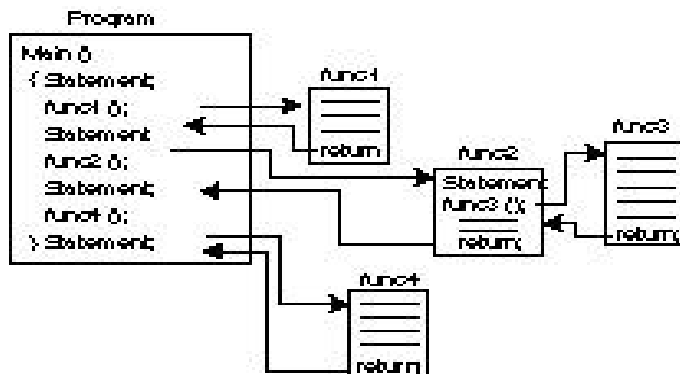


What Is a Function?

- A function is, a subprogram that can act on data and return a value
- Each function has its own name, and when that name is encountered, the execution of the program branches to the body of that function
- When the function returns, execution resumes on the next line of the calling function
- Every C++ program has at least one function, main(). When a program starts, main() is called automatically. main() might call other functions, some of which might call still others.

Illustration of Program Flow



Program Flow

- When a program calls a function, execution switches to the function and then resumes at the line after the function call
- Well-designed functions perform a specific and easily understood task
- Complicated tasks should be broken down into multiple functions, and then each can be called in turn

Function Types

- User-defined functions
- Built-in functions
- Built-in functions are part of your compiler package--they are supplied by the manufacturer for your use

Declaring and Defining Functions

- To use functions in a program one has to first
 - first declare the function. Tell the compiler the
 - name, return type, and parameters of the function
 - then define the function. The definition
 - tells the compiler how the function works
- No function can be called from any other function that hasn't first been declared in the program
- The declaration of a function is called its prototype

Declaring the Function

Three ways to declare a function:

- Write the function declaration or prototype into a file. Then use the `#include` directive to include the file in the program
- Write the function declaration or prototype into the file in which the function is used
- Define the function i.e. write the complete working function before it is called by any other function. In this case the definition acts as its own declaration

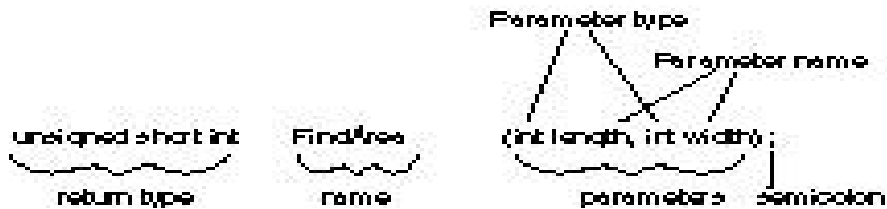
Function Prototypes

- The function prototype is a statement, which means it ends with a semicolon. It consists of the function's
 - return type
 - name, and
 - parameter list (list of all the parameters and their types, separated by commas)
- The function prototype and the function definition must agree exactly about the
 - return type
 - name and
 - parameter list
 - If they do not agree, you will get a compile-time error

Example of Function Prototype

- `Long Area(int length, int width);`
- This prototype declares a function
 - named `Area()`
 - that returns a long and
 - that has two named parameters, both integers
- `Long Area(int, int);`
- This prototype declares a function
 - named `Area()`
 - that returns a long and
 - that has two parameters, both integers

Example of Function Prototype



Function Syntax

- Function Prototype Syntax
 - `return_type function_name ([type [parameterName]]...);`
- Function Definition Syntax
 - `return_type function_name ([type parameterName]...)`
`{`
`statements;`
`}`

Function Prototype Examples

- `long FindArea(long length, long width);`
//returns long, has two parameters
- `void PrintMessage(int messageNumber);`
// returns void, has one parameter int
- `GetChoice();`
// returns int, has no parameters
- `BadFunction();`
// returns int, has no parameters

Function Definition Examples

- `long Area(long l, long w)`
 {
 return l * w;
 }
- `void PrintMessage(int whichMsg)`
 {
 if (whichMsg == 0)
 cout << "Hello.\n";
 if (whichMsg == 1)
 cout << "Goodbye.\n";
 if (whichMsg > 1)
 cout << "I'm confused.\n";
 }

Execution of Functions

- When a function is called, execution begins with the first statement after the opening brace ({)
- Functions can also call other functions and can even call themselves (Recursion)

Local Variables

- Variables can be declared within the body of the function
- These variables are called local variables
- Local variables exist only locally within the function itself
- When the function returns in the main program, the local variables are no longer available.
- Local variables are defined like any other variables
- The parameters passed into the function are also considered local variables
- The parameter variables are used within the function exactly as if they had been defined within the body of the function

Use of local variables and parameters

```
• #include <iostream.h>
• float Convert(float);
• int main()
• {
•     float TempFer;
•     float TempCel;
•     cout << "Please enter the temperature in Fahrenheit: ";
•     cin >> TempFer;
•     TempCel = Convert(TempFer);
•     cout << "\nHere's the temperature in Celsius: ";
•     cout << TempCel << endl;
•     return 0;
• }

• float Convert(float TempFer)
• { float TempCel;
•     TempCel = ((TempFer - 32) * 5) / 9;
•     return TempCel;
• }
```

Output of Program

- Please enter the temperature in Fahrenheit: 212
- Here's the temperature in Celsius: 100
- Please enter the temperature in Fahrenheit: 32
- Here's the temperature in Celsius: 0
- Please enter the temperature in Fahrenheit: 85
- Here's the temperature in Celsius: 29.4444

Variable Scope

- Variable scope determines how long the variable space
- is available to the program and
- where it can be accessed
- Variables declared within a block are scoped to that block
- they can be accessed only within that block and "go out of existence" when that block ends
- Global variables have global scope and are available anywhere within your program

Global Variables

- Variables defined outside of any function have global scope and are called global variables
- Global variables are available from any function in the program, including main().
- A local variable with the same name as a global variable hides the global variable
- In this case the value of the local variables change and the value of the global variables do not change
- If a function has a variable with the same name as a global variable, the name refers to the local variable--not the global--when used within the function.

Global and local variables

- `#include <iostream.h>`
`void myFunction(); // prototype`
`int x = 5, y = 7; // global variables`
`int main() {`
`cout << "x from main: " << x << "\n";`
`cout << "y from main: " << y << "\n\n";`
`myFunction(); 1`
`cout << "Back from myFunction!\n\n";`
`cout << "x from main: " << x << "\n";`
`cout << "y from main: " << y << "\n";`
`return 0; }`

Global and local variables

```
void myFunction() {  
    int y = 10;  
    cout << "x from myFunction: " << x << "\n";  
    cout << "y from myFunction: " << y << "\n\n"; }  

```

Output:

```
x from main: 5  
y from main: 7  
x from myFunction: 5  
y from myFunction: 10  
Back from myFunction!  
x from main: 5  
y from main: 7
```

More on Local Variables

- Variables declared within the function are said to have "local scope."
- These variables are visible and usable only within the function in which they are defined
- In fact, in C++ you can define variables anywhere within the function, not just at its top.
- The scope of the variable is the block in which it is defined.
- If a variable is defined inside a set of braces within the function, that variable is available only within that block.

Scoping within a block

- ```
#include <iostream.h>
void myFunc();
int main() {
 int x = 5;
 cout << "\nIn main x is: " << x;
 myFunc();
 cout << "\nBack in main, x is: " << x;
 return 0; }
```

## Function

```
void myFunc() {
 int x = 8;
 cout << "\nIn myFunc, local x: " << x <<
 endl;
 {
 cout << "\nIn block in myFunc, x is: " << x;
 int x = 9;
 cout << "\nVery local x: " << x;
 }
 cout << "\nOut of block, in myFunc, x: "
 << x << endl; }
}
```

## OUTPUT

Output:

```
In main x is: 5
In myFunc, local x: 8
In block in myFunc, x is: 8
Very local x: 9
Out of block, in myFunc, x: 8
Back in main, x is: 5
```

## Parameters Are Local Variables

- The arguments passed in to the function are local to the function
- Changes made to the arguments do not affect the values in the calling function
- This is known as passing by value, which means a local copy of each argument is made in the function.
- These local copies are treated just like any other local variables.

## Parameters Are Local Variables

- ```
#include <iostream.h>
void swap(int x, int y);
int main() {
    int x = 5, y = 10;
    cout << "Main. Before swap, x: " << x << "
y: " << y << "\n";
    swap(x,y);
    cout << "Main. After swap, x: " << x << "
y: " << y << "\n";
    return 0; }
```

Parameters Are Local Variables

```
void swap (int x, int y) {  
    int temp;  
    cout << "Swap. Before swap, x: " << x << "  
y: " << y << "\n";  
    temp = x;  
    x = y;  
    y = temp;  
    cout << "Swap. After swap, x: " << x << "  
y: " << y << "\n"; }  

```

Parameters Are Local Variables

Output:

```
Main. Before swap, x: 5 y: 10  
Swap. Before swap, x: 5 y: 10  
Swap. After swap, x: 10 y: 5  
Main. After swap, x: 5 y: 10
```

Return Values

- Functions return a value or return void. Void is a signal to the compiler that no value will be returned.
- To return a value from a function, write the keyword return followed by the value you want to return. The value might itself be an expression that returns a value
- It is legal to have more than one return statement in a single function

```
return 5;  
return (x > 5);  
return (MyFunction());
```

Default Parameters

- A default value is a value to use if none is supplied
- `long myFunction (int x = 50);`
- `long myFunction (int = 50);`
- This prototype says, "myFunction() returns a long and takes an integer parameter
- If an argument is not supplied, use the default value of 50."
- The function definition header for this function would be
 - `long myFunction (int x)`
- If the calling function did not include a parameter, the compiler would fill x with the default value of 50.

Demonstration of default parameter values

```
• #include <iostream.h>
  int AreaCube(int length, int width = 25, int height = 1);
  int main() {
    int length = 100, width = 50, height = 2;
    int area = AreaCube(length, width, height);
    cout << "First area equals: " << area << "\n";
    area = AreaCube(length, width);
    cout << "Second time area equals: " << area << "\n";
    area = AreaCube(length);
    cout << "Third time area equals: " << area << "\n";
    return 0; }
  AreaCube(int length, int width, int height)
  { return (length * width * height); }
```

Output:

```
First area equals: 10000
Second time area equals: 5000
Third time area equals: 2500
```

Overloading Functions

- C++ allows creating more than one function with the same name. This is called function overloading or function *polymorphism*
- The functions must differ in their parameter list, with a
 - different type of parameter,
 - a different number of parameters,
 - or both
- `int myFunction (int, int);`
- `int myFunction (long, long);`
- `int myFunction (long);`

function overloading

- `int Double(int);`
- `long Double(long);`
- `float Double(float);`
- `double Double(double);`

Inline Functions

- When a function is defined, the compiler creates just one set of instructions in memory
- When the function is called, execution of the program jumps to those instructions
- When the function returns, execution jumps back to the next line in the calling function
- If the function is called 10 times, the program jumps to the same set of instructions each time. This means there is only one copy of the function, not 10
- There is some performance overhead in jumping in and out of functions
- Some efficiency can be gained if the program can avoid making these jumps just to execute one or two instructions i.e. the program runs faster if the function call can be avoided

Inline Functions

- If a function is declared with the keyword inline, the compiler does not create a real function
- It copies the code from the inline function directly into the calling function
- No jump is made; it is just as if the statements of the function have been written into the calling function
- If the function is called 10 times, the inline code is copied into the calling functions each of those 10 times. The improvement in speed achieved may be swamped by the increase in size of the executable program
- Increased program size brings its own performance cost

Example of Inline Function

```
• #include <iostream.h>
  inline int Double(int);
  int main()
  {
    int target;
    cout << "Enter a number to work with: ";
    cin >> target;
    cout << "\n";
    target = Double(target);
    cout << "Target: " << target << endl;
    target = Double(target);
    cout << "Target: " << target << endl;
    target = Double(target);
    cout << "Target: " << target << endl;
    return 0; }

  int Double(int target)
  { return 2*target; }
```

Output:

```
Enter a number to work with: 20
Target: 40
Target: 80
Target: 160
```

Recursion

- A function can call itself. This is called recursion
- Recursion can be direct or indirect
- It is direct when a function calls itself
- It is indirect recursion when a function calls another function that then calls the first function
- Some problems are easily solved by recursion, usually those in which one acts on data and then act in the same way on the result
- When a function calls itself, a new copy of that function is run
- The local variables in the second version of the function are independent of the local variables in the first, and they cannot affect one another directly

Problem

- Fibonacci series: 1,1,2,3,5,8,13,21,34...
- The first two numbers of the Fibonacci series are 1
- Each subsequent number, after the second, is the sum of the previous two numbers
- Generally, the n th number is the sum of $n - 2$ and $n - 1$, as long as $n > 2$
- Recursive functions need a stop condition. Something must happen to cause the program to stop recursing, or it will never end
- In the Fibonacci series, $n < 3$ is a stop condition
- A Fibonacci problem: Determine the 12th number in the Fibonacci series

Algorithm for Fibonacci Problem

- The algorithm to use is this:
 1. Ask the user for a position in the series.
 2. Call the fib() function with that position, passing in the value the user entered.
 3. The fib() function examines the argument (n).
 - If $n < 3$ it returns 1;
 - otherwise, fib() calls itself (recursively) passing in $n-2$,
calls itself again passing in $n-1$, and returns the sum.

Working of Fibonacci Algorithm

- If fib(1) is called, it returns 1
- If fib(2) is called, it returns 1
- If fib(3) is called, it returns the sum of calling fib(2) and fib(1). Because fib(2) returns 1 and fib(1) returns 1, fib(3) will return 2.
- If fib(4) is called, it returns the sum of calling fib(3) and fib(2). fib(3) returns 2 (by calling fib(2) and fib(1)) and that fib(2) returns 1, so fib(4) will sum these numbers and return 3, which is the fourth number in the series.
- If fib(5) is called, it will return the sum of fib(4) and fib(3). fib(4) returns 3 and fib(3) returns 2, so the sum returned will be 5
- This method is not the most efficient way to solve this problem (in fib(20) the fib() function is

Recursion using Fibonacci series

```
• #include <iostream.h>
  int fib(int n);
  int main() {
    int n, answer;
    cout << "Enter number to find: ";
    cin >> n;
    cout << "\n\n";
    answer = fib(n);
    cout << answer << " is the " << n << "th
    Fibonacci number\n";
    return 0; }
```

Demonstrating Recursion

```
int fib (int n)
{
  cout << "Processing fib(" << n << ")... ";
  if (n < 3 )
  {   cout << "Return 1!\n";
      return (1);
  }
  else
  {   cout << "Call fib(" << n-2 << ") and fib(" << n-
      1 << ").\n";
      return( fib(n-2) + fib(n-1));
  }
}
```

Output of Recursion

- Output:
Enter number to find: 5
Processing fib(5)... Call fib(3) and fib(4)
Processing fib(3)... Call fib(1) and fib(2)
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
Processing fib(4)... Call fib(2) and fib(3)
Processing fib(2)... Return 1!
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
5 is the 5th Fibonacci number

Partitioning RAM

- Global name space: Global variables are
in global name space
- Free store
- Registers
- Code space and
- Stack

Registers

- Special area of memory in the Central Processing Unit (CPU).
- Instruction Pointers: Set of registers responsible for pointing, at any given moment, to the next line of code
- Instruction pointers keep track of which line of code is to be executed next

Code space

- The code itself is in code space
- This is that part of memory set aside to hold the binary form of the instructions created in a program
- Each line of source code is translated into a series of instructions, and each of these instructions is at a particular address in memory
- The instruction pointer has the address of the next instruction to execute

Stack

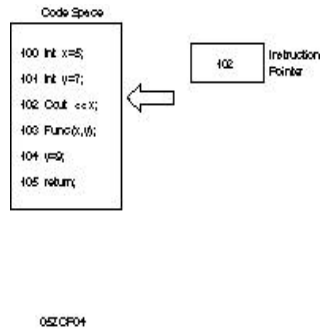
- The stack is a special area of memory allocated for a program to hold the data required by each of the functions in the program
- It is called a stack because it is a last-in, first-out queue
- Last-in, first-out means that whatever is added to the stack last will be the first thing taken
- When data is "pushed" onto the stack, the stack grows; as data is "popped" off the stack, the stack shrinks
- A mental picture of stack is of a series of cubbyholes aligned top to bottom. The top of the stack is whatever cubby the stack pointer (which is another register) happens to be pointing to.

Stack

- Each of the cubbies has a sequential address, and one of those addresses is kept in the stack pointer register. Everything below that address, known as the top of the stack, is considered to be on the stack
- Everything above the top of the stack is considered to be off the stack and invalid.
- When data is put on the stack, it is placed into a cubby above the stack pointer, and then the stack pointer is moved to the new data. When data is popped off the stack, all that really happens is that the address of the stack pointer is changed by moving it down the stack

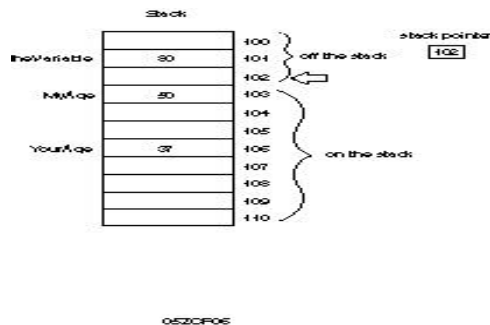
Code Space and Instruction Pointer

-



Stack and Stack Pointer

-



Stack and Functions

- What happens when a program, running on a PC under DOS, branches to a function:
 1. The address in the instruction pointer is incremented to the next instruction past the function call. That address is then placed on the stack, and it will be the return address when the function returns.
 2. Room is made on the stack for the return type you've declared. On a system with two-byte integers, if the return type is declared to be int, another two bytes are added to the stack, but no value is placed in these bytes.
 3. The address of the called function, which is kept in a special area of memory set aside for that purpose, is loaded into the instruction pointer, so the next instruction executed will be in the called function.
 4. The current top of the stack is now noted and is held in a special pointer called the stack frame.
~~Everything added to the stack from now until the~~

Stack and Functions

5. All the arguments to the function are placed on the stack.
6. The instruction now in the instruction pointer is executed, thus executing the first instruction in the function.
7. Local variables are pushed onto the stack as they are defined.

When the function is ready to return, the return value is placed in the area of the stack reserved at step 2. The stack is then popped all the way up to the stack frame pointer, which effectively throws away all the local variables and the arguments to the function.

The return value is popped off the stack and assigned as the value of the function call itself, and the address stashed away in step 1 is retrieved and put into the instruction pointer. The program thus resumes immediately after the function call, with the value of the function retrieved