Functions

- Method of encapsulating a section of code and referencing it by an identifier.
- Functions provide mechanism for code reuse, preventing code duplication.
- A function should perform one logical operation.
- Functions must be prototyped preceding their usage. This is accomplished for standard library functions by including the appropriate header files.

```
#include <stdlib.h>
void Acknowledge();  // function prototype (or declaration)
int main()
{
    • • •
    Acknowledge ();  // function call (or invocation)
    • • •
    return EXIT_SUCCESS;
}
void Acknowledge()   // function definition
{
    cout << "Thank you." << endl;
    cout << "Please come again.";
    return;
}
```

Function Structure

A function definition is almost identical to the structure of main( ).

Header blocks of documentation should be used for each function explaining purpose, processing, etc.

Constants and variables can be declared within a function.

Function definitions cannot be nested inside other function definitions.

Scoping rules determine what entities can be referenced at what levels.

```
int skipBlanks()
{
    const  char BLANK  =  ' ';
    char   letter = BLANK ;
    int  skips = 0 ;

    while (letter == BLANK ) {
        cin.get( letter );
        skips++;
    } // end while

    return(skips);
}
```

Identifier Scope

Local scope	- scope of an identifier declared inside a compound, (block), statement extends from the point of declaration to the end of that compound statement
Global (or file) scope	- scope of identifier declared outside all functions and classes extends from the point of declaration to the end of the entire file containing the program code
Name Precedence	- when function declares identifier with same name as global identifier, local identifier takes precedence within function

Scope Example


```
#include <iostream.h>
void someFunc( float );            // _______________
const int a = 17;                  // _______________
int b;                             // _______________
int c;                             // _______________

int main( ) {
    b=4;

    c=6;

    someFunc(42.8);

    return 0;

}
void someFunc( float c) {          // _______________

    float b;                       // _______________
    b = 3.2;                       // _______________
    cout << "a = " << a;           // _______________
    cout << "b = " << b;           // _______________
    cout << "c = " << c;           // _______________
}
```

Scope Example

```
#include <iostream.h>
void Block1(int a1, char b2);
void Block2();
int  a1;          // _________________
char a2;          // _________________
main  ( )
{
     ...
}
void Block1 (int a1, char b2)
{
int c1;           // _________________
int d2;           // _________________
  ...
}
void Block2()
{
int a1;           // _________________
int b2;           // _________________
while (...)
  {               // Block3
       int   c1; // _________________
       int   b2; // _________________
       ...
  }
}
```

What is the scope of all the identifiers?

IDENTIFIER	SCOPE
a1	
main	____
Block1	____
Block2	____
Block3	____
a2	
main	____
Block1	____
Block2	____
Block3	____
b2	
main	____
Block1	____
Block2	____
Block3	____
c1	
main	____
Block1	____
Block2	____
Block3	____
d2	
main	____
Block1	____
Block2	____
Block3	____

Function Prototypes

The C++ compiler has a very difficult job to do when translating our programs. Anything we can do to make this job easier and to ensure that the compiler does its job correctly is to our benefit. Function prototypes serve this purpose.

In C++, every function must be prototyped (declared) prior to any calls to that function. A C++ function prototype has the form:

type name(parameter-declaration-list); // parameter types specified

Note that the parameter list is specified in the prototype declaration. This allows the compiler to do type checking and type conversion as needed.

The variables/values passed to a function are called parameters or arguments to the function. We will use the term parameter.

Function Prototypes (continued)

Example prototypes:

```
float  sqr(double);
void   pr_int(char, int);
void   RingBell( );                  // function with no parameters
char   RepFloat(float x, int y);
```

A function prototype may specify the names of the formal parameters, or not. In any case, the compiler ignores names of formal parameters in the prototype; but they can serve as valuable documentation.

With the above sqr( ) prototype, the call to sqrt(4) causes a conversion from int value 4 to double value 4.0 to occur before the function call is executed.

Parameter Communication

- Formal Parameter: the accepting parameter in the function definition.
- Actual Parameter: the variable passed into the function at an invocation (call).
- Type and order of parameters in the call and the definition must be the same.
- Names of the parameters in the call and the definition may be different.

Formal parameters are dummy placeholder variables.

```
float CalcNum(int P1, float P2) {
    // function definition

}
```

Formal parameters are matched to actual parameters on the basis of order.

```
y = calcNum(Size, Distance);
    // function invocation
```

C++ Parameter Passing Mechanisms

Pass-by-Value

- default passing mechanism except for one special case discussed in Chapter 8
- allocate a temporary memory location for each formal parameter (when function is called)
- copy the value of the corresponding actual parameter into that location
- called function has no access to the actual parameter, just to a copy of its value

```
.  .  .
int First  = 15,
    Second = 42;
int Least = FindMinimum(First, Second);
.  .  .
```

Variable	Value
First	
Second	
Least	

```
int FindMinimum(int A, int B) {
   if (A <= B)
      return A;
   else
      return B;
}
```

Variable	Value
A	
B	

Created when call occurs and destroyed on return.

C++ Parameter Passing Mechanisms

Pass-by-Reference

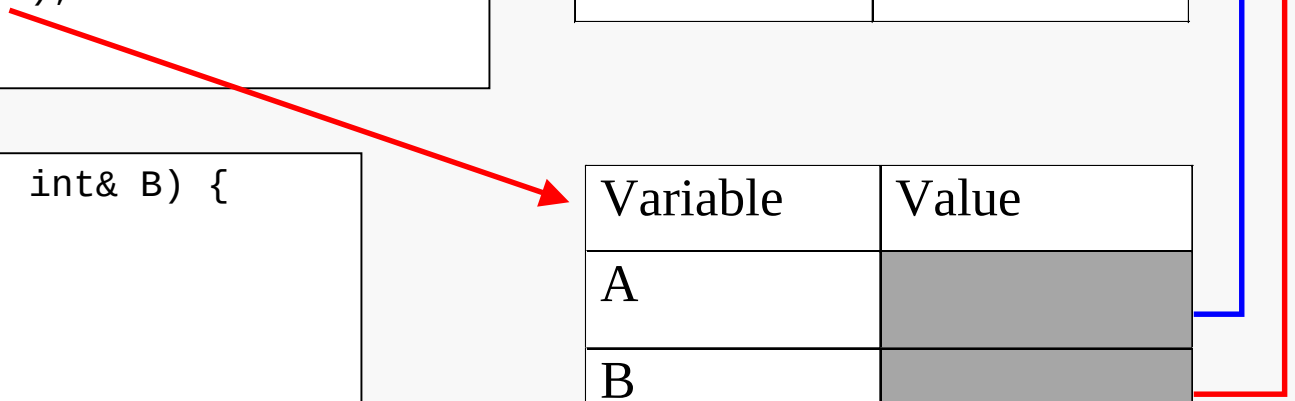
- put ampersand (&) after formal parameter type in prototype and definition
- forces the corresponding actual and formal parameters to refer to the same primary memory location; that is, the formal parameter is then a synonym or alias for the actual parameter
- called function may modify the value of the actual parameter

```
.   .   .
int First  = 15,
    Second = 42;
SwapEm(First, Second);
.   .   .
```

```
void SwapEm(int& A, int& B) {
   int TempInt;
   TempInt = A;
   A       = B;
   B       = TempInt;
}
```

Variable	Value
First	
Second	

Variable	Value
A	
B	
TempInt	



C++ Parameter Passing Mechanisms

Pass-by-Constant-Reference

- precede formal parameter type with keyword `const`, no ampersand (&)
- forces the corresponding actual and formal parameters to refer to the same primary memory location; just as in pass-by-reference
- **but**, the called function is not allowed to modify the value of the parameter; compiler flags such a statement as an error

```
.  .  .
int First  = 15,
    Second = 42,
    Third;
Third = AddEm(First, Second);
.  .  .
```

```
int AddEm(const int A, const int B) {
   int Sum;
   Sum = A + B;
   return Sum;
}
```

Variable	Value
First	
Second	
Third	

Variable	Value
A	
B	
Sum	

Choosing a Parameter Passing Mechanism

Pass-by-Reference

- use **only if** the design of the called function requires that it be able to modify the value of the parameter

Pass-by-Constant-Reference

- use if the called function has no need to modify the value of the parameter, but the parameter is very large (e.g., an array as discussed in Chapter 8)
- use as a **safety net** to guarantee that the called function cannot be written in a way that would modify the value passed in†

Pass-by-Value

- use in all cases where none of the reasons given above apply
- pass-by-value is safer than pass-by-reference

† Note that if a parameter is passed by value, the called function may make changes to that value as the formal parameter is used within the function body. Passing by constant reference guarantees that even that sort of internal modification cannot occur.

Parameter Lists

With pass by value, the actual parameter can be an expression:

F = CalcForce(mass * g, h);

```
double CalcForce(int Weight, int Height){
                 // value    // value

              . . .

}
```

With pass by reference and pass by constant reference, the actual parameter must be a variable.

```
double CalcForce(int weight, int& height){
                  // value    // reference

              . . .

}
```

The function prototype and function definition heading must agree in both order and types of the parameters and return value.

Parameter Example 1

```
#include <iostream.h>
void main( ) {
const int W = 100;
int  X, Y, Z;
void Mix(int P, int& Z);

   X = 10;  Y = 20;  Z  =  30;
   Mix ( X, Y );
   cout <<  W << X << Y<< Z << endl ;
   Mix ( Z,  X );
   cout << W << X << Y << Z << endl ;
}

void Mix (int  P,   int& Z ) {
   int  Y, W ;

   Y = P;   W  =  Z;   Z  =  Z + 10;
   cout << P << W << Y << Z << endl;
}
```

Memory space for main():

W	
X	
Y	
Z	

Memory space for Mix():

P	
Z	
Y	
W	

Output

Parameter Example 2

```
#include <iostream.h>;
void main( ){
   void GetRealValue(float& , float  );

   float TestScore, LowerLimit ;
   LowerLimit = 30.0;
   GetRealValue(TestScore, LowerLimit );
   cout << "Test score entered: "  << TestScore ;
}

void GetRealValue(float& RealValue , float Limit) {

   cout << "Please enter a real value greater than " << Limit;
   cin >> RealValue ;
   while (RealValue < Limit) {
      cout << "ERROR: Value is not large enough" << endl;
      cout << "Please re-enter the real value:";
      cin  >> RealValue ;
   }
}
```

What are the

actual parameters?
formal parameters?

Which are

value parameters ?
reference parameters?

Function Names

All functions should have a prototype so that the compiler can do appropriate error checking.

The function name is not a variable.

The function name should not be used on the right-hand side of an assignment statement inside that function, [except when making a recursive call (see Chapter 14 of notes)].

Every function must have a type (or be void).

```
bool isLeapYear ( const int year )
{
   return ( ( year % 400 == 0 ) ||
          ( ( year %   4 == 0 ) &&
            ( year % 100 != 0 ) ) );
} // end isLeapYear
```

The return Statement

Execution of a return statement accomplishes the following things:

- immediately terminates execution of the function within which the return occurs
- a function definition may contain more than one return statement; however, only one of those will be executed on any given call to that function
- replaces the function invocation with the value of the variable or expression specified in the return statement (if the called function is not void)
- resumes execution of the calling function

```
bool isThisLeapYear = isLeapYear(1999);
```

void functions do not require a return statement, however it is acceptable and useful if multiple exit points are needed. If a void function does not contain a return statement at the end an implied return is executed by the system when a function terminates.

void versus typed functions

If a function needs to communicate just one value to the caller you may accomplish that in either of two ways:

- use a void function with a reference parameter for communication
- use a typed function with an appropriate return statement

The latter approach is generally preferred because it makes the effect of the function call clearer.

Remember that in C++ there is no indication in the function call of which actual parameters are passed by reference (and hence at risk of being modified) and which are passed by value or constant reference (and hence guaranteed not to be modified by the function call).

Use of void

Using a void function with a reference parameter for communication :

```
void SquareIt(float value, float& square) {
   square = value * value;
}
```

. . . could be invoked in the following manner:

```
cin >> xval;
SquareIt(xval, xsquared);     // call is a separate statement
cout << setw(10) << setprecision(4) << xsquared;
```

Use of return

Using a typed function with an appropriate return statement :

```
float SquareIt(float value) {
   float square = value * value;
   return square;
}
```

. . . could be invoked in the following manner:

```
cin >> xval;
xsquared = SquareIt(xval);    // called as part of an
                              //        expression
cout << setw(10) << setprecision(4) << xsquared;
```

Multiple Reference Communication

If a function needs to communicate more than one value to the caller you must use reference parameters to accomplish the communication (pending the introduction of structs).

```
void SwapEm(int& first, int& second) {
   int temp = first;
   first  = second;
   second = temp;
}
```

As a void function, `SwapEm( )` would be invoked as a separate statement:

```
   int x = 17, y = 24;
   SwapEm(x, y);
   cout << x << '\t' << y << endl;
```

Function Prototype Scope

The placement of the prototype of a function determines the scope of the function name, just as with other identifiers.

Placing the prototype outside all function and class bodies gives the function file scope. This allows any function defined in the same file to invoke that function.

Placing the prototype inside the body of another function gives the function name scope local to the compound statement in which the prototype is placed. This can be used to restrict which functions are allowed to invoke that function.

Prototype Scope Example

Consider the simple calculator program on the following slides. The program reads simple integer expressions from an input file and evaluates them:

Input:

```
17 + 43
67 - 14
89 * 12
43 /  7
```

Output:

```
   17 +    43 =    60
   67 -    14 =    53
   89 *    12 =  1068
   43 /     7 =     6
No more expressions.
```

This program illustrates the use placement of function prototypes to restrict access so that only functions that need to make calls can do so.

Prototype Scope Example


```
#include <fstream.h>   // for stream I/O
#include <iomanip.h>   // for setw( )
#include <limits.h>    // for INT_MIN

void main( ) {
   int  NextExpression(ifstream& In, int& Op1, char& Op, int& Op2);
   void Print(int Op1, char Op, int Op2, int Val);
   int  Operand1, Operand2, Value = 0;
   char Operator;
   ifstream iFile;

   iFile.open("calc.dat");
   while (Value != INT_MIN) {
      Value = NextExpression(iFile, Operand1, Operator, Operand2);
      if (Value == INT_MIN) {
         cout << "No more expressions." << endl;
         iFile.close();
         return;
      }
      Print(Operand1, Operator, Operand2, Value);
   }
   iFile.close();  //logically never executed, included for safety
}
```

Prototype Scope Example


```
int  NextExpression(ifstream& In, int& Op1, char& Op, int& Op2) {
   int  DoCalc(int  Op1, char  Op, int  Op2);
   void GetOps(ifstream& In, int& Op1, char& Op, int& Op2);

   GetOps(In, Op1, Op, Op2);                 // get expression

   if (In)
      return DoCalc(Op1, Op, Op2);           // evaluate if OK
   else
      return INT_MIN;                        // return flag if not
}



void GetOps(ifstream& In, int& Op1, char& Op, int& Op2) {

   In >> Op1 >> Op >> Op2;          // read expression
   In.ignore(80, '\n');             // skip to beginning of next line
   return;
}
```

Prototype Scope Example

```
void Print(int Op1, char Op, int Op2, int Val) {
   cout << setw(5) << Op1 << ' ' << Op << ' '
        << setw(5) << Op2 << " = "
        << setw(5) << Val << endl;
}

int DoCalc(int Op1, char Op, int Op2) {
   int Result;

   switch(Op) {                              // consider operation
   case '+':  Result = Op1 + Op2;
              break;
   case '-':  Result = Op1 - Op2;
              break;
   case '*':  Result = Op1 * Op2;
              break;
   case '/':  Result = Op1 / Op2;            // assert( Op2 <> 0 );
              break;
   default:   Result = INT_MIN;              // return flag if operation
   }                                         // is invalid
   return Result;
}
```

File Scope Prototypes

The prototypes could have all been given file scope:

```
#include <fstream.h>   // for stream I/O
#include <iomanip.h>
#include <limits.h>    // for INT_MIN

int  NextExpression(ifstream& In, int& Op1, char& Op, int& Op2);
int  DoCalc(int  Op1, char  Op, int  Op2);
void GetOps(ifstream& In, int& Op1, char& Op, int& Op2);
void Print(int Op1, char Op, int Op2, int Val);

void main( ) {
int  Operand1, Operand2, Value = 0;
   char Operator;
   . . .
   iFile.close();
}
```

… but that would allow unnecessary access to each function.

Prototype Scope Problems

Placing all the prototypes within `main( )` would not work, however:

```
#include <fstream.h>   // for stream I/O
#include <iomanip.h>
#include <limits.h>    // for INT_MIN

void main( ) {
   int  NextExpression(ifstream& In, int& Op1, char& Op, int& Op2);
   int  DoCalc(int  Op1, char  Op, int  Op2);
   void GetOps(ifstream& In, int& Op1, char& Op, int& Op2);
   void Print(int Op1, char Op, int Op2, int Val);

   int  Operand1, Operand2, Value = 0;
   char Operator;
   . . .
   iFile.close();
}
```

… since the references to `GetOps( )` and `DoCalc( )` inside the function `NextExpression( )` would now be to undeclared identifiers.