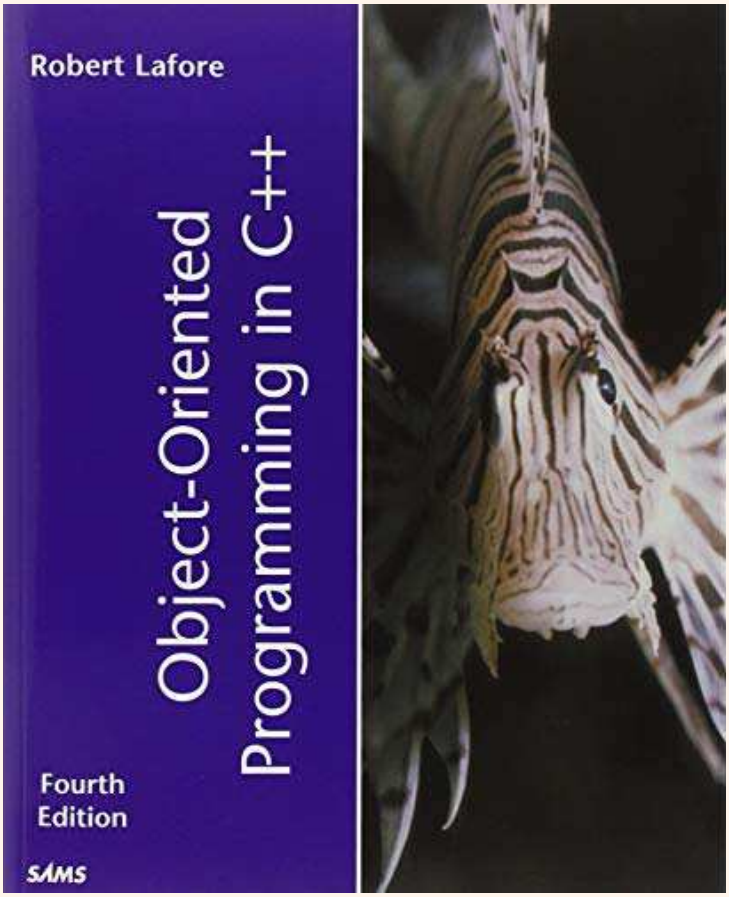


Chapter 5

Functions

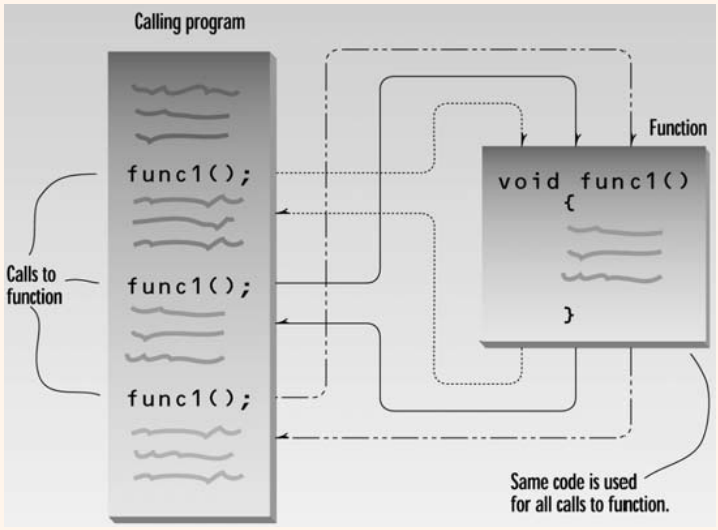
Animated Version

Topics

- Simple Functions
- Passing Arguments to Functions
- Returning Values from Functions
- Reference Arguments
- Overloaded Functions
- Recursion
- Inline Functions
- Default Arguments
- Scope and Storage Class
- Returning by Reference
- `const` Function Arguments

Function

- A function groups a number of program statements into a unit and gives it a name.
- Important reason to use functions
  - to aid in the conceptual organization of a program
  - to reduce program size. the course of the program.
- The function's code is stored in only one place in memory, even though the function is executed many times in the course of the program.



Simple Functions

- The Function Declarations
  - –tells the compiler that at some later point we plan to present a function 'starline'
  - –Terminated with semicolon
  - –also called prototypes
  - –The information in the declaration (the return type and the number and types of any arguments) is also sometimes referred to as the function signature.
- Calling the Function

```
// table.cpp
// demonstrates simple function
#include <iostream>
using namespace std;

void starline();                        //function declaration
                                        //    (prototype)
int main()
   {
   starline();                          //call to function
   cout << "Data type   Range" << endl;
   starline();                          //call to function
   cout << "char         -128 to 127" << endl
        << "short        -32,768 to 32,767" << endl
        << "int          System dependent" << endl
        << "long         -2,147,483,648 to 2,147,483,647" << endl;
   starline();                          //call to function
   return 0;
   }
//--------------------------------------------------------------
// starline()
// function definition
void starline()                         //function declarator
   {
   for(int j=0; j<45; j++)              //function body
      cout << '*';
   cout << endl;
   }
```

```
Output:
*********************************************
Data type       Range
*********************************************
Char            -128 to 127
Short           -32,768 to 32,767
Int             System dependent
Long        -2,147,483,648 to 2,147,483,647
*********************************************
```

Simple Functions (2)

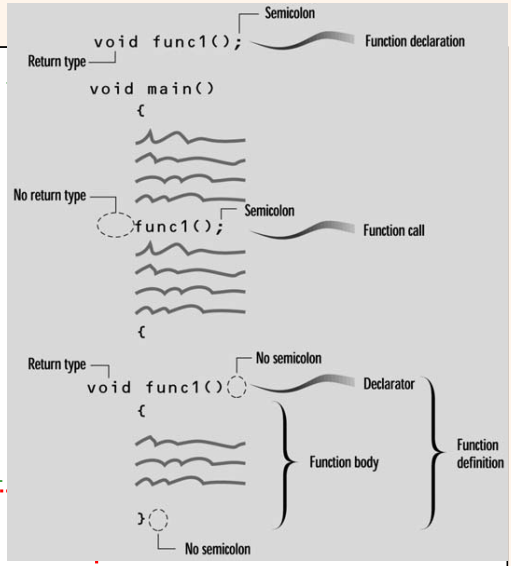
- The Function Definition

TABLE 5.1 Function Components

Component	*Purpose*	*Example*
Declaration (prototype)	Specifies function name, argument types, and return value. Alerts compiler (and programmer) that a function is coming up later.	`void func();`
Call	Causes the function to be executed.	`func();`

Definition	The function itself. Contains the lines of code that constitute the function.	`void func()`
		`{`
		`// lines of code`
		`}`

Declarator	First line of definition.	`void func()`




```
// table.cpp
// demonstrates simple
#include <iostream>
using namespace std;

void starline();

int main()
   {
   starline();
   cout << "Data type
   starline();
   cout << "char
        << "short
        << "int
        << "long
   starline();
   return 0;

//-----------------
// starline()
// function definition
void starline()
   {
   for(int j=0; j<45; j++)              //function body
      cout << '*';
   cout << endl;
   }
```

```
Output:
*********************************************
Data type      Range
*********************************************
Char           -128 to 127
Short          -32,768 to 32,767
Int            System dependent
Long       -2,147,483,648 to 2,147,483,647
*********************************************
```

- Library Functions
  - don't need to write the declaration or definition
  - declaration is in the header file specified at the beginning of the program
  - The definition is in a library file that's linked automatically during program build time

Simple Functions (3)

- Eliminating the Declaration
  - place the function definition (the function itself) in the listing before the first call to the function.

```
// table2.cpp
// demonstrates function definition preceding function calls
#include <iostream>
using namespace std;                    //no function declaration
//-----------------------------------------------------------------
// starline()                           //function definition
void starline()
   {
   for(int j=0; j<45; j++)
      cout << '*';
   cout << endl;
   }
//-----------------------------------------------------------------
int main()                              //main() follows function
   {
   starline();                          //call to function
   cout << "Data type   Range" << endl;
   starline();                          //call to function
   cout << "char        -128 to 127" << endl
        << "short       -32,768 to 32,767" << endl
        << "int         System dependent" << endl
        << "long        -2,147,483,648 to 2,147,483,647" << endl;
   starline();                          //call to function
   return 0;
   }
```

```
Output:
*********************************************
Data type      Range
*********************************************
Char           -128 to 127
Short          -32,768 to 32,767
Int            System dependent
Long       -2,147,483,648 to 2,147,483,647
*********************************************
```

Passing Arguments to Functions

- An argument is a piece of data (an int value, for example) passed from a program to the function.
- Arguments allow a function to operate with different values, or even to do different things, depending on the requirements of the program calling it.

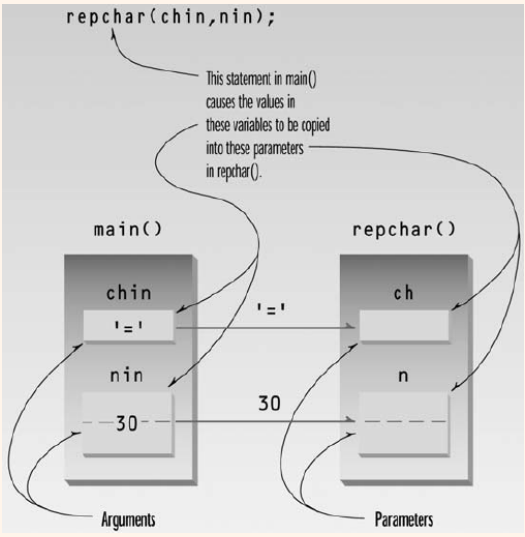
```
// tablearg.cpp
// demonstrates function arguments
#include <iostream>
using namespace std;
void repchar(char, int);                    //function declaration

int main()
   {
   repchar('-', 43);                        //call to function
   cout << "Data type   Range" << endl;
   repchar('=', 23);                        //call to function
   cout << "char        -128 to 127" << endl
        << "short       -32,768 to 32,767" << endl
        << "int         System dependent" << endl
        << "double      -2,147,483,648 to 2,147,483,647" << endl;
   repchar('-', 43);                        //call to function
   return 0;
   }
//--------------------------------------------------------------
// repchar()
// function definition
void repchar(char ch, int n)                //function declarator
   {
   for(int j=0; j<n; j++)                   //function body
      cout << ch;
   cout << endl;
   }
```

Output:

```
-------------------------------------------
Data type     Range
=======================
Char          -128 to 127
Short         -32,768 to 32,767
Int           System dependent
Long          -2,147,483,648 to 2,147,483,647
-------------------------------------------
```


Passing Arguments to Functions (2)

- Passing Variables
  - –Passing by Value
    - function creates copies of the arguments passed to it




```
// vararg.cpp
// demonstrates variable arguments
#include <iostream>
using namespace std;
void repchar(char, int);                    //function declaration

int main()
   {
   char chin;
   int nin;

   cout << "Enter a character: ";
   cin >> chin;
   cout << "Enter number of times to repeat it: ";
   cin >> nin;
   repchar(chin, nin);
   return 0;
   }
//--------------------------------------------------------------
// repchar()
// function definition
void repchar(char ch, int n)                //function declarator
   {
   for(int j=0; j<n; j++)                   //function body
      cout << ch;
   cout << endl;
   }
```

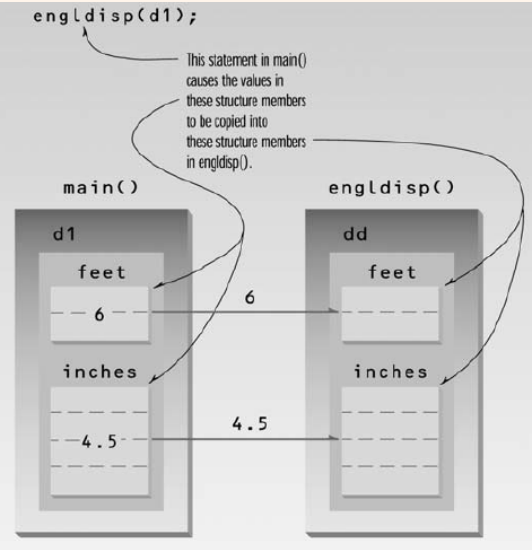
Output:

```
Enter a character: +
Enter number of times to repeat it: 20
++++++++++++++++++++
```

  - –Passing by Reference

Passing Arguments to Functions (3)

- Structures as Arguments

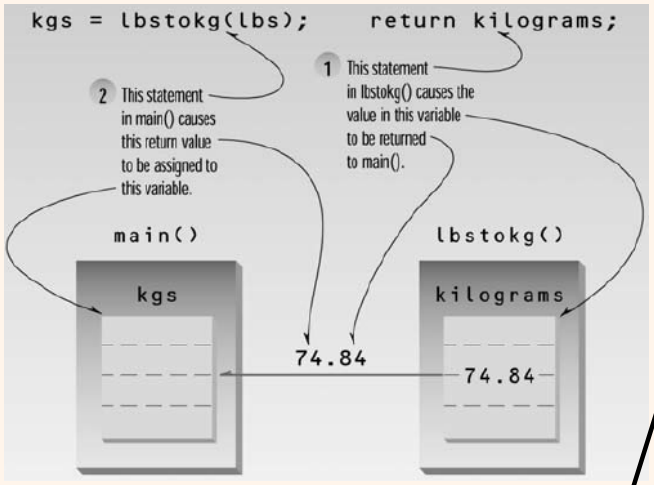



```
Output:
Enter feet: 6
Enter inches: 4

Enter feet: 5
Enter inches: 4.25

d1 = 6'-4"
d2 = 5'-4.25"
```

```
// engldisp.cpp
// demonstrates passing structure as argument
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct Distance               //English distance
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////////
void engldisp( Distance );    //declaration

int main()
   {
   Distance d1, d2;           //define two lengths

                              //get length d1 from user
   cout << "Enter feet: ";  cin >> d1.feet;
   cout << "Enter inches: ";  cin >> d1.inches;

                              //get length d2 from user
   cout << "\nEnter feet: ";  cin >> d2.feet;
   cout << "Enter inches: ";  cin >> d2.inches;

   cout << "\nd1 = ";
   engldisp(d1);              //display length 1
   cout << "\nd2 = ";
   engldisp(d2);              //display length 2
   cout << endl;
   return 0;
   }
//--------------------------------------------------------------
// engldisp()
// display structure of type Distance in feet and inches
void engldisp( Distance dd )   //parameter dd of type Distance
   {
   cout << dd.feet << "\'-" << dd.inches << "\"";
   }
```


Returning Values from Functions

- The return Statement




```
// convert.cpp
// demonstrates return values, converts pounds to kg
#include <iostream>
using namespace std;
float lbstokg(float);   //declaration

int main()
   {
   float lbs, kgs;

   cout << "\nEnter your weight in pounds: ";
   cin >> lbs;
   kgs = lbstokg(lbs);
   cout << "Your weight in kilograms is " << kgs << endl;
   return 0;
   }
//--------------------------------------------------------------
// lbstokg()
// converts pounds to kilograms
float lbstokg(float pounds)
   {
   float kilograms = 0.453592 * pounds;
   return kilograms;
   }
```

```
Output:
Enter your weight in pounds: 182
Your weight in kilograms is 82.553741
```

- Eliminating Unnecessary Variables

```
cout << "Your weight in kilograms is " << lbstokg(lbs)
     << endl;
```

```
return 0.453592 * pounds;
```

Returning Structure Variables

```
// retstrc.cpp
// demonstrates returning a structure
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
struct Distance                         //English distance
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////
Distance addengl(Distance, Distance);  //declarations
void engldisp(Distance);

int main()
   {
   Distance d1, d2, d3;                 //define three lengths
                                        //get length d1 from user
   cout << "\nEnter feet: ";  cin >> d1.feet;
   cout << "Enter inches: ";  cin >> d1.inches;
                                        //get length d2 from user
   cout << "\nEnter feet: ";  cin >> d2.feet;
   cout << "Enter inches: ";  cin >> d2.inches;

   d3 = addengl(d1, d2);                //d3 is sum of d1 and d2
   cout << endl;
   engldisp(d1); cout << " + ";         //display all lengths
   engldisp(d2); cout << " = ";
   engldisp(d3); cout << endl;
   return 0;
   }
```

```
//--------------------------------------------------------
// addengl()
// adds two structures of type Distance, returns sum
Distance addengl( Distance dd1, Distance dd2 )
   {
   Distance dd3;              //define a new structure for sum

   dd3.inches = dd1.inches + dd2.inches;  //add the inches
   dd3.feet = 0;                         //(for possible carry)
   if(dd3.inches >= 12.0)                  //if inches >= 12.0,
      {                                   //then decrease inches
      dd3.inches -= 12.0;                  //by 12.0 and
      dd3.feet++;                          //increase feet
      }                                    //by 1
   dd3.feet += dd1.feet + dd2.feet;       //add the feet
   return dd3;                            //return structure
   }
//--------------------------------------------------------
// engldisp()
// display structure of type Distance in feet and inches
void engldisp( Distance dd )
   {
   cout << dd.feet << "\'-" << dd.inches << "\"";
   }
```

Output:

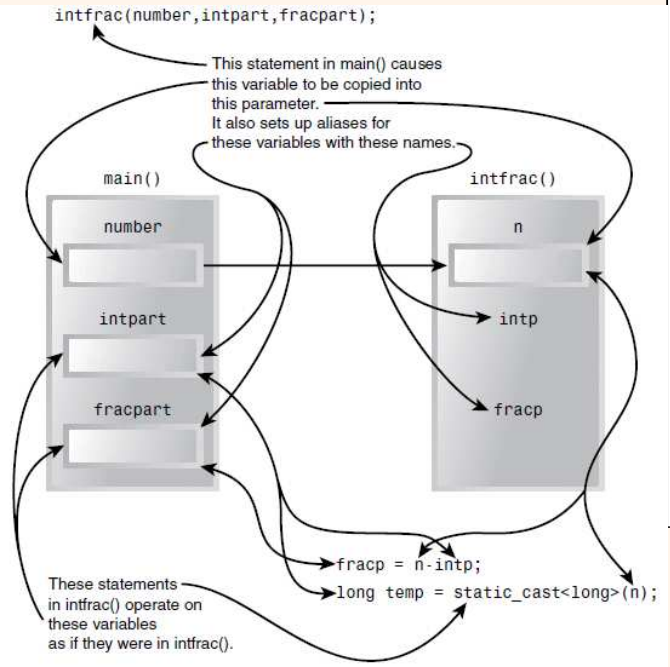
```
Enter feet: 4
Enter inches: 5.5
Enter feet: 5
Enter inches: 6.5
4'-5.5" + 5'-6.5" = 10'-0"
```


Reference Arguments

- A reference provides an alias—a different name—for a variable.
- Reference arguments were introduced into C++ to provide flexibility in a variety of situations involving objects as well as simple variables.
- When arguments passed by value, the called function creates a new variable of the same type as the argument and copies the argument's value into it.
- While arguments passed by reference, a reference to the original variable is passed.
  - It's actually the memory address of the variable.
- The third way to pass arguments to functions, besides by value and by reference, is to use pointers.

Reference Arguments (2)

- Passing Simple Data Types by Reference




```
// ref.cpp
// demonstrates passing by reference
#include <iostream>
using namespace std;

int main()
   {
   void intfrac(float, float&, float&);      //declaration
   float number, intpart, fracpart;          //float variables

   do {
      cout << "\nEnter a real number: ";     //number from user
      cin >> number;
      intfrac(number, intpart, fracpart);    //find int and frac
      cout << "Integer part is " << intpart  //print them
           << ", fraction part is " << fracpart << endl;
      } while( number != 0.0 );              //exit loop on 0.0
   return 0;
   }
//-------------------------------------------------------------
// intfrac()
// finds integer and fractional parts of real number
void intfrac(float n, float& intp, float& fracp)
   {
   long temp = static_cast<long>(n);  //convert to long int,
   intp = static_cast<float>(temp);   //back to float
   fracp = n - intp;                  //subtract integer part
   }
```

```
Output:
Enter a real number: 99.44
Integer part is 99, fractional part is 0.44
```


Reference Arguments (3)

- More complex example

```
// reforder.cpp
// orders two arguments passed by reference
#include <iostream>
using namespace std;

int main()
   {
   void order(int&, int&);          //prototype
   int n1=99, n2=11;          //this pair not ordered
   int n3=22, n4=88;             //this pair ordered

   order(n1, n2);       //order each pair of numbers
   order(n3, n4);

   cout << "n1=" << n1 << endl; //print out all
     numbers
   cout << "n2=" << n2 << endl;
   cout << "n3=" << n3 << endl;
   cout << "n4=" << n4 << endl;
   return 0;
   }
//-------------------------------------------------------------
void order(int& numb1, int& numb2)  //orders two
   numbers
   {
   if(numb1 > numb2)        //if 1st larger than 2nd,
      {
      int temp = numb1;             //swap them
      numb1 = numb2;
      numb2 = temp;
      }
   }
```

```
Output:
n1=11
n2=99
n3=22
n4=88
```

- Passing Structures by Reference

```
// demonstrates passing structure by reference
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
struct Distance                         //English distance
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////
void scale( Distance&, float );         //function
void engldisp( Distance );              //declarations

int main()
   {
   Distance d1 = { 12, 6.5 };           //initialize d1 and d2
   Distance d2 = { 10, 5.5 };
   cout << "d1 = "; engldisp(d1);       //display old d1 and d2
   cout << "\nd2 = "; engldisp(d2);
   scale(d1, 0.5);                      //scale d1 and d2
   scale(d2, 0.25);
   cout << "\nd1 = "; engldisp(d1);     //display new d1 and d2
   cout << "\nd2 = "; engldisp(d2);
   cout << endl;
   return 0;
   }
// scales value of type Distance by factor
void scale( Distance& dd, float factor)
   {
   float inches = (dd.feet*12 + dd.inches) * factor;
   dd.feet = static_cast<int>(inches / 12);
   dd.inches = inches - dd.feet * 12;
   }
// display structure of type Distance in feet and inches
void engldisp( Distance dd )    //parameter dd of type Distance
   {
   cout << dd.feet << "\'-" << dd.inches << "\"";
   }
```

```
Output:
d1 = 12'-6.5"
d2 = 10'-5.5"
d1 = 6'-3.25"
d2 = 2'-7.375"
```

Overloaded Functions

- Functions can share the same name as long as
  - they have a different number of parameters (Rule 1) or
  - their parameters are of different data types when the number of parameters is the same (Rule 2)

```
void myFunction(int x, int y) { ... }
void myFunction(int x) { ... }
```

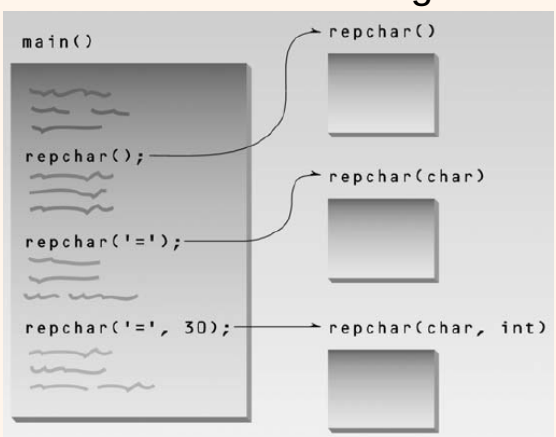
✓ Rule 1

```
void Function(double x) { ... }
void Function(int x) { ... }
```

✓ Rule 2

Overloaded Functions (2)

- Different Numbers of Arguments:
  - Prev example:
    - starline() , repchar() , and charline() —perform similar activities but have different names.
  - more convenient to use the same name for all three functions, even though they each have different arguments.




```
// demonstrates function overloading
#include <iostream>
using namespace std;
void repchar();              //declarations
void repchar(char);
void repchar(char, int);

int main()
   {
   repchar();
   repchar('=');
   repchar('+', 30);
   return 0;
   }
//-----------------------------------------------------------
// displays 45 asterisks
void repchar() // replaces starline()
   {
   for(int j=0; j<45; j++)   // always loops 45 times
      cout << '*';           // always prints asterisk
   cout << endl;
   }
//-----------------------------------------------------------
// displays 45 copies of specified character
void repchar(char ch) // replaces repchar()
   {
   for(int j=0; j<45; j++)   // always loops 45 times
      cout << ch;            // prints specified character
   cout << endl;
   }
//-----------------------------------------------------------
// displays specified number of specified character
void repchar(char ch, int n) // replaces charline()
   {
   for(int j=0; j<n; j++)    // loops n times
      cout << ch;            // prints specified character
   cout << endl;
   }
```

Output:

```
*********************************************
=============================================
++++++++++++++++++++++++++++++
```

Overloaded Functions (2)

- Different Kinds of Arguments:
  - Compiler can also distinguish between overloaded functions with the same number of arguments, provided their type is different.
- Overloaded functions can simplify the programmer's life by reducing the number of function names to be remembered.

```
// demonstrates overloaded functions
#include <iostream>
using namespace std;
struct Distance                 //English distance
   {   int feet;
       float inches;   };
////////////////////////////////////////////////////////////
void engldisp( Distance );      //declarations
void engldisp( float );

int main(){
   Distance d1;
   float d2;
                                //get length d1 from user
   cout << "\nEnter feet: ";  cin >> d1.feet;
   cout << "Enter inches: ";  cin >> d1.inches;
                                //get length d2 from user
   cout << "Enter entire distance in inches: "; cin >> d2;
   cout << "\nd1 = ";
   engldisp(d1);                //display length 1
   cout << "\nd2 = ";
   engldisp(d2);                //display length 2
   cout << endl;
   return 0;  }
//--------------------------------------------------------
// display structure of type Distance in feet and inches
void engldisp( Distance dd )    //parameter dd of type Distance
   {
   cout << dd.feet << "\'-" << dd.inches << "\"";
   }
//--------------------------------------------------------
// display variable of type float in feet and inches
void engldisp( float dd )       //parameter dd of type float
   {
   int feet = static_cast<int>(dd / 12);
   float inches = dd - feet*12;
   cout << feet << "\'-" << inches << "\"";
   }
```

Output:
```
Enter feet: 5
Enter inches: 10.5
Enter entire distance in inches: 76.5
d1 = 5'-10.5"
d2 = 6'-4.5"
```

Recursion

- Recursion involves a function calling itself.

Version	*Action*	*Argument or Return Value*
1	Call	5
2	Call	4
3	Call	3
4	Call	2
5	Call	1
5	Return	1
4	Return	2
3	Return	6
2	Return	24
1	Return	120

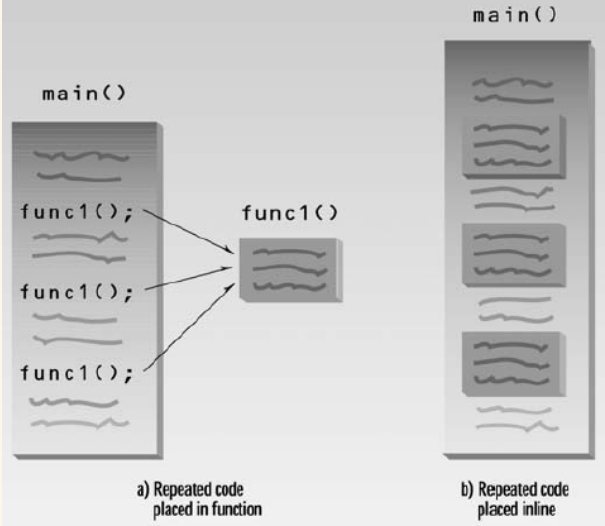
- In memory, each version's variables are stored, but there's only one copy of the function's code.
- Every recursive function must be provided with a way to end the recursion. Otherwise it will call itself forever and crash the program.

```
//factor2.cpp
//calculates factorials using recursion
#include <iostream>
using namespace std;

unsigned long factfunc(unsigned long);  //declaration

int main()
   {
   int n;                   //number entered by user
   unsigned long fact;      //factorial

   cout << "Enter an integer: ";
   cin >> n;
   fact = factfunc(n);
   cout << "Factorial of " << n << " is " << fact <<
    endl;
   return 0;
   }
//--------------------------------------------------------
// factfunc()
// calls itself to calculate factorials
unsigned long factfunc(unsigned long n)
   {
   if(n > 1)
      return n * factfunc(n-1);  //self call
   else
      return 1;
   }
```

Output:
```
Enter an integer: 5
Factorial of 5 is 120
```

Inline Functions

- During function call, compiler normally generates a jump to the function and jumps back after finish.
  - takes extra time; inefficient for short code.
- For short functions:
  - might be better to repeat the code rather jumping?
  - But code becomes longer and complex.



Inline Functions (2)

- Solution: Inline function.
  - function is written like a normal function in the source file
  - However, when the program is compiled, the function body is actually inserted into the program wherever a function call occurs.

```
// inliner.cpp
// demonstrates inline functions
#include <iostream>
using namespace std;

// lbstokg()
// converts pounds to kilograms
inline float lbstokg(float pounds)
   {
   return 0.453592 * pounds;
   }
//-----------------------------------------------------------
int main()
   {
   float lbs;

   cout << "\nEnter your weight in pounds: ";
   cin >> lbs;
   cout << "Your weight in kilograms is " << lbstokg(lbs)
        << endl;
   return 0;
   }
```

Default Arguments

- A function can be called without specifying all its arguments if function declaration provide default values for those arguments.
- Missing arguments must be the trailing arguments.

```
// missarg.cpp
// demonstrates missing and default arguments
#include <iostream>
using namespace std;

void repchar(char='*', int=45);    //declaration with
                                   //default arguments
int main()
   {
   repchar();                      //prints 45 asterisks
   repchar('=');                   //prints 45 equal signs
   repchar('+', 30);               //prints 30 plus signs
   return 0;
   }
//-------------------------------------------------------
// repchar()
// displays line of characters
void repchar(char ch, int n)       //defaults supplied
   {                               //   if necessary
   for(int j=0; j<n; j++)          //loops n times
      cout << ch;                  //prints ch
   cout << endl;
   }
```

Scope and Storage Class (for Local Variables)

- Scope:
  - A variable's scope, also called visibility, describes the locations within a program from which it can be accessed.
  - Two types:
    - Variable with *local* scope: visible only within a block.
      - Local variables: Variables defined within a function body.
    - Variable with *file* scope: visible throughout a file.
  - Block: basically the code between an opening brace and a closing brace, e.g. a function body.

```
void somefunc()
   {
   int somevar;     //local variables
   float othervar;

   somevar = 10;    //OK
   othervar = 11;   //OK
   nextvar = 12;    //illegal: not visible in somefunc()
   }

void otherfunc()
   {
   int nextvar;     //local variable

   somevar = 20;    //illegal: not visible in otherfunc()
   othervar = 21;   //illegal: not visible in otherfunc()
   nextvar = 22;    //OK
   }
```

Scope and Storage Class (2) (for Local Variables)

- Storage class:
  - of a variable determines how long it stays in existence.
  - Lifetime (duration) of a variable: The time period between the creation and destruction of a variable.
  - lifetime of a local variable coincides with the time when the function in which it is defined is executing.
  - Two types:
    - Variable with *automatic* storage class: exist during the lifetime of the function in which they're defined. Automatically created when a function is called and automatically destroyed when it returns.
    - Variable with *static* storage class: exist for the lifetime of the program.
- Initialization:
  - Local variable are usually not auto initialized
  - must initialize explicitly: `int n = 33;`

Scope and Storage Class (3) (for Global Variables)

- Global (external) variables are defined outside of any function and visible to all the functions in a file.
- Role:
  - used when it must be accessible to more than one function in a program.
  - global variables create organizational problems because they can be accessed by any function.
  - In an object-oriented program, there is much less necessity for global variables.

Scope and Storage Class (4) (for Global Variables)

- Initialization:
  - Global variable are usually auto initialized to 0;
- Lifetime and Visibility
  - has static storage class: they exist for the life of the program.
  - don't need to use the keyword static when declaring global variables
  - visible in the file in which they are defined, starting at the point where they are defined.
    - If ch were defined following main() but before getachar(), it would be visible in getachar() and putachar(), but not in main() .

```
// extern.cpp
// demonstrates global variables
#include <iostream>
using namespace std;
#include <conio.h>          //for getch()

char ch = 'a';              //global variable ch

void getachar();            //function declarations
void putachar();

int main()
   {
   while( ch != '\r' )      //main() accesses ch
      {
      getachar();
      putachar();
      }
   cout << endl;
   return 0;
   }
//--------------------------------------------------------
void getachar()             //getachar() accesses ch
   {
   ch = getch();
   }
//--------------------------------------------------------
void putachar()             //putachar() accesses ch
   {
   cout << ch;
   }
```

Scope and Storage Class (5) (Static Local Variables)

- static global variables are meaningful only in multifile programs
- Static local variable:
  - Visibility: automatic local.
  - Lifetime: same as global except that it doesn't come into existence until the first call to the function containing it.
  - used when it's necessary for a function to remember a value when it is not being executed.
- Initialization:
  - only once—the first time their function is called.

```
// static.cpp
// demonstrates static variables
#include <iostream>
using namespace std;
float getavg(float);        //declaration

int main()
   {
   float data=1, avg;

   while( data != 0 )
      {
      cout << "Enter a number: ";
      cin >> data;
      avg = getavg(data);
      cout << "New average is " << avg << endl;
      }
   return 0;
   }
//--------------------------------------------------------
// getavg()
// finds average of old plus new data
float getavg(float newdata)
   {
   static float total = 0;  //static variables are
    initialized
   static int count = 0;    //   only once per program

   count++;                 //increment count
   total += newdata;        //add new data to total
   return total / count;    //return the new average
   }
```

Output:

```
Enter a number: 10
New average is 10
Enter a number: 20
New average is 15
Enter a number: 30
New average is 20
```

Scope and Storage Class (6) (Summary)

- Storage:
  - local variables and function arguments are stored on the stack
  - global and static variables are stored on the heap.

TABLE 5.2 Storage Types

	Local	*Static Local*	*Global*
Visibility	function	function	file
Lifetime	function	program	program
Initialized value	not initialized	0	0
Storage	stack	heap	heap
Purpose	Variables used by a single function	Same as local, but retains value when function terminates	Variables used by several functions

Returning by Reference

```
// retref.cpp
// returning reference values
#include <iostream>
using namespace std;
int x;                      // global variable
int& setx();                // function declaration

int main()
   {
   setx() = 92;             // set x to a value, using
                            // function call on left side
   cout << "x=" << x << endl;// display new value in x
   return 0;
   }
//-----------------------------------------------------
int& setx()
   {
   return x;       / returns the value to be modified
   }
```

const Function Arguments

- If an argument is large, passing by reference is more efficient because, behind the scenes, only an address is really passed, not the entire variable.
- Additionally, if you want a guarantee that the function cannot modify it.
  - Apply const modifier to the variable in the function declaration.

```
//constarg.cpp
//demonstrates constant function arguments

void aFunc(int& a, const int& b);  //declaration

int main()
   {
   int alpha = 7;
   int beta = 11;
   aFunc(alpha, beta);
   return 0;
   }
//--------------------------------------------------
void aFunc(int& a, const int& b)   //definition
   {
   a = 107;    //OK
   b = 111;    //error: can't modify constant
    argument
   }
```

- To pass a const variable to a function as a reference argument, it must be declared const in the function declaration.

Summary (1)

- Functions provide a way to help organize programs, and to reduce program size, by giving a block of code a name and allowing it to be executed from other parts of the program.
  - Function *declarations* (prototypes) specify what the function looks like, function *calls* transfer control to the function, and function *definitions* contain the statements that make up the function.
  - The function declarator is the first line of the definition.
- Arguments can be sent to functions either *by value*, where the function works with a copy of the argument, or *by reference*, where the function works with the original argument in the calling program.
- Functions can return only one value. Functions ordinarily return by value, but they can also return by reference, which allows the function call to be used on the left side of an assignment statement. Arguments and return values can be either simple data types or structures.
- An *overloaded function* is actually a group of functions with the same name. Which of them is executed when the function is called depends on the type and number of arguments supplied in the call.

Summary (2)

- An *inline function* looks like a normal function in the source file but inserts the function's code directly into the calling program.
  - Inline functions execute faster but may require more memory than normal functions unless they are very small.
- If a function uses default arguments, calls to it need not include all the arguments shown in the declaration.
- Variables possess a characteristic called the *storage class*.
  - The most common storage class is *automatic*.
  - *Local* variables have the automatic storage class: they exist only while the function in which they are defined is executing. They are also visible only within that function.
  - *Global* variables have static storage class: they exist for the life of a program. They are also visible throughout an entire file.
  - *Static* local variables exist for the life of a program but are visible only in their own function.
- A function cannot modify any of its arguments that are given the `const` modifier. A variable already defined as `const` in the calling program must be passed as a `const` argument.