scope	(of an identifier) the range of program statements within which the identifier is recognized as a valid name

C++ Scope Rules

1. Every identifier must be declared and given a type before it is referenced (used).

2. The scope of an identifier begins at its declaration.

3. If the declaration is within a compound statement, the scope of the identifier ends at the end of that compound statement. We say the identifier is local to that compound statement (or block).

4. If the declaration is not within a compound statement, the scope of the identifier ends at the end of the file. We say the identifier has global scope.

Intro Programming in C++

> **binding** determining which declaration of an identifier corresponds to a particular use (reference) of that identifier

C++ Binding

Binding identifier references to declarations is the responsibility of the compiler. When the compiler finds a reference it searches for a matching declaration of the name. This search is conducted according to the following rules.

1. A declaration of a constant or variable must match the identifier name exactly.

2. A declaration of a function must also match the number and parameter types.

3. The search proceeds upward (backward) from the location of the reference.

4. If there is no matching declaration in the block containing the reference, and that block is contained within an "enclosing" block, then the search continues into that "enclosing" block.

5. The search will not enter a block enclosed within the block currently being searched. That is, a block is "private" when viewed from outside it.

6. If necessary the compiler will continue this process until global scope is reached and searched.

7. If no matching declaration is found, the reference to the identifier is invalid and an "undeclared identifier" message is issued.

```cpp
#include <iostream>
#include <string>
#include <climits>
using namespace std;

// Constants with global scope:
const int QUITVALUE      = 0;
const string UserPrompt = "Please enter . . . (zero to quit): ";
const string NewMaxMsg  = "That beats the old maximum by: ";

int main() {

    int Max = INT_MIN;    // local to main()
    int UserEntry;

    cout << UserPrompt;   // declarations of cout and cin are . . .?
    cin  >> UserEntry;
. . .
```

```
. . .
    while (UserEntry != QUITVALUE) {

        if (UserEntry > Max) {
            int Increase;              // local to the if body
            Increase = UserEntry - Max;
            cout << NewMaxMsg << Increase << endl;
            Max = UserEntry;
        }
                          // couldn't refer to Increase here
        cout << UserPrompt;
        cin  >> UserEntry;
    }
    // couldn't refer to Max here
    return 0;
}
```

One of the most contentious issues facing novice programmers is the use of identifiers that have global scope.

Since the publication of Edsger Dijkstra's "Use of Globals Considered Harmful" in 1966, there has been a growing consensus among software engineers that identifiers should be declared at the global level only reluctantly.

As a general policy in CS 1044, global scope is to be used <u>only</u> for:
- constants
- function declarations
- type definitions

The use of globally-scoped <u>variables</u> is expressly forbidden. We will discuss why this prohibition is enforced as we consider various examples. Regardless of your prior programming experience and attitudes, <u>do not ignore this rule</u>.

Functions

A function is a mechanism for encapsulating a section of code and referencing it by use of a descriptive identifier.

Functions provide support for code reuse, and help prevent duplicating blocks of code that are used repeatedly within a program.

Every C++ program must include a function named `main`.

Aside from `main`, functions are <u>called</u>, or <u>invoked</u>, by other functions within a program.

The calling function and the called function may communicate, or exchange information, in a number of ways.

The use of functions produces programs that correspond more closely to the procedural decomposition derived during a typical design exercise.

The use of functions also makes implementation more modular since individual programs may be developed separately and then combined to produce a finished program consisting of a group of cooperating functions.

The use of functions also makes testing easier since each function may be tested separately from the rest of the program.

Intro Programming in C++

A function is an agent that is invoked to carry out a particular task.

Every function must have an <u>implementation</u> or <u>definition</u>, which contains the statements that will be executed when the function is invoked.

The function definition is a body that may contain any valid C++ statements.

In CS 1044, the function definition will always be in the same file as `main()`. Technically, the function definitions may be in any order, but we will always place the definition of `main()` at the beginning.

Function definitions may not be nested.

Header blocks of documentation should be written for each function explaining its purpose, setup needs, unusual processing, etc.

```cpp
int getValue() {

    const string UserPrompt = "Please enter . . . ";
    int valueEntered;

    cout << UserPrompt;
    cin >> valueEntered;

    return valueEntered;
}
```

Intro Programming in C++

Execution of a return statement accomplishes the following things:

- immediately terminates execution of the function within which the `return` occurs
- 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

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

```
// in the calling function

UserEntry = getValue();
```

`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.**

The `return` statement provides a way for the called function to send a single value back to the calling function. In many cases it is necessary to also pass information from the calling function to the called function.

Consider:

```cpp
double circleArea() {

    const double PI = 3.141596224;

    return (PI * Radius * Radius);
}
```

Question: where should `Radius` be declared and set?

- local to `circleArea()`?

    The function would only compute the area of one particular circle.

- global scope and set by the calling function?

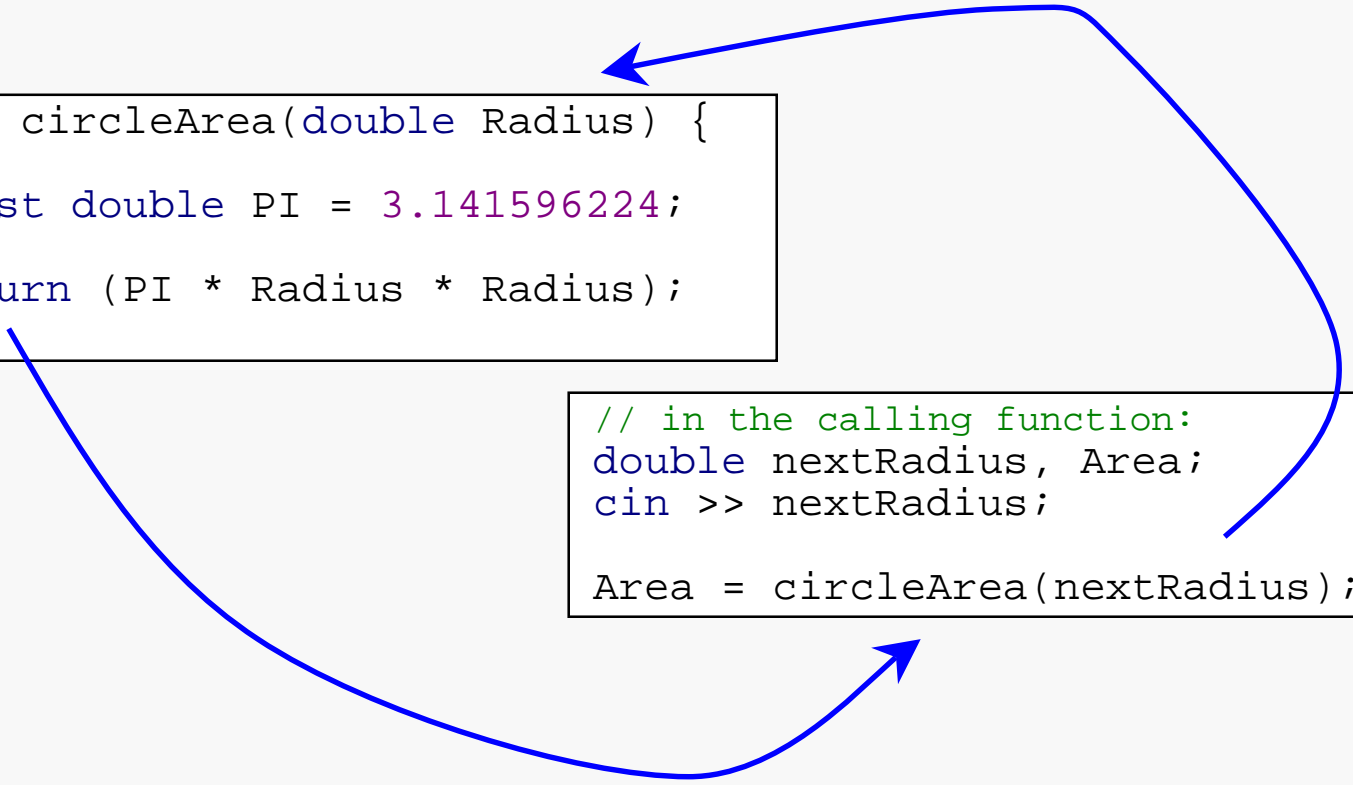   Would work, but global variables should be avoided.

- local to the calling function or some other scope?

    But then the scope of `Radius` would be that other scope, and `Radius` would be inaccessible in `circleArea()`.

The calling function may pass information to the called function by making use of parameters. This requires preparation in the function definition and in the caller:

```cpp
double circleArea(double Radius) {

    const double PI = 3.141596224;

    return (PI * Radius * Radius);
}
```

```cpp
// in the calling function:
double nextRadius, Area;
cin >> nextRadius;

Area = circleArea(nextRadius);
```

As used here, the value of the variable `nextRadius` (in the calling function) is copied into the variable `Radius` (in the called function), where `circleArea()` may make use of that value to perform its calculations.

The function definition must provide a list of declarations of variables that may be used for communication. These variables are declared within the parentheses following the function name, and are called <u>formal parameters</u>.

```cpp
double circleArea(double Radius) {

    const double PI = 3.141596224;

    return (PI * Radius * Radius);
}
```

A formal parameter is essentially just a placeholder into which the calling function will place a value.

The scope of a formal parameter is the body of the function definition.

Formal parameters may be of any valid C++ type.

Formal parameter declarations are comma-separated.

A function may have no formal parameters, or as many as are needed.

The formal parameter list and return type are often called the <u>interface</u> of a function since they make up the view of the function from the perspective of the caller.

The calling function must invoke the called function with a list of <u>actual parameters</u>. The number of actual parameters must match the number of formal parameters in the called function, and these actual parameters must match the formal parameters in type (subject to default conversions).

```cpp
// in the calling function:
double nextRadius, Area;
cin >> nextRadius;

Area = circleArea(nextRadius);
```

Actual parameters and formal parameters are matched by order, not by name or by type.

If the number or types of the actual and formal parameters do not match, the compiler (or possibly the linker) will generate an error message.

Together the actual and formal parameter lists and the return type define the communication possibilities that are open to a function.

The parameter lists are the most basic issues in function design.

A function name is an identifier, so it must be formally declared before it is used.

Unlike a simple variable or constant, a function has a return type and (possibly) a formal parameter list. The function declaration must also specify those.

The function declaration is essentially just a copy of the function header from the function definition.

```
double circleArea(double Radius);
```

A function declaration must specify the types of the formal parameters, but not the formal parameter names.

```
double circleArea(double Radius) {

    const double PI = 3.141596224;

    return (PI * Radius * Radius);
}
```

However, it is good practice to include the formal parameter names in the function declaration.

> **Many authors refer to a function declaration as a <u>prototype</u>.**

Function declarations are typically declared in global scope, although they may be placed anywhere. The placement determines the scope of the function name, and hence where it may be called.

```cpp
#include <iostream>
#include <string>
#include <climits>
using namespace std;
int getValue();                    // function prototypes
void Notify(int Difference);
const int QUITVALUE    = 0;

int main() {
    int Max = INT_MIN;
    int UserEntry;

    UserEntry = getValue();        // function call

    while (UserEntry != QUITVALUE) {
        if (UserEntry > Max) {
            int Increase;
            Increase = UserEntry - Max;
            Notify(Increase);          // function call
            Max = UserEntry;
        }
        . . .
```

Intro Programming in C++

```cpp
         . . .
      UserEntry = getValue();      // function call
   }
   return 0;
}


// function definitions
int getValue() {

   const string UserPrompt = "Please enter a . . . (zero to quit): ";
   int valueEntered;
   cout << UserPrompt;
   cin >> valueEntered;
   return valueEntered;
}


void Notify(int Difference) {

   const string NewMaxMsg  = "That beats the old maximum by: ";
   cout << NewMaxMsg << Difference << endl;
   return;
}
```

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

void F(double c);          // Line  1
const int a = 17;          //        2
int b;                     //        3
int c;                     //        4

int main( ) {
   int  b;                 // Line  5
   char c;                 // Line  6
   b = 4;                  // _____
   c = 'x';               // _____
   F(42.8);               // _____

   return 0;
}
void F(double c) {         // Line  7
   double b;               //        8
   b = 3.2;                // _____
   cout << "a = " << a;   // _____
   cout << "b = " << b;   // _____
   cout << "c = " << c;   // _____
   int a;                  // Line  9
   a = 42;                 // Line 10
   cout << "a = " << a;   // _____
}
```

Examine the sample program and consider:

- what is the scope of each declared identifier?

- what declaration is each identifier reference bound to?

- where are global identifiers referenced?

- where is the global identifier b referenced?

Intro Programming in C++

Pass-by-Value

- default passing mechanism except for one special case discussed later
- 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.
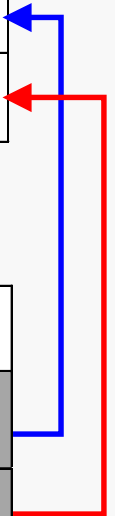
Pass-by-Reference
- put ampersand (&) after formal parameter type in prototype and definition
- forces the corresponding actual and formal parameters to refer to the <u>same</u> memory location; that is, the formal parameter is then a synonym or alias for the actual parameter
- called function may modify the value of the actual parameter

```
.   .   .

int First  = 15,

    Second = 42;

SwapEm(First, Second);

.   .   .
```

Variable	Value
First	
Second	

```
void SwapEm(int& A, int& B) {

    int TempInt;

    TempInt = A;

    A       = B;

    B       = TempInt;

}
```

| Variable	Value
A	
B	
TempInt	

Pass-by-Constant-Reference

- precede formal parameter type with keyword `const`, follow it with an 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; the compiler flags such a statement as an error

```
.   .   .

int First  = 15,

    Second = 42,

    Third;

Third = AddEm(First, Second);

.   .   .
```

Variable	Value
First	
Second	
Third	

```
int AddEm(const int& A, const int& B) {

    int Sum;

    Sum = A + B;

    return Sum;

}
```

Variable	Value
A	
B	
Sum	

Intro Programming in C++
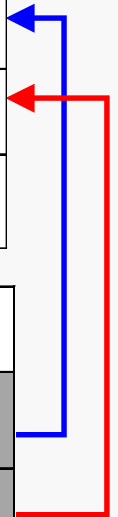
Pass-by-Reference

- 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., a string or a structure or an array, as discussed later)
- 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.

With pass by value, the actual parameter can be an expression (or a variable or a constant):

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

   . . .

}
```

```
F = CalcForce(mass * g, h);
```

With pass by reference and pass by constant reference, the actual parameter must be an *l-value*; that is, something to which a value can be assigned.

```
void getRGB(int& Red, int& Green,
                              int& Blue){

   . . .

}
```

That rules out expressions and constants.

```
. . .
int main( ) {                                    //   1
const int W = 100;                               //   2
int  X = 10, Y = 20, Z = 30;                     //   3
void Mix(int P, int& Z);                         //   4

   Mix ( X, Y );                                 //   5
   cout << W << X << Y << Z << endl;             //   6
   Mix ( Z,  X );                                //   7
   cout << W << X << Y << Z << endl;             //   8
   return 0;                                     //   9
}


void Mix (int  P,    int& Z ) {                  //  10
   int  Y = 0, W = 0;                            //  11

   Y = P;                                        //  12
   W = Z;                                        //  13
   Z = Z + 10;                                   //  14
   cout << P << W << Y << Z << endl;             //  15
  }
```

Memory space for `main()`:

W	
X	
Y	
Z	

Memory space for `Mix()`:

P	
Z	
Y	
W	

Output


```
////////////////////////////////////////////////////// isLeapYear()
//  Determines whether a given year is a leap year.
//
//  Parameters:
//     Year    year to be tested
//
//  Returns:   true if Year is a leap year, false otherwise
//
//  Pre:       Year has been initialized
//  Post:      specified value has been returned
//
//  Called by: buildCalendar()
//  Calls:     none
//
bool isLeapYear (int Year) {

    return ( ( Year % 400 == 0 ) ||
             ( ( Year %   4 == 0 ) &&
               ( Year % 100 != 0 ) ) );

} // end isLeapYear
```

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 <u>call</u> 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).

Example   7. Functions 25

```
void SquareIt(double Value, double& 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;
```

```
double SquareIt(double Value) {

    double 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;

                               // . . . or even as:

cout << setw(10) << setprecision(4) << SquareIt(xval);
```

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 structured variables).

```
void readItem(ifstream& In, string& SKU, int& Units, int& Dollars,
              int& Cents) {

    getline(In, SKU, ':');

    In >> Units;

    In >> Dollars;

    In.ignore(INT_MAX, '.');

    In >> Cents;

    In.ignore(INT_MAX, '\n');


    return;

}
```

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

Placing the declaration outside all function bodies gives the function name file scope. This allows any function defined later (after the declaration) in the same file to invoke that function.

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

The declaration for a function may occur more than once.

The definition (implementation) of a function can occur only one time.

Consider designing a function to determine if three given integer values are potentially the sides of a valid triangle.

Geometry review: there exists a triangle with sides A, B and C if the sum of any two sides is larger than the third side.

Initial Considerations

The decision can be made entirely on the basis of the three values that are given. The function does not require any additional information, so it will not be provided with any.

We will delegate all input and output operations to the client code. The function will take the three integer values as parameters.

The function will return a `bool` value to indicate the result of the test. This could be implemented via a reference parameter, but it is cleaner to use a typed function.

The test and interface seem clear enough… here's an implementation:

```
bool validTriangle(int A, int B, int C) {
    return (A + B > C);
}
```

Function Testing

The function implementation must be tested. The specification gave no indication of how the input will be supplied to the program, nor how the function will be used. However we may still test the function by supplying a "driver" and suitable test data.

The "driver" will read a sequence of data sets from an input file and generate a report file summarizing the results obtained from the function implementation.

The report must be examined by a human to determine whether any cases were handled incorrectly.

The driver is straightforward because the function interface is simple and the function performs a single, constrained task:

```
. . .
int main() {

   . . .
   int A, B, C;
   In >> A >> B >> C;
   In.ignore(INT_MAX, '\n');
   while (In) {
      cout << setw(5) << A
           << setw(5) << B
           << setw(5) << C
           << boolalpha
           << "       "
           << validTriangle(A, B, C) << endl;
      In >> A >> B >> C;
      In.ignore(INT_MAX, '\n');
   }
   . . .
}
```

Of course, this is of no use without test data.

Designing good test data is one of the hardest tasks a developer will face.

Usually the logic of the problem will imply a number of cases; these may be distinct or they may overlap.

A test designer must identify these cases and create data corresponding to them.

The test designer must also determine what the correct results would be, usually by hand, since the implementation being tested obviously cannot be used to determine correctness.

Case Identification

Obviously there are two main cases here: valid triangles and invalid triangles.

Obviously there are an infinite number of possible test cases; we cannot try all of them. This leads to the Fundamental Rules of Testing:

No amount of testing can prove an implementation is entirely correct.

The goal of testing is to discover flaws, not to verify correctness.

Despite the first rule, do not conclude testing is unimportant or pointless.

In view of the second, remember your goal is to "break" the implementation being tested.

Valid triangles can be classified as acute or right or obtuse. It's not clear that those distinctions are relevant here, but it wouldn't hurt to be sure that all are included in the test data.

Invalid triangles will have one side that's "too long" for the other two.

3	6	7	acute
5	12	13	right
5	12	15	obtuse

3	6	10	bad
6	10	3	bad
10	3	6	bad
3	5	8	boundary case
5	8	3	boundary case
8	3	5	boundary case

The test design has identified a basic issue the function designer missed. What?

It has also missed a basic issue also missed by the function designer. What is that?

When the driver and function are executed on the given test data the results are correct for the given valid triangle data. However, the results are incorrect for several of the invalid data cases:

```
   3    6   10        false
   6   10    3        true
  10    3    6        true
   3    5    8        false
   5    8    3        true
   8    3    5        true
```

By examining the test results we can determine the responsible logical flaw in the current design. Knowing that, we can revise the design to eliminate the flaw.

Here, it is obvious the test designer considered the effect of permuting (reordering) the input values and the developer did not. This sort of oversight during program design is all too common and often not so easily detected or repaired.

If the test designer also overlooked this issue then the flaw would have slipped through.

In view of the test results the function must be redesigned. There are at least two sensible approaches. We could first determine the maximum of the three values and then test whether the sum of the other two exceeds the maximum. We could consider all three possible tests, not just whether `A + B > C`.

```
bool validTriangle(int A, int B, int C) {
    return ( (A + B > C) &&
             (A + C > B) &&
             (B + C > A) );
}
```

The revision fixes the detected problem.

What problems remain?

The revised function produces the following output on the test data:

3	6	7	true
5	12	13	true
5	12	15	true
3	6	10	false
6	10	3	false
10	3	6	false
3	5	8	false
5	8	3	false
8	3	5	false

Note well:

- You cannot simply assume that the revision corrected the problem --- retest!

- You cannot simply assume that the revision has not introduced new problems. You must retest on ALL of the test data.

- The amount of test data used here is certainly inadequate.

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
```
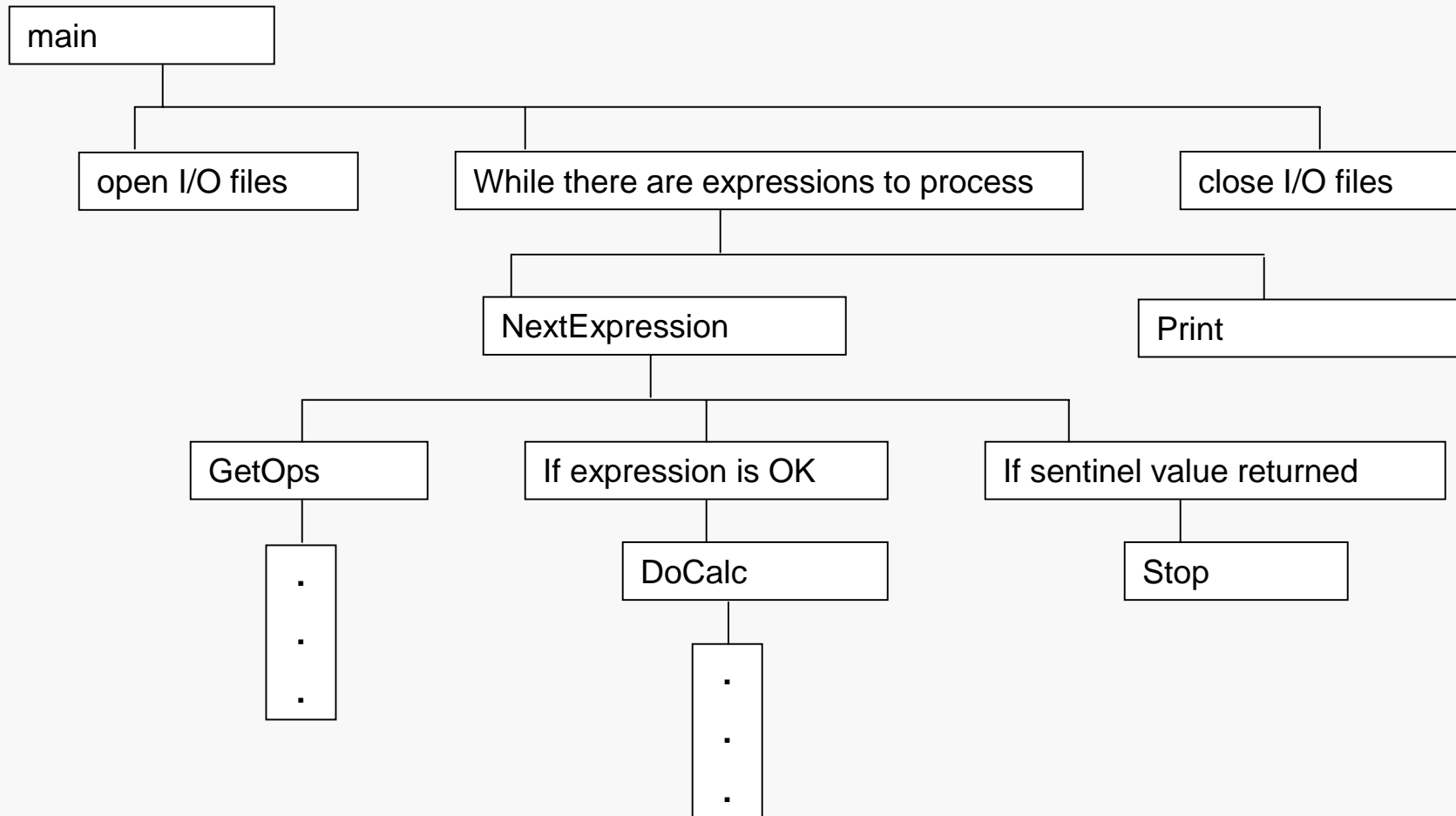
Output:

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

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

An analysis of the formal specification leads to the following design:

```
                                    main
                                      |
        +-----------------------------+-----------------------------+
        |                             |                             |
  open I/O files    While there are expressions to process    close I/O files
                                      |
                        +-------------+-------------+
                        |                           |
                  NextExpression                  Print
                        |
          +-------------+-----------------------+
          |                    |                |
       GetOps         If expression is OK   If sentinel value returned
          |                    |                |
          .                 DoCalc             Stop
          .                    |
          .                    .
                               .
                               .
```

```
. . .
int 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("Calculator.data");
    while (iFile && Value != INT_MIN) {
        Value = NextExpression(iFile, Operand1, Operator, Operand2);
        if (Value == INT_MIN) {
            cout << "No more expressions." << endl;
            iFile.close();
            return 0;
        }
        Print(Operand1, Operator, Operand2, Value);
    }
    iFile.close();  // probably never executed, included for safety
    return 0;
}
```

```
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;

}
```

```cpp
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;   // no protection against Op2 == 0
               break;
    default:   Result = INT_MIN;     // return flag if operation
    }                                //    is invalid
    return Result;
}
```