

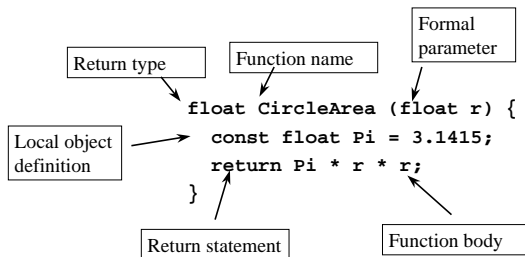
## Chapter 6: Programmer-defined Functions

- Development of simple functions using value and reference parameters

### 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

```

cout << CircleArea(MyRadius) << endl;

```

The diagram shows a function invocation with the following components labeled:

- Actual parameter:** MyRadius

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

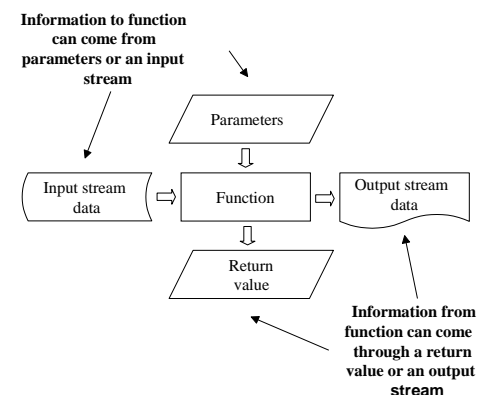
- Also known as pass by value or call by value
  - The actual parameter is evaluated and a copy is given to the invoked function

```
#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



## PromptAndRead()

```
// PromptAndRead(): prompt and extract
// next
// integer

int PromptAndRead() {
    cout << "Enter number (integer): ";
    int Response;
    cin >> Response;

    return Response;
}
```

## Sum()

```
// 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;
}
```

## 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
- ✓ 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 object
    - 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;
}
```

### Using

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

Doesn't do what we want!

### Consider

- ✓ A parameter passing style where
  - Changes to the formal parameter change the actual parameter

That would work!

## 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

```
ptypei &pnamei
```

```
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;
}
```

## Using

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

Return statement not necessary for void functions

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

## Consider

```
int i = 5;      void Swap(int &a,
int j = 6;      int &b) {
Swap(i, j);     int Temp = a;
int a = 7;      a = b;
int b = 8;      b = Temp;
Swap(b, a);     return;
}
```

## Extraction

- 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
- Usefulness

- Question

- Why not just pass the object by value?

- Answer

- For large objects, making a copy of the object can be very inefficient



## Passing Constant Rectangles

```
void DrawBoxes(const RectangleShape &R1,
               const RectangleShape &R2) {
    R1.Draw();
    R2.Draw();
}

int ApiMain() {
    SimpleWindow Demo("Demo Program");
    Demo.Open();
    RectangleShape Rect1(Demo, 3, 2, Blue);
    RectangleShape Rect2(Demo, 6, 5, Yellow);
    DrawBoxes(Rect1, Rect2);
    return 0;
}
```

## 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

$$p\text{type}_i \text{ pname}_i = d\text{value}_i$$

✓ Then

- If there is no  $i^{\text{th}}$  argument in the function invocation,  $\text{pname}_i$  is initialized to  $d\text{value}_i$
- The parameter  $\text{pname}_i$  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

## Random Numbers

- ✓ Generating a sequence of random numbers is often useful
  - In a game, it ensures that a player does not see the same behavior each time
  - In a simulation of a complex system, random numbers can be used to help generate random events
    - Car crash in a simulation of a highway system
    - Likelihood of a gene in cell mutation
    - Weather simulation



## Uniform Random Numbers

- ✓ Uniform random number sequence
  - A sequence of random numbers where
    - Each value in the sequence is drawn from the same range of numbers
    - In each position of the sequence, any value in the number range is equally likely to occur

## Random Numbers

### Examples

- Generate a uniform random number sequence in the range 1 to 6

- Use a fair six-sided die
- Each roll represents a new random number



- Generate a uniform random number sequence in the range 1 to 2

- Use a fair coin
  - Heads: 1, Tails: 2



## Random Numbers

- ✓ We can write an algorithm for generating what looks like random numbers



30 21 9 28 29 ...

- ✓ Because it's an algorithm, we know the rules for generating the next number
  - The generated numbers are not really random
    - They are properly called pseudorandom numbers

## Stdlib Library

- ✓ Provides in part functions for generating pseudorandom numbers

- `rand()`
  - Returns a uniform pseudorandom unsigned int from the inclusive interval 0 to `RAND_MAX`

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;
int main() {
    for (int i = 1; i <= 5; ++i)
        cout << rand() << endl;
    return 0;
}
```

## Different Sequences

- ✓ To produce a different sequence, invoke `void srand(unsigned int);`

- ✓ Consider `seed.cpp`

```
int main() {
    cout << "Enter a seed: ";
    unsigned int Seed;
    cin >> Seed;
    srand(Seed);
    for (int i = 1; i <= 5; ++i)
        cout << rand() << endl;
    return 0;
}
```

## Different Sequences

✓ To automatically get a different sequence each time

- Need a method of setting the seed to a random value
  - The standard method is to use the computer's clock as the value of the seed
  - The function invocation `time()` can be used
    - Returns an integral value of type `time_t`
    - Invocation `time(0)` returns a suitable value for generating a random sequence

## Randseed.cpp

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
using namespace std;

int main() {
    srand((unsigned int) time(0));
    for (int i = 1; i <= 5; ++i)
        cout << rand() << endl;
    return 0;
}
```