Function Templates

- Many functions have the same code body regardless of type.

for (i = 0; i < n; i++) a[i] = b[i];

Function Templates (2)

- Most C programmers automate this with a simple macro.

```
#define COPY (A, B, N) {        \
int i;                          \
for (i = 0; i < N; i++)         \
        A[i] = B[i];            \
}
```

Function Templates (3)

1. This works but not "type-safe". A user could mix types when conversions were inappropriate.

2. Macros present the possibility for serious "side effects" and do not enable compiler to perform type checking.

Function Templates (4)

- Function template approach in C++.

```
template <class T>
void copy(T a[], T b[], int n) {
        for (int i = 0; i < n; i++) a[i] = b[i];
}
```

Function Templates (5)

- Moreover, a generic copying procedure that accepts two distinct class type arguments.

```
template <class T1, class T2>
void copy(T1 a[], T2 b[], int n) {
    for (int i = 0; i < n; i++) a[i] = b[i];
}
```

Function Templates (6)

- A function template itself may be overloaded as well.

Class Templates

- A class template lets you generate multiple versions of a class definition, providing an alternative to derivation.

- Derivation lets you add fields to an existing structure; templates let you change the type definition of an existing field within the structure.

Class Templates (2)

- Class templates are called parameterized types because they require one or more type parameters.

- Class template can contain friends. A friend function that does not use a template specification is universally a friend of all instantiations of the class template (each one is a template class).

Class Templates (3)

- A friend function that incorporates template arguments is specifically a friend of its instantiated class.

```
template <class T>
class foo {
        friend void bar();          // universal
        friend vect<T> product(vect<T> v);
                                    // instantiated
}
```

Class Templates (4)

- Static members are not universal but are specific to each instantiation.

```
template <class T>
class foo {
    static int count;
}

template <class T> int foo<T>::count = 0;
```

Class Templates (5)

foo <int> a;

foo <double> b;

- The static variables *foo<int>::count* and *foo<double>::count* are distinct.

Class Templates (6)

- Class templates can have non-type parameters, which can have default arguments and are treated as "***const***".

template <class T, int number> array .....

array <int, 10> a;

C++ I/O

- Occurs in stream of bytes, where a stream is logical device that either produces or consumes information.
- When a C++ program begins, four streams are automatically opened: "***cin***", "***cout***", "***cerr***", and "***clog***".
- The output operator is left associative and returns a value of type "*ostream &*".

Manipulator

- A manipulator is a value or function that has a special effect on the stream it operates on. A simple example of a manipulator is "***endl***" whose effect is to output a newline and then flush the ostream.
- The manipulators "***dec***", "***hex***", and "***oct***" can be used to change integer bases.

Manipulator (2)

- "***setw***" is a manipulator that changes the default field width to the value of its argument.
- Each time an output operation is performed, the field width returns to its default setting, so it may be necessary to set the minimum field width before each output statement.

Manipulator (3)

- By default, six digits are displayed after the decimal point when floating-point values are output, you can set this number by using the "***setprecision***" manipulator.

This page intentionally left blank.

Chapter 12 - Templates

Outline

12.1 Introduction

- Templates - easily create a large range of related functions or classes
  - function template - the blueprint of the related functions
  - template function - a specific function *made* from a function template

12.2 Function Templates

- overloaded functions
  - perform similar operations on different data types
- function templates
  - perform identical operations on different data types
  - provide type checking
- Format:

  template<class *type*, **class** *type*...**>**
  - can use **class** or **typename** - specifies type parameters

  ```
  template< class T >
  template< typename ElementType >
  template< class BorderType, class FillType >
  ```
  - Function definition follows **template** statement

12.2 Function Templates (II)

```
1  template< class T >

2  void printArray( const T *array, const int
count )
3  {

4     for ( int i = 0; i < count; i++ )

5        cout << array[ i ] << " ";

6

7     cout << endl;

8  }
```

T is the type parameter. **T**'s type is detected and substituted inside the function.

The newly created function is compiled.

The **int** version of **printArray** is

```
void printArray( const int *array, const int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
}
```


12.3 Overloading Template Functions

- related template functions have same name
  - compiler uses overloading resolution to call the right one
- function template can be overloaded
  - other function templates can have same name but different number of parameters
  - non-template function can have same name but different arguments
- compiler tries to match function call with function name and arguments
  - if no precise match, looks for function templates
    - if found, compiler generates and uses template function
  - if no matches or multiple matches are found, compiler gives error

12.4 Class Templates

- class templates
  - allow type-specific versions of generic classes
- Format:

```
template <class T>
class ClassName{
 definition
 }
```

  - Need not use **"T"**, any identifier will work
  - To create an object of the class, type

    ClassName`< type > myObject;`

    Example: `Stack< double > doubleStack;`

12.4 Class Templates (II)

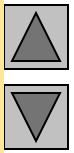
- Template class functions
  - declared normally, but preceded by **`template<class T>`**
    - generic data in class listed as type **`T`**
  - binary scope resolution operator used
  - Template class function definition:

```
template<class T>
MyClass< T >::MyClass(int size)
{
   myArray = new T[size];
}
```

    - constructor definition - creates an array of type **`T`**

```
1  // Fig. 12.3: tstack1.h
2  // Class template Stack
3  #ifndef TSTACK1_H
4  #define TSTACK1_H
5
6  template< class T >
7  class Stack {
8  public:
9     Stack( int = 10 );    // default constructor (stack size 10)
10    ~Stack() { delete [] stackPtr; } // destructor
11    bool push( const T& ); // push an element onto the stack
12    bool pop( T& );        // pop an element off the stack
13 private:
14    int size;             // # of elements in the stack
15    int top;              // location of the top element
16    T *stackPtr;          // pointer to the stack
17
18    bool isEmpty() const { return top == -1; }      // utility
19    bool isFull() const { return top == size - 1; } // functions
20 };
21
22 // Constructor with default size 10
23 template< class T >
24 Stack< T >::Stack( int s )
25 {
26    size = s > 0 ? s : 10;
27    top = -1;               // Stack is initially empty
28    stackPtr = new T[ size ]; // allocate space for elements
29 }
```

Notice how a member function of the class template is defined



Outline

1. Class template definition

1.1 Function definitions

1.2 `Stack` constructor

12.5 Class Templates and Non-type Parameters

- can use non-type parameters in templates
  - default argument
  - treated as **`const`**
- Example:

```
template< class T, int elements >
Stack< double, 100 >
  mostRecentSalesFigures;
```

  - declares object of type **`Stack< double, 100>`**
  - This may appear in the class definition:

```
T stackHolder[ elements ]; //array to hold stack
```

    - creates array at compile time, rather than dynamic allocation at execution time

12.6 Templates and Inheritance

- A class template can be derived from a template class
- A class template can be derived from a non-template class
- A template class can be derived from a class template
- A non-template class can be derived from a class template

12.7 Templates and friends

- friendships allowed between a class template and
  - global function
  - member function of another class
  - entire class
- **`friend`** functions
  - inside definition of class template **`X`**:
  - **`friend void f1();`**
    - **`f1()`** a **`friend`** of all template classes
  - **`friend void f2( X< T > & );`**
    - **`f2( X< int > & )`** is a **`friend`** of **`X< int >`** only. The same applies for **`float`**, **`double`**, etc.
  - **`friend void A::f3();`**
    - member function **`f3`** of class **`A`** is a **`friend`** of all template classes

12.7 Templates and friends (II)

- `friend void C< T >::f4( X< T > & );`
  - `C<float>::f4( X< float> & )` is a `friend` of `class X<float>` only

- **`friend`** classes
  - `friend class Y;`
    - every member function of `Y` a friend with every template class made from `X`
  - `friend class Z<T>;`
    - class `Z<float>` a `friend` of class `X<float>`, etc.

12.8 Templates and static Members

- non-template class
  - **`static`** data members shared between all objects

- template classes
  - each class (**`int`**, **`float`**, etc.) has its own copy of **`static`** data members
  - **`static`** variables initialized at file scope
  - each template class gets its own copy of **`static`** member functions

This pag ntionally le nk.

Chapter 11- C++ Stream Input/Output

Outline

11.1 Introduction

- Many C++ I/O features are object-oriented
  - use references, function overloading and operator overloading
- Extensibility
  - Users may specify I/O of user-defined types as well as standard types

11.2 Streams

- Stream
  - A transfer of information in the form of a sequence of bytes

- I/O Operations:
  - Input: A stream that flows from an input device (i.e.: keyboard, disk drive, network connection) to main memory
  - Output: A stream that flows from main memory to an output device (i.e.: screen, printer, disk drive, network connection)

11.2 Streams (II)

- I/O operations are a bottleneck
  - The time for a stream to flow is many times larger than the time it takes the CPU to process the data in the stream
- Low-level I/O
  - unformatted
  - individual byte unit of interest
  - high speed, high volume, but inconvenient for people
- High-level I/O
  - formatted
  - bytes grouped into meaningful units: integers, characters, etc.
  - good for all I/O except high-volume file processing

11.2.1 Iostream Library Header Files

- **`iostream`** library:
  - **`<iostream.h>:`** Contains **`cin`**, **`cout`**, **`cerr`**, and **`clog`** objects
  - **`<iomanip.h>:`** Contains *parameterized stream manipulators*
  - **`<fstream.h>:`** Contains information important to user-controlled file processing operations

11.2.2 Stream Input/Output Classes and Objects

- **`ios`:**
  - **`istream`** and **`ostream`** inherit from **`ios`**
    - **`iostream`** inherits from **`istream`** and **`ostream`**.
- **`<<`** (left-shift operator): overloaded as *stream insertion operator*
- **`>>`** (right-shift operator): overloaded as *stream extraction operator*
- Used with **`cin`**, **`cout`**, **`cerr`**, **`clog`**, and with user-defined stream objects

11.2.2 Stream Input/Output Classes and Objects (II)

- **`istream:`** input streams
  - **`cin >> someVariable`**;
    - **`cin`** knows what type of data is to be assigned to **`someVariable`** (based on the type of **`someVariable`**).
- **`ostream:`** output streams
  - **`cout << someVariable;`**
    - **`cout`** knows the type of data to output
  - **`cerr << someString;`**
    - Unbuffered. Prints **`someString`** immediately.
  - **`clog << someString;`**
    - Buffered. Prints **`someString`** as soon as output buffer is full or flushed.

11.3.1 Stream-Insertion Operator

- **<<** is overloaded to output built-in types
  - can also be used to output user-defined types.
  - **cout << '\n';**
    - prints newline character
  - **cout << endl;**
    - **endl** is a stream manipulator that issues a newline character and flushes the output buffer
  - **cout << flush;**
    - **flush** flushes the output buffer.

11.3.2 Cascading Stream-Insertion/Extraction Operators

- **<<** : Associates from left to right, and returns a reference to its left-operand object (i.e. **cout**).
  - This enables cascading

```
cout << "How" << " are" << " you?";
```

Make sure to use parenthesis:

```
cout << "1 + 2 = " << (1 + 2);
```

NOT

```
cout << "1 + 2 = " << 1 + 2;
```

How about
cout << 1 < 2;

11.3.4 Character Output with Member Function put; Cascading puts

- **`put`** member function
  - outputs one character to specified stream

    ```
    cout.put( 'A');
    ```

  - returns a reference to the object that called it, so may be cascaded

    ```
    cout.put( 'A' ).put( '\n' );
    ```

  - may be called with an ASCII-valued expression

    ```
    cout.put(65);
    ```

    outputs **`A`**

11.4 Stream Input

- **`>>`** (stream-extraction)
  - used to perform stream input
  - Normally ignores whitespaces (spaces, tabs, newlines)
  - Returns zero (**`false`**) when **`EOF`** is encountered, otherwise returns reference to the object from which it was invoked (i.e. **`cin`**)
    - This enables cascaded input.

```
cin >> x >> y;
```

- **`>>`** controls the state bits of the stream
  - **`failbit`** set if wrong type of data input
  - **`badbit`** set if the operation fails

11.4.1 Stream-Extraction Operator

- **>>** and **<<** have relatively high precedence
  - conditional and arithmetic expressions must be contained in parentheses

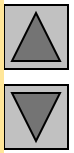
- Popular way to perform loops

```
while (cin >> grade)
```

  - extraction returns **0** (**false**) when **EOF** encountered, and loop ends

```cpp
// Fig. 11.11: fig11_11.cpp
// Stream-extraction operator returning false on end-of-file.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int grade, highestGrade = -1;

   cout << "Enter grade (enter end-of-file to end): ";
   while ( cin >> grade ) {
      if ( grade > highestGrade )
         highestGrade = grade;

      cout << "Enter grade (enter end-of-file to end): ";
   }

   cout << "\n\nHighest grade is: " << highestGrade << endl;
   return 0;
}
```

```
Enter grade (enter end-of-file to end): 67
Enter grade (enter end-of-file to end): 87
Enter grade (enter end-of-file to end): 73
Enter grade (enter end-of-file to end): 95
Enter grade (enter end-of-file to end): 34
Enter grade (enter end-of-file to end): 99
Enter grade (enter end-of-file to end): ^Z
Highest grade is: 99
```



Outline

1. Initialize variables

2. Perform loop

3. Output

Program Output

11.4.2 `get` and `getline` Member Functions

- **`cin.get()`:** inputs a character from stream (even white spaces) and returns it
- **`cin.get( c )`:** inputs a character from stream and stores it in **`c`**

11.4.2 `get` and `getline` Member Functions (II)

- **`cin.get(array, size)`:**
  - accepts 3 arguments: array of characters, the size limit, and a delimiter (default of **`'\n'`**).
  - Uses the array as a buffer
  - When the delimiter is encountered, it remains in the input stream
  - Null character is inserted in the array
  - unless delimiter flushed from stream, it will stay there

- **`cin.getline(array, size)`**
  - operates like **`cin.get(buffer, size)`** but it discards the delimiter from the stream and does not store it in array
  - Null character inserted into array

```
1  // Fig. 11.12: fig11_12.cpp
2  // Using member functions get, put and eof.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  int main()
10 {
11    char c;
12
13    cout << "Before input, cin.eof() is " << cin.eof()
14         << "\nEnter a sentence followed by end-of-file:\n";
15
16    while ( ( c = cin.get() ) != EOF )
17       cout.put( c );
18
19    cout << "\nEOF in this system is: " <<
20    cout << "\nAfter input, cin.eof() is " << cin.eof() << endl;
21    return 0;
22 }
```

`cin.eof()` returns **`false (0)`** or **`true (1)`**

`cin.get()` returns the next character from input stream, including whitespace.

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions^Z
Testing the get and put member functions
EOF in this system is: -1
After input cin.eof() is 1
```

Outline

1. Initialize variables

2. Input data

2.1 Function call

3. Output

11.4.3 `istream` Member Functions `peek`, `putback` and `ignore`

- **`ignore`** member function
  - skips over a designated number of characters (default of one)
  - terminates upon encountering a designated delimiter (default is **`EOF`**, skips to the end of the file)
- **`putback`** member function
  - places the previous character obtained by **`get`** back in to the stream.
- **`peek`**
  - returns the next character from the stream without removing it

11.4.4 Type-Safe I/O

- **<<** and **>>** operators
  - Overloaded to accept data of different types
  - When unexpected data encountered, error flags set
  - Program stays in control

11.6 Stream Manipulators

- stream manipulator capabilities:
  - setting field widths
  - setting precisions
  - setting and unsetting format flags
  - setting the fill character in fields
  - flushing streams
  - inserting a newline in the output stream and flushing the stream inserting a null character in the output stream and skipping whitespace in the input stream.

11.6.1 Integral Stream Base: `dec`, `oct`, `hex` and `setbase`

- **`oct`, `hex`,** or **`dec`**:
  - change base of which integers are interpreted from the stream.

  Example:

```
int n = 15;
 cout << hex << n;
```

  - prints **`"F"`**

- **`setbase`**:
  - changes base of integer output
  - load **`<iomanip>`**
  - Accepts an integer argument (**`10`**, **`8`**, or **`16`**)

```
cout << setbase(16) << n;
```

  - parameterized stream manipulator - takes an argument

11.6.2 Floating-Point Precision (`precision`, `setprecision`)

- **`precision`**
  - member function
  - sets number of digits to the right of decimal point
    ```
    cout.precision(2);
    ```
  - **`cout.precision()`** returns current precision setting
- **`setprecision`**
  - parameterized stream manipulator
  - Like all parameterized stream manipulators, **`<iomanip>`** required
  - specify precision:
    ```
    cout << setprecision(2) << x;
    ```
- For both methods, changes last until a different value is set

11.6.3 Field Width (`setw`, `width`)

- **`ios width`** member function
  - sets field width (number of character positions a value should be output or number of characters that should be input)
  - returns previous width
  - if values processed are smaller than width, fill characters inserted as padding
  - values are not truncated - full number printed
  - **`cin.width(5);`**
- **`setw`** stream manipulator

```
cin >> setw(5) >> string;
```

- Remember to reserve one space for the null character

11.7.2 Trailing Zeros and Decimal Points (`ios::showpoint`)

- **`ios::showpoint`**
  - forces a float with an integer value to be printed with its decimal point and trailing zeros

```
cout.setf(ios::showpoint)
cout << 79;
```

`79` will print as `79.00000`

- number of zeros determined by precision settings

11.7.3 Justification (`ios::left`, `ios::right`, `ios::internal`)

- **`ios::left`**
  - fields to left-justified with padding characters to the right
- **`ios::right`**
  - default setting
  - fields right-justified with padding characters to the left
- Character used for padding set by
  - **`fill`** member function
  - **`setfill`** parameterized stream manipulator
  - default character is space

11.7.4 Padding(`fill`, `setfill`)

- **`fill`** member function
  - specifies the fill character
  - space is default
  - returns the prior padding character

```
cout.fill( '*');
```

- **`setfill`** manipulator
  - also sets fill character

```
cout << setfill ('*');
```