Templates

PIC 10B, UCLA

Templates

One of the most powerful techniques in C++ is the ability to templatize an implementation.

If that were not the case then we would need to have *separately defined* classes such as **std::vectorInt, std::vectorDouble, std::vectorString**, for every type of class we want to store in a vector.

This would also mean that we could not have a **std::vectorEmployeePointer** class because a variable type **Employee** is one we define outside of the C++ Standard.

Then, we'd also need to have separate functions to sort **std::vectorSomething** and **std::dequeSomething**, etc. The function **std::sort**, as it is, can work on any container with random access iterators.

Templates

Beyond this, much of our code can become redundant... consider defining a **max** function that returns the maximum of two values (could be **double**, **std::string**, etc.). The same code essentially works for any type, and we should only have to write that code once, instead of:

```
int max(int, int);
double max(double, double);
char max(char, char);
const std::string& max(const std::string&, const std::string&);
// etc. etc. etc.
```

Just one

```
const T& max(const T&, const T&);
```

defined for arbitrary **T** should be sufficient.

Defining Templates: Simple Example

Observe a templated **max** function that will return the maximum of two inputs (of the same type), with the maximum defined by **operator<**, which must be defined for those types. The following code defines the templated function:

```
/**
Template function returns the maximum (by <) between two arguments.
@tparam T the type of the arguments
@param first the first argument
@param second the second argument
@return a reference to the larger argument (could be a dangling pointer!)
*/
template <typename T> // this is templated based on the type T
const T& max(const T& first, const T& second) { // signature
   return (first < second) ? second : first; // return the larger value
}
```

Defining Templates: Simple Example

```
double x = 41.7, y = 5.31;
std::string s1 = "Dog", s2 = "cat"; // recall 'A'< ... < 'Z'< 'a'< ... < 'z'

std::cout << max(x,y) <<" "<< max(s1,s2);
```

```
41.7 cat
```

Notice that even if fundamental types are passed to the **max** function we wrote, they will be passed as references to const: in order for templates to apply as efficiently as possible to all data types, it is necessary to pretend we are passing large class objects.

Defining Templates: Simple Example

Remark: it is more efficient to define
const T& max(const T&, const T&), returning by reference rather than by value. But this efficiency can also break code so one must be careful.

```
std::string Lvalue("Lvalue"); // fine
```

Below, the string literal instantiates a **std:string prvalue** that's passed to the function. Hence a local variable **right** of type **std::string** is created. When the function call ends, that local variable is destroyed and the output could be referencing a destroyed object!

```
const std::string& out = max(Lvalue, std::string("Rvalue"));

std::cout << out; // BAD: might print nothing, might break code...
```

Defining Templates: Simple Example

A template is declared by using the **template** keyword and then a comma separated list of one or more template parameters types in a set of angled brackets **< >.**

The **typename** keyword signifies that the variable type can be anything: **T** could be a **std::string**, **std::vector**, **long long**, etc.

After signifying that **T** is of any type, we then implement the function “normally” as though **T** were some specific type.

The proper documentation includes the **@tparam** descriptor(s) of the template argument type(s).

Function Templates

A slightly older keyword that is used in many codes is **class** and in our contexts, it means the exact same thing as **typename**:

```
template<class T>
const T& max(const T&, const T&); // declares templated max function
```

Function Templates

Remark on constexpr: for templates, provided a version of the templated function can be **constexpr**, we can actually mark the functions as such. Thus, a better **max** would be:

```
template<typename T>
constexpr const T& max(const T& first, const T& second) {
   return (first<second) ? second : first;
}
```

For templates, the **constexpr** qualification is ignored unless we ask to use the function in a **constexpr** context.

```
constexpr int x = max(4,7); // okay

int a = 9; // not constexpr
int y = max(5,a); // okay: not asking for constexpr

constexpr z = max(5,a); // ERROR: a not constexpr
```

Function Templates

General template function syntax:

// declare template function of one paramter type
template<typename T>
returnType function(arguments);

// declare template function of two parameter types
template<typename T1, typename T2>
returnType function(arguments);

etc.

Note: there is no requirement that all the template parameter types are the same type within a given template. **T1** and **T2** could represent different data types, etc.

Function Templates

It doesn’t matter what name we give to the template parameter types: **S**, **T**, **foobar**, etc.

```
template <typename T, typename S>
void print_both(const T& t, const S& s) {
   std::cout << t << " "<< s << '\n';
}
```

The above works in all the cases

```
print_both(0, 39.6); // print an int and double

print_both(7,8); // print two ints

// print a const char* and Time object
print_both("string literal", Time(4,5,6));
```

etc.

Template Definition

A **template** definition gives the blueprints for a function or class; the precise way the function/class works is determined by the template parameters.

Having a template definition *does not instantiate a class or function*. It only tells the compiler *how to create* such a class or function should the need arise.

There's no such thing as **void print_both(const int&,const double&)** until the compiler sees the need for that in the code.

Template Definition

A template should be thought of as a factory for making function/classes, but the factory only makes a function/class "on demand".

To avoid linker errors, **templates should be declared and defined within a header file!!!** This is in contrast to declaring in a **.h**-file and defining in a **.cpp**-file.

Special rules apply for templates and in the contexts we'll be looking at, our functions are implicitly **inline** so this won't be an issue.

Function Templates

For templated functions, provided the compiler can infer the proper parameter types, *it will do so for us automatically*. However, at times this can be ambiguous. Recall the signature of **max:**

```
template<typename T>
constexpr const T& max(const T&, const T&);
```

Consider:

```
std::cout << max(3.14, 2); /* ERROR: is T a double or an int? */
```

Function Templates

Because our max function accepted two arguments of the same type, the compiler is unsure of what to choose. It could turn 3.14 into the **int** 3 and do the comparison; or it could promote 2 to the **double** 2. and do the comparison. We remedy the situation by explicitly stating the template parameter:

```
std::cout << max<double>(3.14, 2); this or
std::cout << max<int>(3.14, 2); // this
```

Function Templates

The type deduction can even apply to the type of elements being stored:

```
template<typename T>
void print(const std::set<T>& theSet) { // given a set storing any type
   for (const auto& t : theSet) { // prints all its elements
      std::cout << t << '\n';
   }
}
```

Then we can print an arbitrary **std::set**:

```
std::set<int> s1 {1,2,-1,-4};
std::set<double> s2 {1.2, 4.8, 2.2};
print(s1);
print(s2);
```

Function Templates

Warning: technically the **print** function defined on the previous slide is "not general enough" and wouldn't work for all **std::sets** without the signature:

```
template<typename Type, typename Compare, typename Alloc>
void print(const std::set<Type,Compare,Alloc>&);
```

An **std::set** is templated not only by its stored data type, but also by its comparison type and its allocator type.

Function Templates

It is possible to templatize based on integer values, too.

```
template<typename T, size_t sz> // any type T, any size_t parameter s
// arr is a reference to array of T of size sz
void print_array( const T (&arr)[sz]) {
   for (const auto& x : arr) {     std::cout << x << " ";     }
}

// In main:
int arr[] = { 1, 2, 3, 4, 5};
print_array(arr);
```

```
1 2 3 4 5
```

When arrays are passed by reference as above, the size parameter can be found without our specifying it: there is no pointer decay here.

Function Templates: decltype

Suppose we want a function that simply returns the first element of a container (if there is one!):

```
// Out cannot be deduced: deductions are left to right
template<typename Out, typename Container>
Out first(Container& c) {
   return *( std::begin(c) );
}
```

Function Templates: decltype

Starting in C++11, we can use **decltype** with a trailing return type:

```
template<typename Container>
auto first(Container& c) -> decltype( *( std::begin(c) ) ) {
   return *( std::begin(c) );
}
```

Whatever type ***(std::begin(c))** happens to be will be returned.

We used the free function **std::begin** rather than the member function call **.begin()** because C-style arrays do not have member functions.

```
// example:
std::string s("hi");
first(s) = 'H'; // now s == "Hi"
```

Class Templates I

We can also define classes as templates. We consider a simple implementation for a **pair** class.

To illustrate various syntax, some of our definitions will be within the class and others outside.

Class Templates II

```
template <typename T1, typename T2>
struct pair {
   T1 first;
   T2 second;

   using first_type = T1; // aliases for the types
   using second_type = T2;

   // interface to be continued...
```

We included aliases for the types T1 and T2. This is common practice in C++ templated classes and is done for the real **std::pair**, too.

Class Templates III

```
// default constructor
constexpr pair() : first(), second() { }

// interface to be continued ...
```

The empty parentheses after **first** and **second** ensure the data are value initialized, i.e. set to 0 if they are numeric types, and otherwise default constructed.

Provided both data types are **constexpr** type, the **constexpr** version will apply.

Class Templates IV

```
    // constructor to directly initialize values
    constexpr pair(const T1& t1, const T2& t2); // declared within class

}; // end interface
```

Other implementations (to accept by value and move) or accepting rvalue references are also possible.

The C++ Standard implementation of **std::pair** has a constructor accepting lvalue references to const for its inputs (as above) *and a constructor that accepts forwarding references* (to be discussed), which can also maximize efficiency even more than passing by value and moving!

Class Templates V

```
template<typename T1, typename T2>
void swap(pair<T1,T2>& x, pair<T1,T2>& y) {
    using std::swap; // enable ADL, use the best swap
    swap(x.first, y.first);
    swap(x.second, y.second);
}
```

The **using std::swap** enables ADL. We don't know the types of **first** and **second** and it's possible (as we wrote for our own classes) special **swap** functions exist for them that are part of a special namespace.

We instruct the compiler that if it cannot find a suitable **swap** function for the variables then it should consider the **std::swap** function as a backup.

In these "blueprints" for a pair, we really don't know the types **T1** and **T2** and we cannot assume the best swaps are part of the **std** namespace.

Class Templates VI

In defining outside of the class, we must specify again that we are referring to a template and explicitly list names for the parameters. Once within the class scope, we can then refer to variables and the class itself without the template parameters.

```
template<typename A, typename B> // defined outside class
pair<A,B>::pair(const A& a, const B& b) : first(a), second(b) { }
```

We have written a default constructor and another enabling direct initialization. The compiler will automatically generate the copy/move constructors, copy/move assignment operators, and a destructor.

Class Templates

With our own **pair** templated class, we can define our own **make_pair** function.

```
template<typename A, typename B>
pair<A, B> make_pair( const A& first, const B& second) {

   // return pair constructed from parameters
   return pair<A,B>(first, second);
}
```

Class Templates

To construct a **pair** from its template class type directly, we need the explicit listing of its variable types:

```
// first will be 0, second will be ""
pair<int, std::string> pair1;

// first will be 3, second will be "blue"
pair<int, std::string> pair2(3,"blue");
```

Class Templates

On the other hand, the **make_pair** can deduce its input types:

```
auto pair3 = make_pair(10u, 2.26); // will be pair<unsigned, double>
auto pair4 = make_pair(20u, 7.2);
```

Above, we did not need to specify the type of the pairs: **make_pair** figures that out and returns an appropriate type.

```
swap(pair3, pair4);

std::cout << pair3.first << " " <<pair3.second << '\n';
std::cout << pair4.first << " " <<pair4.second << '\n';
```

```
20 7.2
10 2.26
```

Class Templates

Consider a function that uses the aliases/typdefs from our class:

```
/**
This function prints the sizes of the types stored in a pair
@tparam P the type of pair
@param p the object
*/
template<typename P>
void print_pair_sizes( const P& p ) {
   std::cout << sizeof( typename P::first_type) <<","
      <<sizeof( typename P::second_type);
}
```

Recall that **sizeof** specifies how many bytes a value or data type takes up.

Class Templates

```
template<typename P>
void print_pair_sizes( const P& p ) {
   std::cout << sizeof( typename P::first_type) <<","
      <<sizeof( typename P::second_type);
}
```

On Visual Studio 2017, if we run:

```
pair<bool,double> foo(true,3.14159);
print_pair_sizes(foo);
```

We generate:

```
1,8
```

because a **bool** is given 1 byte and a **double** 8 bytes.

Class Templates

```
template<typename P>
void print_pair_sizes( const P& p ) {
   std::cout << sizeof( typename P::first_type) <<","
      <<sizeof( typename P::second_type);
}
```

This is a rather contrived example to illustrate how the typedefs within the pair class can be used.

The **typename** is necessary as the compiler needs to know we're talking about a data type and not a static member variable when we use the **::**. In the function we wrote, we didn't even tell it that **P** was a **pair** so it is clueless.

Welcome to generic programming!

Class Templates

Unlike for function templates, class templates require an explicit listing of the template parameters (unless default values are given for example).

That's why we don't just write:

```
std::vector v = {1, 2, 3}; // ERROR: template type required!
```

Array Class

To illustrate further template syntax and methods, we will write an **Array** class with limited functionality within a **basic** namespace.

Array will...

- have a nested class **ArrayView**, an object that has no ownership of the **Array** but references a subset of the array;
- store a static C-style array of templated type **T** and **size_t** elements;
- have a default constructor that value initializes all members;
- have **subscript operators**, overloaded on const;
- have a **get_vew** function to make an **ArrayView**; and
- have a **coerce** function to coerce an argument into a **T** and place it in the array.

ArrayView will...

- have a **print** function to print all the elements.

Array Class

Let **Foo** be defined by:

```
struct Foo { explicit operator int() const { return 8; } };
```

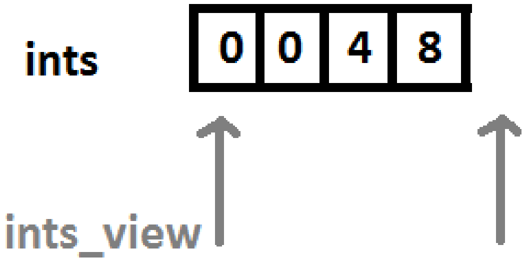
Array use in **main**:

```
// stores 4 zeros
basic::Array<int,4> ints;

// make 3rd item 4
ints[2] = 4;
// make 4th item an 8 (Foo{} made into 8)
ints.coerce(3, Foo{} );

// get a view of the array
basic::Array<int,4>::ArrayView ints_view = ints.get_view();
// call print on it
print(ints_view); // prints "0 0 4 8"
```

Array Class

The **Array** just stores a static array.

The **ArrayView** stores two pointers: to the first, and just past-the-end.



The coding implementation follows.

Array Class I

```
namespace basic { // opening namespace

  /**
  @class Array serves as an example class for a templated array class.
  It wraps around a normal C-style array to become a class-object.
  @tparam T the type being stored
  @tparam N the number of elements
  */
  template<typename T, std::size_t N>
  class Array {
  private:
    T values[N];

    // to be continued ...
```

The only member is a static array.

Array Class II

```
public:
    // some typedefs
    using value_type = T;
    using reference = T&;
    using const_reference = const T&;

    // nested class
    class ArrayView;

    // to be continued ...
```

It is common in the C++ container classes to use semantic naming for the template parameter types, hence the **using**s.

We have also declared that **ArrayView** is a nested class within **Array**.

Array Class III

```
// default constructor
constexpr Array() : values{} {}

// to be continued ...
```

We initialize the array with an empty initializer list. This will result in all entries of the array being value initialized: class types will be default constructed and primitive types will be 0.

This is relevent otherwise the values cannot be constant expressions. Default initialized values of primitive type could be anything...

Array Class IV

```
constexpr const_reference operator[](std::size_t i) const {
    return values[i];
}

constexpr reference operator[](std::size_t i) {
    return values[i];
}

// to be continued ...
```

We overloaded the subscript operator on const...

The use of **std::size_t** is often not necessary. But sometimes with limited header files included, **size_t** may not be defined. Or it might only be defined within **std::** but not without the qualification.

Array Class V

```
template<typename Type>
constexpr void coerce(std::size_t i, const Type& val);

// to be continued ...
```

We only declare this **coerce** function. We will define it outside the class.

Array Class VI

```
    ArrayView get_view(std::size_t begin_index = 0,
        std::size_t past_end_index = N) const &;

}; // END OF ARRAY INTERFACE

// to be continued ...
```

We declare the **get_view** function. It only works on lvalues: an **ArrayView** of an rvalue could reference invalid memory!

Note it has default arguments and we can refer to the template argument **N** like a normal variable, just like we can with **T** being a data type.

The function is not **constexpr**: simply put, to implement the **ArrayView** class, the values of pointers to non-static entities are needed, i.e. those referencing the **Array** class, and this cannot be done with **constexpr**.

Array Class VII

```
// defining the ArrayView class now
template<typename T, std::size_t N>
class Array<T, N>::ArrayView {
private:
   const T* first; // its first entry
   const T* past_end; // past its last entry

public:
   ArrayView(const T* _first, const T* _past_end);

   // to be continued ...
```

Array Class VIII

The **ArrayView** stores two pointers: one to the start of its range, and one past-the-end.

In defining this nested class outside of the class, we have to define it within the scope of **Array.** But because of the templating, we also need to give the template parameters!

Array Class IX

```
  friend void print(const ArrayView& av) {
    std::cout << "[ ";
    // print from first but do not include past_end!
    for (const T* it = av.first; it < av.past_end; ++it) {
      std::cout << *it << " ";
    }
    std::cout << "]";
  }

}; // END OF ARRAYVIEW INTERFACE

// to be continued ...
```

Array Class X

For technical reasons, separately declaring and defining functions that are friends of templated classes is difficult.

The matters are made worse when said functions are actually friends of *nested classes within a templated class*... The compiler and linker can get confused.

One solution we adopt, because it involves the least bizzarre syntax and is quite general, is to simply **define all friend functions, including operator overloads, of template classes or their nested classes directly in the class body.**

Array Class XI

Then, due to other issues called **non-deduced contexts**, we will also choose to make non-member functions of nested classes of templates friends, even if they do not need to be. This isn't elegant but it ensures the functions are accessible via ADL, although not explicit namespace qualified lookup.

Thus, at the end of the day,

print(some_view); // will compile: found via ADL

// will not compile: not declared at namespace scope
basic::print(some_view);

Array Class XII

```
// define ArrayView constructor

template<typename T, std::size_t N>
Array<T, N>::ArrayView::ArrayView(const T* _first,
   const T* _past_end) : first(_first), past_end(_past_end) {}

// to be continued ...
```

Note that in this definition we need the outer scope **Array<T,N>**, which is templated, before we can define the constructor function **ArrayView::ArrayView**.

Array Class XIII

```
// define the coerce

template<typename T, std::size_t N>
template<typename Type>
constexpr void Array<T, N>::coerce(std::size_t i, const Type& val) {
    values[i] = static_cast<T>( val );
}

// to be continued ...
```

The **coerce** function itself is templated. But it appears within the templated class. Hence we must write **both levels of templates** in defining the function outside the class!

We employ a **static_cast** to invoke (possibly) explicit conversion operators for different class types.

Array Class XIV

```
template<typename T, std::size_t N>
typename Array<T, N>::ArrayView Array<T,N>::get_view(std::size_t
    begin_index, std::size_t past_end_index) const & {
    return ArrayView(values + begin_index,
        values + past_end_index);
}

} // CLOSING NAMESPACE
```

The return type is marked with **typename**! This is similar to the **typename** we used for **typename P::first** for the templated **pair** class. Once again it is necessary to instruct the compiler we are referring to a data type, **Array<T,N>::ArrayView**, not a static member variable.

Array Class XV

Actually, we can avoid the need for **typename** here by using the *auto* function notation. The functions below are the same:

return_type function(types) { / ... */ }*
auto function(types) -> return_type { / ... */ }*

And we can write

```
template<typename T, std::size_t N>
auto Array<T, N>::get_view(std::size_t begin_index,
    std::size_t past_end_index) const & -> ArrayView {
    return ArrayView(values + begin_index, values + past_end_index);
}
```

since we passed to the "scope" of **Array<T,N>** where ArrayView is known.

Array Class

Remark:

It is obviously a lot easier to just define as much as possible within the outermost class body, including fully defining the inner classes with their complete interface. This is just an illustration of the many different situations that can arise so things were defined inside/outside to bring about different scenarios.

Templates with Default Arguments

Just as with functions that admit default arguments such as

```
void foo(int x, double y = 4.3, char c = ' ');
```
,

a template can have default parameter types or values (integer types only!):

```
template<typename T = int, size_t sz = 10>
class DefaultedArray { /* ... */ };
```

Templates with Default Arguments

Given the templated **DefaultedArray** from the previous page, we can construct instances of the template as follows:

```
DefaultedArray< > a1; // will be of type DefaultedArray<int, 10>
DefaultedArray<double> a2; // will be of type DefaultedArray<double, 10>
DefaultedArray<double, 500000> a3; /* will be of type
    DefaultedArray<double, 500000> */
```

Even when all parameters have default values, the angled brackets < > are still necessary for the instantiation of the default template.

As with functions, default values/types for templates are given from right to left.

Setting Policy

Understanding templates is important in **setting policy**: specifying, for example, how elements of a **std::set** are sorted. By default they are sorted by a templated class type **std::less<T>** with a call operator to determine if the left argument is less than the right, based on **operator<**.

The callable class types **std::less**, **std::greater**, and others, are defined in the **<functional>** header.

Setting Policy

```
template <typename T>
struct less {
   bool operator()(const T& lhs, const T& rhs) const { // call operator
      return lhs < rhs;
   }
};
```

Likewise, there is also

```
template <typename T>
struct greater {
   bool operator()(const T& lhs, const T& rhs) const { // call operator
      return lhs > rhs;
   }
};
```

Setting Policy I

We consider a warm-up example in setting policy. It illustrates the syntax and usage. Suppose we wanted a class **max_only** that will only track the "largest" value it has seen and nothing else. But we want to define "largest" in different ways. For example:

```
max_only<int> i{3};
i.compare(7);
i.compare(1);
i.get(); // 7: "largest" is regular largest here

// replace normal < by >:
max_only<double, std::greater<double> > d{3.14};
d.compare(9.8);
d.get(); // 3.14: "largest" is smallest here
```

Here is a bare-bones implementation with nothing fancy.

Setting Policy II

```
template<typename T, typename CallableType = std::less<T> >
class max_only{
private:
   T value; // max it has seen
   CallableType less_than; // means of comparison

// to be continued ...
```

If nothing is specified for the second template argument then **CallableType** will be defaulted to the *type* **std::less<T>.** This is a perfectly valid data type.

The member variable **less_than** is of whatever type **CompareType** happens to be. We will use **less_than** in lieu of regular "<".

We want freedom in what **CallableType** is so users can give *any callable type* as a comparison: lambda, function (pointer), callable class, etc.

Setting Policy III

```
public:
    constexpr max_only(const T& init_value,
        const CallableType& _less_than = CallableType() ) :
        value(init_value), less_than(_less_than) {}

    // to be continued ...
```

The user needs to specify an initial value for the class to store, but the second constructor argument is optional.

If nothing is specified for the second argument **_less_than**, *which is of type* **CallableType**, we provide a default *value* of that type, **CallableType()**.

Then, whatever value **_less_than** happens to be from the constructor arguments, we assign that to the member variable **less_than.**

Setting Policy IV

```
    constexpr void compare(const T& other){
        if( less_than(value,other) ){ // if new value "larger", replace it
            value = other;
        }
    }

    constexpr const T& get() const {
        return value;
    }
};
```

Instead of comparing two values with <, we use **less_than** which can be called instead.

Setting Policy

The same idea holds for **std::set.** Ignoring allocator details, effectively an **std::set** looks like:

```
// BY DEFAULT CallableType IS std::less<T>
template<typename T, typename CallableType = std::less<T> >
class set {
private:
   node* root; // pointer to a node
   CallableType less_than; // member lessThan compares

   // other stuff ...
public:
   // BY DEFAULT comp is set to compareType()
   set(const CallableType& _less_than = CallableType() ) : root(nullptr),
      less_than( _less_than ) /* maybe more stuff */ { }

   // other stuff ...
};
```

Setting Policy

Consider the code:

```
std::set<int> s1;
```

Then in construction,
T = int,
CallableType = std::less<int>,
and **less_than** is set to the value **std::less<int>()**.

From **begin** to **end** elements would be sorted like: 3, 5, 8, 22, 40, 108, ..., i.e. ascending.

Setting Policy

Consider the code:

```
std::set<double, std::greater<double> > s2;
```

Then in construction,
T = double,
CallableType = std::greater<double>,
and **less_than** is set to **std::greater<double>()**.

From **begin** to **end** elements would be sorted like: 98.6, 41.2, 18.8, 4.01, ..., i.e. descending.

Setting Policy

Consider the code:

```
auto bySize = [](const std::string& x, const std::string& y)
    ->bool { return s.size() < y.size(); };

std::set<std::string, decltype(bySize)> s3(bySize);
```

Then in construction,
T = std::string,
CallableType = *whatever type the lambda is*,
and **less_than** is set to **bySize.**

From **begin** to **end** elements would be sorted like: "dog", "kitty", "elephant", ..., i.e. ascending order by size.

Setting Policy

Suppose the function compareSquares is defined as such:

```
bool compareSquares(int x, int y) { return x*x < y*y; };
```

Consider the code:

```
std::set<int, bool(*)(int,int)> s4(compareSquares);
```

Then in construction,
T = int,
CallableType = bool(*)(int,int), a function pointer,
and **less_than** is set to **compareSquares.**

From **begin** to **end** elements would be sorted like: 0, -1, 2, -3, 4, ..., i.e. ascending order by size of the squared values.

Declaring/Defining with Default Arguments

Remark: as with functions with default arguments, the default arguments should only appear in the original declaration. For example,

```
void foo(int i=7); // declaration in isolation
void foo(int i=7) {} // INCORRECT separate definition
void foo(int i) { } // CORRECT separate definition
```

and

```
template<typename T = double, size_t N = 42>
struct Bar { void baz() const; /* declaration */ };

template<typename T = double, size_t N = 42> // INCORRECT
void Bar<T>::baz() const { }

template<typename T, size_t N> // CORRECT
void Bar<T>::baz() const { }
```

Template Definitions in Header Files

For ordinary class and function definitions, it is nice practice to separate the interface/declarations (in a header file) from the implementations/definitions (in a cpp file).

The traditional nice coding practice approach of separating declaration and definitions does not work under ordinary circumstances with templates.

Given appropriate declarations, when a compiler comes across a specific instance of a template, it can instantiate the template without a definition, leaving the hard work of finding the definition to the linker. In another cpp file, the definition of the template may appear, but without any specific instances of the template given, the compiler will not generate code to define an instance of the template. The linker will then not be able to find the definition.

Template Definitions in Header Files

Header.h

```
template<typename T> T foo(); // #ifndef stuff not shown
```

Foo.cpp

```
// The compiler does not generate any code from this file:
#include "Header.h"
template<typename T> T foo() { return T(); }
```

main.cpp

```
// The compiler instantiates foo<double>() but has no definition
#include "Header.h"
int main() {
   foo<double>();       return 0;
}
```

LINKER ERROR: unresolved externals.
unresolved external symbol double foo<double>(void) referenced in main

Template Definitions in Header Files

In practice, many programmers declare and define the templates together within a header file.

Header.h

```
// Declares and defines the template: #ifndef stuff not shown
template<typename T> T foo() { return T(); }
```

main.cpp

```
// The compiler instantiates foo<double>() and uses definition in header
#include "Header.h"
int main() {
   foo<double>();
   return 0;
}
```

Remark: we could have declared the template function and later defined it *within the same header file*; we don't have to declare/define simultaneously.

Variadic Templates

Consider a **print** function that will accept a list of comma separated arguments and print them to the console with **operator<<.** This could get messy and rather long...

```
// with just 1 argument
template<typename T>
void print(const T& t) { std::cout << t; }

// ...

// with 5 arguments
template<typename T1, typename T2, typename T3, typename T4,
   typename T5> void print(const T1& t1, const T2& t2, const T3& t3,
   const T4& t4, const T5& t5) {
   std::cout << t1 << t2 << t3 << t4 << t5 ;
}
```

Variadic Templates

Using the elipsis **...** syntax, we can write a template function

```
template<typename ... Tvals> // take any number of arguments
void print(const Tvals& ... args); // print any number of arguments
```

accepting a variable number of inputs (we call it **variadic**).

Tvals is called a **template parameter pack**, a collection of *zero or more template arguments*.

And **args** is called a **function parameter pack**, a collection of *zero or more input parameters*.

Variadic Templates

We will define the templated **void print(const Tvals& ... args);** recursively.

We will strip off one parameter at a time until there are no parameters left (a base case).

We will make use of a helper function **display**.

Variadic Templates

```
void display(){} // base case: do nothing with no inputs

/* pick out the first item to print and recurse on the remaining parameter
   pack */

template<typename Tfirst, typename ... Trest>
void display(const Tfirst& param1, const Trest&... params){
   std::cout <<param1;
   display(params...);
}

// to print the inputs, call the functions above
template<typename ... Tvals>
void print(const Tvals& ... parameters){
   display(parameters...);
}
```

Variadic Templates

In calling

```
print("hel", true, 0, '!');
```

when **display** is called,

- first **"hel"** is **param1** and **params** is **true, 0, '!'**.
- In the next call, **true** is **param1** and **params** is **0, '!'**.
- In the next call, **param1** is **0** and **params** is **'!'**.
- In the next call, **param1** is **'!'** and **params** is **void**.
- This **void** cannot be stripped off into a **param1** so it is then passed to the base case in the final call.

Output:

```
hel10!
```

(note without **std::boolalpha**, **true** is displayed by **std::cout** as **1** and **false** as **0**.)

Variadic Template Patterns

The placement of the **...** is quite important. Suppose that **params...** is a parameter pack and suppose that **f** is a function or constructor that takes an appropriate number of inputs and **g** is a function that accepts a single input of appropriate type. Then

f(g(params)...);

actually means
f(g(params$_1$**), g(params**$_2$**), ... g(params**$_n$**));**

In other words: evaluate **g** with every single input of the parameter pack and then pass all those outputs to **f**.

Variadic Template Patterns

On the other hand,

g(params...);

means to evaluate **g** with the parameter pack.

Emplace

We can write a variadic **emplace** function for our **Array** class to update the value of the a given element by passing an arbitrary number of arguments!

```
class Array {

   // all the other stuff ...

   template<typename... Types>
   constexpr void emplace(size_t i, const Types&... args) {
      values[i] = T(args...); // pass all the parameters to make a T
   }
};
```

Emplace

For example in **main**:

```
basic::Array<std::string, 5> strings;

strings.emplace(2,"hi"); // now strings[2] == "hi"
strings.emplace(2); // now strings[2] == "": made string from 0 arguments
strings.emplace(4,10,'b'); // now strings[4] == "bbbbbbbbbb"
```

Forwarding References and Perfect Forwarding

There is a small problem of inefficiency in how we constructed objects via **emplace**... what if some of the parameters in the function parameter pack were **r-values** (like the **std::string("...")**)?

If a parameter in the pack is an **r-value**, it still gets treated as though it were a **l-value** because *by giving it a name, it is an lvalue*, even if the parameter passed is an rvalue.

Forwarding References and Perfect Forwarding

Ideally, we'd like to know whether a given parameter is a **l-value** or **r-value** and use it differently, possibly taking advantage of move semantics, etc.

This leads us to consider **T&&** in template type deduction. It does not mean r-value reference!

Danger! Things are about to get weird. When dealing with templates, it is possible for the template type and the parameter type to differ.

Forwarding References and Perfect Forwarding

```
template<typename T>
void f(T t);

double d = 2.2;
const double cd = 4.4;
const double & dr = d;
f(d); // T is double, t is double
f(cd); // T is double, t is double
f(dr); // T is double, t is double
f(6.6); // T is double, t is double
```

We essentially have

double t = d;
double t = cd;
double t = dr;
double t = 6.6;

Forwarding References and Perfect Forwarding

```
template<typename T>
void g(const T& t);

double d = 2.2;
const double cd = 4.4;
const double &dr = d;
g(d); // T is double, t is const double&
g(cd); // T is double, t is const double&
g(dr); // T is double, t is const double&
g(6.6); // T is double, t is const double&
```

We essentially have

const **double**& t = d;

const **double**& t = cd;

const **double**& t = dr;

const **double**& t = 6.6;

Forwarding References and Perfect Forwarding

```
template<typename T>
void h(T& t);

double d = 2.2;
const double cd = 4.4;
const double& dr = d;
h(d); // T is double, t is double&
h(cd); // T is const double, t is const double&
h(dr); // T is const double, t is const double&
```

We essentially have

double& t = d;

const double& t = cd;

const double& t = dr;

And we cannot have

```
h(6.6);
double& t = 6.6; // ERROR: cannot bind lvalue to pr value
```

Forwarding References and Perfect Forwarding

```
template<typename T>
void i(T* t);

double d = 2.2;
const double cd = 4.4;
const double& dr = d;
i(&d); // T is double, t is double*
i(&cd); // T is const double, t is const double*
i(&dr); // T is const double, t is const double*
```

We essentially have

double* t = & d;

const double* t = &cd;

const double* t = &dr;

Forwarding References and Perfect Forwarding

```
template<typename T>
void j(const T* t);

double d = 2.2;
const double cd = 4.4;
const double& dr = d;
j(&d); // T is double, t is const double*
j(&cd); // T is double, t is const double*
j(&dr); // T is double, t is const double*
```

We essentially have

const **double*** t = &d;

const **double*** t = &cd;

const **double*** t = &dr;

Forwarding References and Perfect Forwarding

For the situation

```
template<typename T>
void k(T&& t);
```

when

- the input is an (const) lvalue of type **foo**, then **T** is an lvalue reference (to const) for **foo**, as is **t**;
- if the input is an rvalue of type **foo**, then **T** is a **foo** and **t** is an rvalue reference to **foo.**

These are the rules of **reference collapsing**.

Forwarding References and Perfect Forwarding

```
template<typename T>
void k(T&& t);

double d = 2.2;
const double cd = 4.4;
const double & dr = d;

k(d); // T is double&, t is double&
k(cd); // T is const double&, t is const double&
k(dr); // T is const double&, t is const double&
k(6.6); // T is double, t is double&&
```

We essentially have

double& t = d;

const double& t = cd;

const double& t = dr;

double&& t = 6.6;

Forwarding References and Perfect Forwarding

The behaviour here is more interesting. Note that l-values will be permissible parameters but we ordinarily cannot bind r-value references to l-values!

T&& here is called a **forwarding reference**.

Accepting parameters of all types, Scott Meyers, before the term "forwarding reference" was part of the C++ Standard, coined the term **universal reference**.

Forwarding References and Perfect Forwarding

This type deduction pheoneman only arises when types are deduced through a function!

```
template<typename T>
struct Foo {
    void bar(T&& t) const; // no deduction: T is specified by class template!
    void baz(T& t) const; // no deduction
};
```

The above is valid code but the **T&&** really does just mean rvalue reference to **T**.

Forwarding References and Perfect Forwarding

Giving an rvalue reference a name has the unfortunate effect of making it into an lvalue.

```
template<typename T>
void foo(T&& t) { // not optimal
   bar(t); // t always used as an lvalue: goodbye move semantics
}
```

This problem is resolved with the **std::forward** templated function from **<utility>**.

```
template<typename T>
void foo(T&& t) { // T could be lvalue reference or not
   bar(std::forward<T>(t)); // t may be treated as an rvalue or lvalue
}
```

Forwarding References and Perfect Forwarding

The template type of **std::forward** cannot be inferred by its argument and thus the argument must be given explicitly. To properly forward, its template type must be of the value category **type** or **type&.**

When its template argument is...

- a non-reference type (**type**), an rvalue reference will be forwarded **type&&**;
- an lvalue reference type (**type&** or **const type&**), an lvalue reference will be forwarded (as **type&** or **const type&**).

Being able to forward rvalues as rvalues and lvalues as lvalues is **perfect forwarding.**

Forwarding References and Perfect Forwarding

```
std::string str = "moo"; // str is lvalue

std::forward<std::string>(str); // will output std::string&&, rvalue
std::forward<std::string&>(str); // will output std::string&, lvalue
std::forward<const std::string&>(str); /* will output const std::string&,
   lvalue */
```

Forwarding References and Perfect Forwarding

```
struct FWD {
   FWD(std::string&&) : c('R') {}
   FWD(const std::string&) : c('L') {}
   char c;
};
template<typename T>
void deduce(T&& t){
   std::cout <<FWD(std::forward<T>(t)).c;
}

// in main
std::string s1;
const std::string s2 = s1;
deduce(s1);
deduce(s2);
deduce( std::string() );
```

Forwarding References and Perfect Forwarding

s1 is an lvalue of type **std::string** and the deduced type of **T** is **std::string&**. Thus, a **std::string&** is forwarded to the constructor accepting an lvalue reference.

s2 is an lvalue of type **const std::string** and the deduced type of **T** is **const std::string&**. Thus, a **const std::string&** is forwarded to the constructor accepting an lvalue reference.

std::string() is a prvalue and the deduced type of **T** is **std::string**. Thus, a **std::string&&** is forwarded to the constructor (it's an xvalue) accepting an rvalue reference.

Forwarding References and Perfect Forwarding

The compiler is very smart: it will allow code (in templates) that otherwise looks wrong when the types are explicit.

```
template<typename T>
void smart(T&& t) {
   const T& s = t;
}

// ...
const double d = 0.;
smart(d); // fine!
```

T is deduced as **const double&** and **const T&** just becomes **const double&**, not **const const double& &** which would be invalid code to write.

Forwarding References and Perfect Forwarding

The forwarding applies equally well to variadic templates.

A better **Array::emplace** function would be:

```
// within interface ...

template<typename... Types>
constexpr void emplace(std::size_t i, Types&&... args) {
   // forward parameters to make a T
   values[i] = T(std::forward<Types>(args)...);
}
```

Void Pointers and sizeof

A pointer to any type can be stored within a **void***, pointer to **void.** These pointers must be first cast before they can be used.

```
int i = 7;
void *vp = &i;
std::cout << *(static_cast<int *>(vp));

std::cout << *vp; // ERROR: type unknown
```

Some functions accept **void*** inputs to allow any type of pointer as an input; we may also obtain **void*** outputs.

Void Pointers and sizeof

For any type name or variable, we can calculate its number of bytes with the **sizeof** operator. On Visual Studio:

```
int x = 47;
long double y = 1.2e+211;
std::string z("abc");

sizeof(x); // 4: an int is given 4 bytes
sizeof(y); // 8: a long double is given 8 bytes
sizeof(z); // 28: a std::string takes up 28 bytes
sizeof(unsigned short); // == 2: unsigned short is given 2 bytes
sizeof(false); // == 1: a bool takes up 1 full byte
```

operator new, new expression, operator delete, and delete expression

The **new expression** can be used in contexts such as:

```
int *ip = new int; // ip points to default initialized int-value
int *ip2 = new int(); // ip points to 0-initialized int-value
std::string *sp = new std::string("hey"); // sp points to string with "hey"

double *dp = new double[42](); // dp points to array of 42 0-double values
std::string *sp2 = new std::string[100](); /* sp2 points to dynamic array of
   100 default strings */
```

This then creates heap memory to store the objects. We can pass construction parameters to a single heap object or allow for default initialization. For dynamic arrays, we cannot pass parameters and all of the values in the array must be default initialized.

operator new, new expression, operator delete, and delete expression

operator new is an operator that allocates memory without constructing objects/values. We consider

```
void *operator new(size_t sz); // allocates memory of sz bytes
```

The function returns a **void*** to the first address of the memory segment.

operator new, new expression, operator delete, and delete expression

The **placement new** gives us freedom to construct an object at a given address given its construction parameters. The syntax is:

new (pointer) type (initializers);

For example:

```
void *vp = operator new(100*sizeof(std::string));
new (vp) std::string(10, 'b');
```

Above, we constructed space for 100 **std::string**s and created a **std::string** of **"bbbbbbbbbb"** at the beginning.

This is much like how the allocators work: first, memory is allocated, then objects are constructed.

operator new, new expression, operator delete, and delete expression

When the **new expression** is used, three things happen:

- **operator new** is called with appropriate arguments,
- the object is initialized with construction parameters, and
- a pointer is returned to the first element.

Something similar happens with the **new[] expression.**

operator new, new expression, operator delete, and delete expression

operator delete is an operator that frees memory that had previously been allocated by **operator new**.

```
void operator delete(void *) noexcept; // deallocates memory
```

The function is marked as **noexcept** as it will not throw exceptions; this helps the compiler to generate better code if it knows exceptions won't crop up in function.

operator new, new expression, operator delete, and delete expression

When the **delete expression** is used such as in writing

```
delete ip;
```

two things happen:

- the destructor is called on the pointed-to object and
- the memory is freed by the invocation of **operator delete.**

Something similar happens with the **delete [] expression.**

Allocator Class

We can now write a simple allocator class. Ultimately an allocator is very simple:

```
template<typename T>
struct Allocator {
   // allocates a contiguous block to store T's
   T* allocate(size_t size) const;

   // constructs a T given its parameters of construction
   template<typename ... Types>
   void construct(void *vp, Types&&... params) const;

   // destroys the pointed to T object by invoking the destructor
   void destroy(T* tp) const { tp->~T(); }

   // deallocates the block of memory beginning at tp with operator delete
   void deallocate(T* tp) const { operator delete(tp); }
};
```

Allocator Class

```
template<typename T>
T *Allocator<T>::allocate(size_t size) const {
   try{
      void *vp = operator new(sizeof(T)*size);
      return static_cast<T*> (vp);
   }
   catch (std::bad_alloc &bad){
      std::cerr <<"allocation error";
      throw;
   }
}

template<typename T>
template<typename ... Types>
void Allocator<T>::construct(void *vp, Types&&... params) const {
   new (vp) T(std::forward<Types>(params)...);
}
```

Allocator Class

The destructor of a **T** object is explicitly invoked here because **destroy** is supposed to destroy the object; under most ordinary circumstances, we should never call a destructor manually.

The deletion process is simple: we just invoke **operator delete**.

Allocator Class

In allocating the memory, we may not be able to allocate enough memory so we use a **try** and **catch** block. If the memory allocation fails, we print the error message and throw the error again.

We captured the parameter pack as (variadic) universal reference and use these parameters to construct a **T** object at **vp**.

Aside: auto&&

The **auto** keyword has similar rules as template type deductions:

```
int i = 4;
auto&& j = i; // j is int& to i
auto&& k = 13; // k is an int&& storing 13

double d = 0.4;
const double e = 0.5;
auto& f = d; // f is double&
auto& g = e; // g is const double&
```

Aside: the make_unique and make_shared Functions

The **make_unique** and **make_shared** functions can be understood in terms of variadic templates and forwarding references. For example:

```
namespace std {
   template<typename T, typename ... Types>
   unique_ptr<T> make_unique(Types&& ... values) {
      return unique_ptr<T>( new T( std::forward<Types(values) ... ));
   }
}
```

Aside: Setting Policy for Shared Pointers

By default, a **std::shared_ptr** does not have a special means of managing dynamic arrays but with the knowledge of setting policy, we can create **std::shared_ptr** objects to manage dynamic arrays.

An excerpt of the interface of **shared_ptr**:

```
template <typename T>
class shared_ptr {

   /* This constructor accepts a pointer and a Deleter: function taking T*
         input and freeing the memory */

   template <typename U, class Deleter>
   shared_ptr( U* ptr, Deleter d);
};
```

Without a deleter specified, **delete** is called on a **U*** object. This is bad if the pointer was **new []**ed instead of **new**ed...

Aside: Setting Policy for Shared Pointers

Consider the struct:

```
template<typename T>
struct Del {
   void operator()( T* tp ) const { delete [] tp; }
};
```

Then we can initialize a shared_ptr managing a dynamic array now, for example:

```
std::shared_ptr<unsigned> shareC(new unsigned[10],
   Del<unsigned>() );
```

By specifying an object that can be called in lieu of the default call to **delete**, we no longer need to fear undefined behaviour or memory leaks. The **delete[]** expression will be called.

Summary

- Templates allow general code to be written without having to manage many nearly identical instances of writing a class, function, etc.
- Both classes and functions can be templatized; for a class, the template parameters must be specified (or given default values).
- A template only gives the compiler instructions for generating functions/classes; only those used in the code are created.
- Generally templates are declared and defined in one file.
- Proper template instantiations allow us to **set policy** for classes.
- **Type deduction** occurs in templates where a parameter has an inferred type.
- A **universal reference** can accept both l-values and r-values and with **perfect forwarding** as in **std::forward**, a type-aware form of the variable can be used.
- A **void*** can point to anything.
- **operator new** and **operator delete** work behind the scenes for allocators and **new** and **delete** expressions.

Exercises I

1. Write a templated **sum** function that returns the sum of all elements of a **std::vector** of arbitrary type: throw an exception if the **std::vector** is empty.
2. Write a templated **max** function that returns the maximum of all elements of a **std::vector** of arbitrary type, that accepts a binary predicate as a second argument for a “less than” replacement. Throw an exception if the **std::vector** is empty.
3. Repeat the previous two exercises, now where the container type is also arbitrary! You may need **decltype** and/or a trailing return type!
4. Write a templated **triplet** class that extends a **std::pair** in the obvious way: it has members **first**, **second**, and **third.** Include a **make_triplet** function as well.

Exercises II

5. Write a templated function **do_for_each** that accepts a functor for an action to take upon an lvalue and an *arbitrarily* long list of arguments and performs that functor on all of them. For example:
   ```
   int i = 7, j = 8, k = 9;
   do_for_each( [](int& x) ->
     void { x *= 2; }, i, j, k);
   // now i==14, j==16, k==18
   ```
6. Write a templated **queue** class (so items can be inserted only at the back, removed only from the front - maybe use a **std::deque** under the hood). Include an **emplace** function and a **nested iterator** class to go from the front to the back of the **queue**.
7. Implement either **bubble_sort** or **selection_sort** given two iterators of an arbitrary container and an optional functor to replace **operator<**: assume the first iterator must come “before” the second iterator.

Exercises III

8. Write a variadic **toString** function: it accepts an arbitrary list of arguments of arbitrary type, converts them all to **std::string**s with an **std::ostringstream**, and concatenates them, returning the concatenation.
9. What are **forwarding references** and **perfect forwarding**? How are they helpful?

Exercises IV

10. Consider the following functions:
    - **template<typename T> void foo1(T t);**
    - **template<typename T> T& foo2(T& t);**
    - **template<typename T> const T& foo3(const T& t);**
    - **template<typename T> T foo4(T&& t);**
    - **template<typename T> const T& foo5(T* t);**

    along with the following code:

    ```
     std::string s(“hi”);
    const std::string t = s;
    ```

    Determine which arguments to each of the **foo**s is valid; for those that are valid, state the type of **T** and **t**:
    - **s**
    - **t**
    - **std::string()**
    - **std::move(s)**
    - **std::forward<std::string>(s)**
    - **std::forward<const std::string&>(s)**
    - **&s**
    - **&t**

Exercises V

11. What is a **void*** (pointer to void)?
12. How do **operator new** and **operator delete** differ from the **new expression** and **delete expression**?
13. Using **operator new** and the **placement new expression**, write a templated function **New** that allocates space for an object of an arbitrary type and constructs it variadically in place. For example:

    ```
    std::string *str = New<std::string>(10,'b');
    // str points to "bbbbbbbbbb" on the heap
    ```

14. Write a templated function **Delete** that destructs and object and frees the memory. For example:

    ```
    Delete<std::string>(str);
    // destructs the "bbbbbbbbbb" from above, frees
    the memory
    ```