Parasol
Smarter computing.
Texas A&M University

Variadic templates

Bjarne Stroustrup

Spring 2013

Parasol
Smarter computing.
Texas A&M University

Future topics – max 20 lectures to go

- Containers – vector, map, set, unordered_map, …
- Constexpr – compile-time computation
- Type functions (using, constexpr)
- Concepts – C++0x
- Constraints – C++14
- Concept design – STL algorithms
- Haskell, Java, C#
- Allocators
- Template metaprogramming
- Matrix library
- Concurrency library?
- Graph libraries
- FP-style pattern matching
- C++14 report
- **And ??? – send me suggestions**

Problems

- How to construct a class with 1, 2, 3, 4, 5, 6, 7, 8, 9, or … initializers?
  - E.g., **thread t {f,x,y91};**
- How to avoid constructing an object out of parts and then copying the result?
  - **vector<pair<string,int>> v;**
    v.push_back(make_pair(string,7)); *// involves a copy*
- How to construct a tuple?

Variadic templates

- Many would like to do something like this
  - (use boost::format), but

```
const string pi = "pi";
const char* m = "The value of %s is about %g (unless you live in %s).\n";
printf(m,  pi, 3.14159, "Indiana");
```

Variadic templates

- Deal with arbitrary numbers of arguments of arbitrary types at compile time
  - In a type-safe manner

```
// First: handle a printf string without non-format arguments:
void printf(const char* s)
{
    if (s==nullptr) return;

    while (*s) {
        if (*s == '%' && *++s != '%')
            throw runtime_error{"invalid format: missing arguments"};
        std::cout << *s++;
    }
}
```

Variadic templates

```
// Next: handle a printf with non-format arguments:
template<typename T, typename... Args>          // note the "…"
void printf(const char* s, T value, Args... args)      // note the "…"
{
    if (s==nullptr) throw std::runtime_error{"printf: No format string"};

    while (*s) {
        if (*s == '%' && *++s != '%') {
                std::cout << value;          // use first non-format argument
                                             // note: no use of format specifier
                return printf(++s, args...);          // "peel off" first argument
        }
        std::cout << *s++;
    }
    throw std::runtime_error{"extra arguments provided to printf"};
}
```

Variadic templates

```
// we can check T against known argument types:
if (*s=='%') {        // a format specifier or %%
        switch (*++s) {
        case '%':             // not format specifier
                break;
        case 's':  if (!Is_C_style_string<T>() && !Is_string<T>())
                        throw runtime_error("Bad printf() format");
                break;
        case 'd':  if (!Is_integral<T>())
                        throw runtime_error("Bad printf() format");
                break;
        case 'g':  if (!Is_floating_point<T>())
                        throw runtime_error("Bad printf() format");
                break;
}
std::cout << value;                    // use first non-format argument
```

Variadic templates

- "Think **tuple**"
  - And, yes, there is a **std::tuple**

Variadic templates and initializer lists

- Initializer lists require {}

```
void fi(initializer_list<int>);
fi(1);              // error
fi({1});            // ok
fi({1,2});          // ok
fi({{1},{2}});      // ok? (yes: means: fi({1,2}) for C compatibility)
```

- Variadic templates doesn't need { }

```
template<typename... Args> void fv(Args…);
fv(1);              // ok
fv(1,2);            // ok
fv({1});            // a tuple {1}; not the same as f(1)
fv({1,2});          // ok: a single tuple, not the same as fv(1,2)
fv({{1},{2}});      // ok? (yes: tuple of tuples)
```

Variadic templates and initializer lists

- Their area of use overlap for homogenous initializers
  - Imagine that = is suitable defined

```
y = { “asdf”, 1, X, pair<int,complex>{1, {2,3}}};// template
z = {1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0};          // list or template?
```

- A variadic template is a template
  - A function taking an initializer list need not be a template
  - But elements must be read from the initializer lists
    - no moves –a { } list is immutable
- A template generates a new type for each invocation with a different type

```
template<typename... Args> f(Args…);          // potential code bloat
f(1);       // one f
f(1,2);     // another f
f(1,2,3);   // yet another f
```

Variadic templates and initializer lists

- List **min()**:

```
template< typename T, class Compare>
const T& min(initializer_list<T> values, Compare comp);  // std::min()

int i = min({1, 2, 42, a, b, 34}, comp);
```

- Variadic **min()** is a little bit like:

```
template <classT, class Compare , typename ... Args >
const T& min( const T& a, const T& b, const Args &... args , Compare comp );

int i = min(1, 2, 42, a, b, 34, comp);
```

- Note: the real definition is much more complex because the comparator argument is part of the variadic argument pack and need special processing.

Variadic templates in standard containers

- Emplace
  - list, deque, vector

  a.emplace_back(args) *means* **a.emplace(a.end(), forward<Args>(args)...)**
- Declaration

  template <class... Args> void emplace_back(Args&&... args);
- Call

```
vector<pair<string, int>> v;
v.emplace_back("asdf",2);     // no copying of temporary object
                              // but emplace_back is a member template
                              // remember min()
```

forward() and ref()

- How to forward:
  - A reference must remain a reference
  - An rvalue must remain an rvalue
  - An lvalue must remain an lvalue
- **std::forward()**:

```
template<typename T>
T&& forward(remove_reference<T>::type& t) noexcept
{
    return static_cast<T&&>(t);
}
```

Thread construction

- Constructor

  template <class F, class ...Args>

  explicit thread(F&& f, Args&&... args);

- Use

  void f0(); *// no arguments*

  void f1(int); *// one int argument*

  thread t1 {f0};

  thread t2 {f0,1}; *// error: too many arguments*

  thread t3 {f1}; *// error: too few arguments*

  thread t4 {f1,1};

  thread t5 {f1,1,2}; *// error: too many arguments*

  thread t3 {f1,"I'm being silly"}; *// error: wrong type of argument*

Thread construction

- Pass a reference as a reference

```
void my_task(vector<double>& arg);          // note: pass by reference

void test(vector<double>& v)
{
        thread my_thread1 {my_task,v};          // Oops: pass a copy of v
        thread my_thread2 {my_task,ref(v)};     // OK: pass v by reference
        thread my_thread3 {[&v]{ my_task(v); }};// OK: dodge the & problem
        // ...
}
```

Thread construction

- Spot the bug

```
double comp2(vector<double>& v)
{
    using Task_type = double(double*,double*,double);      // type of task

    packaged_task<Task_type> pt0 {accum};      // package the task (i.e., accum)
    packaged_task<Task_type> pt1 {accum};

    future<double> f0 {pt0.get_future()};      // get hold of pt0's future
    future<double> f1 {pt1.get_future()};      // get hold of pt1's future

    double* first = &v[0];
    thread t1 {pt0,first,first+v.size()/2,0};      // start a thread for pt0
    thread t2 {pt1,first+v.size()/2,first+v.size(),0};  // start a thread for pt1
    // ...
    return f0.get()+f1.get();                      // get the results
}
```

Thread construction

- move

```
double comp2(vector<double>& v)
{
    using Task_type = double(double*,double*,double);      // type of task

    packaged_task<Task_type> pt0 {accum};      // package the task (i.e., accum)
    packaged_task<Task_type> pt1 {accum};

    future<double> f0 {pt0.get_future()};      // get hold of pt0's future
    future<double> f1 {pt1.get_future()};      // get hold of pt1's future

    double* first = &v[0];
    thread t1 {move(pt0),first,first+v.size()/2,0};          // start a thread for pt0
    thread t2 {move(pt1),first+v.size()/2,first+v.size(),0};  // start a thread for pt1
    // ...
    return f0.get()+f1.get();                  // get the results
}
```

Let's start simply

- Pair, a useful **std** class:

```
template<typename T, typename U>
struct pair {
        using first_type = T;       // the type of the first element
        using second_type = U;      // the type of the second element

        T first;                    // first element
        U second;                   // second element

        // ...
};
```

Pairs are useful and common

- You can use them directly

```
pair<string,int> p {"Cambridge",1209};
cout << p.first;          // print "Cambridge"
p.second += 800;          // update year
```

- But mostly they are part of libraries
  - E.g. an element of a **map<K,V>** is a **pair<const K,V>**

```
void print_all(const map<string,int>& m)
{
        for (const auto& p : m)
                cout << p.first << ": " << p.second << '\n';
}
```

It is tedious to write pair<T,U>

- We can use **std::makepair()**:

```
pair<string,int> h {"Harvard",1736};
auto p = make_pair("Harvard",1736);
```

pair use

- Alternatives

```
void user(list<pair<string,double>>& lst)
{
        // find an insertion point:
        auto p = lst.begin();
        while (p!=lst.end()&& p->first!="Denmark")  /* do nothing */ ;

        p=lst.emplace(p,"England",7.5);              // nice and terse
        p=lst.insert(p,make_pair("France",9.8));  // helper function
        p=lst.insert(p,pair<string,double>>{"Greece",3.14}}); // verbose
}
```

Emplace?

- All standard containers have one

```
template<typename T, typename C = deque<T>>
class stack {                    // _iso.stack.defn_
public:
        // …
        template<typename... Args>
        void emplace(Args&&... args)
        {
                c.emplace_back(std::forward<Args>(args)...);
        }
protected:
        C c;
};
```

Tuple

- A **pair<T,U>** has two elements
  - Why not 5?
  - Why not 0?
  - In generic code we often want to store N elements
    - For almost arbitrary N
  - If the elements are homogeneous, use something like a **vector**
- If the elements are heterogeneous, we need a **tuple**

```
template<typename... Types>
class tuple {
public:
    // ...
};
```

Tuples

- In the old days
  - Just use N arguments where N is more than you need

```
struct unused {};
template<class T1 = unused, class T2 = unused, class T3 = unused>
class Tuple {  //  can be used with 0 to N arguments
                    // How big does N has to be in real life?
    T1 v1; T2 v2; T3 v3;              // a waste of space
public:
    Tuple();
    Tuple(T1);
    Tuple(T1,T2);
    Tuple(T1, T2, T3);      // a lot of typing
    // …
};
```

Tuple

- Avoid the unused arguments and wasted space

```
// don't bother implementing the general case:
Template<class T1, class T2, class T3> class Tuple;

template<> class Tuple<> {        // specialize: N==0
public:
    Tuple();
    // …
};

template<class T1>
class Tuple<T1> {        // specialize: N==1
    T1 v1;
public:
    Tuple(T1);
    // …
};
// …
```

Tuple

- What do you want tuples for?
  - Heterogeneous structures
    - that you don't want to write "by hand"
  - **pair<T1,T2>** is the simplest tuple
    - (and arguably most useful)
    - E.g. name/value pair
    - E.g. return-code/value pair
  - Compile-time lists
  - List of bound arguments in a bound function (Currying)
    - E.g. the implementation of **std::bind()**

Tuples

- Note
  - We never store the types (only values of types)
    - There is no standard run-time representation of types in C++
    - Like all template-based schemes tuples are compiler-based
  - The tuple values are stored contiguously
    - Not as a linked structures
    - The zero-overhead principle holds

Tuples

- How to make a general (variadic) tuple
  - Rumor has it that you can do it differently "but not easily or elegantly"

```
template<typename... Elements> class tuple;          // general tuple (never used)
template<typename Head, typename... Tail>            // N-tuple for 1<=N
class tuple<Head, Tail...>
    : private tuple<Tail...>  // note: base class depending on template argument
                              // note: private base class
{
    Head head;            // each added type adds one member to the final class
public:
    // …
};
template<>
class tuple<> {           // zero-tuple
    // …
};
```

20.4.2 Class template tuple

```
template <class... Types>
class tuple {
public:     // 20.4.2.1, tuple construction
constexpr tuple();
explicit tuple(const Types&...);
template <class... UTypes>
explicit tuple(UTypes&&...) noexcept;
tuple(const tuple&) = default;
tuple(tuple&&) = default;
template <class... UTypes>
tuple(const tuple<UTypes...>&);
template <class... UTypes>
tuple(tuple<UTypes...>&&) noexcept;
template <class U1, class U2>
tuple(const pair<U1, U2>&); // iff sizeof...(Types) == 2
template <class U1, class U2>
tuple(pair<U1, U2>&&) noexcept; // iff sizeof...(Types) == 2
// allocator-extended constructors
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a);
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a, const Types&...);
template <class Alloc, class... UTypes>
tuple(allocator_arg_t, const Alloc& a, const UTypes&&...);
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a, const tuple&);
template <class Alloc>
tuple(allocator_arg_t, const Alloc& a, tuple&&);
template <class Alloc, class... UTypes>
tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
template <class Alloc, class... UTypes>
tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
template <class Alloc, class U1, class U2>
tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
template <class Alloc, class U1, class U2>
tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);
// 20.4.2.2, tuple assignment
tuple& operator=(const tuple&);
tuple& operator=(tuple&&) noexcept;
template <class... UTypes>
tuple& operator=(const tuple<UTypes...>&);
template <class... UTypes>
tuple& operator=(tuple<UTypes...>&&) noexcept;
template <class U1, class U2>
tuple& operator=(const pair<U1, U2>&); // iff sizeof...(Types) == 2
template <class U1, class U2>
tuple& operator=(pair<U1, U2>&&) noexcept; // iff sizeof...(Types) == 2
void swap(tuple&) noexcept;
};
```

Tuples

- How to make a general (variadic) tuple

```
template<typename Head, typename... Tail>          // N-tuple for 1<=N
class tuple {
public:
    // constructors
    // default destructor
    // assignments
    // swap
      // add rvalue constructors and assignments
      // add allocators to constructors and assignments
      // add noexcept
      // add conversion from pair
};
```

20.4.2 Class template tuple

```
template<typename Head, typename... Tail>     // N-tuple for 1<=N
class tuple {
public:    // 20.4.2.1, tuple construction
    constexpr tuple();
    explicit tuple(const Types&...);
    template <class... UTypes> explicit tuple(UTypes&&...) noexcept;
    tuple(const tuple&) = default;
    tuple(tuple&&) = default;
    template <class... UTypes> tuple(const tuple<UTypes...>&);
    template <class... UTypes> tuple(tuple<UTypes...>&&) noexcept;
    template <class U1, class U2>
        tuple(const pair<U1, U2>&);            // iff sizeof...(Types) == 2
    template <class U1, class U2>
        tuple(pair<U1, U2>&&) noexcept;        // iff sizeof...(Types) == 2
```

20.4.2 Class template tuple

```
    // allocator-extended constructors
template <class Alloc> tuple(allocator_arg_t, const Alloc& a);
template <class Alloc> tuple(allocator_arg_t, const Alloc& a, const Types&...);
template <class Alloc, class... UTypes>
    tuple(allocator_arg_t, const Alloc& a, const UTypes&&...);
template <class Alloc> tuple(allocator_arg_t, const Alloc& a, const tuple&);
template <class Alloc> tuple(allocator_arg_t, const Alloc& a, tuple&&);
template <class Alloc, class... UTypes>
    tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
template <class Alloc, class... UTypes>
    tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
template <class Alloc, class U1, class U2>
    tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
template <class Alloc, class U1, class U2>
    tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);
```

20.4.2 Class template tuple

```
    // 20.4.2.2, tuple assignment
    tuple& operator=(const tuple&);
    tuple& operator=(tuple&&) noexcept;
    template <class... UTypes>
        tuple& operator=(const tuple<UTypes...>&);
    template <class... UTypes>
        tuple& operator=(tuple<UTypes...>&&) noexcept;
    template <class U1, class U2>
        tuple& operator=(const pair<U1, U2>&);        // iff sizeof...(Types) == 2
    template <class U1, class U2>
        tuple& operator=(pair<U1, U2>&&) noexcept; // iff sizeof...(Types) == 2
    void swap(tuple&) noexcept;
};
```

Tuple helper functions

- Make a tuple (deduce the argument types):

```
template<class... Types>
    tuple<VTypes...> make_tuple(Types&&... t);   // Vtypes is shorthand for
                                                 // "Types" with references
                                                 // dereferenced
```

- Access Ith element

```
template <size_t I, class... Types>
typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>& t);
```

- ==, !=, <, <=, >, >=

Tuple and pair

- A 2-**tuple** is not a **pair**, but
  - You can construct a **tuple** from a **pair**
  - You can assign a **pair** to a **tuple**
  - You can access a **pair** like a **tuple** (**get<1>(t)** and **get<2>(t)**)
  - You cannot construct a **pair** from a **tuple**
  - You cannot assign a **tuple** to a **pair**

Simplicity vs. generality

- In general, ideally you want solutions that are
  - General (e.g. works for any N)
  - Simple and efficient for the important cases (e.g. N==1 and N==2)
- I personally have a bias towards “simple and efficient”
- For templates you can
  - Be general using variadic templates
  - Specialized for the simple cases

Variadic template references

- N2151 07-0011 Variadic Templates for the C++0x Standard Library D. Gregor, J. Järvi,
- N2080 06-0150 Variadic Templates (Revision 3) D. Gregor, J. Järvi, G. Powell
- N2087 06-0157 A Brief Introduction to Variadic Templates Douglas Gregor
- N2772 08-0282 Variadic functions: Variadic templates or initializer lists? -- Revision 1 L. Joly, R. Klarer
- N2551 08-0061 A variadic std::min(T, ...) for the C++ Standard Library (Revision 2) Sylvain Pion