Advanced Software Engineering with C++ Templates

Lecture 2: Classes, Namespaces, and Templates

Thomas Gschwind <thg@zurich.ibm.com>
Agenda

- **Classes**
  - When and why?
  - Declaration
  - Encapsulation
  - Constructors
  - Destructors
  - Conversion
  - References
  - Operators
  - Constant Classes

- **Namespaces**

- **Templates**
  - Polymorphisms
  - Declaration and Use
  - Ambiguities
  - Specialization
  - Classes and Members
  - An Example (pvector)
Classes

- Encapsulation
  - Add structure to the program
  - Explicit interface
    (Access to class checked by compiler)
  - User only needs to know this interface
  - Hide implementation details
- Useful for abstract data types such as stack, vector, map, ...
- User-defined types
  - Allows developers to develop types that behave like primitive types
  - One of the key differences between C++ and C (or Java)

Decide which types you want; provide a full set of operations for each type.

Bjarne Stroustrup
Classes: When are They Needed?

- User defined types
- Abstract data types (e.g., vector, map, ...)
- Whenever you have code that would look in C like this

```cpp
typedef struct {
    int s[256]; int sp;
} Stack;

void stack_init(Stack *s) {
    s->sp=256;
}

void stack_push(Stack *s, int i) {
    s->s[--s->sp]=i;
}

int stack_pop(Stack *s) {
    return s->s[s->sp++];
}

int stack_empty(Stack *s) {
    return s->sp==256;
}

void main(int argc, char *argv[]) {
    Stack s; stack_init(&s); // use s
}
Classes: Declaration & Definition

A new type is defined with struct or class (or typedef as in C)

- Both are declared and defined the same way
- Both may have member functions (methods) that operate on the class
- Hence, a class is a structure and vice-versa (except for the default visibility public (struct) and private (class))

```
struct fraction {
    int c, d;
    void set_counter(int cntr) {
        c = cntr;
        // this->c = cntr;  // same
    }
}

class complex {
public:
    int c, d;
    void set_real(int cntr) {
        c = cntr;
        // this->c = cntr;  // same
    }
}
```

Member functions defined like here are implicitly considered to be inline

this is a pointer and is accesses with ->
Sidebar: **inline**

- Replaces function call with function’s body
  - **inline** functions go into header file
  - Code needs to be known during compile time
- When to use it?
  - Useful for small functions
  - Sometimes also for larger ones
- Faster and more compact code
- For C Developers: Combines advantages of macros and functions
  - Own area for parameters and variables
  - No side effects
  - As efficient as macros (if possible)

```cpp
template <class T>
inline void swap(T &a, T &b) {
    T c(a); a=b; b=c;
}

template <class T>
T gcf(T a, T b) {
    if (a<b) {
        swap(a, b); // replaced with
        // T c(a); a=b; b=c;
    }
    while (b!=0) {
        a=a-b;
        if (a<b) {
            swap(a, b); // replaced with
            // T c(a); a=b; b=c;
        }
    }
    return a;
}
```

Th. Gschwind. Fortgeschrittene Programmierung in C++. 6
Classes: Declaration & Definition

- Members may be defined outside the class definition
- Prepend name with classname::
- Visibility? Yes, we may access private members of another instance of the same type.

```cpp
fraction.h

```class``` fraction { // declaration+definition
```declaration+definition```
int c, d;       // counter and denominator
```end declaration+definition```
```public:```
```declaration+definition```
fraction operator*(fraction b);
```end declaration+definition```
```declaration+definition```
fraction operator/(fraction b) {
    swap(b.c, b.d); return *this/b;
```end declaration+definition```
};
```end class```

```cpp
fraction.cc

fraction fraction::operator*(fraction b) {
```declaration+definition```
int f1=gcf(this->c,b.d),
    f2=gcf(b.c,this->d);
```end declaration+definition```
```declaration+definition```
return fraction((this->c/f1)*(b.c/f2),
```end declaration+definition```
    (this->d/f2)*(b.d/f1));
};
```end class```
Classes: Declaration & Definition

- When the compiler compiles our class
  - Converts member functions to individual functions
  - Adds an additional this parameter
- Just using member functions adds no overhead

```cpp
class fraction {
    int c, d;

public:
    fraction operator*(fraction b); // External View
    fraction operator/(fraction b) {
        swap(b.c, b.d);
        return *this/b;
    }
};

fraction fraction::operator*(fraction b) { ... }

```
Classes: Encapsulation, Access Control

- **public:**
  - Members declared in this section maybe used everywhere (default for structures)

- **private:**
  - Things declared here may only be used by this class
  - Useful for helper methods and attributes (default for classes)

- **protected:**
  - Things declared here may only be used by this class and its subclasses
  (We will discuss inheritance and subclasses in a future lecture)

- **Friends**
  - Something different but somewhat similar to package in Java
Classes: That’s what Friends are for

- Allow other functions and classes access to protected and private members
- Useful for
  - Functions, operators, etc. that logically belong to a class but cannot be defined as member thereof such as:

```cpp
class fraction {
    friend istream &operator>>(istream &is, fraction &f);
    friend ostream &operator<<(ostream &os, fraction f);
    ...
};
```

- Classes having a close relationship to each other

```cpp
class my_matrix {
    friend class my_vector;
    ...
};
```
Classes: That’s what Friends are for...

- Now, we can access the private members
- In this case using `get,set`\_\_\{counter,denumerator\} would have been equally performant and at least equally readable
- `operator>>` should only change the 2\textsuperscript{nd} parameter if the value was read successfully

```cpp
inline void check_char(istream &is, char ch) {
    char c; is >> c; if (c!=ch) { is.putback(c); is.setstate(ios::badbit); }
}

istream &operator>>(istream &is, fraction &f) {
    fraction g;
    check_char('('); is >> g.c;
    check_char('/'); is >> g.d;
    check_char(')'); if (is) f=g;
    return is;
}

ostream &operator<<(ostream &os, fraction f) {
    os << '(' << f.c << '/' << f.d << ')';
    return os;
}
```

Stream States
- good: stream is good
- fail: last operation failed
- bad: characters may have been lost
- eof: reached end-of-file
Classes: Constructor & Destructor

- **Constructor** ($T$)
  - Executed after memory allocated for an object

- **Destructor** ($\sim T$)
  - Executed before memory will be deallocated for an object
  - Similar but better than Java’s finalize() method (because the time it is executed is well known)
  - Allows you to free additional resources (we will come back to this)

```cpp
class Fraction {
public:
    Fraction(...) { // … }  // Constructor
    ~Fraction() { // … }    // Destructor
};
```
Classes: Constructors

- **Default Constructor** ($T()$)
  - Created by the compiler, if not defined by yourself
  - Initializes attributes with default constructor

- **Copy Constructor** ($T(const T&)$)
  - Created by the compiler, if not defined by yourself
  - Copies each attribute from the source to the target object ("shallow copy")
  - This constructor is executed frequently
    - Whenever a parameter is passed by value
    - Whenever a value is returned from a function
      (not if a pointer or reference to the value is returned)

- **C++11 adds another Move Constructor** ($T(T&&)$)
  - Currently, we ignore the move constructor
  - We will come back to this when we look at rvalue references
Each class gets the following artifacts for “free” (that is, if they are not declared, C++ will provide them)

- Default constructor
- Copy constructor
- Destructor
- Assignment operator
Classes: Disable Default Artifacts

- If we do not want the default artifacts, we can block their creation
  - Why would I not want it?
  - E.g., an object should not ever be duplicated by „accident“

- Hide the default constructor
  - Declare in the private part of the class

- What if not even the class itself may use it?
  - Declare it in the class
    => Compiler won’t create one for you
  - But never implement it
    => Linker will complain if you ever use it
  - Or starting with C++11 add “= delete” after its declaration
    \( T(\text{const } T&) = \text{delete} \)
Classes: Rule of Three

- The “Rule of Three” states that if you implement either of the following three, you most likely need all three of them:
  - Copy Constructor
  - Assignment Operator
  - Destructor

- In any case, follow this rule and even if you do not need the others, just provide the same as the compiler-generated code to show that you did not forget about the others.
Classes: Alternatively, always Implement Default Artifacts?

- As alternative to “The Rule of Three”, some developer suggest to always provide all default artifacts.
- I find this overly verbose but probably it is a matter of taste.

```cpp
class fraction { // type declaration+definition
    int c; int d;
public:
    fraction(int cntr=0, int denom=1) : c(cntr), d(denom) { /*void*/ }
    // redundant
    fraction(const fraction &f) : c(f.c), d(f.d) { /*void*/ }
    ~fraction(void) { /*void*/ }
    fraction operator+(fraction b);
    fraction operator-(fraction b);
};
```

This initializes the members of this class in a more efficient way, it makes a difference for user-defined types.

Yes, that’s the same as the compiler-generated one and no we did not forget to implement it (/*void*/).
Classes: User-Defined Conversion Operations

- C++ allows users to define implicit conversion functions
- If, e.g., we want to pass a fraction to a function expecting a double

```cpp
double solve(double p, double q) { // solve x^2+px+q=0
    return -p/2+sqrt(p*p/4-q);
}

void foo(fraction &a, fraction &b) {
    cout << "The result is " << solve(a, b) << endl; // implicit
    cout << "a+b= " << (double)(a+b) << endl; // explicit
}
```

```cpp
class fraction {
public:
    // declare and define implicit conversion function to double
    operator double() { return (double)c/d; }
};
```

Should invoke a user-defined conversion

Style-wise, in C++, one should use static_cast<double>(a+b)
Returning References

- Functions may also return references
- Allows the function to be used as lvalue

```cpp
class fraction {
public:
    ...

    // conversion fraction to double
    operator double() { return (double)c/d; }

    // references as return value
    int &counter() { return c; }
    int &denominator() { return d; }
};
```

```cpp
void normalize(fraction &a) {
    int f = gcf(a.counter(), a.denominator());
    a.counter() /= f;
    a.denominator() /= f;
}
```

However, this may make it harder to change the internal representation of fraction numbers in the future.

- `counter()` and `denominator()` may now be used on the left side of `operator=` (i.e., as lvalue)
Use References with Care

- References have the same characteristics as a pointer
  - The variable a reference point should exist as long as the reference itself
  - Never return a reference to a local variable!
  - BTW, avoid static local variables (they are like global variables)

- References are taken implicitly
  - They more invisible then pointers
  - They may be generated “by accident”

- The Google Coding Style suggests to avoid references as return values and instead to always use pointers
  - Advantage, have to write return &variable;
  - Disadvantage, may not always be intuitive
C/C++ const vs Java final

- **C/C++ const** specifies that a given variable or object a variable points to is constant
- **Java final** specifies that the value of a variable cannot be changed
C/C++ const vs Java final

**C++**

```c++
const int a = 17;
vector<string> v1;
v1.push_back("Hello");
const vector<string> v2;
v2.push_back("Hello");
const vector<string> *vp = &v2;
vp = &v1;
vp->push_back("Hello");
vector<string> *const vq = &v2;
vq = &v1;
vq.push_back("Hello");
vector<string> *vr = &v2;
```

**Java**

```java
final int a = 17;
Vector v1 = new Vector();
v1.add("Hello");
final Vector v2 = new Vector();
v2.add("Hello");
final Vector vp = v2;
vp = v1;
vp.add("Hello");

Vector vr = v2;
```
C/C++ const vs Java final

C++

OK const int a=17;
OK vector<string> v1;
OK v1.push_back("Hello");
OK const vector<string> v2;
    v2.push_back("Hello");
OK const vector<string> *vp=&v2;
OK vp=&v1;
    vp.push_back("Hello");
OK vector<string> *const vq=&v2;
    vq=&v1;
OK vq.push_back("Hello");
    vector<string> *vr=&v2;

C++ (C) distinguish whether the pointer is const or the structure pointed to

Java

final int a=17;
final Vector v1=new Vector();
v1.add("Hello");
final Vector v2=new Vector();
v2.add("Hello");
final Vector vp=v2;
    vp=v1;
    vp.add("Hello");

Vector vr=v2;

Java uses an overly simplistic model that only worries about the reference
C/C++ const (cont’d)

- v2.push_back is NOT OK
- BUT v2.size is OK?
- How can the compiler tell the difference?
- The secret is the declaration of the member...

```cpp
const vector<string> v2;
v2.push_back("Hello");
int i=v2.size();
```

```cpp
... class vector {
public:
    void push_back(T elem);
    int size() const;
}
```
Agenda

- Classes
  - When and why?
  - Declaration
  - Encapsulation
  - Constructors
  - Destructors
  - Conversion
  - References
  - Operators
  - Constant Classes

- Namespaces

- Templates
  - Polymorphisms
  - Declaration and Use
  - Ambiguities
  - Specialization
  - Classes and Members
  - An Example (pvector)
Namespaces

- Modular programming
- Avoid name collisions
- User defined data-types

```cpp
namespace fractions {
    // ...
    int gcf(int a, int b);
    int lcm(int a, int b);
    int swap(int &a, int &b);
    // ...
}
```

Namespace declarations need to be repeated again in the corresponding .cc file.
Namespaces (cont’d)

- Namespaces may be anonymous
  - `namespace /* no name given */ { ... }

- A source file may contain
  - Multiple namespace statements
  - Nested namespace statements

```cpp
namespace util {
  int gcf(int a, int b);
  int lcm(int a, int b);
  int swap(int &a, int &b);
}
namespace fractions {
  class fraction { ... };
}
namespace util2 {
  ...
}
```
Using Namespaces

- **Scoping**
  - `::` Operator
  - **Example:** `util::gcf(foo, bar);`

- **Import a single name of a given namespace**
  - `using` declaration
  - **Example:** `using NAMESPACE::VAR;`

- **Import all names defined in a given namespace**
  - `using directive`
  - **Example:** `using namespace NAMESPACE;`
Operators can be defined in multiple ways and scopes

- As member function:
  ```cpp
  fraction fraction::operator+(complex b);
  ```
- Stand-alone:
  ```cpp
  fraction operator+(fraction a, complex b);
  ```
- As stand-alone operator, in the current namespace, or that of fraction, or that of complex

Now, which one will be used by the compiler?
Multiple operators? (cont’d)

Example

fraction a;
complex b;
cout << a+b << endl;

Solution

1. Does A have a member operator+(b)
   => yes => use it, if it has multiple,
   use standard overloading rules

2. Is there an operator+(a,b) defined in the namespace
   where A or B is defined in (in that order)?
   => yes => use it, if it has multiple,
   use standard overloading rules

fraction a;
complex b;
cout << a+b << endl;

A a;
B b;
cout << a+b << endl;
Agenda

- Classes
  - When and why?
  - Declaration
  - Encapsulation
  - Constructors
  - Destructors
  - Conversion
  - References
  - Operators
  - Constant Classes

- Namespaces

- Templates
  - Polymorphisms
  - Declaration and Use
  - Ambiguities
  - Specialization
  - Classes and Members
  - An Example (pvecto)
Types of Polymorphisms

- „Ad-hoc“
  - Overloading
  - Statically resolved by the compiler (using argument types)

- Dynamic
  - Using virtual member functions
  - Method to be invoked identified during run-time (using the virtual method table)

- Static or Parametric
  - Using templates
  - Function to be invoked identified statically
  - Concrete Functions/Classes are generated for the individual parameter types
Templates: Declaration and Definition

- Allow the use of the same function/class for different types
- Are checked and resolved statically (i.e., during compile time)
  - The definition must be available to the compiler
  - Template declarations and definitions (typically) belong into the header file
  - In some cases it makes sense to put them into the .cc file and pre-allocate them for a predetermined set of types

```
template<class T>
inline T min(T a, T b) {
    return a<b ? a : b;
}
```

This is “old style”, one should use `typename` instead but some people still prefer `class`. If you use an antiquated C++ compiler you may have to use `class`. 
Templates: Use

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

const double pi = 3.141596;

void f() {
  min(2.718282, 1.0);
  min('a', 'z');
  min(1, 26);
  min(pi, 2.718282);
  min('a', 26);
  min(2.718282, 1);
}
```
Template parameters must be unambiguously resolved

Otherwise, the ambiguity needs to be resolved manually
Templates: Resolving Ambiguities

- Unlike for “normal” functions, there is no implicit conversion for templates.

- Explicit
  - If necessary
    \[
    \text{min<int>}'a', 26); \\
    \]
  - Or even if unnecessary
    \[
    \text{min<const double>}'pi', 2.718282); \\
    \]
Mixing Templates and Non-Templates

- Templates and non-templates can be mixed
- Can define a template-based function min
- And define a non template-based function min at the same time
- Non-templates are preferred over templates if no type conversion necessary

```cpp
template<class T>
inline T min(T a, T b) {
    return a<b ? a : b;
}

inline double min(double a, double b) {
    return a<b ? a : b;
}
```
Resolving Ambiguities (cont’d)

- We can create separate helper functions
  - Helper functions may be based on the underlying template

```cpp
inline int min(int x, int y) {
    return min<int>(x, y);
}
inline double min(double x, double y) {
    return min<double>(x, y);
}
```

- This looks tedious ...
min Template – A Problem?

- We have seen it works fine with numbers and characters
- What about C strings?
  ```cpp
cout << min("Hello", "World") << endl;
```
- The above will compare the addresses where the strings are stored
  - It will return the string with the smaller address
- This is not what the typical developer wants ...
- For strings it would be better to use a lexicographical comparison such as strcmp ...
Template Specialization (1st Attempt)

- Templates and non-templates can be mixed
- Define a non template-based function min for C strings

```cpp
#include "min.h"

void foo(char *x, char *y, const char *z) {
    cout << min(x,y) << endl;     // yes
    cout << min(x,z) << endl;     // yes
    cout << min<const char*>(x,z) << endl; // compiles but wrong
}
```

We are asking for the template, so we get the template ...
Template Specialization (2\textsuperscript{nd} Attempt)

- C++ allows us to specialize an existing template for specific types

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

template<> inline char *min<char*>(char *a, char *b) {
    return strcmp(a,b)<0 ? a : b;
}

template<> inline char *min<const char*>(const char *a, const char *b) {
    return strcmp(a,b)<0 ? a : b;
}

#include "min.h"
void foo(char *x, char *y, const char *z) {
    cout << min(x,y) << endl;  // yes
    cout << min(x,z) << endl;  // error
    cout << min<const char*>(x,z) << endl;  // yes
}
```

Compiler error; as we discussed, there is no implicit parameter conversions for templates.
Templates: Classes and Members

- Works exactly the same
- Simply put template `<class T, class U, ...>` in front of the declaration
- It is even OK, to introduce new template parameters for individual member functions
A Persistent `pvector` Class

- We want to implement a persistent version of C++’s vector class
- Reads all elements from a file in the constructor
- Writes all elements back to the file in the destructor

```cpp
template<typename T>
class pvector {
  string filename;
  vector<T> v;
  ...

public:
  pvector(string fname) : filename(fname) { readvector(); }
  ~pvector() { writevector(); }
  void push_back(const T &el) { v.push_back(el); }
  void pop_back() { v.pop_back(); }
  ...
```
A Persistent `pvector` Class (cont’d)

```cpp
template<typename T>
class pvector {
    string filename;
    vector<T> v;

    void readvector() {
        ifstream ifs(filename);
        for(;;) {
            T x; ifs >> x; if(!ifs.good()) break;
            v.push_back(x);
        }
    }

    void writevector() {
        ofstream ofs(filename);
        typename vector<T>::iterator fst=v.begin(), lst=v.end();
        while(fst!=lst) ofs << *fst++ << endl;
    }

    ... 
}
```
A Persistent `pvector` Class (cont’d)

- What happens if we pass the `pvector` around?

```cpp
void foo(pvector<int> pv) {
    if(pv.size()>0) cout << pv[0] << endl;
    pv.push_back(17);
}

int main(int argc, char *argv[]) {
    pvector<int> pv("/tmp/pvector-int.txt");
    foo(pv);
}
```
Summary

- Classes
  - When and why?
  - Declaration
  - Encapsulation
  - Constructors
  - Destructors
  - Conversion
  - References
  - Operators
  - Constant Classes

- Namespaces

- Templates
  - Polymorphisms
  - Declaration and Use
  - Ambiguities
  - Specialization
  - Classes and Members
  - An Example (pvector)
Exercise 1

- Implement the persistent vector data type.
- Experiment with the persistent vector and use it in combination with different data types. What do you observe? Why do you observe that behavior? How can it be changed?
- What happens if we pass the `pvector` around?

- Solve this exercise till the next lecture.
Exercise 2

- Implement a program that shows a function that cannot be inlined.
Exercise 3

- Implement an RPN (Reverse Polish Notation Calculator)
  - When the user enters ‘q’ the program terminates
  - When the user enters ‘n’ followed by a number, it is put on the stack
  - When the user enters ‘d’ the last number is removed from the stack
  - When the user enters ‘+’, ‘-’, ‘*’, ‘/’, the calculator takes the last two numbers from the stack applies the operation to the numbers and puts the result on the stack
  - Use the std::vector container and its iterator!
Exercise 3 – An Example (user input in bold)

Command: \textbf{n 2 n 4 n 3}
1: 2
2: 4
3: 3

Command: \textbf{*}
1: 2
2: 12

Command: \textbf{-}
1: -10

Command: \textbf{n 2 /}
1: -5

Command: \textbf{d}
Command: \textbf{q}

Use getline for reading the commands and an istringstream for parsing the commands and arguments.
Exercise 4

- Convert the RPN calculator to make use of templates, so that it can be used with any number data type (e.g., the fraction data type).
- Add the operation ‘m’ that computes the minimum of the top two numbers using the min function shown in these slides.

Hints
- Don’t forget to create a copy of your old RPN calculator for the 1\textsuperscript{st} colloquium
- Rename the min function to mymin to avoid confusion with std::min
Exercise 5

- Extend the RPN calculator such that it uses the persistent vector for the stack of the numbers.

- Hints
  - Don’t forget to create a copy of your old RPN calculator for the 1st colloquium
Next Lecture

- More on Templates
- More on C++’s Standard Library

Have fun solving the examples!

See you next week!