Course Information

- Lecture:
  - James B D Joshi
  - Mondays: 6:00-8:50 PM
  - One (two) 15 (10) minutes break(s)
  - Office Hours: Wed 3:00-5:00PM/Appointment
  - TA: Ming Mao
- Pre-requisite
  - IS 0015 Data Structures and Programming Techniques
- Textbook

Course Information

- Course Description
  - An introduction to the development of programs using C++.
  - Emphasis is given to the development of program modules that can function independently.
  - Object-oriented design
  - The theory of data structures and programming language design is continued.

Grading

- Quiz 10% (in the beginning of the class; on previous lecture)
- Homework/Programming Assignments 50% (typically every week)
- Midterm 20%
- Comprehensive Final 20%
Course Policy

- Your work MUST be your own
  - Zero tolerance for cheating
  - Discussing problems is encouraged, but each must present his own answers
  - You get an F for the course if you cheat in anything however small
  - NO DISCUSSION
- Homework
  - There will be penalty for late assignments (15% each day)
  - Ensure clarity in your answers – no credit will be given for vague answers
  - Homework is primarily the GSA’s responsibility
- Check webpage for everything!
  - You are responsible for checking the webpage for updates

Computer Languages

- Machine language
  - Generally consist of strings of numbers - Ultimately 0s and 1s - Machine-dependent
  - Example: +1300042774 +1400593419
- Assembly language
  - English-like abbreviations for elementary operations
  - Incomprehensible to computers - Convert to machine language
  - Example:
  - LOAD BASEPAY
  - ADD OVERPAY
  - STORE GROSSPAY
- High-level languages
  - Similar to everyday English, use common mathematical notations
  - Compiler/Interpreter
  - Example: grossPay = basePay + overTimePay

History of C and C++

- History of C
  - Evolved from two other programming languages
  - BCPL and B: “Typeless” languages
  - Dennis Ritchie (Bell Lab): Added typing, other features
- History of C++
  - Early 1980s: Bjarne Stroustrup (Bell Lab)
  - Provides capabilities for object-oriented programming
  - Objects: reusable software components
  - Object-oriented programs
- Building block approach” to creating programs
  - C++ programs are built from pieces called classes and functions
  - C++ standard library: Rich collections of existing classes and functions

Structured/OO Programming

- Structured programming (1960s)
  - Disciplined approach to writing programs
  - Clear, easy to test and debug, and easy to modify
  - E.g. Pascal: 1971: Niklaus Wirth
- OOP
  - “Software reuse”
  - “Modularity”
  - “Extensible”
  - More understandable, better organized and easier to maintain than procedural programming
Basics of a Typical C++ Environment

- **C++ systems**
  - Program-development environment
  - Language
  - C++ Standard Library
- **C++ program names extensions**
  - .cpp
  - .cxx
  - .cc
  - .C

---

A Simple Program: Printing a Line of Text

- **Standard output stream object**
  - `std::cout`
  - “Connected” to screen
  - `<<` Stream insertion operator
  - Value to right (right operand) inserted into output stream
- **Namespace**
  - `std`: specifies that entity belongs to “namespace std`
  - `std`: removed through use of `using` statements
- **Escape characters:** `\`
  - Indicates “special” character output

---

Phases of C++ Programs:

1. Edit
2. Preprocess
3. Compile
4. Link
5. Load
6. Execute

---

Common Input/output functions

- `cin`
  - Standard input stream
  - Normally keyboard
- `cout`
  - Standard output stream
  - Normally computer screen
- `cerr`
  - Standard error stream
  - Display error messages

- Comments: C’s comment /* .. */ OR Begin with // or
- Preprocessor directives: Begin with #
  - Processed before compiling
// Fig. 1.2: fig01_02.cpp
// A first program in C++.
#include <iostream>

int main()
{
    std::cout << "Welcome to C++!
";
    return 0; // indicate that program ended successfully
} // end function main

Welcome to C++!
**Decision Making: Equality and Relational Operators**

- **if** structure
  - Make decision based on truth of falsity of condition
    - If condition met, body executed
    - Else, body not executed
  - Equality and relational operators
    - Equality operators
      - Same level of precedence
    - Relational operators
      - Same level of precedence
      - Associate left to right

<table>
<thead>
<tr>
<th>Relational operators</th>
<th>C++ equality operator</th>
<th>Example of C++ condition</th>
<th>Meaning of C++ condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>==</td>
<td><code>x == y</code></td>
<td>x is equal to y</td>
</tr>
<tr>
<td>!=</td>
<td>!=</td>
<td><code>x != y</code></td>
<td>x is not equal to y</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td><code>x &gt; y</code></td>
<td>x is greater than y</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td><code>x &lt; y</code></td>
<td>x is less than y</td>
</tr>
<tr>
<td>&gt;=</td>
<td>&gt;=</td>
<td><code>x &gt;= y</code></td>
<td>x is greater than or equal to y</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&lt;=</td>
<td><code>x &lt;= y</code></td>
<td>x is less than or equal to y</td>
</tr>
</tbody>
</table>

**Algorithms / pseudocode**

- Computing problems
  - Solved by executing a series of actions in a specific order
- Algorithm: a procedure determining
  - Actions to be executed
  - Order to be executed
  - Example: recipe
- Program control
  - Specifies the order in which statements are executed
- Pseudocode
  - Artificial, informal language used to develop algorithms
  - Similar to everyday English

**Control Structures**

- Sequential execution
  - Statements executed in order
- Transfer of control
  - Next statement executed not next one in sequence
  - Structured programming — “goto”-less programming
- 3 control structures to build any program
  - Sequence structure
    - Programs executed sequentially by default
  - Selection structures
    - if, if/else, switch
  - Repetition structures
    - while, do/while, for
Keywords

- C++ keywords
  - Cannot be used as identifiers or variable names

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Common in C and C++ languages</th>
<th>C++ only keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>break case char const</td>
<td>asm bool catch class const_cast delete dynamic_cast explicit false friend inline mutable namespace new operator private protected public reinterpret_cast static_cast template this throw true try typeid typename using virtual wchar_t</td>
</tr>
<tr>
<td>continue</td>
<td>default do double else</td>
<td>auto bool catch class const_cast delete dynamic_cast explicit false friend inline mutable namespace new operator private protected public reinterpret_cast template this throw true try typeid typename using virtual wchar_t</td>
</tr>
<tr>
<td>case</td>
<td>default do double else</td>
<td>auto bool catch class const_cast delete dynamic_cast explicit false friend inline mutable namespace new operator private protected public reinterpret public static_cast template this throw true try typeid typename using virtual wchar_t</td>
</tr>
<tr>
<td>char</td>
<td>const_cast delete dynamic_cast explicit false friend inline mutable namespace new operator private protected public reinterpret_cast template this throw true try typeid typename using virtual wchar_t</td>
<td></td>
</tr>
<tr>
<td>continue</td>
<td>default do double else</td>
<td>auto bool catch class const_cast delete dynamic_cast explicit false friend inline mutable namespace new operator private protected public reinterpret_cast template this throw true try typeid typename using virtual wchar_t</td>
</tr>
<tr>
<td>do</td>
<td>double else for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>else for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>for</td>
<td>goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>goto</td>
<td>if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>int long register return short signed sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>long register return short signed sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>register return short signed sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>return short signed sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>short signed sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>signed sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>signed</td>
<td>sizeof static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>sizeof</td>
<td>static struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>static</td>
<td>struct switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>struct</td>
<td>switch typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>switch</td>
<td>typedef union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>typedef</td>
<td>union unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>union</td>
<td>unsigned void volatile while</td>
<td></td>
</tr>
<tr>
<td>unsigned</td>
<td>void volatile while</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>while</td>
<td></td>
</tr>
<tr>
<td>volatile</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Control Structures

- Flowchart
  - Graphical representation of an algorithm
  - Special-purpose symbols connected by arrows (flowlines)
  - Rectangle symbol (action symbol)
    - Any type of action
  - Oval symbol
    - Beginning or end of a program, or a section of code (circles)

Exercise: Find greater of three numbers

if/else Selection Structure

- Ternary conditional operator (?:)
  - Three arguments (condition, value if true, value if false)
- Code could be written:
  ```
  cout << ( grade >= 60 ? "Passed" : "Failed" );
  ```

while Repetition Structure

- Repetition structure
  - Counter-controlled
    - While/do while: loop repeated until condition becomes false
    - For: loop repeated until counter reaches certain value Flowchart representation?
  - Sentinel value
    - Indicates "end of data entry"
    - Sentinel chosen so it cannot be confused with regular input
- Example
  ```
  int product = 2;
  while ( product <= 1000 ) {
    product = 2 * product;
    cout << product;
  }
  ```
  Flowchart representation?
  What is the output?
**switch** Multiple-Selection Structure

- Test variable for multiple values
- Series of **case** labels and optional **default** case

```
switch ( variable ) {
  case value1:      // taken if variable == value1
    statements
    break;     // necessary to exit switch
  case value2:
  case value3: // taken if variable == value2 or == value3
    statements
    break;
  default:          // taken if none matches
    statements
    break;
}
```

**break and continue** Statements

- **break** statement
  - Immediate exit from while, for, do/while, switch
- **continue** statement
  - Program continues with first statement after structure

- Common uses
  - Escape early from a loop
  - Skip the remainder of switch

**Logical Operators**

- Used as conditions in loops, **if** statements
- **&&** (logical **AND**)
  - **true** if both conditions are **true**
    - if ( gender == 1 && age >= 65 )
      - ++seniorFemales;
- **||** (logical **OR**)
  - **true** if either of condition is **true**
    - if ( semesterAverage >= 90 || finalExam >= 90 )
      - cout << "Student grade is A" << endl;

- **!** (logical **NOT**, logical negation)
  - Returns **true** when its condition is **false**, & vice versa
    - if ( !( grade == sentinelValue ) )
      - cout << "The next grade is " << grade << endl;
    - Alternative:
      - if ( grade != sentinelValue )
      - cout << "The next grade is " << grade << endl;
Confusing Equality (==) and Assignment (=) Operators

- Common error
  - Does not typically cause syntax errors
- Aspects of problem
  - Expressions that have a value can be used for decision
    - Zero = false, nonzero = true
  - Assignment statements produce a value (the value to be assigned)

if == was replaced with =
if ( payCode = 4 )
cout << "You get a bonus!" << endl;

What happens?

Lvalues
- Expressions that can appear on left side of equation
  - Can be changed
  - x = 4;
- Rvalues
  - Only appear on right side of equation
  - Constants, such as numbers (i.e. cannot write 4 = x);
- Lvalues can be used as rvalues, but not vice versa

Structured-Programming Summary

- Structured programming
  - Programs easier to understand, test, debug and modify
- Rules for structured programming
  - Only use single-entry/single-exit control structures
  - Rules
    1) Begin with the “simplest flowchart”
    2) Any rectangle (action) can be replaced by two rectangles (actions) in sequence
    3) Any rectangle (action) can be replaced by any control structure (sequence, if, if/else, switch, while, do/while or for)
    4) Rules 2 and 3 can be applied in any order and multiple times
Program Components in C++

- **Modules:** functions and classes
- Programs use new and “prepackaged” modules
  - New: programmer-defined functions, classes
  - Prepackaged: from the standard library
- Functions invoked by function call
  - Function name and information (arguments) it needs
- Function definitions
  - Only written once
  - Hidden from other functions

Functions

- Functions
  - Modularize a program
  - Software reusability
    - Call function multiple times
- Local variables
  - Known only in the function in which they are defined
  - All variables declared in function definitions are local variables
- Parameters
  - Local variables passed to function when called
  - Provide outside information

Math Library Functions

- Perform common mathematical calculations
  - Include the header file `<cmath>`
- Functions called by writing
  - `functionName(argument);`
    - `functionName(argument1, argument2,...);`
- Example
  - `cout << sqrt( 900.0 );`
    - All functions in math library return a `double`
- Function arguments can be
  - Constants: `sqrt( 4 );`
  - Variables: `sqrt( x );`
  - Expressions:
    - `sqrt( sqrt( x ) );`
    - `sqrt( 3 - 6 * x );`
- Other functions
  - `sin(x), floor(x), log10(x), etc.`

Function Definitions

- Function prototype
  - `int square( int );`
- Calling/invoking a function
  - `square(x);`
- Format for function definition
  - `return-value-type function-name ( parameter-list )`
    - declarations and statements
  - Prototype must match function definition
    - Function prototype
      - `double maximum( double, double, double );`
    - Definition
      - `double maximum( double x, double y, double z )`
        - `...`
Function Definitions

• Example function
  
  ```c
  int square(int y)
  {
    return y * y;
  }
  ```

• **return** keyword
  - Returns data, and control goes to function’s caller
  - If no data to return, use `return;`
  - Function ends when reaches right brace
  - Control goes to caller

• Functions cannot be defined inside other functions

Function Prototypes

• **Function signature**
  - Part of prototype with name and parameters
    ```c
    double maximum(double, double, double);
    ```

• **Argument Coercion**
  - Force arguments to be of proper type
    - Converting `int(4)` to `double(4.0)
    ```c
    cout << sqrt(4)
    ```

• **Conversion rules**
  - Arguments usually converted automatically
  - Changing from `double` to `int` can truncate data
    - 3.4 to 3
  - Mixed type goes to highest type (promotion)

Data types

- `long double`
- `double`
- `float`
- `unsigned long int` (synonymous with `unsigned long`)
- `signed long int` (synonymous with `signed long`)
- `signed int` (synonymous with `signed`)
- `unsigned short int` (synonymous with `unsigned short`)
- `short int` (synonymous with `short`)
- `long long int` (synonymous with `signed long long`)
- `float` (alias for `float.h` or `float_t`)

Fig. 3.5 Promotion hierarchy for built-in data types.

Header Files

• **Header files contain**
  - Function prototypes
  - Definitions of data types and constants

• **Header files ending with .h**
  - Programmer-defined header files
    ```c
    #include "myheader.h"
    ```

• **Library header files**
  - ```c
    #include <cmath>
    ```
Enumenration: enum

- Set of integers with identifiers
  
  ```
  enum typeName {constant1, constant2...};
  ```
- Constants start at 0 (default), incremented by 1
- Constants need unique names
- Cannot assign integer to enumeration variable
  - Must use a previously defined enumeration type

- Example
  ```
  enum Status {CONTINUE, WON, LOST};
  Status enumVar;
  enumVar = WON; // cannot do enumVar = 1
  ```

Storage Classes

- Variables have attributes
  - Have seen name, type, size, value
  - Storage class
    - How long variable exists in memory
    - Scope
      - Where variable can be referenced in program
    - Linkage
      - For multiple-file program which files can use it

- Automatic storage class
  - Variable created when program enters its block
  - Variable destroyed when program leaves block
  - Only local variables of functions can be automatic
  - Automatic by default
  - Keyword `auto` explicitly declares automatic
  - `register` keyword
    - Hint to place variable in high-speed register
    - Good for often-used items (loop counters)
    - Often unnecessary, compiler optimizes
  - Specify either `register` or `auto`, not both

- Static storage class
  - Variables exist for entire program
  - For functions, name exists for entire program
  - May not be accessible, scope rules still apply
  - Keyword `auto` and `register`
    - Local variables in function
  - `register` variables are kept in CPU registers
  - `static` keyword
    - Local variables in function
    - Keeps value between function calls
    - Only known in own function
  - `extern` keyword
    - Default for global variables/functions
    - Globals defined outside of a function block
    - Known in any function that comes after it
Scope Rules

• Scope
  Portion of program where identifier can be used

• File scope
  – Defined outside a function, known in all functions
  – Global variables, function definitions and prototypes

• Function scope
  – Can only be referenced inside defining function
  – Only labels, e.g., identifiers with a colon (case)

• Block scope
  – Begins at declaration, ends at right brace }
  – Can only be referenced in this range
  – Local variables, function parameters
  – Local static variables still have block scope

• Storage class separate from scope

• Function-prototype scope
  – Parameter list of prototype
  – Names in prototype optional
  – Compiler ignores
  – In a single prototype, name can be used once

Outline

// Fig. 3.12: fig03_12.cpp
// A scoping example.
#include <iostream>

void useLocal( void );        // function prototype
void useStaticLocal( void );  // function prototype
void useGlobal( void );      // function prototype

int x = 1;     // global variable

int main()
{
  int x = 5;   // local variable to main
  {
    int x = 7;
    cout << "local x in main's inner scope is " << x << endl;
  }
  cout << "local x in main's outer scope is " << x << endl;
  useLocal();       // useLocal has local x
  useStaticLocal(); // useStaticLocal has static local x
  useGlobal();      // useGlobal uses global x
  useLocal();      // useLocal reinitializes its local x
  useStaticLocal(); // static local x retains its prior value
  useGlobal();      // global x also retains its value
  cout << "local x in main is " << x << endl;
  return 0;   // indicates successful termination
} // end main

// Fig. 3.12: fig03_12.cpp
// A scoping example.
#include <iostream>

void useLocal( void );        // function prototype
void useStaticLocal( void );  // function prototype
void useGlobal( void );      // function prototype

int x = 1;     // global variable

int main()
{
  int x = 5;   // local variable to main
  {
    int x = 7;
    cout << "local x in main's inner scope is " << x << endl;
  }
  cout << "local x in main's outer scope is " << x << endl;
  useLocal();       // useLocal has local x
  useStaticLocal(); // useStaticLocal has static local x
  useGlobal();      // useGlobal uses global x
  useLocal();      // useLocal reinitializes its local x
  useStaticLocal(); // static local x retains its prior value
  useGlobal();      // global x also retains its value
  cout << "local x in main is " << x << endl;
  return 0;   // indicates successful termination
} // end main
// useLocal reinitializes local variable x during each call
void useLocal( void )
{
    int x = 25;  // initialized each time useLocal is called
    // local x is 25 on entering useLocal
    ++x;
    // local x is 26 on exiting useLocal
}

// useStaticLocal initializes static local variable x only the first time the function is called; value of x is saved between calls to this function
void useStaticLocal( void )
{
    // initialized only first time useStaticLocal is called
    static int x = 50;
    // local static x is 50 on entering useStaticLocal
    ++x;
    // local static x is 51 on exiting useStaticLocal
}

// useGlobal modifies global variable x during each call
void useGlobal( void )
{
    // global x is 1 on entering useGlobal
    x *= 10;
    // global x is 10 on exiting useGlobal
}

// useGlobal modifies global variable x during each call
void useGlobal( void )
{
    // global x is 1 on entering useGlobal
    x *= 10;
    // global x is 10 on exiting useGlobal
}

Recursion

- Functions that call themselves
- Can only solve a base case
- If not base case
  - Break problem into smaller problem(s)
  - Launch new copy of function to work on the smaller problem (recursive call/recursive step)
    - Slowly converges towards base case
    - Function makes call to itself inside the return statement
  - Eventually base case gets solved
    - Answer works way back up, solves entire problem
Recursion

• Example: factorial

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 1 \]

- Recursive relationship: \( n! = n \times (n-1)! \)
  
  \[ 5! = 5 \times 4! \]
  
  \[ 4! = 4 \times 3! \ldots \]

- Base case: \( 1! = 0! = 1 \)

Example Using Recursion: Fibonacci Series

• Fibonacci series: 0, 1, 1, 2, 3, 5, 8...

  - Each number sum of two previous ones
  - Example of a recursive formula:
    
    \[ \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \]

  - C++ code for Fibonacci function

    ```cpp
    long fibonacci(long n)
    {
        if ( n == 0 || n == 1 ) // base case
            return n;
        else
            return fibonacci(n - 1) + fibonacci(n - 2);
    }
    ```

Data type: unsigned long

- Can hold an integer from 0 to 4 billion.
Example Using Recursion: Fibonacci Series

- **Order of operations**
  - `return fibonacci(n - 1) + fibonacci(n - 2);`
- **Recursive function calls**
  - Each level of recursion doubles the number of function calls
  - 30th number = $2^{30} \approx 4$ billion function calls
  - Exponential complexity

---

Recursion vs. Iteration

- **Repetition**
  - Iteration: explicit loop
  - Recursion: repeated function calls
- **Termination**
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance between performance (iteration) and good software engineering (recursion)

---

Inline Functions

- **Inline functions**
  - Keyword `inline` before function
  - Asks the compiler to copy code into program instead of making function call
    - Reduce function-call overhead
    - Compiler can ignore `inline`
  - Good for small, often-used functions
- **Example**
  ```
  inline double cube(const double s)
  {
    return s * s * s;
  }
  ```

---

References and Reference Parameters

- **Call by value**
  - Copy of data passed to function
  - Changes to copy do not change original
  - Prevent unwanted side effects
- **Call by reference**
  - Function can directly access data
  - Changes affect original
- **Reference parameter**
  - Alias for argument in function call
    - Passes parameter by reference
    - Use `&` after data type in prototype
    - `void myFunction(int &data)`
    - Real “data is a reference to an int”
  - Function call format the same
    - However, original can now be changed
References and Reference Parameters

- **Pointers**
  - Another way to pass-by-reference
- **References as aliases to other variables**
  - Refer to same variable
  - Can be used within a function
    ```c
    int count = 1; // declare integer variable count
    int &cRef = count; // create cRef as an alias for count
    ++cRef; // increment count (using its alias)
    ```
- **References must be initialized when declared**
  - Otherwise, compiler error
  - Dangling reference

Default Arguments

- **Function call with omitted parameters**
  - If not enough parameters, rightmost go to their defaults
  - Default values
    - Can be constants, global variables, or function calls
- **Set defaults in function prototype**
  ```c
  int myFunction( int x = 1, int y = 2, int z = 3 );
  ```
  - `myFunction(3)`
    - `x = 3`, `y` and `z` get defaults (rightmost)
  - `myFunction(3, 5)`
    - `x = 3`, `y = 5` and `z` gets default

Unitary Scope Resolution Operator

- **Unary scope resolution operator (::)**
  - Access global variable if local variable has same name
  - Not needed if names are different
  - Use `::variable`
    ```c
    *y = ::x + 3; // Good to avoid using same names for locals and globals
    ```

Function Overloading

- **Function overloading**
  - Functions with same name and different parameters
  - Should perform similar tasks
    ```c
    int square(int x) { return x * x; }
    float square(float x) { return x * x; }
    ```
  - Overloaded functions distinguished by signature
    - Based on name and parameter types (order matters)
    - Name mangling
      - Encode function identifier with no. and types of parameters
    - Type-safe linkage
      - Ensures proper overloaded function called
Function Templates

- Compact way to make overloaded functions
  - Generate separate function for different data types
- Format
  - Begin with keyword `template`
  - Formal type parameters in brackets `<>
  - Every type parameter preceded by `typename` or `class` (synonyms)
  - Placeholders for built-in types (i.e., `int`) or user-defined types
  - Specify arguments types, return types, declare variables
- Function definition like normal, except formal types used

Example

```cpp
template < class T >  // or template< typename T >
T square( T value1 )
{
  return value1 * value1;
}
```

- `T` is a formal type, used as parameter type
- Above function returns variable of same type as parameter
- In function call, `T` replaced by real type

```cpp
int x;
int y = square(x);
```

Example

```cpp
// Fig. 3.27: fig03_27.cpp
// Using a function template.
#include <iostream>

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

// definition of function template maximum
template < class T >  // or template < typename T >
T maximum( T value1, T value2, T value3 )
{
  T max = value1;
  if ( value2 > max )
    max = value2;
  if ( value3 > max )
    max = value3;
  return max;
}

int main()
{
  // demonstrate maximum with int values
  int int1, int2, int3;
  cout << "Input three integer values: ";
  cin >> int1 >> int2 >> int3;
  cout << "The maximum integer value is: 
    " << maximum( int1, int2, int3 );

  // demonstrate maximum with double values
  double double1, double2, double3;
  cout << "Input three double values: ";
  cin >> double1 >> double2 >> double3;
  cout << "The maximum double value is: 
    " << maximum( double1, double2, double3 );
}
```

- `maximum` called with various data types
```cpp
// demonstrate maximum with char values
char char1, char2, char3;

cout << "Input three characters: 
";

cin >> char1 >> char2 >> char3;

// invoke char version of maximum
cout << "The maximum character value is: " <<
maximum( char1, char2, char3 ) << endl;

return 0; // indicates successful termination
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```