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);` or
  - `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 - 6x );`
- Other functions
  - `ceil(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 )
    {
    
    }

• Example function
  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
    - `double maximum( double, double, double );`

- Argument Coercion
  - Force arguments to be of proper type
    - Converting `int (4)` to `double (4.0)`
      `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

<table>
<thead>
<tr>
<th>Data types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>long double</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
</tr>
<tr>
<td>unsigned long int (synonymous with unsigned long)</td>
<td></td>
</tr>
<tr>
<td>long int</td>
<td>(synonymous with long)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>(synonymous with unsigned)</td>
</tr>
<tr>
<td>int</td>
<td></td>
</tr>
<tr>
<td>unsigned short int (synonymous with unsigned short)</td>
<td></td>
</tr>
<tr>
<td>short int</td>
<td>(synonymous with short)</td>
</tr>
<tr>
<td>unsigned char</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>(false becomes 0, true becomes 1)</td>
</tr>
</tbody>
</table>

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>
  ```

Enumeration: `enum`

- Enumeration
  - Set of integers with identifiers
    ```c
    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
  ```c
  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
    - **register int counter = 1;**
Storage Classes

- Static storage class
  - Variables exist for entire program
    - For functions, name exists for entire program
  - May not be accessible, scope rules still apply
- **auto** and **register** keyword
  - 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:**)
Scope Rules

• 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

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

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

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

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
  cout << "local x in main's outer scope is " << x << endl;
  {
    int x = 7;
    cout << "local x in main's inner scope is " << x << endl;
  } // end new scope
} // end main

© 2003 Prentice Hall, Inc. All rights reserved.
void useLocal( void )
{
    int x = 25;  // initialized each time useLocal is called
    cout << endl << "local x is " << x
       << " on entering useLocal" << endl;
    ++x;
    cout << "local x is " << x
       << " on exiting useLocal" << endl;
    // end function useLocal
}

void useStaticLocal( void )
{
    // initialized only first time useStaticLocal is called
    static int x = 50;
    cout << endl << "local static x is " << x
       << " on entering useStaticLocal" << endl;
    ++x;
    cout << "local static x is " << x
       << " on exiting useStaticLocal" << endl;
    // end function useStaticLocal
}
Recursion

- Recursive functions
  - 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

Example: factorial

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

- Recursive relationship \( n! = n \times (n-1)! \)
  \[ 5! = 5 \times 4! \]
  \[ 4! = 4 \times 3! \]
- 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:
    - \(fib(n) = fib(n-1) + fib(n-2)\)

- C++ code for Fibonacci function
  
  ```cpp
  long fibonacci( long n )
  {  
      if ??? // base case
        return ???;
      else
        ???
    }
  ```

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
    - \(30^{th}\) number = \(2^{30} - 4\) 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
  ```cpp
  inline double cube( const double s )
  {
    return s * s * s;
  }
  ```
  - `const` tells compiler that function does not modify `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 )`
    • Read “`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
    ```
    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
    • Reference to undefined variable
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
  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
    - 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
    - i.e., function to square int and function to square float
      
      ```cpp
      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
Function Templates

• 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
  - If `int`, all **T**'s become `ints`
    ```cpp
    int x;
    int y = square(x);
    ```
int main()
{
  // demonstrate maximum with int values
  int int1, int2, int3;
  cout << "Input three integer values: ";
  cin >> int1 >> int2 >> int3;
  // invoke int version of maximum
  cout << "The maximum integer value is: " << maximum(int1, int2, int3) << endl;

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

  // 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
} // end main