Part 5: Generic Programming

Contents

[DOCUMENT NOT FINALIZED YET]

Introduction

Code is often independent of actual types. E.g.

- Sorting routines (C's qsort)
- Containers (vectors, lists, sets, maps,...)

Generic programming:

Implement once and reuse code for arbitrary types

Benefit: code is easier to maintain!

C way: use `void*` as generic pointer type and pass function pointers

C++ way: function templates, class templates, and functors

Example: Sorting Arrays

C:

```
          start address   #elements    #bytes per element
                 |            |              |
                 |            |              |
void qsort(void *base, size_t number, size_t size,

             int(*compar)(void *, void *));
                 |
                 |
pointer to function that compares two keys via generic pointers


E.g.: Foo a[N]; ... qsort(a, N, sizeof(a[0]), Foo_compare);
```

C++: `sort(v.begin(), v.end(), Compare());`

Shorter and faster because compiler can inline code, such as simple integer comparisons for which CPUs have dedicated machine instructions

We will soon see how this works

Function Templates

```
int min(int a, int b) { return a < b ? a : b; }
float min(float a, float b) { return a < b ? a : b; }
...
```

In C++, there is no need for defining long lists of (almost) identical functions!

The following generic definition covers (almost) all:

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

Function `min` is now paramerized by type `T`

The compiler generates implementations for actual type instances when the function is used

Note: function templates usually make some assumptions about type parameters

For instance, `min` assumes that operator < is defined for values of type `T`. If this is not the case, you'll see a compiler error message

Examples

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

template <typename T>
void swap(T &a, T &b)
{
  T temp(a); a = b; b = temp;
}

int main()
{
  int a = 10, b = 5, c = max(a,b); // calls max<int,int>
  float e = 2.0, f = 1.0, g = max(e,f); // max<float,float>

  swap(a, b);  // swap<int, int>
  swap(e, f);  // swap<float, float>
}
```

swap assumes that type T can be copy contructed and assigned

If this is not the case (because maybe the author of class T was lazy and made T's AO private), the compiler will complain

More Examples

```
template <typename T>
T max(T a, T b);
// OK, forward declaration; eventually,
//code needs to be provided in header file

template <typename T>
void swap(T &a, T &b);
// OK, forward declaration; eventually,
// code needs to be provided in header file

template <typename U, typename V>
U foo(U a, V b)
{
  return a;
}
// OK, class/typename are synonyms
// U,V appear in function signature

template <typename U, V>
U bar(V a);
// ERROR: no typename/class in front of V
// PROBLEM: when calling bar(x) compiler cannot
// infer type U =>  need to write bar<int>(a)
//                  compiler infers U=int
```

Function Template Instantiation

- Function templates specify how individual functions can be constructed given a set of actual types (instantiation)
- This happens as side-effect of either invoking or taking the address of a function template
- Compiler and/or the linker has to remove multiple identical instantiations
- Template instantiation may be slow – dumb compilers repeat compilation

Type Parameter Binding

```
template <typename T>
T max(const T* a, int size)
{
  ...
}

float a[100], x; ... x = max(a, 100);

// formal parameter: const T *a  -> T *a
// actual parameter: float *a    => T = float
```

1. Each formal argument of the function template is examined for the presence of formal type parameters

2. If a formal type parameter is found, the type of the corresponding actual argument is determined

3. The types of the formal and actual argument are matched. Type qualifiers reference, const, and volatile are ignored. No non-trivial type conversions take place. Safer $\rightarrow$ Good!

Function Templates vs. Macros

- Function templates are type-aware, macros are not
- Function templates can be specialized easily, macros cannot
- Inlined function templates are as fast as macro-generated code
- No "unwanted" side-effects with function templates Macro parameters may be evaluate more than once which can cause trouble:

E.g., `min(x++, y++)`

```
#define min(a,b) ((a) < (b) ? (a) : (b))

// vs.

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

```
#define array_size_macro(A) (sizeof(A)/sizeof(A[0]))

// vs.

template <typename T, std::size_t n>
constexpr std::size_t array_size(const T (&A)[n])
// note that we pass a reference to an array
// otherwise, size information is lost as arrays
// are then treated as pointers
{
  return n;
}

...
int A[10];
cout << array_size(A) << endl; // 10
int B[array_size(A)]; // works

int *i;
cout << array_size_macro(i) << endl; // shouldn't!
```

$\leadsto$ In C++ macros are not that useful anymore and should be avoided if possible

Class Templates

Overview

- Reuse type independent code
- Classes are parameterized by types

Simplest syntax:

```
template <typename T>
class X
{
... T can be used here ...
};
```

int, bool values can also be used as template parameters

Class templates are very useful for container types, such as vector, set, list, whose code does not depend on the type of values stored in them

Class template instantiation is explicit. E.g.:

```
  vector<int> a; // a is a vector of integers
```

whereas instantiation of function templates is usually implicit — the compiler infers types — but can also be explicit: `min<int>(a, b)`

The compiler instantiates class templates on demand

In this process, formal type parameters get replaced by actual parameters

This also means: class tmplate methods must be implemented in header files, because the compiler needs to know the code to generate when instantiating class templates

Advanced application: compile-time type reflection and compile-time computations (later)

Stack Class Template

```
template <typename T>
class Stack
{
public:
  Stack();
  ~Stack();

  // Stack is a container class =>
  // method code independent of T
  void push_back(T &v) {...} // append element
  T    &back() {...}  // last element (if !empty)
  void pop_back() {...}      // remove last
  bool empty() const {...}   // true iff empty

private:
  T *p; // implemented using dynamic array
  int size;
};

Stack<int>   si;  // int stack
Stack<float> sf;  // float stack
si.push_back(5);
cout << si.back();
if (si.empty()) exit(0);
sf.push_back(3.5); ...
```

When the compiler sees `Stack<int> si;` it compiles class `Stack` for `T=int`. From then on, we can create and manipulate `int`-stacks without subsequent compilations of `Stack<int>`

Likewise, when the compiler sees `Stack<float> sf;` later, it compiles class `Stack` for `T=float`, and so on

So, our life as programmers just got simpler: similar to function templates, we only have to maintain one generic class definition. The compiler adapts it to various element types `T` for us on demand

Because class templates may be instantiated for multiple types on demand in each compilation unit (.cpp file), the compiler has more work to do and the size of object code it generates can be much bigger compared to regular template-less programs

It is the job of the linker, which combines all object files to create the final executable file, to remove all but one instance of identical template instantiations

E.g., if we use `stack<int>` in file `Foo.cpp` and `Bar.cpp` the linker must not complain about multiple definitions of stack class methods, but silently drop one set of functions it generated code for twice

Vector Class Template

```
template <typename T>
class Vector
{
public:
  Vector(int size=0);
  ~Vector();
  ...
  T &operator[](int i) {
    check(i);
    return p[i];
  }
  // const version (chosen when called on const Vector)
  const T &operator[](int i) const {
    check(i);
    return p[i];
  }

private:
  // check index
  void check(int i) const { assert(0 <= i && i < size); }
  T *p; // implemented using dynamic array
  int size;
};

Vector<int> vi(10);         // 10 ints
Vector<const char*> vs(20); // 20 C-strings
vi[0] = 10;
vs[1] = "text";
```

Pair Class Template

```
template <typename T>
struct Pair
{
  Pair(T v1, T v2); // implementations can be deferred
  T first, second;
};

// implementation can be provided outside class template
// definition but still in header file!
// Improves readability
template <typename T>
Pair<T>::Pair(T v1, T v2)
 : first(v1), second(v2)
{ ...
}

//... in main():
Pair<int> pi(0, 0);      // pair of ints
Pair<float> pf(0.0, 2); // pair of floats
// pair of int pairs
Pair<Pair<int>> pp(Pair<int>(0, 2), Pair<int>(3, 1));
pi = pp.first;
```

Tricky Bits

Dependent Names

In class templates sometimes we have to clarify whether an identifier is a type or not

```
template <typename T>
class X
{
  typename T::SubType *ptr;
};
```

Without "typename" `SubType` would be considered a static member of type `T`! Thus, the compiler would view

```
T::SubType * ptr
```

as a multiplication!

When a name depends on a template type you need to use `typename`

Using this

In class templates with base classes, using a name `x` by itself is not always equivalent to `this->x`, even though a member `x` is inherited

```
template <typename T>
class X
{
public:
  void bar();
};

template <typename T>
class Y : public X<T>
{
public:
  void foo() { bar(); }
  // call global bar() if exists, error otherwise
};
```

Solution:

`void foo() { this->bar(); }`

or

`void foo() { X<T>::bar(); }`

Explicit Call of Default Constructor

POD types such as int, char, double have no default constructor

To ensure that template variables of such types are initialized we must call constructors explicitely:

```
template <typename T>
void foo()
{
  T x = T(); // POD is initialized with 0
             // T x; leaves x unitialized for
             // POD types T...
  // ...
}
```

```
template <typename T>
class X
{
public:
  X() : a() {} // this also works for POD T

private:
  T a;
};
```

Above initialization works for all POD types, including structs, not just for class template variables, but also for arrays:

```
int *p = new int[100]();
```

initializes all values with 0

Template Specializations

Suppose the general implementation of a class template can be improved for specific types

E.g. `vector<bool>`

bools occupy one byte each, a waste of 7 bits!

A customized `vector<bool>` class template could store 8 bits in a byte instead

Template specialization does the trick:

```
// (A) general form
template <typename T> class X { ... };
// (B) specialized form
template <> class X<bool> { ... };
// (C) specialized form
template <> class X<int>  { ... };

X<float> x;  // instantiates (A)
X<bool> x;   // instantiates (B)
X<int> x;    // instantiates (C)
```

Adapts class templates to special needs. There is no relation between above implementations — except for the same class template name X!

→ different code for different types

Flashback: Function Templates

There, too, we sometimes need specialization

Consider this min template:

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

Does it work for `T = char *` ?

```
const char *foo = "foo", *bar = "bar";
cout << min(foo, bar) << endl;
```

The code compiles, but will do something unexpected ...

The result is kind of random, because < applied to pointers returns true iff the first pointer points to a lower address in memory than the second

Oops. Our intention was to print the lexicographically smaller string

Remedy: function template specialization

You can either write:

```
template <>
const char *min<const char*>(
  const char *a, const char *b
)
{
  // compare the two C-strings lexicographically
  return strcmp(a, b) < 0 ? a : b;
}
```

or simply (preferred, see lec12/temporder?.c)

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

Specialized functions take precedence over templated versions in the process of the compiler searching for the function to call

Partial Class Template Specialization

```
// (A) general case
template <typename T>
class List { ... };

// (B) special case where we handle pointers
// to types: share void* implementation and
// use pointer casts ...
template <typename T>
class List<T*> {
  List<void*> impl;  // infinite recursion?!
  ...
  T* get(int i) const {
    return reinterpret_cast<T*>(impl.get(i));
  }
};

// (C) if we don't provide this, we run into an
// infinite recursion at compile-time:
template <> class List<void*> { ... };

List<int>   x;  // instantiates (A)
List<int*>  y;  // instantiates (B)
List<void*> z;  // instantiates (C)
```

Compiler looks for best (= most specialized) type match

What is the advantage?

For pointer types we only fully instantiate `List<void*>` once

All other `List<T*>` classes are just wrappers which delegate method calls to `impl`

Partial class template specialization adapts class templates even more

Compiler chooses the most specialized match

More details in “C++ Templates” by Vandevoorde and Josuttis

Type Traits

We sometimes want to know something about types at compile-time, to create faster code, for instance

For this, we need to infer type properties at compile-time, a process known as compile-time type reflection

For example, POD types can be copied with memcpy, but others can't

A smart copy method could take advantage of this fact

Refresher: The code that follows makes use of enumeration types, which define integer constants in the program

```
// named integer constants, start with 0 and increment
enum { APPLE, ORANGE, PLUM };

struct X {
  // can evaluate constant expressions at compile-time
  enum { value = 5, foo = value + 1 };
};

cout << APPLE << " " << ORANGE << " " << PLUM << endl;
// 0 1 2
cout << X::value << " " << X::foo << endl;
// 5 6
```

With this and partial class template specialization we can find out whether a type is a pointer:

```
template <typename T>
struct TypeTraits
{
  // general case
  template <typename U>
  struct PointerTraits
  {
    enum { value = 0 }; // no pointer
  };

  // special case
  template <typename U>
  struct PointerTraits<U*>
  {
    enum { value = 1 }; // is pointer
  };

  enum { is_pointer = PointerTraits<T>::value };

  // other interesting type properties ...
  enum { is_reference = ... };
  enum { is_const = ... };
  enum { has_trivial_assign = ... }; // bit-wise copy OK?
};
```

Here, enums are initialized with the result of a recursive COMPILE-TIME COMPUTATION
(such as `PointerTraits<T>::value`)

Examples

```
cout << TypeTraits<int*>::is_pointer << endl; // 1
```

Why 1? Let's trace the compile-time computation top down:

```
TypeTraits<T> ...

enum { is_pointer = PointerTraits<T>::value };
```

So, for `T = int*` we have

```
enum { is_pointer = PointerTraits<int*>::value };
```

What is `PointerTraits<int*>::value` ?

For `int*` this definition matches the partial template specialization:

```
// special case
template <typename U>
struct PointerTraits<U*>
{
  enum { value = 1 }; // is pointer
};
```

So, `PointerTraits<int*>::value = 1 !`

In general, for arbitrary types `X`:

```
cout << TypeTraits<X*>::is_pointer << endl; // 1
```

What about

```
cout << TypeTraits<int>::is_pointer << endl; // 0
```

?

Here, the general template version version matches:

```
// general case
template <typename U>
struct PointerTraits
{
  enum { value = 0 }; // no pointer
};
```

Fast Array Copy Function

The fast memcpy function can be used to copy arrays holding primitive (POD) objects

Otherwise, we need to iterate assignments for all array elements

Approach: class template that defines a flag telling the "smart" copy routine when to use memcpy

This is the goal:

```
const int N = 1000;

int a[N];
int b[N];
copy(b, a, N);  // uses memcpy: FAST

double a[N];
double b[N];
copy(b, a, N);  // also uses memcpy

struct Bar { virtual ~Bar() {...} }; // not POD
Bar a[N];
Bar b[N];
copy(b, a, N);  // uses Bar's AO: SLOW
```

copy is a function template:

```
template <typename T>
void copy(T *dst, T *src, int n)
{
  CopyImpl<has_triv_assign<T>::value>::copy(dst, src, n);
  //                              *****  0: slow,  1: fast
}
```

The class in which we call the copy function is dependent on a compile-time constant that tells us whether T has a trivial assignment operator or not

Note: here we use an int as template parameter

```
// default (slow)

template <int U>
struct CopyImpl
{
  // static method is a function template
  template <typename T>
  static void copy(T *dst, T *src, int n) {
    for (int i=0; i < n; ++i) {
      dst[i] = src[i];
    }
  }
};

// specialization (fast)

template <>
struct CopyImpl<1>
{
  // static method is a function template
  template <typename T>
  static void copy(T *dst, T *src, int n) {
    memcpy(dst, src, n*sizeof(T));
  }
};
```

How to figure out whether type T has a trivial assignment?

For example:

```
// safe default: no trivial assignment
template <typename T>
struct has_triv_assign { enum { value=0 }; };
// basic types can be assigned trivially
template <>
struct has_triv_assign<char> { enum { value=1 }; };
...
template <>
struct has_triv_assign<double> { enum { value=1 }; };
// all pointers can be assigned trivially
template <>
struct has_triv_assign<T*> { enum { value=1 }; };
```

For all other types let the user decide:

```
struct Foo {  // can be assigned trivially
  int a, b;
};

// followed by this line to make copy() aware of it
template <>
struct has_triv_assign<Foo> { enum { value=1 }; };

struct Bar {  // can't be assigned trivially
  virtual ~Bar() {...}
};
```

For `Bar` we don't have to say anything, because having no trivial assignment is the default

Result: copying 1,000,000 integers with the smart copy function is 2.3x faster (measured with time a.out) on my computer

Starting with C++11 a wide variety of type traits are supported. E.g.:

```
#include <type_traits>
struct X { virtual ~X() { } };
cout << std::is_polymorphic<X>::value;         // 1
cout << std::is_polymorphic<X*>::value;        // 0
cout << std::is_trivially_copyable<X*>::value; // 1
```

See `http://www.cplusplus.com/reference/type_traits/` and the boost section below

Compile-Time Recursion

Factorial function:

$0! = 1$

$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$, for $n > 0$

$1\ 1\ 2\ 6\ 24\ 120\ 720 \ldots$

```
template <int n>
struct Fac // general case n! = n*(n-1)!
{
  // compile-time constant
  enum { value = n * Fac<n-1>::value };
};

template <>
struct Fac<0>   // base case 0! = 1
{
  enum { value = 1 };       // compile-time constant
};

int main()
{
  // Fac<N>::value is now a compile-time constant!
  cout << Fac<5> ::value << endl;  // =  5! = 120
  cout << Fac<10>::value << endl;  // = 10! = 3628800
  cout << Fac<0> ::value << endl;  // =  0! = 1
}
```

Computation done at compile-time — beware, this can take a long time

It has been shown that C++'s template mechanism is Turing complete

This means that in principal you can run any kind of computation while compiling your program

Upside: compile-time computation speeds up runtime computation

Downside: you can't even be sure anymore whether compilation eventually stops ...

Other useful applications: unrolling loops, peeling off recursion levels, etc.

How bad can it be? Try this:

```
// general case: type recursion

template<int Depth, int A, typename B>
struct X {
  static const int x =
  // spawns two recursive calls
      X <Depth-1, 0, X<Depth,A,B>>::x
  // can't reuse previous result (1 vs. 0)
    + X <Depth-1, 1, X<Depth,A,B>>::x;
};

// base case

template <int A, typename B>
struct X<0,A,B> { static const int x = 1; };

static const int z = X<17,0,int>::x;

int main() { return 0; }
```

On my computer this program takes 17 seconds to compile. Replace the 17 by 18,19,20... and enjoy ... with each increment, the compilation time roughly doubles

Boost: A Collection of C++ Libraries

Playground for C++ libraries available at www.boost.org

Community proposes/reviews/improves libraries

Large variety of useful libraries:

regular expressions, formatted output, type traits, portable threading, random number generators, ...

Some of them like `shared_ptr` and `type_traits` already made it into the C++ standard

To use boost libraries with g++ include the need boost header files, e.g.

```
 #include <boost/filesystem.hpp>
```

and link with libraries (if necessary, most boost libraries are just headers)

```
 g++ -o hello hello.cpp -lboost_filesystem
```

On the lab machines g++ 5.4+ is installed which implements the full C++14 standard

Boost format

Formatting output with output stream operator << can be a pain (look it up, it's ugly!)

If you like C's printf, you'll love this:

```
cout << boost::format("%+4d %.2f %40s") % 399 % 1.34567 % "foo";
                       |                    |      |        |
              printf format string   arguments separated by %,
                                     those match the % in the
                                     format string
```

If you get the number of parameters or their type wrong, the runtime system will let you know

Above output:

```
+399 1.35                                      foo
```

C++ Arrays

C++ arrays are safe (and equally fast) replacements for standard C-arrays. They made it into the C++11 standard after being tested in the Boost library

```
#include <array>

std::array<int, 100> a;  // array of 100 ints

a[99] = 1;  // C-array syntax
a[100] = 0; // runtime error in debug mode (see below)
```

g++ generates `std::array` index bound checks when using `-D_GLIBCXX_DEBUG`

I suggest to say good-bye to unsafe C-arrays and use `std::array` as safe replacement in your code

Boost lexical_cast

Imagine converting numbers into strings and strings into numbers which can be accomplished by using string streams like so:

```
#include <sstream>

istringstream is("9999");
int n;
is >> n;    // converts string into integer
if (!is) { ... failed ... }

ostrstream os;
os << n;
... os.c_str() ...  // string representing n
```

Streams however are dynamic data structures and therefore sometimes too slow

Meet the boost::lexical_cast function template, which is faster:

```
#include <boost::lexical_cast.hpp>

n   = boost::lexical_cast<int>("9999");
str = boost::lexical_cast<std::string>(9999);

or

typedef std::array<char, 100> Buffer;
Buffer buf = boost::lexical_cast<Buffer>(n);
// no dynamic memory allocation
// but beware of overflow
```

There is much much more in the Boost library

Have a look