (const T1 &v1, const T2 &v2) {
  if (v1 < v2)
    return -1;
  if (v2 < v1)
    return 1;
  return 0;
}
```

Function with a return value

- A template function can also have a template type, to return a variable type

```
template <typename T>
T compare(const T &v1, const T &v2) {
  if (v1 < v2)
    return -1;
  if (v2 < v1)
    return 1;
  return 0;
}
```

Return value

One type

Two types

```
template <typename T1, typename T2>
T3 compare(const T1 &v1, const T2 &v2) {
  if (v1 < v2) { return -1; }
  if (v2 < v1) { return 1; }
  return 0;
}
```

T3 does not need the keyword "template"

Parameter type

- Parameters can be passed by value, pointer, or reference

```
int a, b;

swap1(a,b);
swap2(&a,&b);
swap3(a,b);
```

swap1 will not swap a and b

```
float a, b;

swap1(a,b);
swap2(&a,&b);
swap3(a,b);
```

```
template<typename T>
void swap1 (T a, T b) {
  T tmp;
  tmp = a; a = b; b = tmp;
  return;
}
template<typename T>
void swap2 (T *a, T *b) {
  T tmp;
  tmp = *a; *a = *b; *b = tmp;
  return;
}
template<typename T>
void swap3 (T &a, T &b) {
  T tmp;
  tmp = a; a = b; b = tmp;
  return;
}
```

Limitations

- Pay attention to the requirements forced on T
  - The previous compare template requires that objects T could be compared with operator "<"
  - Now if we use operator "==" we have a new constrain on T

This implementation forces more requirements on T

```
template <typename T>
int compare (const T &v1, const T &v2) {
  if (v1 < v2) { return -1; }
  if (v1 == v2) { return 0; }
  return 1;
}
```

The arguments are passed by value, so it **must** be possible to **copy** objects of T
Objects of T **must** implement **comparisons** with < and ==

Class templates

- As functions, also a class can be programmed to deal with generic data types
  - The definition of a class template is similar to the definition of a function template
  - The main difference is that the compiler cannot deduce the type of T
    - For class templates we must specify the type of T when we instantiate it

Explicit type definition

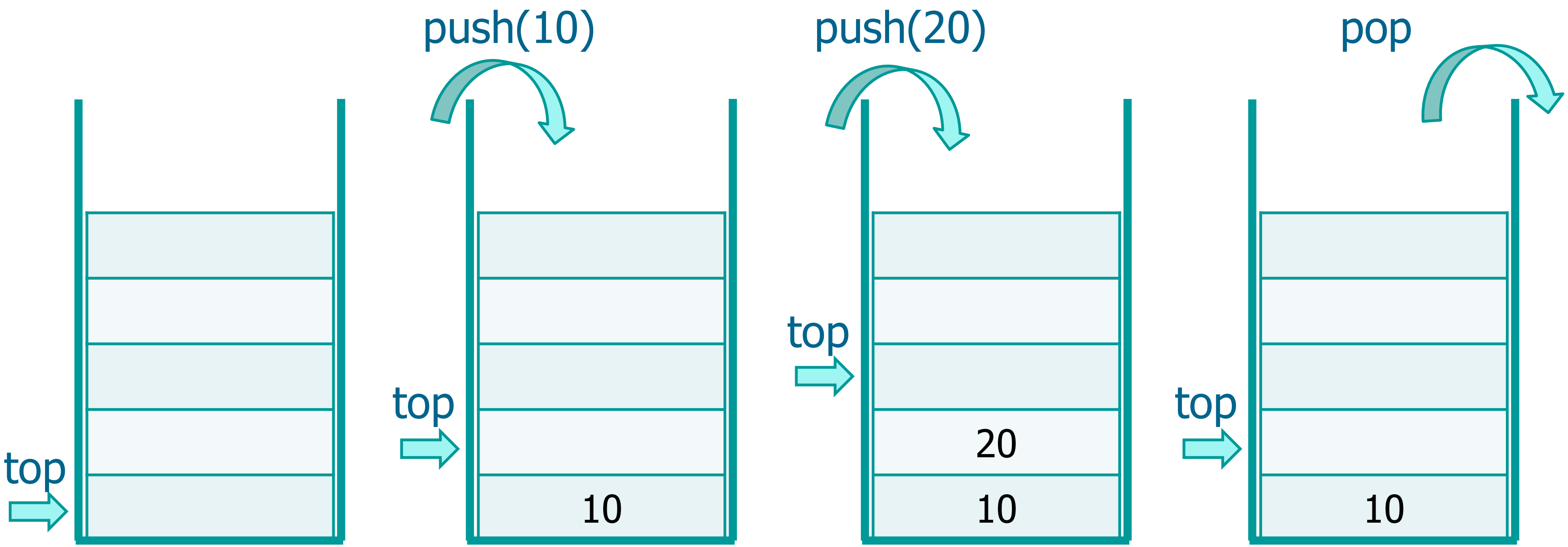
```
vector<int> vi;
vector<string> vs;
map<string,int> word_count;
shared_ptr<string> p1;
```

Class templates

- Templates do not have a proper implementation
  - The compiler is going to take care of it
  - There is no source code to compile
    - A template specifies a set of instructions that the compiler will use to generate the class definition
    - As with function, a particular realization of a template is called an **instantiation** or **specialization**
- Thus, given a template, we must understand where to insert its
  - Declaration (interface)
  - Implemention (definition)
- Several approaches are indeed possible

Exercise

- Write a template to manipulate a stack such that
  - The standard stack operations are allowed (i.e., push, pop, empty, etc.)
  - The stack can store objects of different types



Approach 1

my.h
Declarations and definitions

```
#ifndef MY_H
#define MY_H
#include <iostream>
#include <vector>

template <class T>
class my_vector {
public:
  void my_push(const T &elem) {
    v.push_back(elem);
  }
  T my_pop() {
    T tmp = v.back();
    v.pop_back();
    return tmp;
  }
private:
  std::vector<T> v;
};

#endif
```

main.c
Client

```
#include "my.h"

using std::cin;
using std::cout;
using std::endl;
using std::string;

int main() {
  my_vector<int> v;
  v.my_push (1);
  v.my_pop ();
  return 0;
}
```

Approach 2

my.h
Declarations

```
#ifndef MY_H
#define MY_H
#include <iostream>
#include <vector>
template <class T>
class my_vector {
public:
   void my_push(const T &elem);
   T my_pop();
private:
   std::vector<T> v;
};
template <class T>
void my_vector<T>::my_push(
   const T &elem) {
   v.push_back(elem);
}
template <class T>
T my_vector<T>::my_pop() {
   T tmp = v.back();
   v.pop_back();
   return tmp;
}
#endif
```

Definitions are outside

Template information must be repeated

main.c
Client

```
#include "my.h"

using std::cin;
using std::cout;
using std::endl;
using std::string;

int main() {
   my_vector<int> v;
   v.my_push (1);
   v.my_pop ();
   return 0;
}
```

Approach 3

my.h
Declarations

```
#ifndef MY_H
#define MY_H
#include <iostream>
#include <vector>
template <class T>
class my_vector {
public:
  void my_push(const T &elem);
  T my_pop();
private:
  std::vector<T> v;
}; #endif
```

my.hpp
Definitions

```
#ifndef MY_HPP
#define MY_HPP
#include <iostream>
#include <vector>
template <class T>
void my_vector<T>::my_push(const T &elem) {
  v.push_back(elem);
}
template <class T>
T my_vector<T>::my_pop() {
  T tmp = v.back();
  v.pop_back();
  return tmp;
}
#endif
```

main.c
Client

```
#include "my.h"
#include "my.hpp"

using std::cin;
using std::cout;
using std::endl;
using std::string;

int main() {
  my_vector<int> v;
  v.my_push (1);
  v.my_pop ();
  return 0;
}
```

Approach 4

my.h
Declarations

```
#ifndef MY_H
#define MY_H
#include <iostream>
#include <vector>
template <class T>
class my_vector {
public:
  void my_push(const T &elem);
  T my_pop();
private:
  std::vector<T> v;
};
#endif
```

main.c
Definitions and client

```
#include "my.h"
using ...
template <class T>
void my_vector<T>::my_push(const T &elem) {
  v.push_back(elem);
}
template <class T>
T my_vector<T>::my_pop() {
  T tmp = v.back();
  v.pop_back();
  return tmp;
}
int main() {
  my_vector<int> v;
  v.my_push (1);
  v.my_pop ();
  return 0;
}
```