A <Basic> C++ Course

6
Fonctions et classes templates

Julien Deantoni

adapted from Jean-Paul Rigault courses

References vs. pointers

- References and pointers
  - A reference must be initialized
  - There is nothing such as a NULL (nullptr) reference
  - There is nothing such as a reference to a function
  - References cannot be assigned to (the referenced objects are)

- Address of a reference

```
int i;
int& ri = i;
int* pi = &ri;   //pi == &i
```

Structure of programs

Header file (.h or .hpp)

- Specification of a module
- Not a compilation unit:
  included in other source files
  - global variables declaration
  - constant and static (file scope) variable and function declarations
  - inline function definition
  - class definitions
  - free functions declarations
  - template declarations and definitions

Non header (.cpp)

- Implemenation of a module
- Compiled separately
  - global variable definitions
  - function definitions

Structure of programs

Header file (. h or . hpp)

- Specification of a module
- Not a compilation unit:
  included in other source files
  - global variables declaration
  - constant and static (file scope) variable and function declarations
  - inline function definition
  - class definitions
  - free functions declarations
  - template declarations **AND definitions**

Non header (.cpp)

- Implemenation of a module
- Compiled separately
  - global variable definitions
  - function definitions

Motivation and principles

- Several functions with identical bodies...

```
int Min(int a, int b)
{
   return a < b ? a : b;
}

float Min(float a, float b)
{
   return a < b ? a : b;
}

// etc.
```

Motivation and principles

- Several functions with identical bodies…

```
int Min(int a, int b)
{
   return a < b ? a : b;
}

float Min(float a, float b)
{
   return a < b ? a : b;
}

// etc.
```

- … but with different type parameters

➔ **Parameterized overloading**

```
template <typename T>
   T Min(T a, T b)
   {
       return a < b ? a : b;
   }
```

Motivation and principles

- Several functions with identical bodies…

```
int Min(int a, int b)
{
   return a < b ? a : b;
}

float Min(float a, float b)
{
   return a < b ? a : b;
}

// etc.
```

- … but with different type parameters

➔ **Parameterized overloading**

```
template <typename T>
   T Min(T a, T b)
   {
        return a < b ? a : b;
   }

   double x = Min(2.7, 3.14);
   int i = Min(3, 17);
   c = Min('a', 't');
```

Les types sont inférés des appels

Regular functions and templates

- Template instantiation is blind!

```
char* s = Min("aaa", "zzzzzzz");
```

  - Instantiation of `char* Min(char* , char*)`

Regular functions and templates

- Template instantiation is blind!

```
char* s = Min("aaa", "zzzzzzz");
```

  - Instantiation of `char* Min(char* , char*)`
  - The behaviour of operator **<** on character pointer is not what is expected!

Regular functions and templates

- Template instantiation is blind!

```
char* s = Min("aaa", "zzzzzzz");
```

  - Instantiation of **`char* Min(char*, char*)`**
  - The behaviour of operator **<** on character pointer is not what is expected!

- You may define a regular function which supersedes the generic form

```
char* Min(char* s1, char* s2)
{
   return strcmp(s1, s2) < 0 ? s1 : s2;
}
```

Regular functions and templates

- Template instantiation is blind!

```
char* s = Min("aaa", "zzzzzzz");
```

  - Instantiation of **`char* Min(char*, char*)`**
  - The behaviour of operator **<** on charact
    expected!

What if we use string instead of char* ?

- You may define a regular function which supersedes the generic form

```
char* Min(char* s1, char* s2)
{
    return strcmp(s1, s2) < 0 ? s1 : s2;
}
```

Regular functions and templates

- Template instantiation is blind!

```
char* s = Min("aaa", "zzzzzzz");
```

  - Instantiation of **`char* Min(char*, char*`**
  - The behaviour of operator **<** on chara
    expected!

What if we use Rectangle
instead of char* ?

- You may define a regular function which supersedes the generic form

```
char* Min(char* s1, char* s2)
{
   return strcmp(s1, s2) < 0 ? s1 : s2;
}
```

Declaration (prototype)

```
template <typename Item>
void Sort(int n, Item t[]);

template <typename A,typename B>
void f(A, A&, B);

template <int N>
void g(int N);
```

Declaration (prototype)

```
template <typename Item>
void Sort(int n, Item t[]);

template <typename A,
          typename B>
void f(A, A, B);

template <int N>
void g(int N);
```

```
template <int N>
void g(int t[N]); // guess N ??

int t[10];
g(t);     // ???
```

Declaration (prototype)

```
template <typename Item>
void Sort(int n, Item t[]);

template <typename A,
          typename B>
void f(A, A, B);

template <int N>
void g(int N);
```

```
template <int N>
void g(int t[N]); // guess N ??

int t[10];
g(t);     // KO
```

Declaration (prototype)

```
template <typename Item>
void Sort(int n, Item t[]);

template <typename A,
          typename B>
void f(A, A, B);

template <int N>
void g(int N);
```

```
template <int N>
void g(int t[N]); // guess N ??

int t[10];
g(t);     // KO

template <typename T>
T f(int);      //  guess T ??

double x;
x = f(3); // ???
```

Declaration (prototype)

```
template <typename Item>
void Sort(int n, Item t[]);

template <typename A,
          typename B>
void f(A, A, B);

template <int N>
void g(int N);
```

```
template <int N>
void g(int t[N]); // guess N ??

int t[10];
g(t);     // KO

template <typename T>
T f(int);       //  guess T ??

double x;
x = f(3); // KO
```

Declaration (prototype)

```
template <typename Item>
void Sort(int n, Item t[]);

template <typename A,
          typename B>
void f(A, A, B);

template <int N>
void g(int N);
```

```
template <int N>
void g(int t[N]); // guess N ??
// ...
int t[10];
g<10>(t);    // OK

template <typename T>
T f(int);      //  guess T ??
// ...
double x;
x = f<double>(3); // OK
```

Declaration (prototype)

```
template <typename Item>
void Sort(int n, Item t[]);

template <typename A,
          typename B>
void f(A, A, B);

template <int N>
void g(int N);
```

```
template <int N>
void g(int t[N]); // guess N ??
// ...
int t[10];
g<10>(t);// OK

template <typename T>
T f(int);      //  guess T ??
// ...
double x;
x = f<double>(3); // OK
```

Les types ne peuvent pas être inférés des appels
→ l'instanciation des templates doit être explicite !

Declaration (prototype)

```
template <typename Item>
void Sort(int n, Item t[]);

template <typename A,
          typename B>
void f(A, A, B);

template <int N>
void g(int N);
```

```
template <int N>
void g(int t[N]); // guess N ??
// ...
int t[10];
g<10>(t);// OK

template <typename T>
T f(int);     //  guess T ??
// ...
double x;
x = f<double>(3); // OK
```

Les types ne peuvent pas être inférés des appels
→ l'instanciation des templates doit être explicite !

```
vector<Person *> children;
```

Les types aussi peuvent être “template”

...an example

A Stack without template...

```
#ifndef _INTSTACK15_H_
#define _INTSTACK15_H_


const int N = 15;
class IntStack15{
  private:
       int _values[N];
       int _top;
  public:
       IntStack15();
       void push(int);
       int pop();
       bool is_full();
       bool is_empty();
};
#endif
```

IntStack15.h

...an example

A Stack without template...

```cpp
#ifndef _INTSTACK15_H_
#define _INTSTACK15_H_


const int N = 15;
class IntStack15{
  private:
       int _values[N];
       int _top;
  public:
       IntStack15();
       void push(int);
       int pop();
       bool is_full();
       bool is_empty();
};
#endif
```

IntStack15.h

```cpp
#include “IntStack15.h”
IntStack15::IntStack15(){
   _top=0;
}
IntStack15::push(int newVal){
   _values[_top++]=newVal;
}
int IntStack15::pop(){
   return _values[--_top];
}
bool IntStack15::is_full(){
   return _top >= N;
}
bool IntStack15::is_empty(){
   return _top == 0;
}
```

IntStack15.cpp

...an example

A Stack with an int template...

```
#ifndef _INTSTACK_H_
#define _INTSTACK_H_

template <int N>

class IntStack{
  private:
       int _values[N];
       int _top;
  public:
       IntStack();
       void push(int);
       int pop();
       bool is_full();
       bool is_empty();
};
#include "IntStack.cpp"
#endif
```

IntStack.h

...an example

A Stack with an int template...

```cpp
#ifndef _INTSTACK_H_
#define _INTSTACK_H_

template <int N>

class IntStack{
  private:
       int _values[N];
       int _top;
  public:
       IntStack();
       void push(int);
       int pop();
       bool is_full();
       bool is_empty();
};
#include “IntStack.cpp”
#endif
```

IntStack.h

No separated compilation

...an example

A Stack with an int template...

```
#ifndef _INTSTACK_H_
#define _INTSTACK_H_

template <int N>

class IntStack{
  private:
       int _values[N];
       int _top;
  public:
       IntStack();
       void push(int);
       int pop();
       bool is_full();
       bool is_empty();
};
#include "IntStack.cpp"
#endif
```

IntStack.h

```
#include "IntStack.h"
template <int N>
IntStack<N>::IntStack(){
   _top=0;
}
template <int N>
IntStack<N>::push(int newVal){
   _values[_top++]=newVal;
}
template <int N>
int IntStack<N>::pop(){
   return _values[--_top];
}
template <int N>
bool IntStack<N>::is_full(){
   return _top >= N;
}
template <int N>
bool IntStack<N>::is_empty(){
   return _top == 0;
}
```

IntStack.cpp

...an example

A Stack with an int template...

```
#ifndef _INTSTACK_H_
#define _INTSTACK_H_

template <int N>

class IntStack{
  private:
       int _values[N];
       int _top;
  public:
       IntStack();
       void push(int);
       int pop();
       bool is_full();
       bool is_empty();
};
#include "IntStack.cpp"
#endif
```

IntStack.h

```
#include "IntStack.h"
template <int N>
IntStack<N>::IntStack(){
   _top=0;
}
template <int N>
IntStack<N>::push(int newVal){
   _values[_top++]=newVal;
}
template <int N>
int IntStack<N>::pop(){
   return _values[--_top];
}
template <int N>
bool IntStack<N>::is_full(){
   return _top >= N;
}
         <int N>
         tack<N>::is_empty(){
          N == 0;
```

IntStack.cpp

```
...
IntStack<15> s1;
IntStack<10> s2;
...
```

...an example

A Stack with an int template...

```
#ifndef _INTSTACK_H_
#define _INTSTACK_H_

template <int

class IntStack
  private:
       int _val
       int _top
  public:
       IntStack
       void push(int);
       int pop();
       bool is_full();
       bool is_empty();
};
#include "IntStack.cpp"
#endif
```

Ir

```
#include "IntStack.h"
template <int N>
IntStack<N>::IntStack(){
   _top=0;
}
template <int N>
                             ewVal){
                             al;


                             ];

template <int N>
bool IntStack<N>::is_full(){
   return _top >= N;
}
          int N>
          ack<N>::is_empty(){
          N == 0;
```

IntStack.cpp

Deux instanciations d'une même classe "template" avec des paramètres différents donnent deux types différents

```
...
IntStack<15> s1;
IntStack<10> s2;
...
```

...an example

A Stack with an 2 templates...

```
#ifndef _STACK_H_
#define _STACK_H_

template <typename T,int N>

class Stack{
  private:
       T _values[N];
       int _top;
  public:
       Stack();
       void push(T);
       T pop();
       bool is_full();
       bool is_empty();
};
#include “Stack.cpp”
#endif
```

Stack.h

...an example

A Stack with an 2 templates...

```cpp
#ifndef _STACK_H_
#define _STACK_H_

template <typename T,int N>

class Stack{
  private:
        T _values[N];
        int _top;
  public:
        Stack();
        void push(T);
        T pop();
        bool is_full();
        bool is_empty();
};
#include "Stack.cpp"
#endif
```

Stack.h

```cpp
#include "Stack.h"
template <typename T,int N>
Stack<T,N>::Stack(){
   _top=0;
}
template <typename T,int N>
Stack<T,N>::push(T newVal){
   _values[_top++]=newVal;
}
template <typename T,int N>
T Stack<T,N>::pop(){
   return _values[--_top];
}
template <typename T,int N>
bool Stack<T,N>::is_full(){
   return _top >= N;
}
template <typename T,int N>
bool Stack<T,N>::is_empty(){
   return _top == 0;
}
```

Stack.cpp

...an example

A Stack with an 2 templates...

```
#ifndef _STACK_H_
#define _STACK_H_

template <typename T,int N>

class Stack{
  private:
       T _values[N];
       int _top;
  public:
       Stack();
       void push(T);
       T pop();
       bool is_full();
       bool is_empty();
};
#include "Stack.cpp"
#endif
```

S

```
#include "Stack.h"
template <typename T,int N>
Stack<T,N>::Stack(){
   _top=0;
}
template <typename T,int N>
Stack<T,N>::push(T newVal){
   _values[_top++]=newVal;
}
template <typename T,int N>
T Stack<T,N>::pop(){
   return _values[--_top];
}
template <typename T,int N>
bool Stack<T,N>::is_full(){
   return _top >= N;

pename T,int N>
,N>::is_empty(){
op == 0;
```

Stack.cpp

```
...
Stack<int,15> s1;
Stack<std::string, 10> s2;
...
```

...an example

A Stack with an 2 templates...

```
#ifndef _STACK_H_
#define _STACK_H_

template <typename T,int N>

class Stack{
  private:
      T _values[N];
      int _top;
  public:
      Stack();
      void push(T);
      T pop();
      bool is_full()
      bool is_empty(
};
#include "Stack.cpp"
#endif
```

```
#include "Stack.h"
template <typename T,int N>
Stack<T,N>::Stack(){
   _top=0;
}
template <typename T,int N>
Stack<T,N>::push(T newVal){
   _values[_top++]=newVal;
}
template <typename T,int N>
T Stack<T,N>::pop(){
   return _values[--_top];
}
template <typename T,int N>
s_full(){
N;

T,int N>
s_empty(){
0;
```

Stack.cpp

```
...
Stack<int,15> s1;
Stack<std::string, 10> s2;
Stack<Stack<int,5>, 10> s3;
...
```

...an example

A Stack with an 2 templates...

```
#ifndef _STACK_H_
#define _STACK_H_

template <typename T,int N>

class Stack{
  private:
       T _values[N];
       int _top;
  public:
       Stack();
       Stack(const Stack&);
       void push(T);
       T pop();
       bool is_full();
       bool is_empty();
       void operator=(const Stack&)
};
#include "Stack.cpp"
#endif
```

Stack.h

```
#include "Stack.h"
template <typename T,int N>
Stack<T,N>::Stack(){
    _top=0;
}
template <typename T,int N>
Stack<T,N>::push(T newVal){
    _values[_top++]=newVal;
}
template <typename T,int N>
```

And the associated definitions (i.e. implementations)

```
...
Stack<int,15> s1;
Stack<std::string, 10> s2;
Stack<Stack<int,5>, 10> s3;
...
```

...an example

A Stack with an 2 templates...

```
#ifndef _STACK_H_
#define _STACK_H_

template <typename T,int N>

class Stack{
  private:

  pu
      Stack(const Stack&);
      void push(T);
      T pop();
      bool is_full();
      bool is_empty();
      void operator=(const Stack&)
};
#include "Stack.cpp"
#endif
```

Stack.h

```
#include "Stack.h"
template <typename T,int N>
Stack<T,N>::Stack(){
   _top=0;
}
template <typename T,int N>
Stack<T,N>::push(T newVal){
```

Ici, le compilateur va “instancier” les templates
pour chacun des types, créer les “.o” et les ajouter au projet !!

definitions
(i.e. implementations)

```
...
Stack<int,15> s1;
Stack<std::string, 10> s2;
Stack<Stack<int,5>, 10> s3;
...
```

...an example

A Stack with an 2 templates...

```
#ifndef _STACK_H_
#define _STACK_H_

template <typename T,int N>

class Stack{
  private:
      T _values[N];
      int _top;
  p
      bool is_empty();
      void operator=(const Stack&)
};
#include "Stack.cpp"
#endif
```

Stack.h

```
#include "Stack.h"
template <typename T,int N>
Stack<T,N>::Stack(){
   _top=0;
}
template <typename T,int N>
Stack<T,N>::push(T newVal){
   _values[_top++]=newVal;
}
```

Ici, le compilateur va "instancier" les templates
pour chacun des types, créer les ".o" et les ajouter au projet !!

Autre manière de compiler !! Donc Makefile à changer !!

```
...
Stack<int,15> s1;
Stack<std::string, 10> s2;
Stack<Stack<int,5>, 10> s3;
...
```

compilation séparée

```
# sketchy makefile example
EXE_NAME=executable
LINK_CXX=g++
COMPIL_CXX=g++ -c

example: main.o rectangle.o
	$(LINK_CXX) main.o rectangle.o -o $(EXE_NAME)
main.o: main.cpp
	$(CXX)  main.cpp
rectangle.o: rectangle.cpp rectangle.h
	$(CXX) rectangle.cpp
```

Makefile

Compiling

A Makefile example for template...

Pas de compilation séparée

```
# Common targets
# Variables ALL must be defined in specific makefiles; in addition
# FOR NON TEMPLATE FILES

#
# project_Stack: main_Stack_char_10.o Stack_char_10.o
#      $(LINK_CXX) main_Stack_char_10.o Stack_char_10.o -o $(EXE_NAME)
# Stack_char_10.o: Stack_char_10.cpp Stack_char_10.h
#      $(CXX) Stack_char_10.cpp
# main_Stack_char_10.o: main_Stack_char_10.cpp
#      $(CXX)  main_Stack_char_10.cpp

#
# FOR TEMPLATE FILES
# we do not make separate compilation of the templated entities

project_Stack: main_Stack.o
       $(LINK_CXX) main_Stack.o -o $(EXE_NAME)
main_Stack.o: main_Stack.cpp Stack.h Stack.cpp
       $(CXX)  main_Stack.cpp
```

Makefile