Chapter 16

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

Introduction

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

Function Templates

- 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 Syntax

- 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

Template Prefix

- Recall:
  template<class T>
- In this usage, "class" means "type", or "classification"
- Can be confused with other "known" use of word "class"!
  - C++ allows keyword "typename" in place of keyword "class" here
  - But most use "class" anyway

Template Prefix 2

- Again:
  template<class T>
- T can be replaced by any type
  - Predefined or user-defined (like a C++ class type)
- In function definition body:
  - T used like any other type
- Note: can use other than "T", but T is "traditional" usage

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

Calling a Function Template

- Consider following call:
  swapValues(int1, int2);
  - C++ compiler "generates" function definition for two int parameters
    using template
- Likewise for all other types
- Needn’t do anything "special" in call
  - Required definition automatically generated

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
  - For templates → not supported on most compilers!
- Safest to place template function definition in file where invoked
  - Many compilers require it appear 1st
  - 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

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

Improper 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>& thePair);
//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

- Recall vector class
  - It's a template class!
- Another: basic_string 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

- Already used it!
- Recall "string"
  - 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