CSCI251/CSCI851 Spring-2021
Advanced Programming **(S6f)**

Generic Programming VI: Template compilation models

Outline

- Template compilation models:
  - The inclusion model.
  - The explicitly instantiation model.
  - The separation model.

Template Compilation

- So far, we have assumed that all template examples place fully-defined templates within each compilation unit.
- This is not the conventional practice.
- Conventional practice:
  – All declarations are placed in .h files.
  – Definitions and implementation are placed in .cpp files.

- Rationale:
  - Non-inline function bodies in header files lead to multiple function definitions, resulting in linker errors. ☹
  - Hiding the implementation from clients helps reduce compile-time coupling. ☺
  - Vendors can distribute pre-compiled code, for a particular compiler, along with headers so that users cannot see the function implementations.
    - The headers might, for example, allow local constants to be defined for setting in the linking phase.
      - This is often done in Makefiles anyway.
  - Compile times are shorter since header files are smaller.

- Remember that templates are not code as such, but instructions for code generation.
- Only template instantiations are real code. Code is generated by the compiler when…
  - … the compiler has seen a complete template definition during compilation, and
  - … encounters a point of instantiation for that template in the same translation unit.

- The problem is that we need to know where references requiring instantiations are.
  - There are a few ways in which we can do this.

Template Compilation Models

- Inclusion model:
  - The most common approach consists of generating the code for the instantiation in every translation unit and letting the linker weed out duplicates.
- Explicit instantiation model:
- The separation model:

A Stack Template

```
// file: Stack.h
#ifndef _STACK_H_
#define _STACK_H_
#include <iostream>

template<class T>
class Stack {
private:
  int size;     // # of elements in the stack
  int top;      // location of the top element
  T *stackPtr   // pointer to the stack
public:
  Stack(int=10);       // default constructor (stack size=10)
  ~Stack() { delete [] stackPtr;} // destructor
  bool push(const T&); // push an element onto the stack
  bool pop(T&);               // pop an element off the stack
private:
  bool isFull() const {return top == size-1;}
  bool isEmpty() const {return top == -1;}
};
#endif
```

```cpp
// file: Stack.cpp
#include “Stack.h”
template<class T>
Stack<T>::Stack(int s)
{
  size = s>0?s:10;
  // stack is initially empty
  top = -1;
  stackPtr = new T[size];
}

template<class T>
bool Stack<T>::push(const T& val)
{
  if(!isFull()) {
    stackPtr[++top] = val;
    return true;
  }
  return false;
}

bool Stack<T>::pop(T& val)
{
  if(!isEmpty()) {
    val = stackPtr[top--];
    return true;
  }
  return false;
}
```

```cpp
// file: TestStack.cpp
#include <iostream>
#include "Stack.h"
using namespace std;

template<class T>
void testStack(
        Stack<T>& theStack, T value, T increment, const char* stackName)
{
  cout<<"\nPushing elements onto "<<stackName<<endl;
  while (theStack.push(value)) {
    cout<<value<<" ";
    value +=increment;
  }
  cout<<"\nStack is full, cannot push "<<value;
  cout<<"\n\nPop elements from "<<stackName<<endl;

  while(theStack.pop(value))
    cout<<value<<" ";

  cout<<"\nStack is empty, Cannot pop\n";
}
```

Compilation failed due to following error(s).

```
main.cpp:(.text+0x16): undefined reference to `Stack::Stack(int)'
main.cpp:(.text+0x27): undefined reference to `Stack::Stack(int)'
/tmp/ccQzIQya.o: In function `void testStack<double>(Stack<double>&, double, double, char const*)':
main.cpp:(.text._Z9testStackIdEvR5StackIT_ES1_S1_PKc[_Z9testStackIdEvR5StackIT_ES1_S1_PKc]+0x58): undefined reference to `Stack::push(double const&)'
main.cpp:(.text._Z9testStackIdEvR5StackIT_ES1_S1_PKc[_Z9testStackIdEvR5StackIT_ES1_S1_PKc]+0xf6): undefined reference to `Stack::pop(double&)'
/tmp/ccQzIQya.o: In function `void testStack<int>(Stack<int>&, int, int, char const*)':
main.cpp:(.text._Z9testStackIiEvR5StackIT_ES1_S1_PKc[_Z9testStackIiEvR5StackIT_ES1_S1_PKc]+0x54): undefined reference to `Stack::push(int const&)'
main.cpp:(.text._Z9testStackIiEvR5StackIT_ES1_S1_PKc[_Z9testStackIiEvR5StackIT_ES1_S1_PKc]+0xde): undefined reference to `Stack::pop(int&)'
collect2: error: ld returned 1 exit status
```

```
// file: TestStack.cpp
#include <iostream>
#include “Stack.h”
#include “Stack.cpp”
using namespace std;

template<class T>
void testStack(
        Stack<T>& theStack, T value, T increment, const char* stackName)
{
  cout<<“\nPushing elements onto “<<stackName<<endl;
  while (theStack.push(value)) {
    cout<<value<<“ “;
    value +=increment;
  }
  cout<<“\nStack is full, cannot push “<<value;
  cout<<“\n\nPop elements from “<<stackName<<endl;

  while(theStack.pop(value))
    cout<<value<<“ “;

  cout<<“\nStack is empty, Cannot pop\n”;
}
int main()
{
  Stack<double> doubleStack(5);
  Stack<int> intStack;
  testStack(doubleStack, 1.1, 1.1, “doubleStack”);
  testStack(intStack,1,1,”intStack”);
}
```

- Consider the following example that consists of five files:
  - **OurMin.h**: contains the declaration of the **min( )** function template.
  - **OurMin.cpp**: contains the definition of the **min( )** function template.
  - **UseMin1.cpp**: attempts to use an **int**-instantiation of **min( )**.
  - **UseMin2.cpp**: attempts to use an **int**-instantiation of **min( )**.
  - **MinMain.cpp**: calls **usemin1( )** and **usemin2( )**.

```
// file: OurMin.h
#ifndef OURMIN_H
#define OURMIN_H
// The declaration of min()
template<typename T> const T& min(const T&, const T&);
#endif // OURMIN_H
```

```
// file: OurMin.cpp
#include "OurMin.h"
// The definition of min()
template<typename T> const T& min(const T& a, const T&
b) { return (a < b) ? a : b;}
```

duplicated definition

min<int,int>

```
//file:UseMin1.cpp {O}
#include <iostream>
#include "OurMin.h"
#include "OurMin.cpp"
void usemin1() {
  std::cout << min(1,2) << std::endl;
}
```

```
//file:UseMin2.cpp {O}
#include <iostream>
#include "OurMin.h"
#include "OurMin.cpp"
void usemin2() {
  std::cout << min(3,4) << std::endl;
} ///:~
```

```
//file:MinMain.cpp
//{L} UseMin1 UseMin2 MinInstances
void usemin1();
void usemin2();

int main() {
  usemin1();
  usemin2();
} ///:~
```

- Disadvantages of the inclusion model:
  - Duplicated definitions:
    - Most compilers can deal with this.
  - All template source code is visible to the client, so there is little opportunity for library vendors to hide their implementation strategies.
  - Header files tend to be much larger than they would be if function bodies were compiled separately.
    - This can increase compile times dramatically over traditional compilation models.

Template Compilation Models

- Explicit instantiation model:
  - You can manually direct the compiler to instantiate any template specializations of your choice.
  - When you use this technique, there must be one and only one such directive for each such specialization; otherwise you might get multiple definition errors.

- Consider the following example that consists of five files:
  - **OurMin.h**: contains the declaration of the **min( )** function template.
  - **OurMin.cpp**: contains the definition of the **min( )** function template.
  - **UseMin1.cpp**: attempts to use an **int**-instantiation of **min( )**.
  - **UseMin2.cpp**: attempts to use a **double**-instantiation of **min( )**.
  - **MinMain.cpp**: calls **usemin1( )** and **usemin2( )**.

A problem!

```
// file: OurMin.h
#ifndef OURMIN_H
#define OURMIN_H
// The declaration of min()
template<typename T> const T& min(const T&, const T&);
#endif // OURMIN_H
```

```
// file: OurMin.cpp
#include "OurMin.h"
// The definition of min()
template<typename T> const T& min(const T&
b) { return (a < b) ? a : b;}
```

Linker errors:

Unresolved external references for

min<int> and min<double>

```
//file:UseMin1.cpp {O}
#include <iostream>
#include "OurMin.h"
void usemin1() {
  std::cout << min(1,2) << std::endl;
}
```

```
//file:UseMin2.cpp {O}
#include <iostream>
#include "OurMin.h"
void usemin2() {
  std::cout << min(3.1,4.2) << std::endl;
} ///:~
```

```
//file:MinMain.cpp
//{L} UseMin1 UseMin2 MinInstances
void usemin1();
void usemin2();

int main() {
  usemin1();
  usemin2();
} ///:~
```

- To solve the “linker errors”, we will need introduce a new file, **MinInstances.cpp**, that explicitly instantiates the needed specializations of **min( )**:

```
// file: MinInstances.cpp {O}
#include "OurMin.cpp"
// Explicit Instantiations for int and double
template const int& min<int>(const int&, const int&);
template const double& min<double>(const double&, const double&);
```

and modify **OurMin.cpp** slightly.

```
// file: OurMin.cpp
#ifndef _OURMIN_CPP_
#define _OURMIN_CPP_
#include "OurMin.h"
// The definition of min()
template<typename T> const T& min(const T& a, const T& b)
  { return (a < b) ? a : b;}
#endif _OURMIN_CPP_
```

Template Compilation Models

- The separation model:
  - Here the function template definitions are separated from their declarations across translation units.
  - This is done by *exporting* templates.
  - The keyword **export** is not supported by many compilers.

The separation model:

```
// file:OurMin2.h
// Declares min as an exported template
#ifndef OURMIN2_H
#define OURMIN2_H
export template<typename T> const T& min(const T&, const T&);
#endif
```

```
// C05:OurMin2.cpp
// The definition of the exported min template
#include "OurMin2.h"
export template<typename T> const T& min(const T& a, const T& b)
{
  return (a < b) ? a : b;
}
```

The translation unit is defined as the code in a file, including header information, but excluding compilation dependent sections such as where we have #ifndef statements.

CSCI251/CSCI851 Spring-2021
Advanced Programming (S7a)

Miscellaneous topics:

Bits and pieces…

Outline

- Enumerations: enums.
  - Scoped.
- Stream manipulators …
- RAII.
- Wrappers.
- Smart pointers.

Enumerations: `enum`

- An `enum` is a means of grouping together integral constants, and each enumeration is a new literal type.
- Classical `enum`s are unscoped, and look like …

```
enum colour {red, green, blue};
```

- C++11 introduces scoped `enum`s, which looks more like …

```
enum class lights {red, green, blue};
```

- The appearance in itself isn’t very helpful, but the idea is the names entered in the braces take on literal values.
- The default values are 0 for the first, and 1 more than the previous for other entries.
- So 0, 1, 2 for the example …

```
enum colour {red, green, blue};

cout << red << endl;
cout << green << endl;
cout << blue << endl;
```

Unscoped generally

- The general unscoped notation is

```
enum typeName {list-of-values}
```

- And once we have done this …

```
enum colour {red, green, blue};
```

- … we can declare and use variables of that type …

```
colour hatColour;
hatColour=blue;
```

- You can set values as well, as in this example from the textbook:

```
enum {floatPrec = 6, doublePrec = 10,
          double_doublePrec = 10};
```

- What if you set some?

```
enum {a = 4, b, c, d};
```

- Or set some equal …

```
enum {a = 4, b, c, d = 4};
```

- Unscoped `enum`s are accessible anywhere, scoped are not.
- The general notation for this …

```
enum class typename {list-of-values};
```

- To access these values we need to use the scope resolution operator, so `typename::value`.
- Let’s look at an example.

- Set up a couple of `enum`s, one unscoped and one scoped.

```
enum colour {red, yellow, green};
enum class peppers {red, yellow, green};
```

- Which of these will work?

```
colour eyes = green;
peppers p1 = green;
colour hair = colour::red;
peppers p2 = peppers::red;
```

- All but the second, where you try to initialise `peppers` with a `colour`.

`enum` type types: C++11

- By default, scoped `enum`s have `int` as the underlying type, but they don't have to…
  - Best illustrated with an example from the textbook …
- `enum intValues : unsigned long long { charType = 255, shortType = 65535, intType = 65535, longType = 4294967295UL, long_longType = 18446744073709551615ULL};`

- Unscoped `enum`s don't have a default type, just something large enough to hold the specified values.
- This makes a difference when we use forward declaration for `enum`s, which is allowed from C++11 on.
  - The underlying size must be specified and since unscoped doesn't have a default we must be explicit for them.

```
enum intValues : unsigned long long;
enum class open_modes;
```

```cpp
// unscoped enum
enum Color : unsigned long long
{
    Red = 2ULL, Green, blue
};

Color c1 = Red;
Color c2 = Color::blue;
int c3 = Color::Green;
std::cout << c3 << std::endl;

//int Red = 1;
//会产生重定义错误
```

```cpp
// scopend enum
enum class Color : unsigned int
{
    Red, Green, blue
};

//Color c1 = Red;
//不能直接使用

Color c2 = Color::blue;

//int c3 = Color::Red;
//不能隐式转换

int Red = 3;
```

Stream manipulators

- Stream manipulators are used to modify streams, so input or output.
- All of you have used at least one existing manipulator:
  - `endl` is a stream manipulator.
- Another existing one is `setw(n)`, a parameterised stream manipulator used to set the width of the stream to `n`.
- But you can write your own as well.

Writing stream manipulators …

- Consider `cout << endl;`
- `basic_ostream` contains the following declaration

```
 basic_ostream<charT,traits>& operator<<
     ( basic_ostream<charT,traits>&
       (*f)(basic_ostream<charT,traits>&) ) {
   return f(*this)
 };
```

- where `f` is a type “pointer to a function with one argument of type `basic_ostream&` that returns type `basic_ostream`”.
- In practice, `f` is a pointer to a manipulator.
  - The method `operator<<` invokes the manipulator to which `f` points.

- So

```
cout << endl;
```

- … is equivalent to

```
cout.operator<<(endl);
```

- And the prototype of the `endl` function is …

```
ostream& endl( ostream& os )
{
    os << '\n';
    return os.flush();
}
```

Rewriting `endl` ...

```
#include <iostream>
using namespace std;

ostream& endl( ostream& os)
{
    os <<"This is my endl !"<<'\n';
    return (os.flush());
}

int main()
{
    cout << endl;
}
```

- We could write a manipulator for outputting currencies, using functionality in the header `iomanip`.

```
ostream& Currency( ostream& os ) {
    os << '$';
    os << setprecision(2);
    os << fixed << setfill('*');
    os.width(12);
    return os;
}
int main() {
    double balance=1023.456;
    cout << Currency << balance << endl;
}
```

Parameterised manipulators …

- We will just look at an example.
- For whatever reason we want a manipulator, `star(n)`, that prints `n` stars.
- Two steps are needed to implement a manipulator with parameters.
  - Call a constructor of the class star to set class data members:

```
star(10)
```

  - Call the operator:

```
operator<<( ostream&, const star& );
```

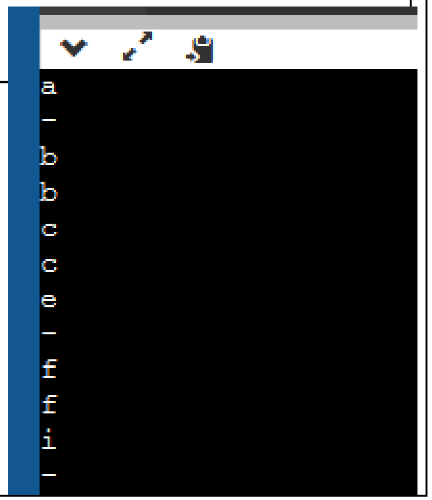
```
class star {
  friend ostream& operator<<(ostream &, const star &);
  private:
      int n;
  public:
      star( int  m ){n=m;}
};

ostream& operator<<(ostream& os, const star& r) {
    for(int i=0; i<r.n; i++)
        os << '*';
    return os;
}

int main() {
    cout << star(10) <<endl;
    return 0;
}
```

Input to output …

- Stream manipulators can be used to transform data.

```
void devowel(istream& in, ostream& out){
char c;
   while (in.get(c))
   {
        if (isVowel(c))
                c='-';
        out.put(c);
   }
}
int main()
{
        devowel(cin, cout);
}
```

```
bool isVowel(char x)
{
        if ( x=='a' || x=='e' || x=='i' || x=='o' || x=='u')
                return 1;
        else
                return 0;
}
```



RAII: Resource Acquisition Is Initialisation

- Less commonly, but more clearly, called Scope-Bound Resource Management (SBRM).
- To quote from

http://en.cppreference.com/w/cpp/language/raii

- *Resource Acquisition Is Initialization* or RAII, is a C++ programming technique which binds the life cycle of a resource that must be acquired before use to the lifetime of an object.
  - allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection—anything that exists in limited supply

- So?
  - Binding the resource to the lifetime of the object before use means that the resources are tidied up when they go out of scope.
  - The resource is freed up correctly when the object is destroyed.
- Memory leaks were probably the earlier instance we looked at in wanting to make sure we appropriately tidy up.
- See https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#e6-use-raii-to-prevent-leaks

… continuing the quote …

- RAII can be summarized as follows:
  - encapsulate each resource into a class, where
    - The constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
    - The destructor releases the resource and never throws exceptions;
  - always use the resource via an instance of a RAII-class that either
    - has automatic storage duration or temporary lifetime itself, or
    - has a lifetime that is bounded by the lifetime of an automatic or temporary object.

- The idea of using the resource via a RAII-class is bundled up with the idea of resource ownership.
- We expect that if object A owns object B, object A managing the lifetime of object B and a user of object A shouldn't be able to directly manage B by making calls like `delete B`, `fclose(B)` ...
- Standard container classes will be RAII compliant.
  - Note though that iterators, or pairs of iterators defining a range, don't own the data elements they reference.

An example with mutexs …

- This example is also from http://en.cppreference.com/w/cpp/language/raii.
- A mutex, short for a mutual exclusion object, is created to make sure that when multiple threads of a program need to access the same resource, such as a file, they don’t do so at the same time.
- Mutexs are used to deal with concurrency problems.

```
std::mutex m;

void bad()
{
    m.lock();                    // acquire the mutex
    f();                         // if f() throws an exception, the mutex is never released
    if(!everything_ok()) return; // early return, the mutex is never released
    m.unlock();                   // if bad() reaches this statement, the mutex is released
}

void good()
{
    std::lock_guard<std::mutex> lk(m); // RAII class: mutex acquisition is initialization
    f();                               // if f() throws an exception, the mutex is released
    if(!everything_ok()) return;       // early return, the mutex is released
}                                      // if good() returns normally, the mutex is released
```

Smart pointer types, in brief

- Smart pointer types:
  - Usually a class template.
  - Behaves syntactically in a similar way to pointers.
  - Has the special member functions, for construction, destruction, moving and copying, defined to maintain certain invariants.
    - Roughly they make sure we don’t run into problem with: Memory leaks, use-after-freeing using, heap corruption via pointer arithmetic (we free the wrong location).
    - They automatically delete the object to which they point.

Standard smart pointers …

- There are two standard smart pointer types from C++11, defined in the `memory` header.
- The first, `std::unique_ptr<T>`, defines a pointer that owns the thing pointed to.
  - So if you call the destructor for the `unique_ptr`, the thing pointed too will be destroyed too.
  - As would seem sensible, at any time there can only be a single `unique_ptr` to a given object.
  - There is a version `std::unique_ptr<T,D>` with D being the defined deleter, defaulting to `std::default_delete<T>`, which calls operator `delete`.

- The RAII idea:

  Whenever we allocate a resource, we initialize a `unique_ptr` to manage it.

- The second is for managing situations where we want multiple references to the same object…

`std::shared_ptr<T>`

- They can be set up using statements like:

```
shared_ptr<string> p1;
shared_ptr<list<int>> p2;
```

- The `shared_ptr` records how many references there are to an object and destroys the object iff the last reference to the object is being destroyed.
  - And in doing so tidies up the memory.