

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 called, or invoked, 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.

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

Every function must have an implementation or definition, 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.

```
int getValue() {  
  
    const string UserPrompt = "Please enter . . . ";  
    int valueEntered;  
  
    cout << UserPrompt;  
    cin >> valueEntered;  
  
    return valueEntered;  
}
```

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:

```
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:

```
double circleArea(double Radius) {  
    const double PI = 3.141596224;  
    return (PI * Radius * Radius);  
}
```

```
// 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 formal parameters.

```
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 interface 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 actual parameters. 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).

```
// 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 prototype.

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.

```
#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;
        }
        . . .
    }
}
```

```
    . . .
    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;
}
```

Scope Example

```
#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?

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.

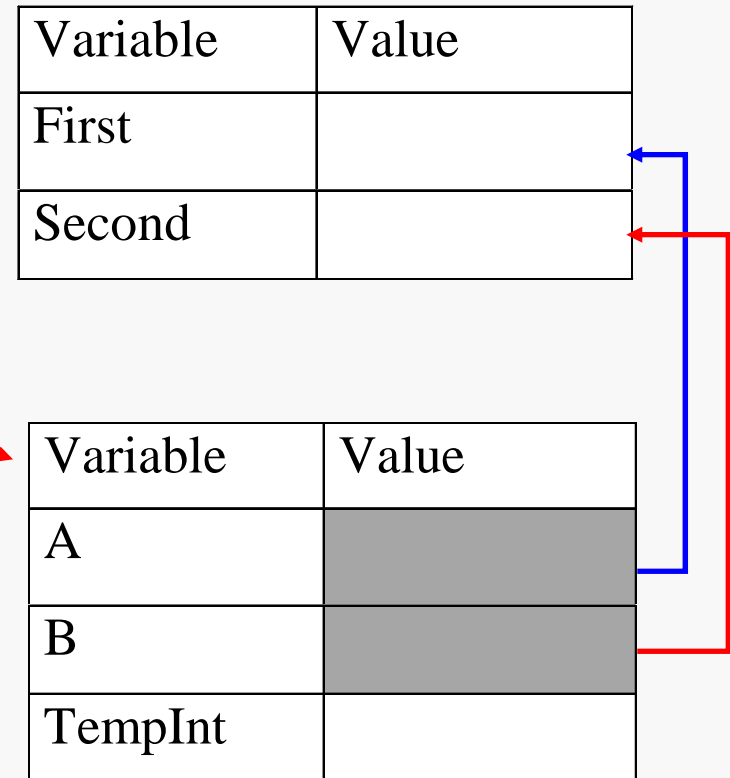
Pass-by-Reference Parameters

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 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;  
}
```

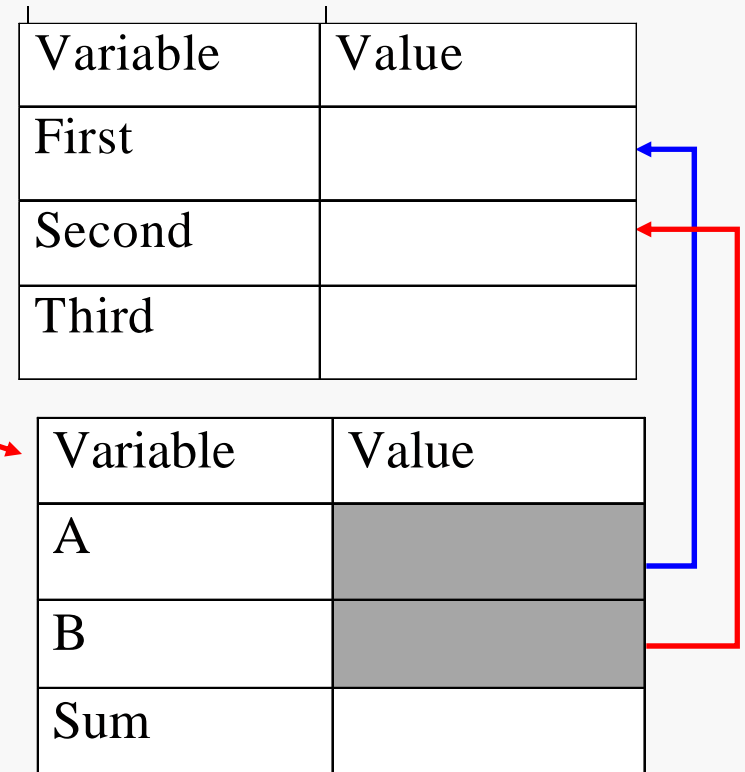


Pass-by-Constant-Reference

- 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);  
. . .
```

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



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.

Parameter Communication Trace

```
. . .
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
