Answers to Mastery Checks

Module 1: C++ Fundamentals

1. C++ is at the center of modern programming because it was derived from C and is the parent of Java and C#. These are the four most important programming languages.

2. True, a C++ compiler produces code that can be directly executed by the computer.

3. Encapsulation, polymorphism, and inheritance are the three guiding principles of OOP.

4. C++ programs begin execution at **main( )**.

5. A header contains information used by the program.

6. **<iostream>** is the header the supports I/O. The statement includes the **<iostream>** header in a program.

7. A namespace is a declarative region in which various program elements can be placed. Elements declared in one namespace are separate from elements declared in another.

8. A variable is a named memory location. The contents of a variable can be changed during the execution of a program.

9. The invalid variables are d and e. Variable names cannot begin with a digit or be the same as a C++ keyword.

10. A single-line comment begins with // and ends at the end of the line.
    A multiline comment begins with /* and ends with */.

11. The general form of the **if**:

    if(*condition*) *statement*;

    The general form of the **for**:

    for(*initialization*; *condition*; *increment*) *statement*;

12. A block of code is started with a **{** and ended with a **}**.

13.
```
// Show a table of Earth to Moon weights.

#include <iostream>
using namespace std;

int main() {
  double earthweight; // weight on earth
  double moonweight;  // weight on moon
  int counter;

  counter = 0;
  for(earthweight = 1.0; earthweight <= 100.0; earthweight++) {
    moonweight = earthweight * 0.17;
    cout << earthweight << " earth-pounds is equivalent to " <<
                       moonweight << " moon-pounds.\n";
    counter++;
    if(counter == 25) {
      cout << "\n";
      counter = 0;
    }
  }

  return 0;
}
```

14.
```
// Convert Jovian years to Earth years.

#include <iostream>
using namespace std;

int main() {
  double e_years; // earth years
  double j_years;  // Jovian years

  cout << "Enter number of Jovian years: ";
  cin >> j_years;

  e_years = j_years * 12.0;

  cout << "Equivalent Earth years: " << e_years;

  return 0;
}
```

15. When a function is called, program control transfers to that function.

16.
```
// Average the absolute values of 5 numbers.

#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
  int i;
  double avg, val;

  avg = 0.0;
```

```
    for(i=0; i<5; i++) {
      cout << "Enter a value: ";
      cin >> val;

      avg = avg + abs(val);
    }
    avg = avg / 5;


    cout << "Average of absolute values: " << avg;

    return 0;
  }
```

Module 2: Introducing Data Types and Operators

1. The C++ integer types are

int	short int	long int
unsigned int	unsigned short int	unsigned long int
signed int	signed short int	signed long int

   The type **char** can also be used as an integer type.

2. 12.2 is type **double**.
3. A **bool** variable can be either **true** or **false**.
4. The long integer type is **long int**, or just **long**.
5. The **\t** sequence represents a tab. The **\b** rings the bell.
6. True, a string is surrounded by double quotes.
7. The hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
8. To initialize a variable, use this general form:

   type var = *value*;

9. The **%** is the modulus operator. It returns the remainder of an integer division. It cannot be used on floating-point values.
10. When the increment operator precedes its operand, C++ will perform the corresponding operation prior to obtaining the operand's value for use by the rest of the expression. If the operator follows its operand, then C++ will obtain the operand's value before incrementing.

11. A, C, and E

12. `x += 12;`

13. A cast is an explicit type conversion.

14. Here is one way to find the primes between 1 and 100. There are, of course, other solutions.

```
// Find prime numbers between 1 and 100.

#include <iostream>
using namespace std;

int main() {
  int i, j;
  bool isprime;

  for(i=1; i < 100; i++) {
    isprime = true;

    // see if the number is evenly divisible
    for(j=2; j <= i/2; j++)
      // if it is, then it is not prime
      if((i%j) == 0) isprime = false;

    if(isprime)
      cout << i << " is prime.\n";
  }

  return 0;
}
```

Module 3: Program Control Statements

1.
```
// Count periods.

#include <iostream>
using namespace std;

int main() {
  char ch;
  int periods = 0;

  cout << "Enter a $ to stop.\n";

  do {
    cin >> ch;
    if(ch == '.') periods++;
  } while(ch != '$');

  cout << "Periods: " << periods << "\n";

  return 0;
}
```

2. Yes. If there is no **break** statement concluding a **case** sequence, then execution will continue on into the next **case**. A **break** statement prevents this from happening.

3.
```
if(condition)
  statement;
else if(condition)
  statement;
else if(condition)
  statement;
.
.
.
else
  statement;
```

4. The last **else** associates with the outer **if**, which is the nearest **if** at the same level as the **else**.

5. `for(int i = 1000; i >= 0; i -= 2) // ...`

6. No. According to the ANSI/ISO C++ Standard, **i** is not known outside of the **for** loop in which it is declared. (Note that some compilers may handle this differently.)

7. A **break** causes termination of its immediately enclosing loop or **switch** statement.

8. After **break** executes, “after while” is displayed.

9.
```
0 1
2 3
4 5
6 7
8 9
```

10.
```
/*
   Use a for loop to generate the progression

   1 2 4 8 16, ...
*/
```

```
#include <iostream>
using namespace std;

int main() {

  for(int i = 1; i < 100; i += i)
    cout << i << " ";

  cout << "\n";

  return 0;
}
```

11.
```
// Change case.

#include <iostream>
using namespace std;

int main() {
  char ch;
  int changes = 0;

  cout << "Enter period to stop.\n";

  do {
    cin >> ch;
    if(ch >= 'a' && ch <= 'z') {
      ch -= (char) 32;
      changes++;
      cout << ch;
    }
    else if(ch >= 'A' && ch <= 'Z') {
      ch += (char) 32;
      changes++;
      cout << ch;
    }
  } while(ch != '.');

  cout << "\nCase changes: " << changes << "\n";

  return 0;
}
```

12. C++'s unconditional jump statement is the **goto**.

Module 4: Arrays, Strings, and Pointers

1. `short int hightemps[31];`

2. zero

3.
```
// Find duplicates

#include <iostream>
using namespace std;

int main()
{
  int nums[] = {1, 1, 2, 3, 4, 2, 5, 4, 7, 7};

  for(int i=0; i < 10; i++)
    for(int j=i+1; j < 10; j++)
      if(nums[i] == nums[j])
         cout << "Duplicate: " << nums[i] << "\n";

  return 0;
}
```

4. A null-terminated string is an array of characters that ends with a null.

5.
```
// Ignore case when comparing strings.

#include <iostream>
#include <cctype>
using namespace std;

int main()
{
  char str1[80];
  char str2[80];
  char *p1, *p2;

  cout << "Enter first string: ";
  cin >> str1;
  cout << "Enter second string: ";
  cin >> str2;

  p1 = str1;
  p2 = str2;
  // loop as long as p1 and p2 point to non-null characters
  while(*p1 && *p2) {
    if(tolower(*p1) != tolower(*p2)) break;
    else {
      p1++;
      p2++;
    }
  }

  /* strings are the same if both p1 and p2 point
     to the null terminator.
  */
  if(!*p1 && !*p2)
    cout << "Strings are the same except for " <<
         "possible case differences.\n";
  else
    cout << "Strings differ\n";

  return 0;
}
```

6. When using **strcat( )**, the recipient array must be large enough to hold the contents of both strings.

7. In a multidimensional array, each index is specified within its own set of brackets.

8. `int nums[] = {5, 66, 88};`

9. An unsized array declaration ensures that an initialized array is always large enough to hold the initializers being specified.

10. A pointer is an object that contains a memory address. The pointer operators are **&** and *****.

11. Yes, a pointer can be indexed like an array. Yes, an array can be accessed through a pointer.

12.
```
// Count uppercase letters.
#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;

int main()
{
  char str[80];
  int i;
  int count;

  strcpy(str, "This Is A Test");
  count = 0;
  for(i=0; str[i]; i++)
    if(isupper(str[i])) count++;

  cout << str << " contains " << count << " uppercase letters.";

  return 0;
}
```

13. *Multiple indirection* is the term used for the situation in which one pointer points to another.

14. By convention, a null pointer is assumed to be unused.

Module 5: Introducing Functions

1. The general form of a function is

   return-type name(*parameter list*)
   {
     // *body of function*
   }

2.
```
#include <iostream>
#include <cmath>
using namespace std;

double hypot(double a, double b);

int main() {

  cout << "Hypotenuse of a 3 by 4 right triangle: ";
  cout << hypot(3.0, 4.0) << "\n";

  return 0;
}

double hypot(double a, double b)
{
  return sqrt((a*a) + (b*b));
}
```

3. Yes, a function can return a pointer. No, a function cannot return an array.

4.
```
// A custom version of strlen().
#include <iostream>
using namespace std;
```

```
int mystrlen(char *str);

int main()
{
  cout << "Length of Hello There is: ";
  cout << mystrlen("Hello There");

  return 0;
}

// A custom version of strlen().
int mystrlen(char *str)
{
  int i;

  for(i=0; str[i]; i++) ; // find the end of the string

  return i;
}
```

5. No, a local variable's value is lost when its function returns. (Or, more generally, its value is lost when its block is exited.)

6. The main advantages to global variables are that they are available to all other functions in the program and that they stay in existence during the entire lifetime of the program. Their main disadvantages are that they take up memory the entire time the program is executing, using a global where a local variable will do makes a function less general, and using a large number of global variables can lead to unanticipated side effects.

7.
```
#include <iostream>
using namespace std;

int seriesnum = 0;

int byThrees();
void reset();

int main() {

  for(int i=0; i < 10; i++)
    cout << byThrees() << " ";

  cout << "\n";

  reset();

  for(int i=0; i < 10; i++)
```

```
      cout << byThrees() << " ";

    cout << "\n";

    return 0;
  }


  int byThrees()
  {
    int t;

    t = seriesnum;
    seriesnum += 3;

    return t;
  }

  void reset()
  {
    seriesnum = 0;
  }
```

8.
```
#include <iostream>
#include <cstring>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc != 2) {
    cout << "Password required!\n";
    return 0;
  }

  if(!strcmp("mypassword", argv[1]))
    cout << "Access permitted.\n";
  else
    cout << "Access denied.\n";

  return 0;
}
```

9. True. A prototype prevents a function from being called with the improper number of arguments.

10.
```
#include <iostream>
using namespace std;
```

```
void printnum(int n);

int main()
{
  printnum(10);

  return 0;
}

void printnum(int n)
{
  if(n > 1) printnum(n-1);
  cout << n << " ";
}
```

Module 6: A Closer Look at Functions

1. An argument can be passed to a subroutine using call-by-value or call-by-reference.
2. A reference is an implicit pointer. A reference parameter is created by preceding the parameter name with an **&**.
3. `f(ch, &i);`
4.
```
#include <iostream>
#include <cmath>
using namespace std;

void round(double &num);

int main()
{
  double i = 100.4;

  cout << i << " rounded is ";
  round(i);
  cout << i << "\n";

  i = -10.9;
  cout << i << " rounded is ";
  round(i);
  cout << i << "\n";

  return 0;
}
```

```
void round(double &num)
{
  double frac;
  double val;

  // decompose num into whole and fractional parts
  frac =  modf(num, &val);

  if(frac < 0.5) num = val;
  else num = val+1.0;
}
```

5.
```
#include <iostream>
using namespace std;

// Swap args and return minimum.
int & min_swap(int &x, int &y);

int main()
{
  int i, j, min;

  i = 10;
  j = 20;

  cout << "Initial values of i and j: ";
  cout << i << ' ' << j << '\n';

  min = min_swap(j, i);

  cout << "Swapped values of i and j: ";
  cout << i << ' ' << j << '\n';


  cout << "Minimum value is " << min << "\n";
  return 0;
}

// Swap args and return minimum.
int &min_swap(int &x, int &y)
{
  int temp;

  // use references to exchange the values of the arguments
  temp = x;
  x = y;
  y = temp;

  // return reference to minimum arg
  if(x < y) return x;
  else return y;
}
```

6. A function should not return a reference to a local variable, because that variable will go out-of-scope (that is, cease to exist) when the function returns.

7. Overloaded functions must differ in the type and/or number of their parameters.

8.
```
/*
   Project 6-1 -- updated for Mastery Check

   Create overloaded println() functions
   that display various types of data.

   This version includes an indentation parameter.
*/

#include <iostream>
using namespace std;

// These output a newline.
void println(bool b, int ident=0);
void println(int i, int ident=0);
void println(long i, int ident=0);
void println(char ch, int ident=0);
void println(char *str, int ident=0);
void println(double d, int ident=0);

// These functions do not output a newline.
void print(bool b, int ident=0);
void print(int i, int ident=0);
void print(long i, int ident=0);
void print(char ch, int ident=0);
void print(char *str, int ident=0);
void print(double d, int ident=0);

int main()
{
  println(true, 10);
  println(10, 5);
  println("This is a test");
  println('x');
  println(99L, 10);
  println(123.23, 10);

  print("Here are some values: ");
  print(false);
  print(88, 3);
  print(100000L, 3);
  print(100.01);

  println(" Done!");

  return 0;
}

// Here are the println() functions.
void println(bool b, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';

  if(b) cout << "true\n";
  else cout << "false\n";
}

void println(int i, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';

  cout << i << "\n";
}

void println(long i, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';

  cout << i << "\n";
}

void println(char ch, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';

  cout << ch << "\n";
}
```

```
void println(char *str, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';

  cout << str << "\n";
}

void println(double d, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';

  cout << d << "\n";
}

// Here are the print() functions.
void print(bool b, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';

  if(b) cout << "true";
  else cout << "false";
}

void print(int i, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';

  cout << i;
}

void print(long i, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';

  cout << i;
}

void print(char ch, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';
  cout << ch;
}

void print(char *str, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';

  cout << str;
}

void print(double d, int ident)
{
  if(ident)
    for(int i=0; i < ident; i++) cout << ' ';

  cout << d;
}
```

9.
```
myfunc('x');
myfunc('x', 19);
myfunc('x', 19, 35);
```

10. Function overloading can introduce ambiguity when the compiler cannot decide which version of the function to call. This can occur when automatic type conversions are involved and when default arguments are used.

Module 7: More Data Types and Operators

1. `static int test = 100;`

2. True. The **volatile** specifier tells the compiler that a variable might be changed by forces outside the program.

3. In a multifile project, to tell one file about a global variable declared in another file, use **extern**.

4. The most important attribute of a **static** local variable is that it holds its value between function calls.

5.
```
// Use static to count function invocations.

#include <iostream>
using namespace std;

int counter();

int main()
```

```
{
  int result;

  for(int i=0; i<10; i++)
    result = counter();

  cout << "Function called " <<
       result << " times." << "\n";

  return 0;
}

int counter()
{
  static count = 0;

  count++;

  return count;
}
```

6. Specifying **x** as **register** will have the most impact on performance, followed by **y**, and then **z**. The reason is that **x** is accessed most frequently within the loop, **y** the second most, and **z** is used only when the loop is initialized.

7. The **&** is a bitwise operator that acts on the individual bits within a value. **&&** is a logical operator that acts on true/false values.

8. The statement multiplies the current value of **x** by 10 and assigns that result to **x**. It is the same as

```
x = x * 10;
```

9.
```
// Use rotations to encode a message.

#include <iostream>
#include <cstring>
using namespace std;

unsigned char rrotate(unsigned char val, int n);
unsigned char lrotate(unsigned char val, int n);
void show_binary(unsigned int u);

int main()
{
  char msg[] = "This is a test.";
  char *key = "xanadu";
  int klen = strlen(key);
  int rotnum;

  cout << "Original message: " << msg << "\n";

  // Encode the message by left-rotating.
  for(int i = 0 ; i < strlen(msg); i++) {
    /* Left-rotate each letter by a value
       derived from the key string. */
    rotnum = key[i%klen] % 8;
    msg[i] = lrotate(msg[i], rotnum);
  }

  cout << "Encoded message: " << msg << "\n";

  // Decode the message by right-rotating.
  for(int i = 0 ; i < strlen(msg); i++) {
    /* Right-rotate each letter by a value
       derived from the key string. */
    rotnum = key[i%klen] % 8;

    msg[i] = rrotate(msg[i], rotnum);
  }

  cout << "Decoded message: " << msg << "\n";

  return 0;
}

// Left-rotate a byte n places.
unsigned char lrotate(unsigned char val, int n)
{
  unsigned int t;

  t = val;

  for(int i=0; i < n; i++) {
    t = t << 1;

    /* If a bit shifts out, it will be in bit 8
       of the integer t. If this is the case,
       put that bit on the right side. */
    if(t & 256)
      t = t | 1; // put a 1 on the right end
  }

  return t; // return the lower 8 bits.
```

```
}

// Right-rotate a byte n places.
unsigned char rrotate(unsigned char val, int n)
{
  unsigned int t;

  t = val;

  // First, move the value 8 bits higher.
  t = t << 8;

  for(int i=0; i < n; i++) {
    t = t >> 1;

    /* If a bit shifts out, it will be in bit 7
       of the integer t. If this is the case,
       put that bit on the left side. */
    if(t & 128)
      t = t | 32768; // put a 1 on left end
  }

  /* Finally, move the result back to the
     lower 8 bits of t. */
  t = t >> 8;

  return t;
}

// Display the bits within a byte.
void show_binary(unsigned int u)
{
  int t;

  for(t=128; t>0; t = t/2)
    if(u & t) cout << "1 ";
    else cout << "0 ";

  cout << "\n";
}
```

Module 8: Classes and Objects

1. A class is a logical construct that defines the form of an object. An object is an instance of a class. Thus, an object has physical reality within memory.

2. To define a class, use the **class** keyword.

3. Each object has its own copy of the member variables of a class.

4.
```
class Test {
  int count;
  int max;
  // ...
}
```

5. A constructor has the same name as its class. A destructor has the same name as its class except that it is preceded by a ~.

6. Here are three ways to create an object that initializes **i** to 10:

```
Sample ob(10);
Sample ob = 10;
Sample ob = Sample(10);
```

7. When a member function is declared within a class, it is automatically inlined, if possible.

8.
```
// Create a Triangle class.

#include <iostream>
#include <cmath>
using namespace std;

class Triangle {
  double height;
  double base;
public:
  Triangle(double h, double b) {
    height = h;
    base = b;
  }

  double hypot() {
    return sqrt(height*height + base*base);
  }

  double area() {
    return base * height / 2.0;
  }
};

int main()
{
  Triangle t1(3.0, 4.0);
  Triangle t2(4.5, 6.75);

  cout << "Hypotenuse of t1: " <<
    t1.hypot() << "\n";
  cout << "Area of t1: " <<
    t1.area() << "\n";

  cout << "Hypotenuse of t2: " <<
    t2.hypot() << "\n";
  cout << "Area of t2: " <<
    t2.area() << "\n";

  return 0;
}
```

9.

```
/*
   Enhanced Project 8-1

   Add user ID to Help class.
*/

#include <iostream>
using namespace std;

// A class that encapsulates a help system.
class Help {
  int userID;
public:
  Help(int id) { userID = id; }

  ~Help() { cout << "Terminating help for #" <<
            userID << ".\n"; }

  int getID() { return userID; }

  void helpon(char what);
  void showmenu();
  bool isvalid(char ch);
};

// Display help information.
void Help::helpon(char what) {
  switch(what) {
    case '1':
      cout << "The if:\n\n";
      cout << "if(condition) statement;\n";
```

```
        cout << "else statement;\n";
        break;
      case '2':
        cout << "The switch:\n\n";
        cout << "switch(expression) {\n";
        cout << "  case constant:\n";
        cout << "    statement sequence\n";
        cout << "    break;\n";
        cout << "  // ...\n";
        cout << "}\n";
        break;
      case '3':
        cout << "The for:\n\n";
        cout << "for(init; condition; increment)";
        cout << " statement;\n";
        break;
      case '4':
        cout << "The while:\n\n";
        cout << "while(condition) statement;\n";
        break;
      case '5':
        cout << "The do-while:\n\n";
        cout << "do {\n";
        cout << "  statement;\n";
        cout << "} while (condition);\n";
        break;
      case '6':
        cout << "The break:\n\n";
        cout << "break;\n";
        break;
      case '7':
        cout << "The continue:\n\n";
        cout << "continue;\n";
        break;
      case '8':
        cout << "The goto:\n\n";
        cout << "goto label;\n";
        break;
    }
    cout << "\n";
  }

  // Show the help menu.
  void Help::showmenu() {
```

```
  cout << "Help on:\n";
  cout << "  1. if\n";
  cout << "  2. switch\n";
  cout << "  3. for\n";
  cout << "  4. while\n";
  cout << "  5. do-while\n";
  cout << "  6. break\n";
  cout << "  7. continue\n";
  cout << "  8. goto\n";
  cout << "Choose one (q to quit): ";
}

// Return true if a selection is valid.
bool Help::isvalid(char ch) {
  if(ch < '1' || ch > '8' && ch != 'q')
    return false;
  else
    return true;
}

int main()
{
  char choice;
  Help hlpob(27); // create an instance of the Help class.

  cout << "User ID is " << hlpob.getID() <<
          ".\n";

  // Use the Help object to display information.
  for(;;) {
    do {
      hlpob.showmenu();
      cin >> choice;
    } while(!hlpob.isvalid(choice));

    if(choice == 'q') break;
    cout << "\n";

    hlpob.helpon(choice);
  }

  return 0;
}
```

Module 9: A Closer Look at Classes

1. A copy constructor makes a copy of an object. It is called when one object initializes another. Here is the general form:

    classname (const *classname* &*obj*) {
      // body of constructor
    }

2. When an object is returned by a function, a temporary object is created as the return value. This object is destroyed by the object's destructor after the value has been returned.

3.
```
int sum() {
  return this.i + this.j;
}
```

4. A structure is a class in which members are public by default. A union is a class in which all data members share the same memory. Union members are also public by default.

5. ***this** refers to the object on which the function was called.

6. A **friend** function is a nonmember function that is granted access to the private members of the class for which it is a friend.

7. *type classname*::operator#(type *op2*)
    {
      // left operand passed via "this"
    }

8. To allow operations between a class type and a built-in type, you must use two **friend** operator functions, one with the class type as the first parameter, and one with the built-in type as the first parameter.

9. No, the **?** cannot be overloaded. No, you cannot change the precedence of an operator.

10.
```
// Determine if one set is a subset of another.
bool Set::operator <(Set ob2) {
  if(len > ob2.len) return false; // ob1 has more elements

  for(int i=0; i < len; i++)
    if(ob2.find(members[i]) == -1) return false;
  return true;
}

// Determine if one set is a superset of another.
bool Set::operator >(Set ob2) {
  if(len < ob2.len) return false; // ob1 has fewer elements
```

```
    for(int i=0; i < ob2.len; i++)
      if(find(ob2.members[i]) == -1) return false;
    return true;
  }
```

11.
```
// Set intersection.
Set Set::operator &(Set ob2) {
  Set newset;

  // Add elements common to both sets.
  for(int i=0; i < len; i++)
    if(ob2.find(members[i]) != -1) // add if element in both sets
      newset = newset + members[i];

  return newset; // return set
}
```

Module 10: Inheritance, Virtual Functions, and Polymorphism

1. A class that is inherited is called a *base* class. The class that does the inheriting is called a *derived* class.

2. A base class does *not* have access to the members of derived classes, because a base class has no knowledge of derived classes. A derived class *does* have access to the non-private members of its base class(es).

3.
```
// A circle class.
class Circle : public TwoDShape {
public:
  Circle(double r) : TwoDShape(r) { } // specify radius

  double area() {
    return getWidth() * getWidth() * 3.1416;
  }
};
```

4. To prevent a derived class from having access to a member of a base class, declare that member as private in the base class.

5. Here is the general form of a derived class constructor that calls a base class constructor:

   derived-class( ) : *base-class*( ) { // ...

6. Constructors are always called in order of derivation. Thus, when a **Gamma** object is created, the constructors are called in this order: **Alpha**, **Beta**, **Gamma**.

7. A **protected** member in a base class can be accessed by its own class and by derived classes. It is private, otherwise.

8. When a virtual function is called through a base class pointer, it is the type of the object being pointed to that determines which version of the function will be called.

9. A pure virtual function is a function that has no body inside its base class. Thus, a pure virtual function must be overridden by derived classes. An abstract class is a class that contains at least one pure virtual function.

10. No, an abstract class cannot be used to create an object.

11. A pure virtual function represents a generic description that all implementations of that function must adhere to. Thus, in the phrase "one interface, multiple methods," the pure virtual function represents the *interface,* and the individual implementations represent the *methods.*

Module 11: The C++ I/O System

1. The predefined streams are **cin**, **cout**, **cerr**, and **clog**.

2. Yes, C++ defines both 8-bit and wide-character streams.

3. The general form for overloading an inserter is shown here:

```
ostream &operator<<(ostream &stream, class_type obj)
{
  // class specific code goes here
  return stream;  // return the stream
}
```

4. **ios::scientific** causes numeric output to be displayed in scientific notation.

5. The **width( )** function sets the field width.

6. True, an I/O manipulator is used within an I/O expression.

7. Here is one way to open a file for text input:

```
ifstream in("test");
if(!in) {
  cout << "Cannot open file.\n";
  return 1;
}
```

8. Here is one way to open a file for text output:

```
ofstream out("test");
if(!out) {
  cout << "Cannot open file.\n";
  return 1;
}
```

9. **ios::binary** specifies that a file be opened for binary rather than text-based I/O.

10. True, at end-of-file, the stream variable will evaluate as false.

11. `while(strm.get(ch)) // ...`

12. There are many solutions. The following shows just one way:

```
// Copy a file.

#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Usage: copy <source> <target>\n";
    return 1;
  }

  ifstream src(argv[1], ios::in | ios::binary);
  if(!src) {
    cout << "Cannot open source file.\n";
    return 1;
  }

  ofstream targ(argv[2], ios::out | ios::binary);
  if(!targ) {
    cout << "Cannot open target file.\n";
    return 1;
  }

  do {
    src.get(ch);
    if(!src.eof()) targ.put(ch);
  } while(!src.eof());

  src.close();
  targ.close();

  return 0;
}
```

13. There are many solutions. The following shows one simple way:

```
// Merge two files.

#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc != 4) {
    cout << "Usage: merge <source1> <source2> <target>\n";
    return 1;
  }

  ifstream src1(argv[1], ios::in | ios::binary);
  if(!src1) {
    cout << "Cannot open 1st source file.\n";
    return 1;
  }

  ifstream src2(argv[2], ios::in | ios::binary);
  if(!src2) {
    cout << "Cannot open 2nd source file.\n";
    return 1;
  }

  ofstream targ(argv[3], ios::out | ios::binary);
  if(!targ) {
    cout << "Cannot open target file.\n";
    return 1;
  }

  // Copy first source file.
  do {
    src1.get(ch);
    if(!src1.eof()) targ.put(ch);
  } while(!src1.eof());

  // Copy second source file.
  do {
    src2.get(ch);
    if(!src2.eof()) targ.put(ch);
  } while(!src2.eof());
  src1.close();
  src2.close();
  targ.close();

  return 0;
}
```

14. `MyStrm.seekg(300, ios::beg);`

Module 12: Exceptions, Templates, and Other Advanced Topics

1. C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**. In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block. If an exception (that is, an error) occurs within the **try** block, it is thrown (using **throw**). The exception is caught, using **catch**, and processed.

2. When catching exceptions of both base and derived classes, the derived classes must precede the base class in a **catch** list.

3. `void func() throw(MyExcpt)`

4. Here is one way to add an exception to **Queue**. It is one of many solutions.

```
/*
   Add an exception to Project 12-1

   A template queue class.
*/
#include <iostream>
#include <cstring>
using namespace std;

// This is the exception thrown by Queue on error.
class QExcpt {
public:
  char msg[80];
};

const int maxQsize = 100;

// This creates a generic queue class.
template <class QType> class Queue {
  QType q[maxQsize]; // this array holds the queue
  int size; // maximum number of elements that the queue can store
```

```
    int putloc, getloc; // the put and get indices
    QExcpt Qerr; // add an exception field
  public:

    // Construct a queue of a specific length.
    Queue(int len) {
      // Queue must be less than max and positive.
      if(len > maxQsize) len = maxQsize;
      else if(len <= 0) len = 1;

      size = len;
      putloc = getloc = 0;
    }

    // Put data into the queue.
    void put(QType data) {
      if(putloc == size) {
        strcpy(Qerr.msg, "Queue is full.\n");
        throw Qerr;
      }

      putloc++;
      q[putloc] = data;
    }

    // Get data from the queue.
    QType get() {
      if(getloc == putloc) {
        strcpy(Qerr.msg, "Queue is empty.\n");
        throw Qerr;
      }

      getloc++;
      return q[getloc];
    }
  };

  // Demonstrate the generic Queue.
  int main()
  {
    // notice that iQa is only 2 elements long
    Queue<int> iQa(2), iQb(10);

    try {
      iQa.put(1);
      iQa.put(2);
      iQa.put(3); // this will overflow!

      iQb.put(10);
      iQb.put(20);
      iQb.put(30);

      cout << "Contents of integer queue iQa: ";
      for(int i=0; i < 3; i++) // this will underflow!
        cout << iQa.get() << " ";
      cout << endl;

      cout << "Contents of integer queue iQb: ";
      for(int i=0; i < 3; i++)
        cout << iQb.get() << " ";
      cout << endl;

      Queue<double> dQa(10), dQb(10);  // create two double queues

      dQa.put(1.01);
      dQa.put(2.02);
      dQa.put(3.03);

      dQb.put(10.01);
      dQb.put(20.02);
      dQb.put(30.03);

      cout << "Contents of double queue dQa: ";
      for(int i=0; i < 3; i++)
        cout << dQa.get() << " ";
      cout << endl;

      cout << "Contents of double queue dQb: ";
      for(int i=0; i < 3; i++)
        cout << dQb.get() << " ";
      cout << endl;

    } catch(QExcpt exc) {
      cout << exc.msg;
    }

    return 0;
  }
```

5. A generic function defines the general form of a routine, but does not specify the precise type of data upon which it operates. It is created using the keyword **template**.

6. Here is one way to make **quicksort( )** and **qs( )** into generic functions:

```
// A generic Quicksort.

#include <iostream>
#include <cstring>

using namespace std;


// Set up a call to the actual sorting function.
template <class X> void quicksort(X *items, int len)
{
  qs(items, 0, len-1);
}

// A generic version of Quicksort.
template <class X> void qs(X *items, int left, int right)
{
  int i, j;
  X x, y;

  i = left; j = right;
  x = items[( left+right) / 2 ];

  do {
    while((items[i] < x) && (i < right)) i++;
    while((x < items[j]) && (j > left)) j--;

    if(i <= j) {
      y = items[i];
      items[i] = items[j];
      items[j] = y;
      i++; j--;
    }
  } while(i <= j);

  if(left < j) qs(items, left, j);
  if(i < right) qs(items, i, right);
}

int main() {

  // sort characters.
  char str[] = "jfmckldoelazlkper";

  cout << "Original order: " << str << "\n";
  quicksort(str, strlen(str));

  cout << "Sorted order: " << str << "\n";

  // sort integers
  int nums[] = { 4, 3, 7, 5, 9, 8, 1, 3, 5, 4 };

  cout << "Original order: ";
  for(int i=0; i < 10; i++)
    cout << nums[i] << " ";
  cout << endl;

  quicksort(nums, 10);

  cout << "Sorted order: ";
  for(int i=0; i < 10; i++)
    cout << nums[i] << " ";
  cout << endl;


  return 0;
}
```

7. Here is one way to store **Sample** objects in a **Queue**:

```
/*
   Use Project 12-1 to store Sample objects.

   A template queue class.
*/
#include <iostream>
using namespace std;

class Sample {
  int id;
public:
  Sample() { id = 0; }
  Sample(int x) { id = x; }
  void show() { cout << id << endl; }
};

const int maxQsize = 100;

// This creates a generic queue class.
template <class QType> class Queue {
  QType q[maxQsize]; // this array holds the queue
  int size; // maximum number of elements that the queue can store
  int putloc, getloc; // the put and get indices
public:

  // Construct a queue of a specific length.
  Queue(int len) {
    // Queue must be less than max and positive.
    if(len > maxQsize) len = maxQsize;
    else if(len <= 0) len = 1;

    size = len;
    putloc = getloc = 0;
  }

  // Put data into the queue.
  void put(QType data) {
    if(putloc == size) {
      cout << " -- Queue is full.\n";
      return;
    }

    putloc++;
    q[putloc] = data;
  }

  // Get data from the queue.
  QType get() {
    if(getloc == putloc) {
      cout << " -- Queue is empty.\n";
      return 0;
    }

    getloc++;
    return q[getloc];
  }
};

// Demonstrate the generic Queue.
int main()
{
  Queue<Sample> sampQ(3);

  Sample o1(1), o2(2), o3(3);

  sampQ.put(o1);
  sampQ.put(o2);
  sampQ.put(o3);
  cout << "Contents of sampQ:\n";
  for(int i=0; i < 3; i++)
    sampQ.get().show();
  cout << endl;

  return 0;
}
```

8. Here, the **Sample** objects are allocated:

```
/*
   Use Project 12-1 to store Sample objects.

   Allocate the Sample objects dynamically.

   A template queue class.
*/
#include <iostream>
using namespace std;

class Sample {
  int id;
public:
  Sample() { id = 0; }
  Sample(int x) { id = x; }
  void show() { cout << id << endl; }
};

const int maxQsize = 100;

// This creates a generic queue class.
template <class QType> class Queue {
  QType q[maxQsize]; // this array holds the queue
  int size; // maximum number of elements that the queue can store
  int putloc, getloc; // the put and get indices
public:

  // Construct a queue of a specific length.
  Queue(int len) {
    // Queue must be less than max and positive.
    if(len > maxQsize) len = maxQsize;
    else if(len <= 0) len = 1;

    size = len;
    putloc = getloc = 0;
  }
  // Put data into the queue.
  void put(QType data) {
    if(putloc == size) {
      cout << " -- Queue is full.\n";
      return;
    }

    putloc++;
    q[putloc] = data;
  }

  // Get data from the queue.
  QType get() {
    if(getloc == putloc) {
      cout << " -- Queue is empty.\n";
      return 0;
    }

    getloc++;
    return q[getloc];
  }
};

// Demonstrate the generic Queue.
int main()
{
  Queue<Sample> sampQ(3);

  Sample *p1, *p2, *p3;

  p1 = new Sample(1);
  p2 = new Sample(2);
  p3 = new Sample(3);

  sampQ.put(*p1);
  sampQ.put(*p2);
  sampQ.put(*p3);

  cout << "Contents of sampQ:\n";
  for(int i=0; i < 3; i++)
    sampQ.get().show();
  cout << endl;

  delete(p1);
  delete(p2);
  delete(p3);
  return 0;
}
```

9. To declare a namespace called **RobotMotion** use:

```
namespace RobotMotion {
  // ...
}
```

10. The C++ standard library is contained in the **std** namespace.

11. No, a **static** member function cannot access the non-**static** data of a class.

12. The **typeid** operator obtains the type of an object at runtime.

13. To determine the validity of a polymorphic cast at runtime, use **dynamic_cast**.

14. **const_cast** overrides **const** or **volatile** in a cast.