Genericity
C++ Templates

Object Oriented Programming (236703)
Winter 2015-6
Parametric Polymorphism

- Allows definitions to be parametrized at compile-time
  - Such a definition is actually a “compile-time function” that returns a new program element (method, class, etc.)

- Template parameters =
  - The “function”’s arguments
  - AKA: type parameters

- **Instantiation** of a template =
  - Evaluating the “function”
  - Code for actual class is generated
    - Executable (machine) code, not C++ code
Instantiation Process

- The compiler recompiles the template definition
  - Substitutes formal with actual template parameters
  - Generates new object (.o/.obj) code
  - A single (observable) instantiation per type argument(s)

- Almost no compilation when the definition is read
  - Minimal validation (parsing); actual compilation work done upon instantiation

- The template definition must be part of the source code
  - That is, reachable via include(s) from the instantiation point
  - Supporting separate compilation of templates is too difficult in practice
Template Code Arrangement

Non-template class compilation:

Declaration
class C {...}

Definition
int C::f() {...}

Use
#include “C.h”
C* c = …; c->f();

Template class erroneous compilation:

Declaration
template<class T>
class C {...}

Definition
template<T>
int C<T>::f() {...}

Use
#include “C.h”
C<int>* c = …; c->f();

Template class correct compilation:

Declaration +
Definition

Instantiation:
… C<int> …

include

link ×
C++ Class Templates

template<typename T> class Stack {
  T buff[50];
  int sp;
public:
  Stack() : sp(0) {}
  void push(T val);
  T pop() { return buff[--sp]; }
  int empty() const { return sp == 0; }
  int full() const { return sp == 50; }
};

template<typename T> void Stack<T>::push(T val) {
  if (sp >= size) { error("Stack is full"); return; }
  buff[sp++] = val;
}
C++ Function Templates

- A function can have type parameters too
  
  ```
  template<typename T> T* create() { return new T(); }
  void f() { string* sp = create<string>(); }
  ```

- Here, we must use `create()` with *explicit instantiation*
  - I.e., specify the actual types inside a pair of `< >`

- In some cases the compiler can deduce the template arguments
  
  ```
  template<typename T> void print(T t) { cout << t << endl; }
  void g() { print(5); }
  ```

- Now, we can use `print()` with *implicit instantiation*
  - Possible if a type argument is also a value argument
  - Makes overload resolution more difficult – what if there’s a `print(double)`?
Example: Function Template Instantiation

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

int main()
{
    int x = 10, y = getIntVal();
    int mi = max(x, y);

    double r = 0.5, s = getDoubleVal();
    double md = max(r, s);

    return max(mi, md);
}

Can the return type help?

error: no matching function for call to 'max(int&, double&)'
note: template argument deduction/substitution failed:
note:  deduced conflicting types for parameter 'T' ('int' and 'double')
Function Return Type?

The function return type is not used for resolving ambiguities:

```cpp
template <class T>
long double sum(const T a[], int size) {
    return (long double) avg(a, size) * size;
}

template <class T>
T sum(const T a[], int size) {
    return avg(a, size) * size;
}
```

The compiler will complain if the sum template is used
Functions: Template */3 Virtual

```cpp
struct Base {
    template<typename T>
    virtual void f(T) { ... }
};
```

file1.cpp:
Base* b1 = ...;
b1->f(5);

file2.cpp:
Base* b2 = ...;
b2->f(b2);

What should Base’s vtable hold?
- A single entry for f<T>?
  - f<int> != f<Base*>?
- One entry per initialization?
  - How would the compiler identify all initializations?

So:
A function cannot be virtual and a template!

Note: the Base class is not a template, and is compiled once as any other non-template class.

Note: template classes can definitely have virtual functions!
Implicit Template Instantiation

- Occurs whenever the programmer uses an instance of a generic for the first time
  - Examples: Class templates in C++
  - Advantages:
    - Convenient – no special commands are required
    - Accurate – you get the instantiated types you need
    - Allows partial compilation – reduces executable size
  - Disadvantages:
    - Difficult separation of definitions and declarations
    - Allows partial compilation – might yield surprising errors later

```cpp
// no C<int>
...
// C<int> created
C<int> c;
// C<int>::f created
c.f();
// C<int>, ::f exist
C<int> d;
d.f();
```
Example: Implicit Stack Instantiation

- Whenever a new particular version of the class template Stack is used (e.g., Stack<int> or Stack<char*>), the compiler will instantiate appropriate code
  - Each instantiation of template class is a class by itself
    - The template classes Stack<int>, Stack<Complex>, and Stack<void*> are different classes
- Every instantiation might yield different compilation errors
- Only the members that are used are instantiated
  - If you only use `push`, the code will compile even if `pop` contains errors!
Explicit Template Instantiation

- Occurs when the programmer explicitly instantiates a generic, before the generic is actually used
  - Examples: Ada, C++ explicit instantiation
- Advantages:
  - Easier to arrange code
  - Complete compilation – errors immediately detected
- Disadvantages:
  - If used to ease definitions and declarations management – how can you anticipate all possible instantiations?
- What if the same generic is instantiated twice?
  - That’s an error
Explicit Instantiation in C++

- `template class Stack<int>;
  template void Stack<int>::push(int &);`

- Can Pre-create libraries of instantiations, independent of the using program

- Template definitions can reside in a separate `cpp` file if that file contains all possible (explicit) instantiations

- Explicit instantiation of a class template: *forces instantiation of all its members, consequently forcing constraints*
Possible Template Parameters

- Generics are “functions executed at compile time, producing code”. Therefore, all arguments must be entities that are known at compile time:
  - Types
    - Most frequent and most important; all languages which support genericity allow type parameter
  - Numerical Constants (e.g., the integer constant 3)
    - Useful for building generic arrays
    - C++11 – any constexpr value (explained in a minute)
  - Addresses
    - Variables
    - Functions
A Template Taking a Type

template<typename T>
struct Pair {
    void set(const T& x, const T& y) { a = x; b = y; }
    void print() { cout << a << ", " << b << endl; }
    private: T a, b;
};

int main() {
    Pair<char*> sp;
    Pair<int> ip;
    sp.set(\"ab\", \"cd\"); sp.print();
    ip.set(10, 20); ip.print();
}
A Template Taking a Constant

template<int N>
struct FixedArray {
    double values[N];
    int size() { return N; }
};

int main() {
    FixedArray<10> arr;
    arr.values[0] = 3.14;
    cout << "first element=" << arr.values[0] << endl;
    cout << "size=" << arr.size() << endl;
    return 0;
}
A Template Taking a constexpr

```cpp
template<int N>
struct FixedArray {  
    double values[N];  
    constexpr int size() { return N; }  
};

int main() {  
    FixedArray<10> arr1;  
    FixedArray<arr1.size() + 1> arr2;  
    static_assert(arr1.size() == arr2.size() - 1,  
                  "Unexpected sizes!");  
}
A Template Taking a Function Pointer

typedef void (*ErrorFunction)(const char *);

template <typename Type, ErrorFunction error>
class Array {
    size_t n; Type* buff;

public:
    Array(size_t n_): n(n_), buff(new Type[n]) {
        if (buff == NULL) error("Memory failure");
    }

    Type& operator[](size_t i) {
        if (i >= n) { error("Array overflow");
            return buff[i];
        }
    }
};
void err_abort(const char *msg) {
    cerr << "Error: " << msg
    << " . Aborting!" << endl;
    exit(1);
}

typedef Array<int, err_abort> SafeIntArray;
...
SafeIntArray a(20);
String Literals as Type Arguments?

```
template <typename Type, const char* Name>
class NamedType : public Type {
   public:
   const char* name() { return Name; }
};
```

Does not work! Consider

File1.cpp: NamedType<MyClass, "MyClass"> m1;
File2.cpp: NamedType<MyClass, "MyClass"> m2;

The language neither guarantees a single “MyClass” string, nor multiple ones. So, are m1 and m2 of the same type?

Note: const char* can sure instantiate a template<T>!

How is this pattern called?

template<double D> is also disallowed, due to rounding issues!
Specialization

template<typename T>
struct Pair {
   // T a, b; set; print; ...
};

template<>
struct Pair<bool> {
   void set(bool x, bool y) { v = (x?1:0) + (y?2:0); }
   void print() { cout << (v&1) << "", " << (v&2) << endl; }
   private: uint8_t v:2; // v is a bitmap
};

int main() {
   Pair<bool> pb; pb.set(true, false); pb.print();
   Pair<char> pc; pc.set('x', 'y'); pc.print();
}
Non Conforming Specialization is Legal

template<typename T>
struct Pair {
    // T a, b; set; print; ...
};

template<>
struct Pair<bool> {
    void set(bool x, bool y) { v = (x?1:0) + (y?2:0); }
    public: uint8_t v:2;
};

void main() {
    Pair<bool> pb; pb.set(true, false);
    cout << pb.v << endl;
    pb.print(); // Error. Pair<bool>::print() is undefined
}
C++ Templates: Interim Summary

- A template is a “function”
  - Arguments: types, constants
  - Return value: A new class or a new function
    - Recognized by the compiler

- The template is recompiled with each instantiation
  - Linker can sometimes remove the bloat

- Specialization == “if”
  - Different result based on the actual arguments
template <int N>
struct Factorial {
    enum { Value = N * Factorial<N - 1>::Value }
};

template <>
struct Factorial<0> {
    enum { Value = 1 }
};

int main() {
    constexpr int x = Factorial<4>::Value; // == 24
    constexpr int y = Factorial<0>::Value; // == 1
}
A Non-Terminating Compilation

template <typename T>
struct Loop
{
    typedef typename Loop<Loop<T> >::Temp Temp;
};

int main() {
    Loop<int> n;
    return 0;
}
Typename Disambiguation

```cpp
template<class T>
struct A {
    typedef int M;
};

int p = 0;

template<>
struct A<double> {
    static int M;
};

By default, dependent names are parsed as non-types. If the instantiation yields a type, that’s an error. To disambiguate, define:

```cpp
typename A<T>::M* p
```
Type parameters significantly affects template code

- Constraining type arguments can ease template author work
- But limit user’s ability to use template

Main approaches:
- No restrictions (on arguments)
- Check upon use
- Explicit list of constraints
- By derivation

```cpp
template<class T>
struct A {
    ...;
};
```

Must be a pointer
Must have operator `<`
Must have an `f()`
No Restriction on Arguments

- Any type is allowed. Generic limited to what can be done on all types (i.e., only Object methods)
  - Example:
    - Early versions of Eiffel
  - Advantages:
    - Code sharing – all instantiations use the same code
  - Disadvantages:
    - Restricts generics code
    - Only the most primitive operations are allowed on arguments
    - Useless when no shared root (C++)
Checking Arguments Upon Use

- The generic equivalent of dynamic typing
  - A.k.a. duck typing

- The compiler tries to instantiate a template. If an invalid operation is attempted, then an error message is issued
  - Example:
    - C++

- Advantages:
  - Compiler writing is easy
  - Flexibility to programmer

- Disadvantages:
  - Extra burden on the library designer
  - Surprises to the user
  - Constraints are dependent on actual usage
  - Difficult to understand

```cpp
std::set<MyClass> sm; sm.insert(MyClass());
```

In instantiation of `bool std::less<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = S]`:
```
/usr/include/c++/4.8/bits/stl_function.h: In instantiation of 'bool std::less<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = S]':
/usr/include/c++/4.8/bits/stl_tree.h:1324:11: required from 'std::pair<std::_Rb_tree_node_base*, std::_Rb_tree_node_base*> std::_Rb_tree<_Key, _Val, _KeyOfValue, _Compare, _Alloc>::_M_get_insert_unique_pos(const key_type&) [with _Key = S; _Val = S; _KeyOfValue = std::_Identity<S>; _Compare = std::less<S>; _Alloc = std::allocator<S>; std::_Rb_tree<_Key, _Val, _KeyOfValue, _Compare, _Alloc>::key_type = S]'
```
Explicit List of Constraints

- The programmer specifies the set of operations that might be used on a type argument
  - Example:
    - Ada, future C++
  - Advantages:
    - Readability
    - No surprises
  - Disadvantages:
    - Lists could be very long
    - Less flexibility in the design of classes
    - List of operations instead of abstraction of functionality
Constraints using Inheritance

By derivation: the type argument must be the same or inherited from the declared type parameter

Example:
- Current Eiffel, Java, C#

Advantages:
- Code sharing
- OOP Like
  - Abstraction at the right level

Disadvantages:
- Could lead to overuse of inheritance
  - The template might need a feature *(getName())*, not a type
- Not appropriate for built-in types
Quiz: What are the 6 Constraints?

template <class T>
T avg(const T a[], int size)
{
    T sum = a[0];
    for (int i = 1; i < size; i++)
        sum += a[i];
    return sum / size;
}
Quiz: The 5 Constraints

template<class T>
class Stack {
    T buff[50];
    int sp;

public:
    Stack(): sp(0) {}
    void push(const T &e) {
        buff[sp++] = e;
    }
    T pop(void) { return buff[--sp]; }
    int empty(void) const { return sp == 0; }
    int full(void) const { return sp == 50; }
};
Concepts

- C++ way of explicitly defining constraints
  - Rely on structural compatibility of types and concepts
    - Resembles Go interfaces. Remember them?
- Work in progress – didn’t make it into C++11 and C++14, maybe in C++1z (17?)
- A few early implementations, different syntaxes and abilities
  - Most noticeable implementation – *Concepts Lite*
  - So, don’t mind the details – we care about *the concept!*
Motivation

- Template errors are detected too late – during instantiation
  
  - If templates are compile-time functions that generate code, they currently work like dynamically-typed languages: errors detected at “run time”, and not on the call site
  
- Needed: a way to detect errors on the call site
  
  - I.e., before actual instantiation
Using a Concept (Lite)

template<EqualAndLess T>
int compare(const T& x, const T& y)
{
    return (x < y) ? -1 :
    (x == y) ? 0 :
    1;
}
template<typename T>
constexpr bool EqualAndLess()
{
    return requires(T a, T b) {
        {a == b} -> bool;
        {a < b} -> bool;
    };
}
struct S {} x, y;

int res = compare(x, y);

Error: no matching call to ‘compare(S, S)’
Note: template constraints not satisfied
Note: ‘S’ has no operator ==
Note: ‘S’ has no operator <
Pros and Cons of Concepts

Pros:
- Errors are detected *before instantiation* – yields much clearer messages
- Better documentation – template states what it expects

Cons:
- Yet another layer of language complexity
- Current work only deals with syntactic requirements
  - Semantic work expected in final proposal: don’t just ask for an operator ==, ask for equality comparison
- Actual introduction into the language: unknown