Advanced Software Engineering with C++ Templates

Standard Library (IO), C++ Metaprogramming

Thomas Gschwind <thg@zurich.ibm.com>
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
constexpr 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)];
```
Agenda

- Standard Input/Output
  - Structure
  - Output
  - Input
  - Special Streams

- C++ Metaprogramming
  - With templates
  - With constexpr
  - Variadic templates

- Error Handling
  - Stream State
  - Exceptions
  - Formatting
Streams

- Many languages only support built-in types to be read using the standard I/O mechanisms
- C++ programs contain many user-defined types
- Input and output should be easy to use, convenient, and complete
  - Input & output should be extensible
  - User defined input & output routines
  - User definable input & output format
  - Conflicting requirements => No universally satisfactory solution
I/O System

- **ios_base**: locale independent format state
- **basic_ios<>**: locale dependent format state
- **basic_iostream<>**: formatting (<<, >>, etc.)
- **basic_streambuf<>**: buffering
- **locale**: format information
- **character buffer**: real destination/source

Th. Gschwind. Fortgeschrittene Programmierung in C++.
The `ios_base` class is responsible for storing locale independent state information.

The `basic_ios` class stores locale specific format state information.
- The term *locale* typically refers to the country-specific customization information. Which locale to use is specified in the `locale` class.

The `basic_streambuf` class is responsible for buffering characters read from or written to lower level devices and interacts with a character buffer and the raw destination or source.

The `basic_iostream` class provides input and output functionality as well as the functionality to setup and cleanup the I/O subsystem.
I/O Subsystem Initialization

- We have seen previously that library initialization can be accomplished with constructors of helper classes
  - However, the order of their construction is linker-dependent
  - The I/O Subsystem should be initialized before all other modules
  - How can we ensure its initialization independently of the link-order?
    - We ensure that the initialization is performed in all object files.
    - This is where anonymous namespaces are useful

```cpp
class ios_base::Init {
    static int count;

public:
    Init();
    ~Init();
};

namespace { ios_base::Init __ioinit; }
```
basic_ostream<>

```cpp
template <class Ch, class Tr=char_traits<Ch> >
class basic_ostream: virtual public basic_ios<Ch,Tr> {!
public:
    virtual ~basic_ostream();
    // ...
}
```

- An ostream class converts arbitrary types into a sequence of characters
- Can be specialized for a specific type of characters
  - Specialization using char_traits

<table>
<thead>
<tr>
<th>Standard Output Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>ostream cout;</td>
</tr>
<tr>
<td>wostream wcout;</td>
</tr>
<tr>
<td>ostream cerr;</td>
</tr>
<tr>
<td>wostream wcerr;</td>
</tr>
<tr>
<td>ostream clog;</td>
</tr>
<tr>
<td>wostream wclog;</td>
</tr>
</tbody>
</table>
basic_ostream (cont’d)

- `operator<<` is defined for built-in types
  - Is executed from left to right (left-associative)
  - Returns a reference to `ostream`
  - Does not have to be defined as member function
  - => basic_ostream does not have to be modified

```cpp
cout << "Hello World!" << endl;
(cout << "Hello World!" ) << endl;

operator<<(cout, "Hello World!");
operator<<(cout, endl);

operator<<( 
    operator<<(cout, "Hello World!"),
    endl);
```
basic_istream<>

- Similar to basic_ostream
- `operator>>` for built-in types
  - Ignores white space (as defined by `isspace()`)
  - Input types must be known
- `get`, `getline`
  - Do not ignore white space
  - Size of the input buffer must be specified
**basic_istream<> (cont’d)**

```cpp
template <class Ch, class Tr=char_traits<Ch> >
class basic_istream: virtual public basic_ios<Ch,Tr> { 
  public:
    // ...

    streamsize gcount() const;
    int_type get();
    basic_istream& get(Ch &c);
    basic_istream& get(Ch *p, streamsize n);
    basic_istream& get(Ch *p, streamsize n, Ch term);
    basic_istream& getline(Ch *p, streamsize n);
    basic_istream& getline(Ch *p, streamsize n, Ch term);
    basic_istream& ignore(streamsize n=1, int_type t=Tr::eof());
    basic_istream& read(Ch *p, streamsize n);

    // ...
};
```

The `get` members include the term character in the result, whereas the `getline` members do not.
User-Defined Input & Output (cont’d)

### Input operator
- Only modify the target variable if the data was read successfully
- Try to keep the input stream in an untouched state if the data could not be read successfully
- If there are characters possibly lost on the input stream, set the bad bit

### Output operator is straightforward
Virtual Input and Output Functions?

- Input and output operators of the Standard Library are not virtual
  - Consistent with user defined output functions that are not virtual either
  - Virtual are only some of the character buffer management functions
  - This design accomplishes optimal speed

- What if you need a virtual output function?
  - Put a virtual output (or formatting) function into your base class
Error Handling & Stream States

- good
  As the name suggests, all is fine

- fail
  The last operation has failed but no characters have been lost

- bad
  The last operation failed, characters may have been lost

- eof
  end of file
Error Handling & Stream States (cont’d)

- Check the stream after every input and output operation
- Raise an exception as soon as an error occurs

```cpp
template <class Ch, class Tr=char_traits<Ch> >
class basic_ios: public ios_base {
public:
  bool good() const;  // Stream is in state good
  bool eof() const;  // and shorthands for good and fail.
  bool fail() const;  // Stream is in state
  bool bad() const;  // good, fail, eof, bad and
  operator void*() const;  // shorthands for good
  bool operator!() const { return fail(); }  // and fail.
  iostate rdstate() const;
  void clear(iostate f=goodbit);
  void setstate(iostate f) { clear(rdstate()|f); }

  class failure;
  iostate exceptions() const;
  void exceptions(iostate except);
```

Read and set raw iostate bits.

Get and set states for which exceptions are thrown.
Error Handling with Exceptions

- By default errors are handled without exceptions
- The exceptions members allow to change this

```c++
int foo(istream &is, ostream &os) {
  ios::iostate s=is.exceptions();  // read and store old state
  is.exceptions(ios::failbit);     // throw exception for failbit

do {
  try {
    // some really fancy code here
  } catch (ios_base::failure) { /* exception handling */ }
} while(cin);
  is.exceptions(s);         // Possible source of error!
}
```
Error Handling with Exceptions

- **Cleanup using the destructor**
  - Ensures that the original exception state is restored
  - For instance, in case of an early return or an uncaught exception

```cpp
class set_exceptions {
    ios &s; ios::iostate s_state;

public:
    set_exceptions(ios &stream, ios::iostate e) : s(stream)
    { s_state=s.exceptions(); s.exceptions(e); }
    ~set_exceptions() { s.exceptions(s_state); }
};

int foo(istream &is, ostream &os) {
    set_exceptions finalizer(is, ios::failbit);
    do {
        // some really fancy code here
    } while(cin);
}
```
Formatting `(ios_base)`

C++ supports many different output formatting options

- **Floating point precision**
  - `precision() // get`
  - `precision(n) // set (& get) old state`

- **Width of output for the next numerical output and setting the fill character**
  - `width(n)`
  - `fill(c)`

- **Manipulators**

- **Flush the output buffer**
  - `flush()`

- **Raw Output Flag Manipulation**
  - `fmtflags flags() const;`
  - `fmtflags flags(fmtflags flags);`
  - `fmtflags setf(fmtflags flags);`
  - `fmtflags setf(fmtflags flags, fmtflags mask);`
  - `fmtflags unsetf(fmtflags mask);`

- **Formatflags fmtflags**
  - `skipws, boolalpha,`
  - `dec, hex, oct,`
  - `left, right, internal, …`
Sidebar: bitfields

- Multiple state flags are frequently not stored as a series of bool fields but as some integer type whose bits represent the flags
  - Multiple flags can be changed at the same time
  - Memory is used more efficiently

- This also applies to more complex states such as whether to display numbers in dec, hex, or oct notation which does not fit into a single bit (here, masks are used)

```cpp
// 0 binary flags

bit flags
1 skipws (...0001)
0 boolalpha (...0010)
0 dec (...0000), hex (...0100),
0 oct (...1000)

fmtflags setf(fmtflags flags) {
  myflags = myflags|flags; ... 
}

fmtflags setf(fmtflags flags, fmtflags mask) {
  myflags = myflags&~mask |
  flags&mask; ... 
}

setf(skipws);       // set skipws flag
setf(hex, basefield); // set basefield “wide” flag
```
Let us write a program to convert octal to hexadecimal values
We set most options in the main routine

```cpp
#include <iostream>
#include <algorithm>
#include <iterator>

// ... set_exceptions class from previous slide, converter, etc.

int main(int argc, char *argv[]) {
    cin.setf(ios::oct, ios::basefield);
    cout.setf(ios::hex, ios::basefield);
    cout.setf(ios::right, ios::adjustfield);
    cout.fill('0');

    set_exceptions finalizer(cin, ios::badbit);
    try {
        clog << converter(cin, cout) << " numbers converted"
    } catch(exception) { exit(1); }
}
```
Number Converter (**converter**)

- The converter function performs the actual conversion
- Sets some local flags such as the output width

```cpp
int converter(istream &is, ostream &os) {
    int i=0, n=0;
    set_exceptions finalizer(is, ios::failbit);
    do {
        try {
            while (cin >> i) {
                cout.width(8); cout << i << endl; ++n; /* ... */
            }
        } catch (ios_base::failure) { /* command handling such as termination */ }
    } while (cin);
}
```
Manipulators

- Always using setf can be cumbersome
  - For instance if a number needs to be displayed in oct, dec, and hex at the same time
- Ideal would be something like
  ```cpp
  cout << oct << i << dec << i ...;
  ```
- And this functionality is provided by the standard manipulators...
Standard Manipulators

- flush
- (no)skipws, (no)boolalpha, (no)showbase
- (no)showpos, (no)showpoint, (no)uppercase
- internal, left, right
- dec, hex, oct
- fixed, scientific
- setbase(n), setfill(c), setprecision(n)

```c
int main(int argc, char *argv[]) {
    int i = atoi(argv[1]);
    cout << "dec:" << dec << i << " hex:" << hex << i << endl;
}
```

Th. Gschwind. Fortgeschrittene Programmierung in C++.
Using those manipulators is trivial, but how are they implemented?

```cpp
template <class Ch, class Tr=char_traits<Ch> >
basic_ostream<Ch,Tr>& oct(basic_ostream<Ch,Tr> &s) {
    s.setf(ios_base::oct, ios_base::basefield);
    return s;
}

template <class Ch, class Tr=char_traits<Ch> >
class basic_ostream: virtual public basic_ios<Ch,Tr> {
public:
    basic_ostream& operator<<(basic_ostream &(*f)(basic_ostream&)) {
        return f(*this);
    }
    basic_ostream& operator<<(basic_ios<Ch,Tr> &(*f)(basic_ios<Ch,Tr>&)) {
        f(*this); return *this;
    }
}
```
User-Defined I/O Manipulators

- Definition of your own function of type
  `ios_base& my_manip(basic_ios&);`
- Definition of your own class containing formatting information
- Definition of the `operator<<` or `operator>>` for said class that applies the formatting instructions
User-Defined I/O Manipulators with State

- Using functions to change input output behavior provides limited functionality since they cannot store state or configuration options.
- Another option is to define a class such as `Form` and define an output operator for `Form`.

```cpp
struct Form {
    int prc, wdt, fmt, ...;
    Form(int p=6) : prc(p) { fmt=wdt=0; }
    Form& scientific() { fmt=ios_base::scientific; return *this; }
    Form& precision(int p) { prc=p; return *this; }
};

ostream& operator<<(ostream& os, const Form &f) {
    os.setf(f.fmt, ios_base::floatfield);
    // ...change other output settings...
    return os;
}
```
User-Defined I/O Manipulators with State (cont’d)

- This manipulator can be used like any other manipulator
- One downside, once applied, the settings take affect for all future output operations

```cpp
Form gen4(4);

void f(double d) {
    Form sci8=gen4;
    sci8.scientific().setprecision(8);
    cout << sci8 << d << endl;
    // from now on, all output will use the format defined by sci8
    // (until changed again)
}
```
The stream buffer

- Each stream has an underlying stream buffer
  - A stream buffer represents a series of characters
  - Can be obtained with rdbuf()
  - Can be obtained and set with rdbuf(...)

```cpp
ostringstream oss;
streambuf *orig_streambuf = cout.rdbuf(oss.rdbuf));
// from now, all data to cout is actually written to oss
// now we can call functions that write to cout and capture their output
// when we are done, we revert to cout's original buffer
cout.rdbuf(orig_streambuf);
```
The output stream buffer

- The output buffer is defined by three pointers
  - `pbase` points to the beginning of the buffer
  - `pptr` points to where the next character is put
  - `epptr` points one beyond the end of the buffer

- If we implement a streambuf (inheriting `std::streambuf`), we can set these pointers (`setp`) but they are managed by `std::streambuf`
  - We can (and should) override `overflow` which is called when the buffer is full

```cpp
class encbuf : public streambuf {
    char buf[256];
    ostream *os;
    streambuf *basebuf;
    int off;

public:
    encbuf(ostream &os, int off=13) : os(&os), basebuf(os.rdbuf(this)), off(off) {
        setp(buf, buf+sizeof(buf));
    }
    virtual ~encbuf() { if (os!=NULL) owner->rdbuf(basebuf); }
}
```
class encbuf : public streambuf {
    ...

// no more characters fit into buffer => sync to sink
virtual encbuf::int_type overflow(int_type ch) {
    if (ch!=traits_type::eof() && *os && sync()==0) return sputc(ch);
    return traits_type::eof();
}

// sync buffer to sink
virtual encbuf::int_type sync(void) {
    if (pbase()==pptr()) return 0;
    for (char *p=pbase(); p<pptr(); ++p) {
        char ch = *p;
        if ('a'<=ch && ch<='z') *p = 'a' + (ch-'a'+13)%26;
        else if ('A'<=ch && ch<='Z') *p = 'A' + (ch-'A'+13)%26;
    }
    streamsize n=basebuf->sputn(pbase(), pptr()-pbase());
    setp(buf, buf+sizeof(buf));
    return n ? 0 : -1;
}
};

How is it used?
encbuf(encrypt) cout;
From now until end of scope, cout is encrypted.
The input stream buffer

The input buffer is again defined by three pointers

- `eback` points to the beginning of the buffer
- `gptr` points to where the next character is read from
- `egptr` points one beyond the end of the buffer

If we implement a streambuf (inheriting `std::streambuf`), we can set these pointers (setg) but they are managed by `std::streambuf`

- We can (and should) override underflow which is called when the buffer is empty
- Additionally we need to ensure that there is always enough space in the get buffer such that at least one character may be put back
Agenda

- Standard Input/Output
  - Structure
  - Output
  - Input
  - Special Streams

- C++ Metaprogramming
  - With templates
  - With constexpr
  - Variadic templates

- Error Handling
  - Stream State
  - Exceptions
  - Formatting
Meta-Programming with Templates

What does this code do? Where would it be useful?

template<int P, int N> struct compute2 {
    static const int res=compute2<P,P%N?N-1:0>::res;
};
template<int P> struct compute2<P,1> {
    static const int res=1; }
-template<int P> struct compute2<P,0> {
    static const int res=0; }

-template<int N> struct compute {
    static const int res=compute2<N,N-1>::res; }

int main() {
    cout << compute<3>::res <<"," << compute<4>::res <<"," << compute<5>::res << endl;
}
Yes, the code checks whether the number is a prime number

```cpp
template<int P, int N> struct isprime2 {
    static const int res=isprime2<P,P%N?N-1:0>::res;
};
template<int P> struct isprime2<P,1> {
    static const int res=1; }

template<int P> struct isprime2<P,0> {
    static const int res=0; }

template<int N> struct isprime {
    static const int res=isprime2<N,N-1>::res;
};

int main() {
    cout << isprime<3>::res << "," << isprime<4>::res << "," 
    << isprime<5>::res << endl;
}
Meta-Programming with Templates

- Where is the previous code useful?
- If we need somewhere a prime if we add a template to compute the next prime

```cpp
template<typename T> class my_hash_table {
    T table[compute_next_prime<20000>::res];
    ...
};
```
Meta-Programming with constexpr in C++11

- C++11 makes our life easier (and more complicated again)
- Meta-programming with constexpr

```cpp
constexpr bool isprime2(int i, int n) {
    return (n%i==0) ? false :
                (i*i<n) ? isprime2(i+2,n) : true;
}
constexpr bool isprime(int n) {
    return (n%2==0) ? (n==2) : isprime2(3,n);
}
constexpr int nextprime(int i) {
    return isprime(i) ? i : nextprime(i+1);
}
int main(int argc, char *argv[]) {
    constexpr int res=nextprime(1234567890);
    cout << res << endl;
}
```
Variadic Templates

Wouldn’t it be nice to have printf that supports user-defined types?

Let’s implement a simplified printf function

- Arguments just specified as % (not %d, %g, etc)
- % represented as %%
- Our printf identifies the type automatically
- Supports user-defined types
Variadic Templates (cont’d)

```cpp
void mprintf(const char *s) {
  if (s==nullptr) return;
  while (*s) {
    if (*s=='%' && ++s!='%') throw error("missing argument");
    cout << *s++;
  }
}

template<typename T, typename... Args>
void mprintf(const char *s, T value, Args... args) {
  if (s==nullptr) throw error("too many arguments");
  while (*s) {
    if (*s=='%' && +++s!='%') {
      cout << value; mprintf(s, args...);
      return;
    }
    cout << *s++;
  }
  throw error("too many arguments");
}
```

Th. Gschwind. Fortgeschrittene Programmierung in C++.
Summary

- Standard Input/Output
  - Structure
  - Output
  - Input
  - Special Streams

- C++ Metaprogramming
  - With templates
  - With constexpr
  - Variadic templates

- Error Handling
  - Stream State
  - Exceptions
  - Formatting
Exercise 1: Changing format for a single output operation

- It would be nice if we could change format for only a single value to be output such as shown by the following snippet

```cpp
Form gen4(4);

void f(double d) {
    Form sci8=gen4;
    sci8.scientific().setprecision(8);
    cout << gen4(d) << " back to old options: " << d << endl;
}
```

- `gen4` should become some “kind of” operator that applies the output parameters to only that single output operation

- Make use of this new manipulator in your RPN example
Exercise 2: Output formatting through streambuf

- Write a streambuf that nicely indents the output
  - When a ( or { or [ is encountered the following lines are indented by \(n\) spaces
  - When a ) or } or ] is encountered the following lines are outdented by \(n\) spaces
  - When a string is encountered enclosed by “ or ’, characters until the string terminates with the same characters “ or ’ are ignored
  - Whitespaces at the beginning of a line are ignored (since you are handling the indenting yourself)
  - To make this exercise easier, you do not need to maintain your own buffer (isetp); if you do not specify a buffer, for every character the overflow function will be called

- Use your streambuf on some sample C, C++, Java, JSON, ... code
  - You may slightly change the exercise if you have a specific need in mind (explain)
Next Lecture

- Concept Classes
- Expression Templates
- Repetitorium

Have a nice weekend, see you in two weeks.