# Templates

Object-Oriented Programming with C++

# Why templates?

# Why templates?

- Suppose you need a list of X and a list of Y
  - The lists would use similar code
  - They differ by the type stored in the list

# Why templates?

- Suppose you need a list of X and a list of Y
  - –The lists would use similar code
  - –They differ by the type stored in the list
- Choices
  - –Clone code
    - preserves type-safety
    - hard to manage

# Why templates?

- Suppose you need a list of X and a list of Y
  - –The lists would use similar code
  - –They differ by the type stored in the list
- Choices
  - –Clone code
    - preserves type-safety
    - hard to manage
  - –Require common base class
    - May not be desirable

# Why templates?

- Suppose you need a list of X and a list of Y
  - –The lists would use similar code
  - –They differ by the type stored in the list
- Choices
  - –Clone code
    - preserves type-safety
    - hard to manage
  - –Require common base class
    - May not be desirable
  - –Untyped lists
    - type unsafe

# Templates

- Reuse source code
  - –generic programming
  - –use types as parameters in class or function definitions

# Templates

- Reuse source code
  - –generic programming
  - –use types as parameters in class or function definitions
- Function Template
  - –Example: `sort` function

# Templates

- Reuse source code
  - –generic programming
  - –use types as parameters in class or function definitions
- Function Template
  - –Example: `sort` function
- Class Template
  - –Example: containers such as `stack,list,queue...`
    - `stack` operations are independent of the type of items in the stack
  - –template member functions

# Function templates

- Perform similar operations on different types of data.
- Swap function for two int arguments:

```
void swap ( int& x, int& y ) {
    int temp = x;
    x = y;
    y = temp;
}
```

- What if we want to swap floats, strings, Currency, Person?

# Example: swap function templates

```
template < class T >
void swap( T& x, T& y ){
    T temp = x;
    x = y;
    y = temp;
}
```

- The **`template`** keyword introduces the template
- The **`class T`** specifies a parameterized type name
  - class means any built-in type or user-defined type
- Inside the template, use T as a type name

# Function templates syntax

- Type parameters represent:
  - –types of arguments to the function
  - –return type of the function
  - –define variables within the function

# Template instantiation

- Generating a definition from a template class/function and template arguments:
  - –Types are substituted into template
  - –New body of function or class definition is created
    - syntax errors, type checking
  - –Specialization -- a version of a template for a particular argument(s)

# Example: using swap

```
int i = 3; int j = 4;
swap(i, j);  // use explicit int swap

float k = 4.5; float m = 3.7;
swap(k, m);  // instantiate float swap
std::string s("Hello");
std::string t("World");
swap(s, t);  // instantiate std::string swap
```

- A template function is an instantiation of a function template

# Interactions

- Only *exact* match on types is used

# Interactions

- Only *exact* match on types is used
- *No* conversion operations are applied
  - swap(int, int); // ok
  - swap(double, double); // ok
  - swap(int, double); // error!

# Interactions

- Only *exact* match on types is used
- *No* conversion operations are applied
  - swap(int, int); // ok
  - swap(double, double); // ok
  - swap(int, double); // error!
- Even *implicit* conversions are ignored

# Interactions

- Only *exact* match on types is used
- *No* conversion operations are applied
  - swap(int, int); // ok
  - swap(double, double); // ok
  - swap(int, double); // error!
- Even *implicit* conversions are ignored
- Template functions and regular functions coexist

# Overloading rules

- Check first for unique function match
- Then check for unique function template match
- Then implicit conversions on regular functions

```
void f(float i, float k) {};
template <class T>
void f(T t, T u) {};
f(1.0f, 2.0f);
f(1.0, 2.0);
f(1, 2);
f(1, 2.0);
```

# Function instantiation

- The compiler deduces the template type from the actual arguments passed into the function.
- Can be *explicit*:
  - for example, if the parameter is not in the function signature (older compilers won't allow this...)

```
template <class T>
void foo() { /* ... */ }
```

# Function instantiation

- The compiler deduces the template type from the actual arguments passed into the function.
- Can be *explicit*:
  - for example, if the parameter is not in the function signature (older compilers won't allow this...)

```
template <class T>
void foo() { /* ... */ }

foo<int>();  // type T is int
foo<float>();  // type T is float
```

# Class templates

# Class templates

- Classes parameterized by types
  - –Abstract operations from the types being operated upon
  - –Define potentially infinite set of classes
  - –Another step towards reuse!

# Class templates

- Classes parameterized by types
  - –Abstract operations from the types being operated upon
  - –Define potentially infinite set of classes
  - –Another step towards reuse!

- Typical use: container classes
  - `stack <int>`
    - is a stack that is parameterized over int
  - `list <Person*>`
  - `queue <Job>`

# Example: Vector

```
template <class T>
class Vector{
public:
    Vector(int);
    ~Vector();
    Vector(const Vector&);
    Vector& operator=(const Vector&);
    T& operator[](int);
private:
    T* m_elements;
    int m_size;
}
```

# Usage

```
Vector<int> v1(100);
Vector<Complex> v2(256);

v1[20] = 10;
v2[20] = v1[20];  // ok if int->Complex defined
```

# Vector members

```
template <class T>
Vector<T>::Vector(int size): m_size(size) {
    m_elements = new T[m_size];
}
template <class T>
T& Vector<T>::operator[](int index)
{
    if(index < m_size && index >= 0) {
        return m_elements[index];
    } else {
        …
    }
}
```

# A simple sort function

```
// bubble sort - don't use it!
template <class T>
void sort(Vector<T>& arr) {
    const size_t last = arr.size() - 1;
    for(int i=0; i<last; i++) {
        for(int j = last; j>i; j--) {
            if(arr[j] < arr[j-1]){
                // which swap?
                swap(arr[j], arr[j-1]);
            }
        }
    }
}
```

# Sorting the Vector

```
Vector<int> vi(4);
vi[0] = 4; vi[1] = 3; vi[2] = 7; vi[3] = 1;
sort(vi);  // sort(Vector<int>&)

Vector<string> vs(5);
vs[0] = "Fred";
vs[1] = "Wilma";
vs[2] = "Barney";
vs[3] = "Dino";
vs[4] = "Prince";
sort(vs);  // sort(Vector<string>&);
//NOTE: sort use operator< for comparison
```

# Templates

- Templates can use multiple types

```
template < class Key, class Value >
class HashTable {
    const Value& lookup (const Key&) const;
    void insert (const Key&, const Value&);
    …
}
```

# Templates

- Templates can use multiple types

```
template < class Key, class Value >
class HashTable {
    const Value& lookup (const Key&) const;
    void insert (const Key&, const Value&);
    …
}
```

- Templates nest – they're just new types!

```
Vector< Vector<double*> >
```

# Templates

- Templates can use multiple types

```
template < class Key, class Value >
class HashTable {
    const Value& lookup (const Key&) const;
    void insert (const Key&, const Value&);
    …
}
```

- Templates nest – they're just new types!

```
Vector< Vector<double*> >
```

- Type arguments can be complicated

```
Vector< int (*) (Vector<double>&, int) >
```

# Expression parameters

- Template arguments can be *constant* expressions
- Non-Type parameters
  - can have a default argument

```
template <class T, int bounds = 100>
class FixedVector {
public:
    FixedVector();
    T& operator[](int);
private:
    T elements[bounds];    // fixed-size array!
}
```

# Non-Type parameters

```
template <class T, int bounds>
T& FixedVector<T, bounds>::operator[] (int i){
    return elements[i];    //no error checking
}
```

# Usage: non-type parameters

- Usage
  - `FixedVector<int, 50> v1;`
  - `FixedVector<int, 10*5> v2;`
  - `FixedVector<int> v3;  // uses default`
- Summary
  - Embedding sizes not necessarily a good idea
  - Can make code faster
  - Makes code more complicated
    - size argument appears everywhere!
  - Can lead to (even more) code bloat

# Templates and inheritance

- Templates can inherit from non-template classes

```
template <class A>
class Derived : public Base {…}
```

# Templates and inheritance

- Templates can inherit from non-template classes

```
template <class A>
class Derived : public Base {…}
```

- Templates can inherit from template classes

```
template <class A>
class Derived : public List<A> {…}
```

# Templates and inheritance

- Templates can inherit from non-template classes

```
template <class A>
class Derived : public Base {…}
```

- Templates can inherit from template classes

```
template <class A>
class Derived : public List<A> {…}
```

- Non-template classes can inherit from templates

```
class SupervisorGroup : public
  List<Employee*> {…}
```

# Recurring template pattern

- General form

```
// The Curiously Recurring Template Pattern (CRTP)
template <class T>
class Base
{
  // ...
};
class Derived : public Base<Derived>
{
  // ...
};
```

# Recurring template pattern

- Simulate virtual function in generic programming

```
template <class T>
class Base {
  void interface() {
    static_cast<T*>(this)->implementation(); // ...
  }
  static void static_func() {
    T::static_sub_func(); // ...
  }
};
class Derived : public Base<Derived> {
  void implementation();
  static void static_sub_func();
};
```

# Notes

- In general put the definition and the declaration for the template in the header file
  - won't allocate storage for the class at that point
  - compiler/linker has mechanism for removing multiple definitions

# Writing templates

- Get a non-template version working first

# Writing templates

- Get a non-template version working first
- Establish a good set of test cases

# Writing templates

- Get a non-template version working first
- Establish a good set of test cases
- Measure performance and tune

# Writing templates

- Get a non-template version working first
- Establish a good set of test cases
- Measure performance and tune

- Review implementation
  – Which types should be parameterized?

# Writing templates

- Get a non-template version working first
- Establish a good set of test cases
- Measure performance and tune

- Review implementation
  - Which types should be parameterized?
- Convert non-parameterized version into template

# Writing templates

- Get a non-template version working first
- Establish a good set of test cases
- Measure performance and tune

- Review implementation
  - Which types should be parameterized?
- Convert non-parameterized version into template
- Test against established test cases