Functions and Libraries

Functions

- Previous examples
  - Programmer-defined functions
    - `main()`
    - …
  - Library-defined functions
    - `cin.get()`
    - `string` member functions `size()`
    - `getline(…)`
- Advice
  - Don’t reinvent the wheel! There are lots of libraries out there

Terminology

- A function is invoked by a *function call / function invocation*

```
y = f(a);
```

- A function call specifies
  - The *function name*
    - The name indicates what function is to be called

```
y = f(a);
```

  - The *actual parameters* to be used in the invocation
    - The values are the information that the called function requires from the invoking function to do its task

```
y = f(a);
```

- A function call produces a *return value*

  The return value is the value of the function call

```
y = f(a);
```

Invocation Process

- *Flow of control* is temporarily transferred to the invoked function
  - Correspondence established between *actual* parameters of the invocation with the *formal* parameters of the definition

```
cout << "Enter number: ";
double a;
cin >> a;
y = f(a);
cout << y;
```

  - Value of **a** is given to **x**

```
double f(double x) {
   double result =
         x*x + 2*x + 5;
   return result;
}
```

Invocation Process

- *Flow of control* is temporarily transferred to the invoked function
  - Local objects are also maintained in the invocation's *activation record.* Even main() has a record

```
cout << "Enter number: ";
double a;
cin >> a;
y = f(a);
cout << y;
```

  - Activation record is large enough to store values associated with each object that is defined by the function

```
double f(double x) {
   double result =
         x*x + 2*x + 5;
   return result;
}
```

Invocation Process

- *Flow of control* is temporarily transferred to the invoked function
  - Other information may also be maintained in the invocation's *activation record*

```
cout << "Enter number: ";
double a;
cin >> a;
y = f(a);
cout << y;
```

```
double f(double x) {
   double result =
         x*x + 2*x + 5;
   return result;
}
```

  - Possibly a pointer to the current statement being executed and a pointer to the invoking statement

Invocation Process

- *Flow of control* is temporarily transferred to the invoked function
  - Next statement executed is the first one in the invoked function

```
cout << "Enter number: ";
double a;
cin >> a;
y = f(a);
cout << y;
```

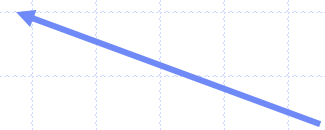
```
double f(double x) {
   double result =
         x*x + 2*x + 5;
   return result;
}
```

Invocation Process

- *Flow of control* is temporarily transferred to the invoked function
  - After function completes its action, flow of control is returned to the invoking function and the return value is used as value of invocation

```
cout << "Enter number: ";
double a;
cin >> a;
y = f(a);
cout << y;
```

```
double f(double x) {
   double result =
        x*x + 2*x + 5;
   return result;
}
```

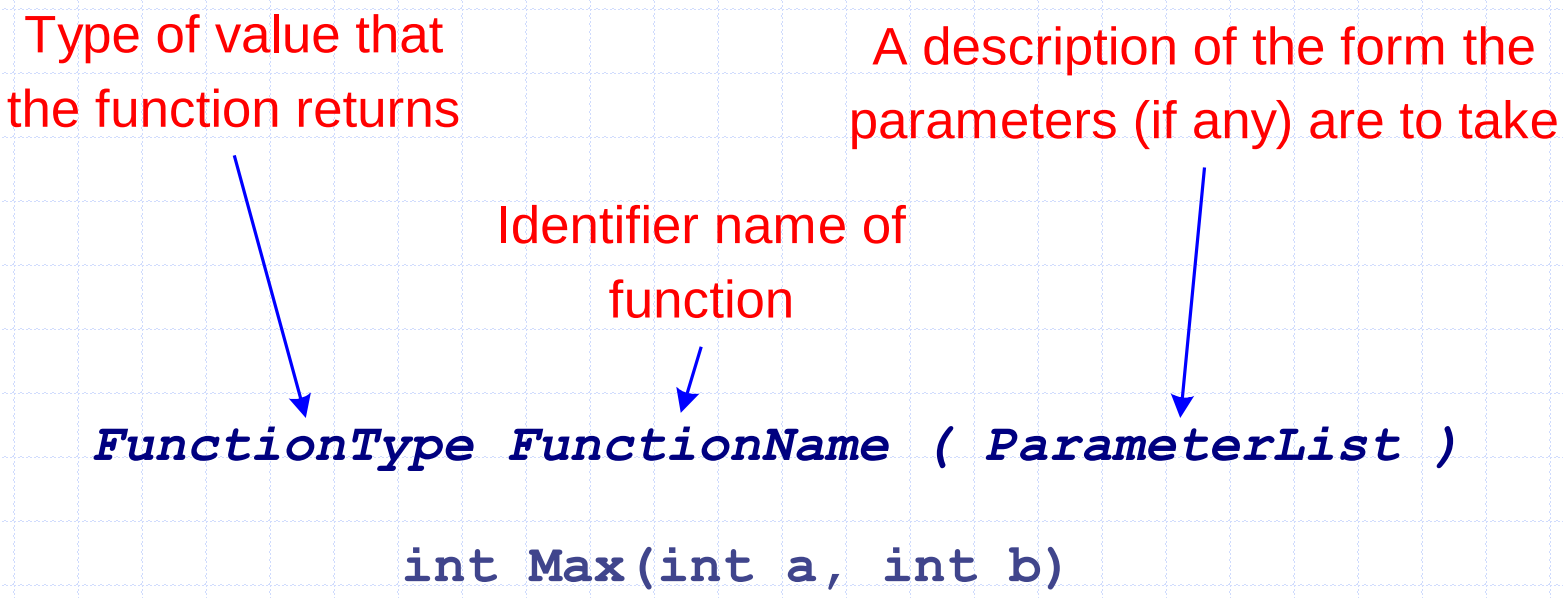


Execution Process

- Function body of invoked function is executed
- Flow of control then returns to the invocation statement
- The return value of the invoked function is used as the value of the invocation expression

Function Prototypes

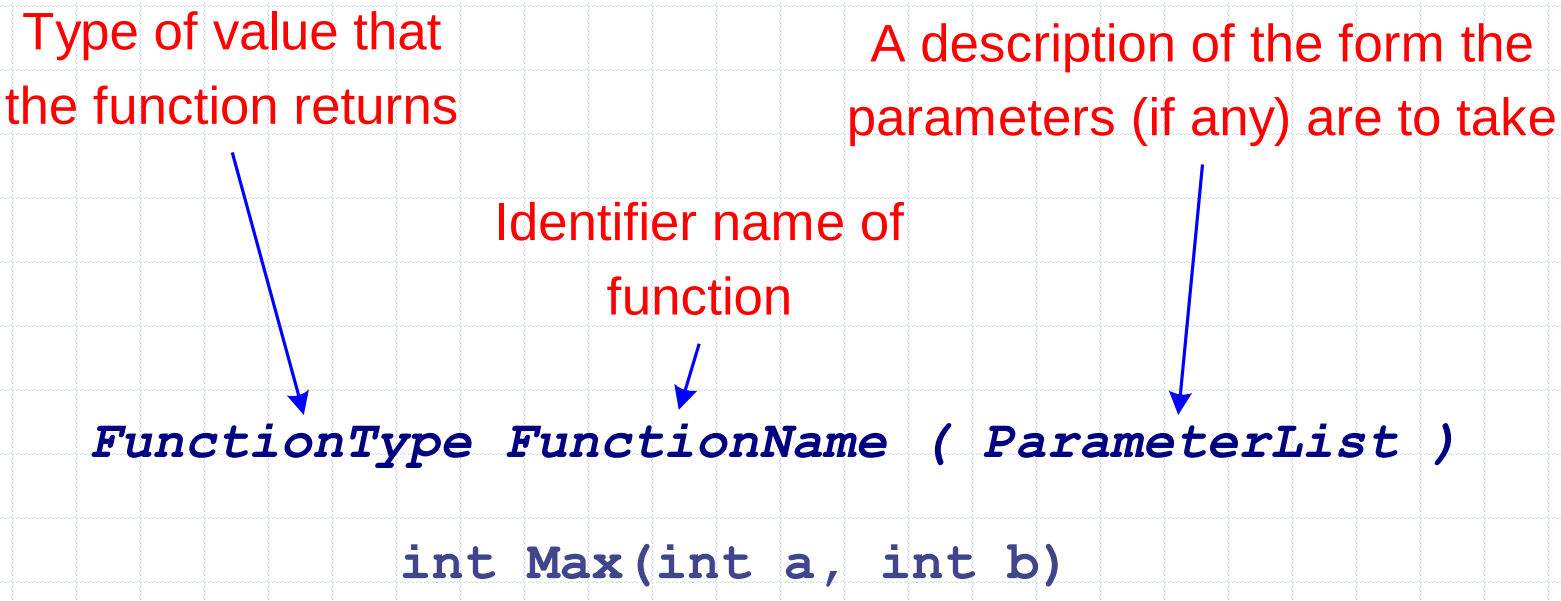
- Before a function can appear in an invocation its interface must be specified
  - *Prototype* or complete definition




```
int Max(int a, int b)
```

Function Prototypes

- Before a function can appear in an invocation its interface must be specified
  - Prototypes are normally kept in library header files

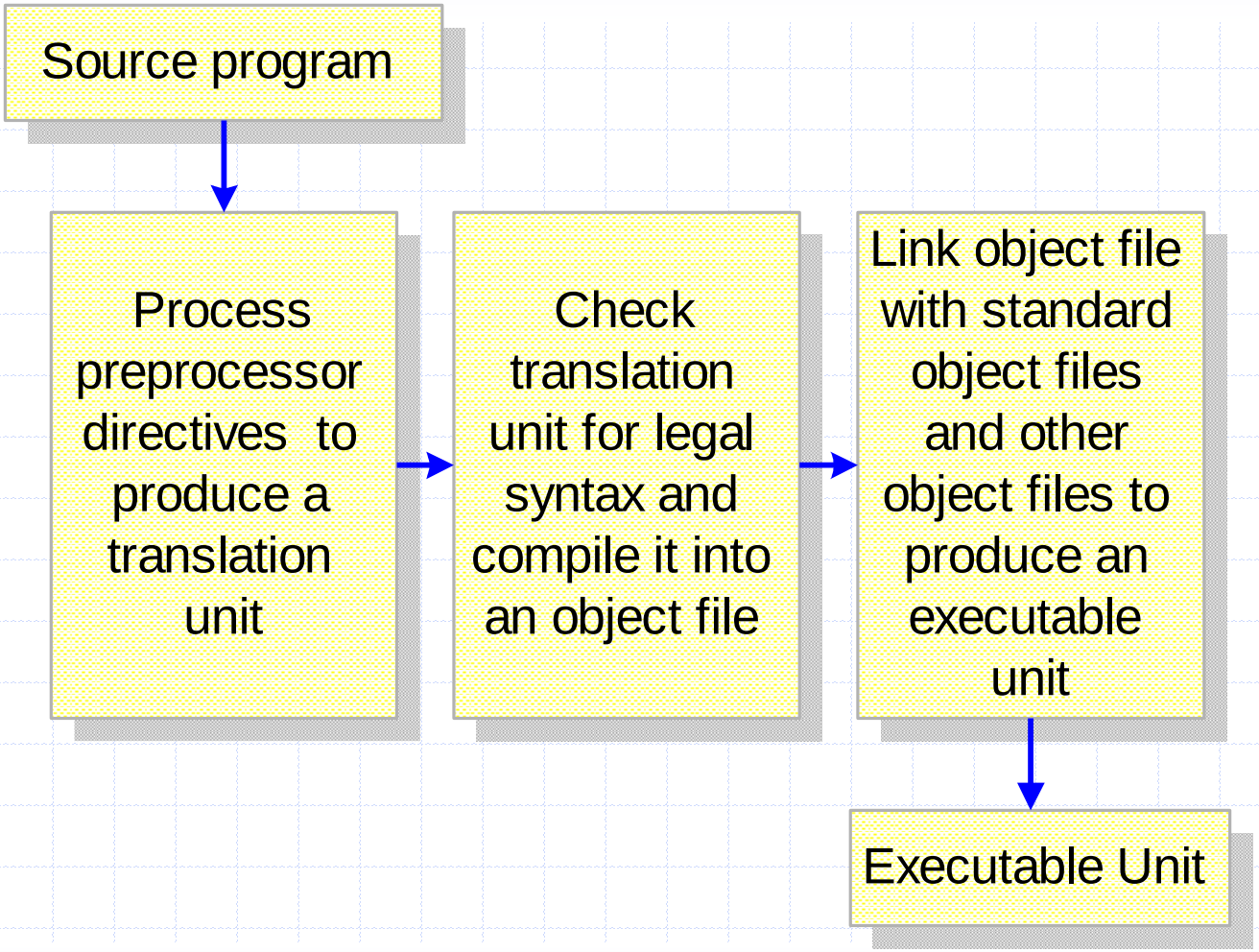



```
int Max(int a, int b)
```

Libraries

- Library
  - Collection of functions, classes, and objects grouped by commonality of purpose
  - Include statement provides access to the names and descriptions of the library components
  - Linker connects program to actual library definitions
- Previous examples
  - String: STL's string class

Basic Translation Process



Some Standard Libraries

- fstream
  - File stream processing
- assert
  - C-based library for assertion processing
- iomanip
  - Formatted input/output (I/O) requests
- ctype
  - C-based library for character manipulations
- math
  - C-based library for trigonometric and logarithmic functions

- Note
  - C++ has many other libraries

Library Header Files

- Describes library components
- Typically contain
  - Function prototypes
    - Interface description
  - Class definitions
- Sometimes contain
  - Object definitions
    - Example: **`cout`** and **`cin`** in **`iostream`**
- Typically do not contain function definitions
  - Definitions are in source files
  - Access to compiled versions of source files provided by a linker

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
   cout << "Enter Quadratic coefficients: ";
   double a, b, c;
   cin >> a >> b >> c;
   if ( (a != 0) && (b*b - 4*a*c > 0) ) {
       double radical = sqrt(b*b - 4*a*c);
       double root1 = (-b + radical) / (2*a);
       double root2 = (-b - radical) / (2*a);
       cout << "Roots: " << root1 << " " << root2;
   }
   else {
       cout << "Does not have two real roots";
   }
   return 0;
}
```

Library header files

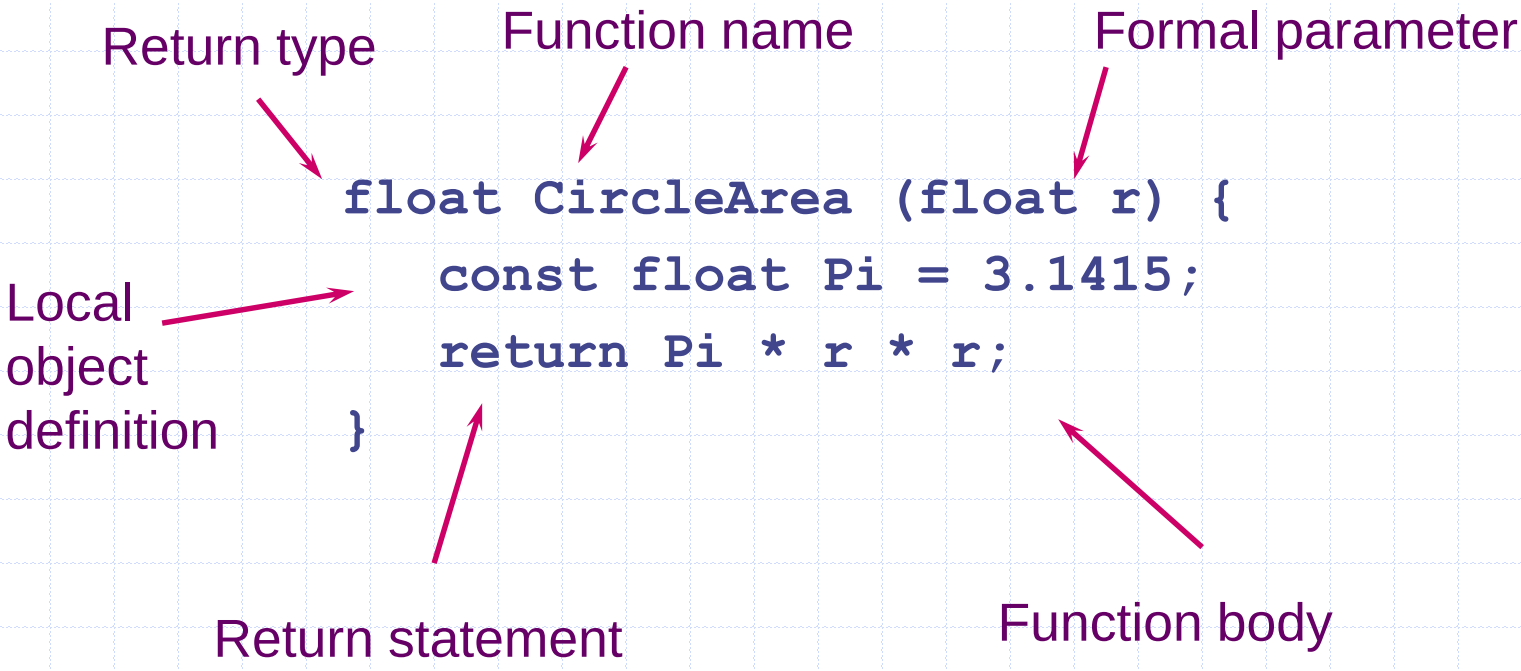
Invocation

```
ifstream sin("in1.txt");    // extract from in1.txt
ofstream sout("out1.txt"); // insert to out1.txt
string s;
while (sin >> s) {
   sout << s << endl;
}
sin.close();                // done with in1.txt
sout.close();               // done with out1.txt
sin.open("in2.txt");    // now extract from in2.txt
sout.open("out.txt",    // now append to out2.txt
(ios_base::out | ios_base::app));
while (sin >> s) {
   sout << s << endl;
}
sin.close();                // done with in2.txt
sout.close();               // done with out2.txt
```

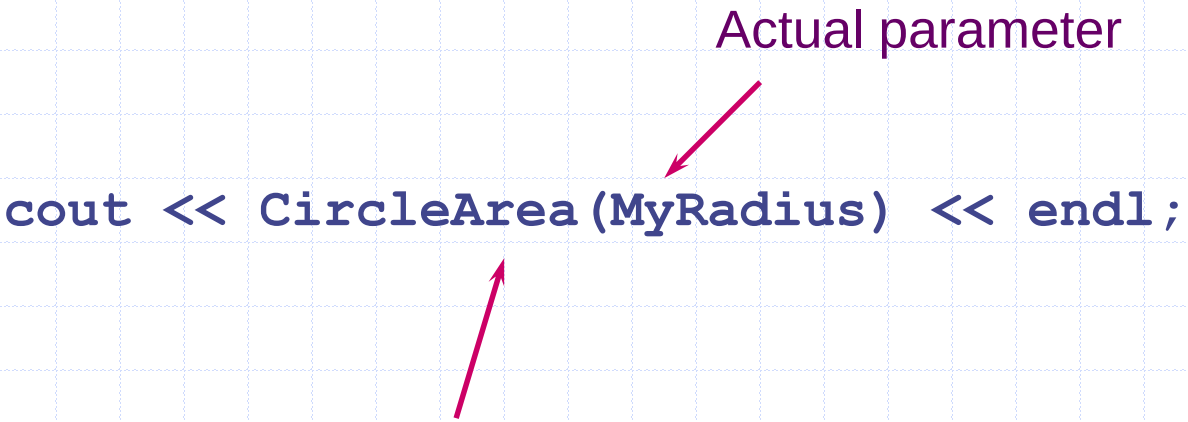
Function Definition

- Includes description of the interface and the function body
  - Interface
    - Similar to a function prototype, but parameters' names are required
  - Body
    - Statement list with curly braces that comprises its actions
    - Return statement to indicate value of invocation

Function Definition



Function Invocation



To process the invocation, the function that contains the insertion statement is suspended and **`CircleArea()`** does its job. The insertion statement is then completed using the value supplied by **`CircleArea()`**.

Simple Programs

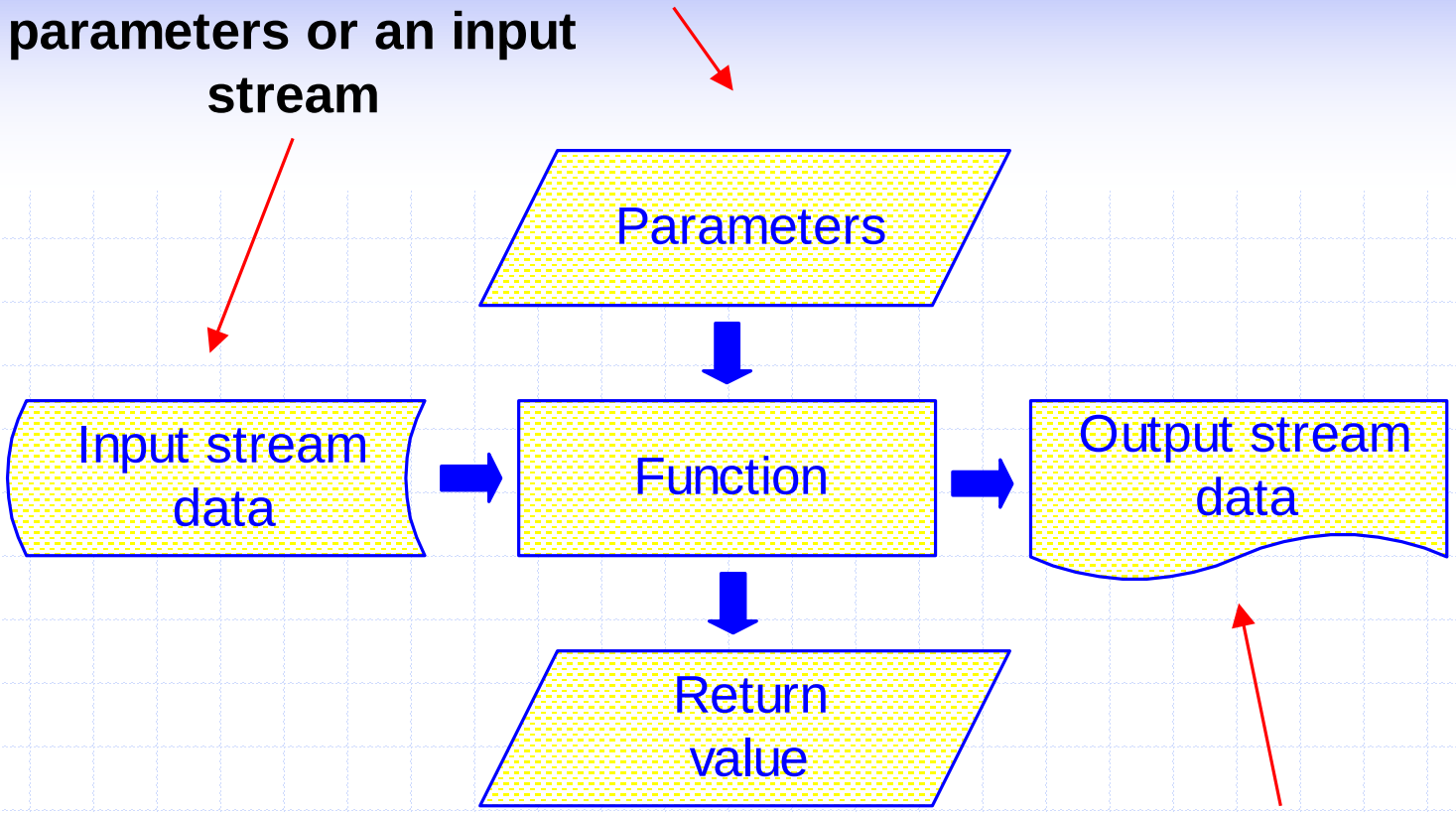
- Single file
  - Include statements
  - Using statements
  - Function prototypes
  - Function definitions
- Functions use value parameter passing
  - Also known as pass by value or call by value
    - The actual parameter is evaluated and a copy is given to the invoked function

```
#include <iostream>
using namespace std;
float CircleArea(float r);
// main(): manage circle computation
int main() {
    cout << "Enter radius: ";
    float MyRadius;
    cin >> MyRadius;
    float Area = CircleArea(MyRadius);
    cout << "Circle has area " << Area;
    return 0;
}
// CircleArea(): compute area of radius r circle
float CircleArea(float r) {
    const float Pi = 3.1415;
    return Pi * r * r;
}
```

Value Parameter Rules

- Formal parameter is created on function invocation and it is initialized with the value of the actual parameter
- Changes to formal parameter do not affect actual parameter
- Reference to a formal parameter produces the value for it in the current activation record
- New activation record for every function invocation
- Formal parameter name is only known within its function
- Formal parameter ceases to exist when the function completes
- Activation record memory is automatically released at function completion

Information to function can come from parameters or an input stream



Information from function can come through a return value or an output stream

Problem

- Definition
  - Input two numbers that represent a range of integers and display the sum of the integers that lie in that range
- Design
  - Prompt user and read the first number
  - Prompt user and read the second number
  - Calculate the sum of integers in the range smaller...larger by adding in turn each integer in that range
  - Display the sum

Range.cpp

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

int PromptAndRead();
int Sum(int a, int b);

int main() {
   int FirstNumber = PromptAndRead();
   int SecondNumber = PromptAndRead();
   int RangeSum = Sum(FirstNumber , SecondNumber);
   cout << "The sum from " << FirstNumber
     << " to " << SecondNumber
     << " is " << RangeSum << endl;
   return 0;
}
```

Range.cpp

```
// PromptAndRead(): prompt & extract next integer
int PromptAndRead() {
   cout << "Enter number (integer): ";
   int Response;
   cin >> Response;
   return Response;
}

// Sum(): compute sum of integers in a ... b
int Sum(int a, int b) {
   int Total = 0;
   for (int i = a; i <= b; ++i) {
        Total += i;
   }
   return Total;
}
```

Blocks and Local Scope

- A block is a list of statements within curly braces
- Blocks can be put anywhere a statement can be put
- Blocks within blocks are *nested* blocks
- An object name is known only within the block in which it is defined and in nested blocks of that block
- A parameter can be considered to be defined at the beginning of the block corresponding to the function body

Local Object Manipulation

```
void f() {
    int i = 1;
    cout << i << endl;              // insert 1
    {
        int j = 10;
        cout << i << j << endl;     // insert 1 10
        i = 2;
        cout << i << j << endl      // insert 2 10
    }
    cout << i << endl;              // insert 2
    cout << j << endl;              // illegal
}
```

Name Reuse

- If a nested block defines an object with the same name as enclosing block, the new definition is in effect in the nested block

However, Don’t Do This At Home

```
void f() {
    {
        int i = 1;
        cout << i << endl;          // insert 1
        {
            cout << i << endl;      // insert 1
            char i = 'a';
            cout << i << endl;      // insert a
        }
        cout << i << endl;          // insert 1
    }
    cout << i << endl;              // illegal insert
}
```

Global Scope

- Objects not defined within a block are global objects
- A global object can be used by any function in the file that is defined after the global object
  - It is best to avoid programmer-defined global objects
    - Exceptions tend to be important constants
- Global objects with appropriate declarations can even be used in other program files
  - **`cout`**, **`cin`**, and **`cerr`** are global objects that are defined in by the iostream library
- Local objects can reuse a global object's name
  - Unary scope operator :: can provide access to global object even if name reuse has occurred

Don’t Do This At Home Either

```
int i = 1;
int main() {
    cout << i << endl;                  // insert 1
    {
        char i = 'a';
        cout << i << endl;              // insert a
        ::i = 2;
        cout << i << endl;              // insert a
        cout << ::i << endl;            // insert 2
    }
    cout << i << endl;
    return 0;
}
```

Consider

```
int main() {
   int Number1 = PromptAndRead();
   int Number2 = PromptAndRead();
   if (Number1 > Number2)
       Swap(Number1, Number2);
   cout << "The numbers in sorted order:"
    << Number1 << ", " << Number2 << endl;
   return 0;
}
void Swap(int a, int b) {
   int Temp = a;
   a = b;
   b = Temp;
   return;
}
```

Parameter passing

- For the previous program to be “effective”, we need a parameter passing style where
  - *Changes to the formal parameter change the actual parameter*

Reference Parameters

- If the formal argument declaration is a *reference* parameter then
  - Formal parameter becomes an *alias* for the actual parameter
    - *Changes to the formal parameter change the actual parameter*
- Function definition determines whether a parameter's passing style is by value or by reference
  - Reference parameter form

$$\texttt{ptype}_i \ \texttt{\&pname}_i$$

```
void Swap(int &a, int &b)
```

Reconsider

```
int main() {
   int Number1 = PromptAndRead();
   int Number2 = PromptAndRead();
   if (Number1 > Number2)
       Swap(Number1, Number2);
   cout << "The numbers in sorted order: "
    << Number1 << ", " << Number2 << endl;
   return 0;
}
void Swap(int &a, int &b) {
   int Temp = a;
   a = b;
   b = Temp;
   return;
}
```

Passed by reference -- in an invocation the actual parameter is given rather than a copy

Return statement not necessary for void functions

Consider

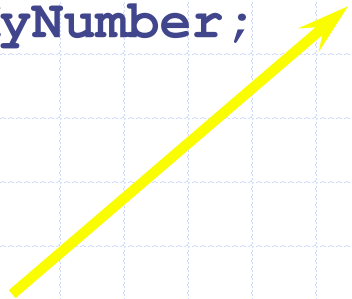
```
int i = 5;
int j = 6;
Swap(i, j);
int a = 7;
int b = 8;
Swap(b, a);
```

```
void Swap(int &a, int &b) {
   int Temp = a;
   a = b;
   b = Temp;
   return;
}
```

Extraction

- A function to extract a value from a given stream

```
void GetNumber(int &MyNumber, istream &sin) {
    sin >> MyNumber;
    return;
}
```



Why is MyNumber a reference parameter?

Why is the stream a reference parameter?

Getnum.cpp

```
int main() {
   ifstream fin("mydata.txt");
   int Number1;
   int Number2;
   cout << "Enter number: ";
   GetNumber(Number1, cin);
   // not needed: cout << "Enter number: ";
   GetNumber(Number2, fin);
   if (Number1 > Number2) {
        Swap(Number1, Number2);
   }
   cout << "The numbers in sorted order: "
    << Number1 << ", " << Number2 << endl;
   return 0;
}
```

Constant Parameters

- The `const` modifier can be applied to formal parameter declarations
  - `const` indicates that the function may not modify the parameter

```
void PromptAndGet(int &n, const string &s) {
   cout << s ;
   cin >> n ;
   // s = "Got it";      // illegal assignment
}                         // caught by compiler
```

  - Sample invocation

```
int x;
PromptAndGet(x, "Enter number (n): ");
```

Constant Parameters

- Usefulness
  - When we want to pass an object by reference, but we do not want to let the called function modify the object
- Question
  - Why not just pass the object by value?
- Answer
  - For large objects, making a copy of the object can be very inefficient

Default Parameters

- Observations
  - Our functions up to this point required that we explicitly pass a value for each of the function parameters
  - It would be convenient to define functions that accept a varying number of parameters

- Default parameters
  - Allows programmer to define a default behavior
    - A value for a parameter can be implicitly passed
    - Reduces need for similar functions that differ only in the number of parameters accepted

Default Parameters

- If the formal argument declaration is of the form

  $\texttt{ptype}_i\ \texttt{pname}_i = \texttt{dvalue}_i$

- then
  - If there is no i^{th} argument in the function invocation, $\texttt{pname}_i$ is initialized to $\texttt{dvalue}_i$
  - The parameter $\texttt{pname}_i$ is an optional value parameter
    - Optional reference parameters are also permitted

Consider

```
void PrintChar(char c = '=', int n = 80) {
   for (int i = 0; i < n; ++i)
       cout << c;
}
```

- What happens in the following invocations?

```
PrintChar('*', 20);
PrintChar('-');
PrintChar();
```

Default Parameters

- Default parameters must appear after any mandatory parameters

- Bad example

```
void Trouble(int x = 5, double z, double y) {
   ...
}
```

Cannot come before mandatory parameters

Default Parameters

- Consider

```
bool GetNumber(int &n, istream &sin = cin) {
   return sin >> n ;
}
```

- Some possible invocations

```
int x, y, z;
ifstream fin("Data.txt");
GetNumber(x, cin);
GetNumber(y);
GetNumber(z, fin);
```

- Design your functions for ease and reuse!

Function Overloading

- A function name can be overloaded
  - Two functions with the same name but with different interfaces
    - Typically this means different formal parameter lists
      - Difference in number of parameters

```
Min(a, b, c)
Min(a, b)
```

      - Difference in types of parameters

```
Min(10, 20)
Min(4.4, 9.2)
```

Function Overloading

```
int Min(int a, int b) {
   cout << "Using int min()" << endl;
   if (a > b)
        return b;
   else
        return a;
}
double Min(double a, double b) {
   cout << "Using double min()" << endl;
   if (a > b)
        return b;
   else
        return a;
}
```

Function Overloading

```
int main() {
   int a = 10;
   int b = 20;
   double x = 4.4;
   double y = 9.2;
   int c = Min(a, b);
   cout << "c is " << c << endl;
   int z = Min(x, y);
   cout << "z is " << z << endl;
   return 0;
}
```

Function Overloading

- Compiler uses function overload resolution to call the most appropriate function
  - First looks for a function definition where the formal and actual parameters exactly match
  - If there is no exact match, the compiler will attempt to cast the actual parameters to ones used by an appropriate function
- The rules for function definition overloading are very complicated
  - Advice
    - Be very careful when using this feature