Advanced Software Engineering with C++ Templates

C++11

Regular Expressions, Constant Expressions, Threads

Thomas Gschwind <thg@zurich.ibm.com>
Bitte bis 4. Dezember ausfüllen...

http://t.uzh.ch/9R7HT
Agenda

- Factories in C++
- C++11 Language Features
  - `nullptr`
  - `static_assert`
  - Constructor changes
  - Initializer lists
  - Range for
- C++11 Standard Library
  - Containers
  - Regular Expressions
  - Smarter Pointers
  - Lambda functions
  - The auto type
  - Constant Expressions
  - Rvalue References
  - Raw strings
  - Threads
  - Tuples
Factories (Virtual Constructor)

Create an object from data describing the object

- Type of object depends on the data itself

- Parse and construct different Shape types sort of like

```
Shape *s=new Shape("S(0,0,2)");
```

```
#include "Shape.h"
#include "ShapeFactory.h"

int main(int argc, char *argv[]) {
    Shape *s1=ShapeFactory::make("R(0,0,3,2)");
    Circle *s2=ShapeFactory::make("C(1.5,1,0.75)");
    s1->show();
    s2->show();
}
```
Factories: Creating the object

```cpp
struct Shape {
    virtual ~Shape() {};
    ...
};

class Rectangle : public Shape {
public:
    static Shape *make(const char *shape);
    ...
};

#include "Shape.h"
#include "ShapeFactory.h"

Shape *Rectangle::make(const char *shape) {
    ...
}
```
Factories: The Factory

typedef Shape* makeType(const char *);

class ShapeFactory {
    static list<makeType*> cl;

public:
    static void add(makeType *m) { cl.push_back(m); }
    static Shape *make(const char *shape) {
        Shape *s=NULL;
        list<makeType*>::const_iterator b=cl.begin(), e=cl.end();
        while (b!=e && (s=(*b)(shape))==NULL) ++b;
        return s;
    }
};

template<class T> struct ShapeFH {
    ShapeFH(makeType *m=(makeType*)&T::make) {
        ShapeFactory::add(m);
    }
};
Factories: Registration of Objects

```cpp
#include "Shape.h"
#include "ShapeFactory.h"

Shape *Rectangle::make(const char *shape) {
    ...
}

static ShapeFH<Rectangle> registerRectangle;
```
Virtual Constructors: Factory (cont’d)

- This design allows us to put the different Shape classes into different source & object files.
- Those object files linked into the program automatically make their shapes available.
Agenda

- Factories in C++
- C++11 Language Features
  - `nullptr`
  - `static_assert`
  - Constructor changes
  - Initializer lists
  - Range `for`
- C++11 Standard Library
  - Containers
  - Regular Expressions
  - Smarter Pointers
  - Lambda functions
  - The auto type
  - Constant Expressions
  - rvalue References
  - Raw strings
  - Threads
  - Tuples
C++11 Introduction

- **History**
  - Last change to C++ was about 10 years ago (C++03)
  - Many features already included in several compilers
  - Final specification in 2011

- **Difficult to change language**
  - Compatibility – new keywords could clash with old code
  - Goal to make it easier for “newbies” and better for experts

- **Keep the original design goals**
  - Generate programs of high performance
  - Versatile
  - Don’t force a specific paradigm
  - Keep zero overhead principle – A feature that is unused must not cost anything

- **Language and libraries are both part of the standard**
  - Java was the “first” language to include the library as part of the standard
nullptr

- New way for saying NULL
- Frequently defined as 0 => problems with ints
static_assert

- Allows to assert that a given condition is met at compile time

- Alternatives
  - `assert`: The classic assert statement is evaluated at run-time (not desirable if condition can be evaluated at compile time)
  - `#error`: Directive by the C++ preprocessor cannot deal properly with template instantiations

```cpp
template<typename T, int N>
class Buffer {
  static_assert(N>16, "Buffer size too small");
  ...
};
```
Constructors

- Constructors may invoke other constructors
  - For instance, for our own String class, we could provide
    ```cpp
    String() : String(NULL) {}
    ```

- Constructors may be derived
  - For instance, in the subclass to derive all constructors from Base, include
    ```cpp
    using Base::Base;
    ```

- Copy constructor may be explicitly removed (same for the assignment operator)
  - Simply declare as
    ```cpp
    pvector(const pvector&) = delete;
    pvector& operator=(const pvector&) = delete;
    ```
Initializer Lists (template<class T> class initializer_list)

- Represents a list of values
- Accept a set of elements in your function or constructor such as in some_function({1, 2, 3, 4, 5, 6})
  - If you don’t like to use a series of push_back calls, initializer lists are for you
- Standard library containers modified to support this

```cpp
#include <initializer_list>

struct MyVec {
    int *rep;
    MyVec(initializer_list<int> seq) : rep(new int[seq.size()]) {
        copy(seq.begin(), seq.end(), rep);
    }
};

MyVec squares({0, 1, 4, 9, 16});
for (int i : {1, 2, 3, 4, 5}) { cout << i << endl; }
```

May leave out parenthesis and write squares{0, 1, 4, 9, 16};
Range For (foreach) Operator

- Copied over from Java (which copied it over from C# (which ...))
- Easier to read than C++'s standard for function

```cpp
void print(const vector<int> &v) {
    for (int i : v) { cout << i << endl; }
    for (int i : {0, 1, 4, 9, 16}) { cout << i << endl; }
}
```

- The object o returned by the expression after : needs to
  - Provide an o.begin() and o.end() member that return an iterator OR
  - A begin(o) and end(o) function returning an iterator must be available

```cpp
int *begin(MyVec &mv) { return mv.rep; }
int *end(MyVec &mv) { return mv.rep+mv.repsz; }
MyVec squares({0, 1, 4, 9, 16});
for (int i : squares) { cout << i << endl; }
```

Yes, begin and end as members of MyVec would be nicer.
Lambda Functions

- Prior to C++11 functions and function objects had to be created
  - These are defined in the global context and not always where needed

```cpp
int cnt;
for_each(v.begin(), v.end(), [](int i) -> void {
    cout << i << endl; // ++cnt;
});
```

- [] captures the environment that visible within the lambda function
  - [&cnt, …] would pass a reference to the local variable cnt
  - [cnt, …] indicates a value
  - [&] and [=] allow to indicate the entire current scope
  - Careful, [=] can generate a lot of copies
  - C++14 adds support for initialization captures such as [mycnt=cnt+1]
New auto “Data Type”

- Compiler will infer the type from the assigned expression
- Type must be uniquely determined during compile-time
- Makes C++ programs easier to write and read (if used correctly)

```cpp
int i=3;
auto j=i; // correct (int)

string s("Hello World");
auto k=cond()? i : s; // incorrect (int or string?)

auto begin=v.begin(); // correct (typename vector<int>::iterator)
vector<int> v=bizarre_function();
```

Could have used auto as well but possibly not more readable. If the return type of bizarre_function() is not obvious, using vector<int> makes the code easier to read. If the code needs a vector<int>, vector<int> will make the compiler error easier to understand, if the return type of the function ever changes.
New auto “Data Type” (cont’d)

- Makes your functions easier to reuse

```cpp
void myswap(int &x, int &y) {
    auto z=x; x=y; y=z;
}
```

```cpp
void printTotalWidth(const vector<Shape> &shapes) {
    if (shapes.size()) {
        auto begin = shapes.begin();
        auto width = begin->getWidth();
        for_each(++begin, shapes.end(), [&width](Shape &s) {
            width+=s.getWidth();
        });
        cout << "width is " << width << endl;
    }
}
```

Thanks to auto, we only need to change the signature, if we want to change swap to use different parameter types.

Thanks to auto the representation of a shape’s width may change but we do not need to update printTotalWidth.

- C++14 also allows the return type of a function to be auto
Constant Expressions

- Certain initializers require a constant expression
  - Arrays stored on the stack (e.g., char buf[256])
  - Switch expressions (same in Java)
  - Functions may not be used in such constant expressions (again, same in Java)

- C++11 allows to use functions in such expressions
  - Must be marked as constexpr
  - Must be possible to evaluate them at compile time

- Useful in combination with static_assert

```cpp
const int min(int a, int b) { return a<b? a : b; }
static_assert(min(consta, constb)>1, "min(consta, constb)>1");
int numbers[min(consta, constb)];
```
Rvalue References

- Remember our String example?

```cpp
int main(int argc, char *argv[]) {
    String a("Hello"), b(" "), c("World"), d("!"), e;
    e = a + d;
    ...
}
```

- The C++ designers realized to be potentially inefficient

- operator+ returns an rvalue but it is clear after the assignment, the value is no longer necessary

- It would be more efficient to just swap the resources

- To solve this problem, rvalue references were introduced (&&

What is invoked here?
- String operator+(...)
- Potentially, the copy constructor
- Assignment operator
Rvalue References (cont’d)

- **String(String &&s)**
- **operator=(String &&s)**
  - Swap the resources with s
  - If not provided, the normal (rvalue) version of the function will be invoked

```cpp
class String {
public:
    String(String &&s) : strg(s.strg), len(s.len) {
        s.strg=NULL;
        s.len=0;
    }
    String &operator=(String &&s) {
        swap(this->strg, s.strg);
        swap(this->len, s.len);
    }
};
```

We swap the resources. This ensures that the other object’s internal state stays valid; eventually its constructor will be invoked. This swapping is important! In the assignment operator, we must not destroy the object and set the other object to NULL.
Rvalue References (cont’d)

- Why is it important that we swap the resources?
- Because this can help us in other situations such as in our swap function

```cpp
template<class T>
void myswap(T &x, T &y) {
    auto z=x; x=y; y=z; }
```

In this code, the assignment operator is invoked. If we want to swap two vectors, this function will copy the vector 3 times.

- In the above case, we assign x to z, which means that x needs to keep a copy of its old value, otherwise, C++ would be in trouble
- In this case, we need to actively tell the compiler to swap resources

```cpp
template<class T>
void myswap(T &x, T &y) {
    // std::move converts its argument to an rvalue
    auto z=std::move(x); x=std::move(y); y=std::move(z); }
```
Rvalue References (cont’d)

- Now, if you did not (yet) catch all the details about rvalue references and how they work, don’t despair

- For more than ten years, C++ developers lived happily without them writing efficient code

- If you are still intrigued to learn more, there is an excellent tutorial available here: http://thbecker.net/articles/rvalue_references/section_01.html
Agenda

- Factories in C++
- C++11 Language Features
  - nullptr
  - static_assert
  - Constructor changes
  - Initializer lists
  - Range for
- C++11 Standard Library
  - Containers
  - Regular Expressions
  - Smarter Pointers
  - Lambda functions
  - The auto type
  - Constant Expressions
  - rvalue References
  - Raw strings
  - Threads
  - Tuples
Containers: hash tables

- Introduced because they are faster than maps for typical applications
- The following hash tables have been added
  - unordered_set
  - unordered_multiset
  - unordered_map
  - unordered_multimap
- Yes the name is not the most beautiful
  - Avoids name clashes with user defined hash_tables
  - Still better than ArrayList ;)


Containers: initializer_lists and emplace

- In C++, populating a container is slightly inefficient
- Containers now support initializer_lists for construction
- Containers support in-situ generation of elements when adding elements to the back (emplace calls)
  - For small data types, not an issue, for big ones, this is interesting

```cpp
vector<fraction> v;
v.push_back(fraction(1,4));    // inefficient, copies the fraction

// emplace a fraction at the end
v.emplace_back(fraction(2,4)); // steals the resources
v.emplace_back(3,4);           // vector creates a fraction at end
```
Regular Expressions: Match strings based on patterns

- “x” followed by any character followed by “y”: x.y
- Zero (One) or more occurrences of “x”: x* (x+)
- “x” possibly followed by y: xy?
- Two, three, or four occurrences of “x”: x{2,4}
- Sequence of one or more alphabetic characters: [a-zA-Z]+
- Match beginning of line: ^
- Match end of line: $
- “x” or “y”: x|y
- Match one or another group: (xy)|(ab)
- Decimal digit, space, letter: \d, \s, \w
C++11 introduced a new type of strings
  • Regular expressions frequently use ‘\’, the escape character in strings
  • Escaping every ‘\’ in a regular expression is tedious
  • Windows share name with string: “\\\\\\w\\\\\\w”

Raw string: R"Delim(cccc)Delim"
  • Signifies the string cccc
  • Delim may be omitted or any other string
  • Delim at the beginning must be the same as Delim at the end
  • Windows share name with raw string: R”(\\w\\w)”

This is interesting in combination with regular expressions
Regular Expressions

- Regular expressions can be created with the regex class
- Regular expressions can be matched with the regex_match function
- The following matches integers such as 1, -2, +567

```cpp
#include <regex>

int main(int argc, char *argv[]) {
    string input;
    regex integer("(\d+|[-+]?[[:digit:]]+)");
    while (true) {
        cout << "Give me an integer! " << endl;
        cin >> input;
        if (input=="q") break;
        if (regex_match(input,integer)) cout << "good" << endl;
        else cout << "bad" << endl;
    }
}
```
Oh, and before you try, use at least g++-4.9, don’t waste your time with g++-4.8. I tried with g++-5, works fine.
Regular Expressions: Looking for Matches

- Looking for matches is done with `regex_search`
  - Results returned in `smatch`
- The example parses us ZIP codes such as CA 12345, CA 12345-1234

```cpp
#include <regex>

int main(int argc, char *argv[]) {
    regex zip(R"(\w{2}\s*\d{5}(-\d{4})?)") ;
    for (string line; getline(cin, line);) {
        smatch matches;
        if (regex_search(line, matches, zip)) {
            cout << "Match: " << matches[0] ;
            if (matches.size() > 1 && matches[1].matched)
                cout << "Submatch: " << matches[1] ;
            cout << endl ;
        }
    }
}
```
Regular Expression Flags

- These are in regex_constants and can be specified as second parameter to regex
- Ignore case: lcase
- No sub expressions in smatch: nosubs
- Optimize for matching speed: optimize
- Consider locale: collation
- Various other expression matching engines: ECMAScript, basic, extended, awk, grep, egrep

- A good overview of the ECMAScript syntax can be found here: http://www.cplusplus.com/reference/regex/ECMAScript/
Smarter Pointers

- **unique_ptr**
  - Same as the pre C++11 `auto_ptr`
  - Uses now the move constructor and hence is more efficient

- **shared_ptr**
  - Provides a pointer with reference counting
  - Like our `smart_pointer`

- **weak_ptr**
  - Useful in combination with `shared_ptr` if cyclic structures are used
  - For details, see the documentation
Threads (Starting and Joining)

- So far, C++ developers had to rely on
  - BOOST thread library, POSIX threads, or other non-standard solutions

- C++ provides a thread class
  - Needs to be passed to a functor (function, function object, lambda function)
  - The new thread is started immediately
  - Threads may be joined with the join() member function

```cpp
#include <thread>

struct printchar {
    char c;
    printchar(char ch) : c(ch) {}
    void operator()() { for (int i=0; i<9999; ++i) cout << c; }
}

int main(int argc, char *argv[]) {
    thread t1(printchar('a')), t2(printchar('b'));
    t1.join(); t2.join();
}
```

Add -pthread to the g++ compiler flags.
Threads (Mutexes)

- Similar to the Java synchronized(object) { ... } mechanism, C++ provides mutexes.
- Instead of changing the language, it is implemented as library (but almost invisible to developers).

```cpp
#include <thread>
mutex mtx;
int cnt=0;
struct printkchar {
    char c;
    printkchar(char ch) : c(ch) {}
    void operator()() {
        for (int i=0; i<9999; ++i) {
            { lock_guard lock(mtx); ++cnt; }
            cout << c;
        }
    }
};
```

C++17 should add a shared_mutex class as well.

The mechanism used here, should already be familiar to many/most of you.
Threads (Async Executions, Futures)

- Asynchronous execution is provided by the async function
- Async is clever enough to know the platform and only executes the function in a separate thread if beneficial on the platform

```cpp
template <typename Iter>
void parsum(Iter begin, Iter end) {
    int sum=0, sz=end-begin;
    if (sz<1000) {
        while (begin!=end) sum+=*begin++;
    } else {
        mid=begin+sz/2;
        auto handle=async([=]{ return parsum(begin,mid); }) ;
        sum+=parsum(mid, end);
        sum+=handle.get();
    }
    return sum;
}
```
Tuples

- C++ always provided a pair\langle T, U \rangle data type
- If more than two elements are needed, C++11 now provides tuples
- For instance, \texttt{tuple<int, int, int, int>}, represents a tuple with 4 integers as elements
  - To create such a tuple, use \texttt{make_tuple(1, 2, 3, 4)}
  - To retrieve an element, use \texttt{get<idx>(mytuple)}
  - This data type uses a lot of template magic for its implementation, we will look in a future lecture into the details of how that works
Summary

- Factories in C++
- C++11 Language Features
  - `nullptr`
  - `static_assert`
  - Constructor changes
  - Initializer lists
  - `Range for`
- C++11 Standard Library
  - Containers
  - Regular Expressions
  - Smarter Pointers
  - Lambda functions
  - The auto type
  - Constant Expressions
  - rvalue References
  - Raw strings
  - Threads
  - Tuples
Exercise 1: Connect 4 with Factory for Computer Player

- You obtained already computer players from your colleagues
- Create those computer players with a factory modelled after the one from the lecture
  - Those students who create a library from all the object files and link the library, link the object files directly with the main program, or otherwise the objects will be ignored as no linker-visible dependencies exist

- (Yes, ideally, the lecturer would have given you a standardized factory so you would only have to link your colleagues computer players BUT then I would spoil you the fun from implementing the factory yourself)
Exercise 2: Spell checker with regular expressions

- Reimplement your spell checker
- Use an sregex_iterator with "(\w+)" as regular expression for parsing full words from the input, i.e., omitting punctuation characters
- Your dictionary file, this time contains regular expressions, for each word you read check each regular expression, whether it matches the word, if none matches, display the incorrectly spelled word on the terminal
- This time it only needs to check words, and display the unknown words on the terminal, no need for replacing incorrect words, etc.
Exercise 3: Connect 4 with Threads

- At the end of a computer player’s move, create one or more threads that computes the computer player’s next move (i.e., while the other player is thinking)

- A trivial implementation would
  - Create 7 threads (one for each possibility of the opponent)
  - When it’s the computer player’s turn again wait for the 7 threads to finish
  - And pick the move suggested by the thread that simulated the other player’s move
Exercise 4: Constructors and Destructors

- In case of inheritance, in which order are constructors and destructors executed?
  - Write a small program that demonstrates your claim
Next Lecture

- C++ Metaprogramming
- C++11 Metaprogramming
- Standard Library: Input & Output

Have a nice weekend, see you next week