Templates

Learning Objectives

- Function Templates
  - Syntax, defining
  - Compiler complications
- Class Templates
  - Syntax
  - Example: array template class
- Templates and Inheritance
  - Example: partially-filled array template class

Finding the Maximum of Two Integers

- Here's a small function that you might write to find the maximum of two integers.

```
int maximum(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Finding the Maximum of Two Doubles

- Here's a small function that you might write to find the maximum of two double numbers.

```
double maximum(double a, double b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

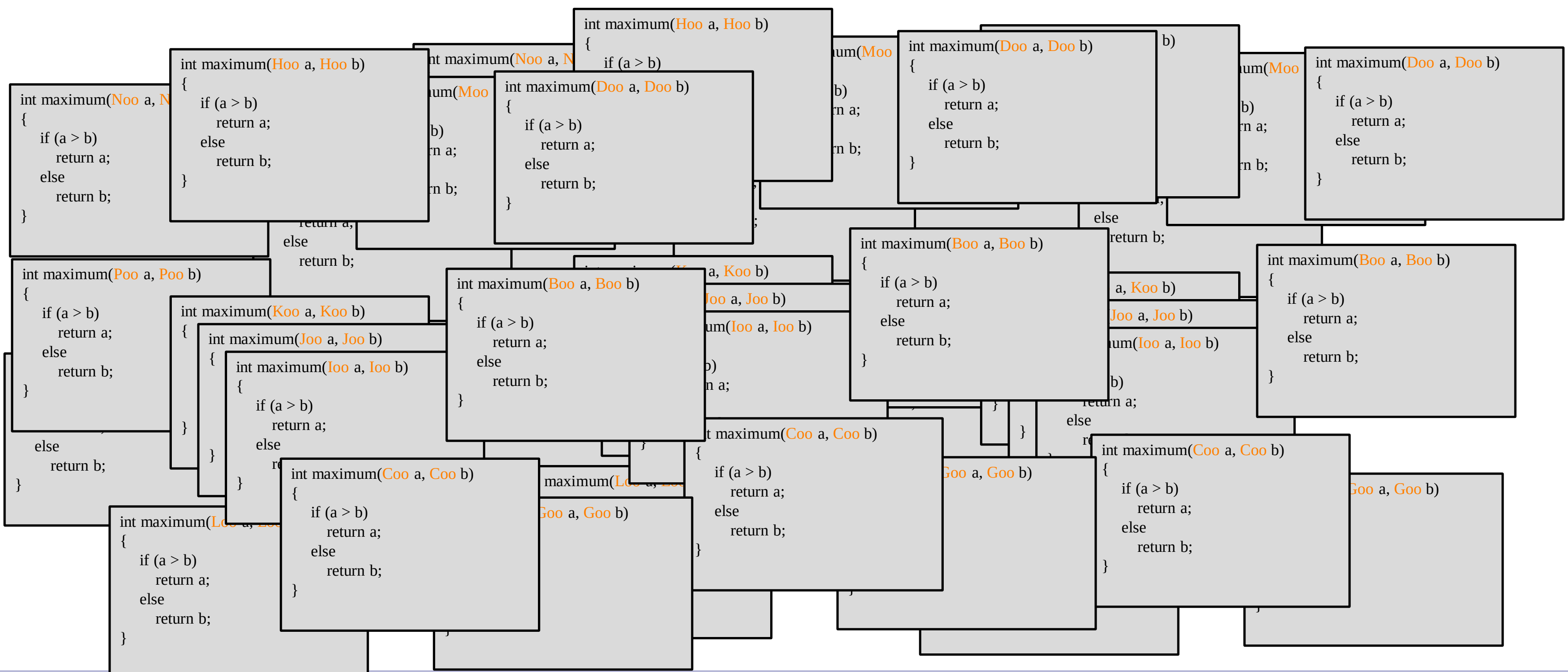
Finding the Maximum of Two foos

- Here's a small function that you might write to find the maximum of two foos.

```
foo maximum(foo a, foo b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

One Hundred Million Functions...

- Suppose your program uses 100,000,000 different data types, and you need a maximum function for each...



A Template Function for Maximum

- This template function can be used with many data types.

```
template <class T>
T maximum(T a, T b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

A Template Function for Maximum

- A template prefix is needed immediately before the function's implementation
- When you write a template function, you choose a data type for the function to depend upon...

```
template <class T>
T maximum(T a, T b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Using a Template Function

- Once a template function is defined, it may be used with any adequate data type in your program...

```
template <class T>
T maximum(T a, T b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
cout << maximum(1,2);
cout << maximum(1.3, 0.9);
...
```

Finding the Maximum Item in an Array

- Here's another function that can be made more general by changing it to a template function:

```
int array_max(int data[  ], size_t n)
{
    size_t i;
    int answer;

    assert(n > 0);
    answer = data[0];
    for (i = 1; i < n; i++)
        if (data[i] > answer) answer = data[i];
    return answer;
}
```

Finding the Maximum Item in an Array

- Here's the template function:

```
template <class T>
T array_max(T data[  ], size_t n)
{
    size_t i;
    T answer;

    assert(n > 0);
    answer = data[0];
    for (i = 1; i < n; i++)
        if (data[i] > answer) answer = data[i];
    return answer;
}
```

Templates

- C++ templates
  - Allow very "general" definitions for functions and classes
  - Type names are "parameters" instead of actual types
  - Precise definition determined at run-time

Another Function Template

- Recall function swapValues:

```
void swapValues(int& var1, int& var2)
{
        int temp;
        temp = var1;
        var1 = var2;
        var2 = temp;
}
```

- Applies only to variables of type int
- But code would work for any types!

Function Templates vs. Overloading

- Could overload function for char's:

```
void swapValues(char& var1, char& var2)
{
    char temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

- But notice: code is nearly identical!
  - Only difference is type used in 3 places

Function Template

- Allow "swap values" of any type variables:

```
template<class T>
void swapValues(T& var1, T& var2)
{
        T temp;
        temp = var1;
        var1 = var2;
        var2 = temp;
}
```

- First line called "template prefix"
  - Tells compiler what's coming is "template"
  - And that T is a type parameter

Function Template Definition

- swapValues() function template is actually large "collection" of definitions!
  - A definition for each possible type!

- Compiler only generates definitions when required
  - But it's "as if" you'd defined for all types

- Write one definition → works for all types that might be needed

Another Function Template

- Declaration/prototype:

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3);
```

- Definition:

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3)
{
	cout	<< stuff1 << endl
		<< stuff2 << endl
		<< stuff3 << endl;
}
```

showStuff Call

- Consider function call:
  showStuff(2, 3.3, 4.4);
- Compiler generates function definition
  - Replaces T with double
    - Since second parameter is type double
- Displays:
  2
  3.3
  4.4

Compiler Complications

- Function declarations and definitions
  - Typically we have them separate
- Safest to place template function definition in file where invoked
  - Many compilers require it appear 1^{st}
  - Often we #include all template definitions

More Compiler Complications

- Check your compiler's specific requirements
  - Some need to set special options
  - Some require special order of arrangement of template definitions vs. other file items

- Most usable template program layout:
  - Template definition in same file it's used
  - Ensure template definition precedes all uses
    - Can #include it

Multiple Type Parameters

- Can have:
  template<class T1, class T2>
- Not typical
  - Usually only need one "replaceable" type
  - Cannot have "unused" template parameters
    - Each must be "used" in definition
    - Error otherwise!

Algorithm Abstraction

- Refers to implementing templates
- Express algorithms in "general" way:
  - Algorithm applies to variables of any type
  - Ignore incidental detail
  - Concentrate on substantive parts of algorithm
- Function templates are one way C++ supports algorithm abstraction

Defining Templates Strategies

- Develop function normally
  - Using actual data types
- Completely debug "ordinary" function
- Then convert to template
  - Replace type names with type parameter as needed
- Advantages:
  - Easier to solve "concrete" case
  - Deal with algorithm, not template syntax

Inappropriate Types in Templates

- Can use any type in template for which code makes "sense"
  - Code must behave in appropriate way
- e.g., swapValues() template function
  - Cannot use type for which assignment operator isn't defined
  - Example: an array:
    int a[10], b[10];
    swapValues(a, b);
    - Arrays cannot be "assigned"!

Class Templates

- Can also "generalize" classes

  template<class T>

  - Can also apply to class definition
  - All instances of "T" in class definition replaced by type parameter
  - Just like for function templates!
- Once template defined, can declare objects of the class

Class Template Definition

- ```
  template<class T>
  class Pair
  {
  public:
          Pair();
          Pair(T firstVal, T secondVal);
          void setFirst(T newVal);
          void setSecond(T newVal);
          T getFirst() const;
          T getSecond() const;
  private:
          T first; T second;
  };
  ```
  ```

Template Class Pair Members

```
template<class T>
Pair<T>::Pair(T firstVal, T secondVal)
{
    first = firstVal;
    second = secondVal;
}

template<class T>
void Pair<T>::setFirst(T newVal)
{
    first = newVal;
}
```

Template Class Pair

- Objects of class have "pair" of values of type T
- Can then declare objects:
  Pair<int> score;
  Pair<char> seats;
  - Objects then used like any other objects
- Example uses:
  score.setFirst(3);
  score.setSecond(0);

Pair Member Function Definitions

- Notice in member function definitions:
  - Each definition is itself a "template"
  - Requires template prefix before each definition
  - Class name before :: is "Pair<T>"
    - Not just "Pair"
  - But constructor name is just "Pair"
  - Destructor name is also just "~Pair"

Class Templates as Parameters

- Consider:
  int addUP(const Pair<int>& thePair);
  - The type (int) is supplied to be used for T in defining this class type parameter
  - It "happens" to be call-by-reference here
- Again: template types can be used anywhere standard types can

Class Templates Within Function Templates

- Rather than defining new overload:

```
template<class T>
T addUp(const Pair<T>& the Pair);
//Precondition: Operator + is defined for values of type T
//Returns sum of two values in thePair
```

- Function now applies to all kinds of numbers

Restrictions on Type Parameter

- Only "reasonable" types can be substituted for T
- Consider:
  - Assignment operator must be "well-behaved"
  - Copy constructor must also work
  - If T involves pointers, then destructor must be suitable!
- Similar issues as function templates

Type Definitions

- Can define new "class type name"
  - To represent specialized class template name
- Example:
  typedef Pair<int> PairOfInt;
- Name "PairOfInt" now used to declare objects of type Pair<int>:
  PairOfInt pair1, pair2;
- Name can also be used as parameter, or anywhere else type name allowed

Friends and Templates

- Friend functions can be used with template classes
  - Same as with ordinary classes
  - Simply requires type parameter where appropriate
- Very common to have friends of template classes
  - Especially for operator overloads (as we've seen)

Predefined Template Classes

- ‘vector’ class is a template class!

- Another: ‘basic_string’ is a template class
  - Deals with strings of "any-type" elements
  - e.g.,

basic_string<char>	works for char’s
basic_string<double>	works for doubles
basic_string<YourClass>	works for YourClass objects

basic_string Template Class

- ‘string’ class
  - It’s an alternate name for basic_string<char>
  - All member functions behave similarly for basic_string<T>

- basic_string defined in library <string>
  - Definition is in std namespace

Templates and Inheritance

- Nothing new here
- Derived template classes
  - Can derive from template or nontemplate class
  - Derived class is then naturally a template class
- Syntax same as ordinary class derived from ordinary class

Summary

- Function templates
  - Define functions with parameter for a type
- Class templates
  - Define class with parameter for subparts of class
- Predefined vector and basic_string classes are template classes
- Can define template class derived from a template base class