W 3.1, 3.2

C++ Functions

Modular Program Design

Functions

- Functions allow a program to be broken down into smaller units of code.
- Modular programming is facilitated thus in C and C++
- C is a small language, extendible by using libraries of functions.
- Check the standard function libraries.

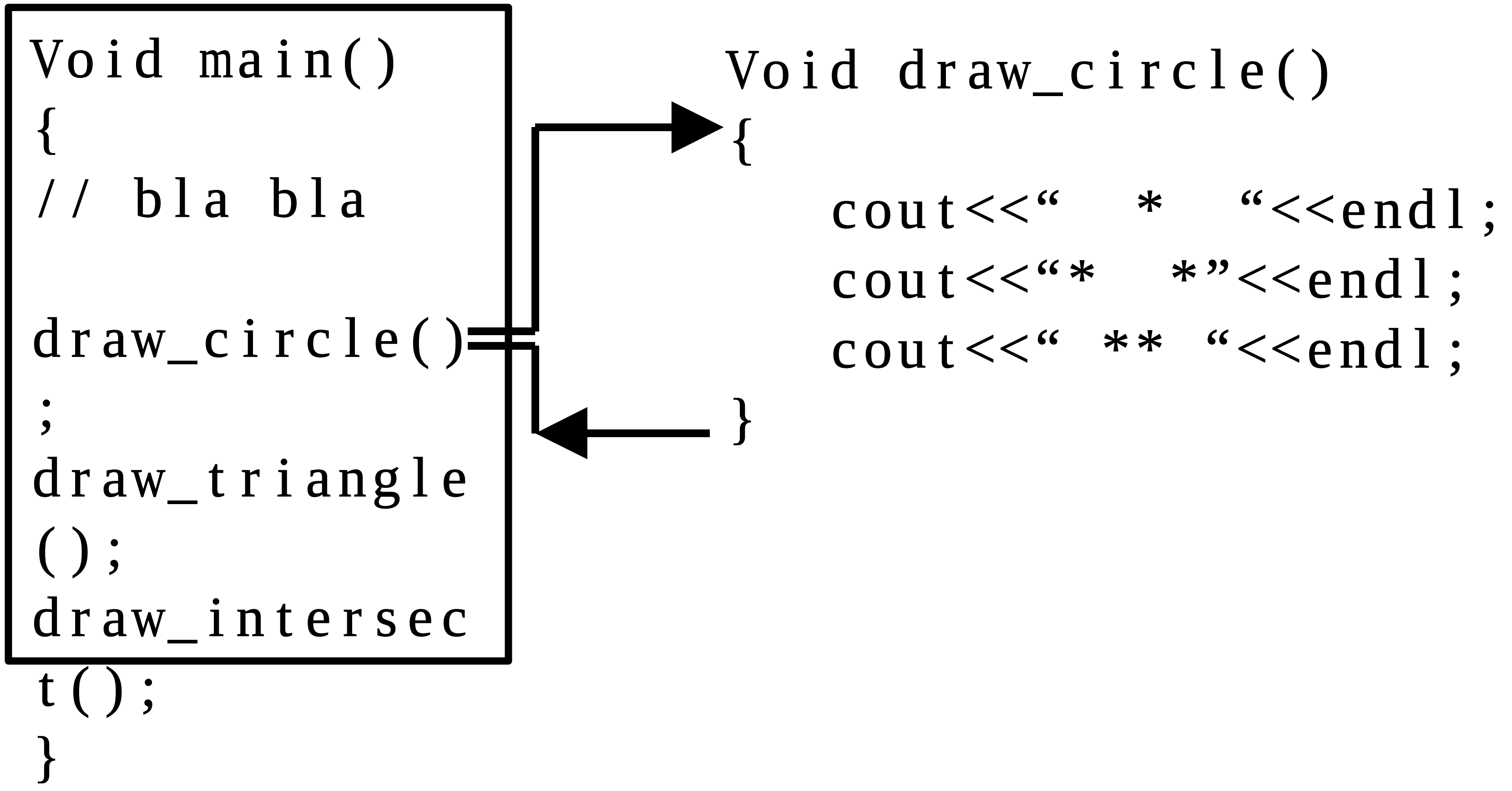
Functions

- **main()** the main function of any program.
- Libraries available, e.g. maths library.
  - sqrt(x)
  - exp(x)
  - log(x)
- User defined functions, build your own.

Simplest Functions

- Some functions do not communicate any data
- Following function simply draws a figure
- Illustrates some of the important ideas
- Functions break up the problem facing programmers

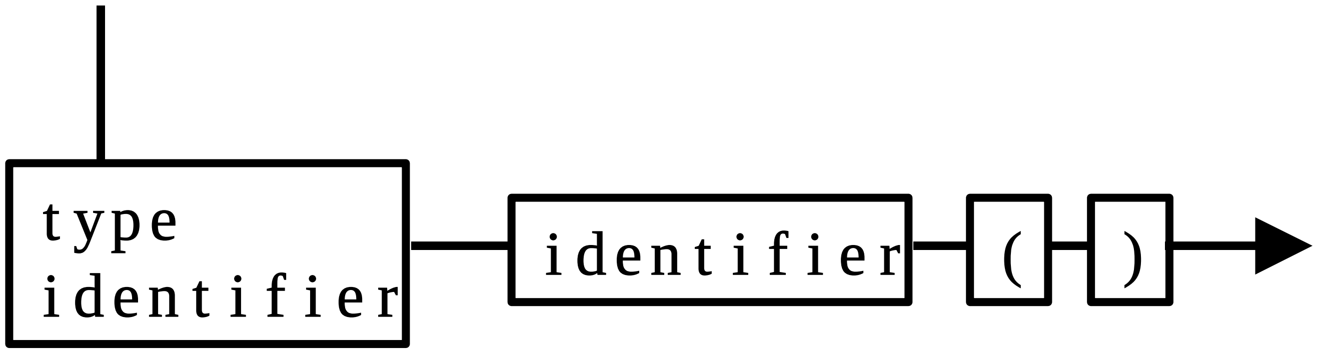
Flow of Control



```
Void main()
{
// bla bla

draw_circle()
;
draw_triangle
();
draw_intersec
t();
}
```

```
Void draw_circle()
{
    cout<<"   *   "<<endl;
    cout<<"*   *"<<endl;
    cout<<" ** "<<endl;
}
```

Minimal Function Definition Syntax



A Simple Program

```
// FILE: stikfig.cpp
#include <iostream.h>
void main ()
{
   // Draw the figure.
   draw_circle (); // Draw a circle.
   draw_triangle (); // Draw a triangle.
   draw_intersect (); // Draw intersecting lines.
}
```

```
// DRAWS A CIRCLE
void draw_circle ()
{
   cout << "    *  " << endl;
   cout << "  *   *" << endl;
   cout << "   * * " << endl;
}  // end draw_circle


// DRAWS A TRIANGLE
void draw_triangle ()
{
   // Draw a triangle.
   draw_intersect ();
   draw_base ();
}  // end draw_triangle
```

```
// DRAWS INTERSECTING LINES
void draw_intersect ()
{
   cout << "   / \\  " << endl;
   cout << "  /   \\ " << endl;
   cout << " /     \\" << endl;
}  // end draw_intersect


// DRAWS A HORIZONTAL LINE
void draw_base ()
{
   cout << " -------" << endl;
}  // end draw_base
```

Common Math Functions

- Commonly used math library functions include
  - cos(x) x is in radians, returns a value which must be assigned as follows
    - `myvar = cos(rads);`
  - log(x), pow(x, y), sin(x), sqrt(x), fmod(x, y)
- These functions accept data through the parameter list
- See Deitel & Deitel 3rd Ed. Page 161

Using Math Functions

- Math library functions
  - Allow the programmer to perform common mathematical calculations
  - Are used by including the header file **`<cmath>`**
- Functions called by writing

  functionName (*argument*)
- Example

```
cout << sqrt( 900.0 );
```

  - Calls the **`sqrt`** (square root) function. The preceding statement would print **`30`**
  - The **`sqrt`** function takes an argument of type **`double`** and returns a result of type **`double`**, as do all functions in the math library

Arguments (also known as parameters)

- Function arguments (or parameters) can be
  - Constants

```
        sqrt( 4 );
```

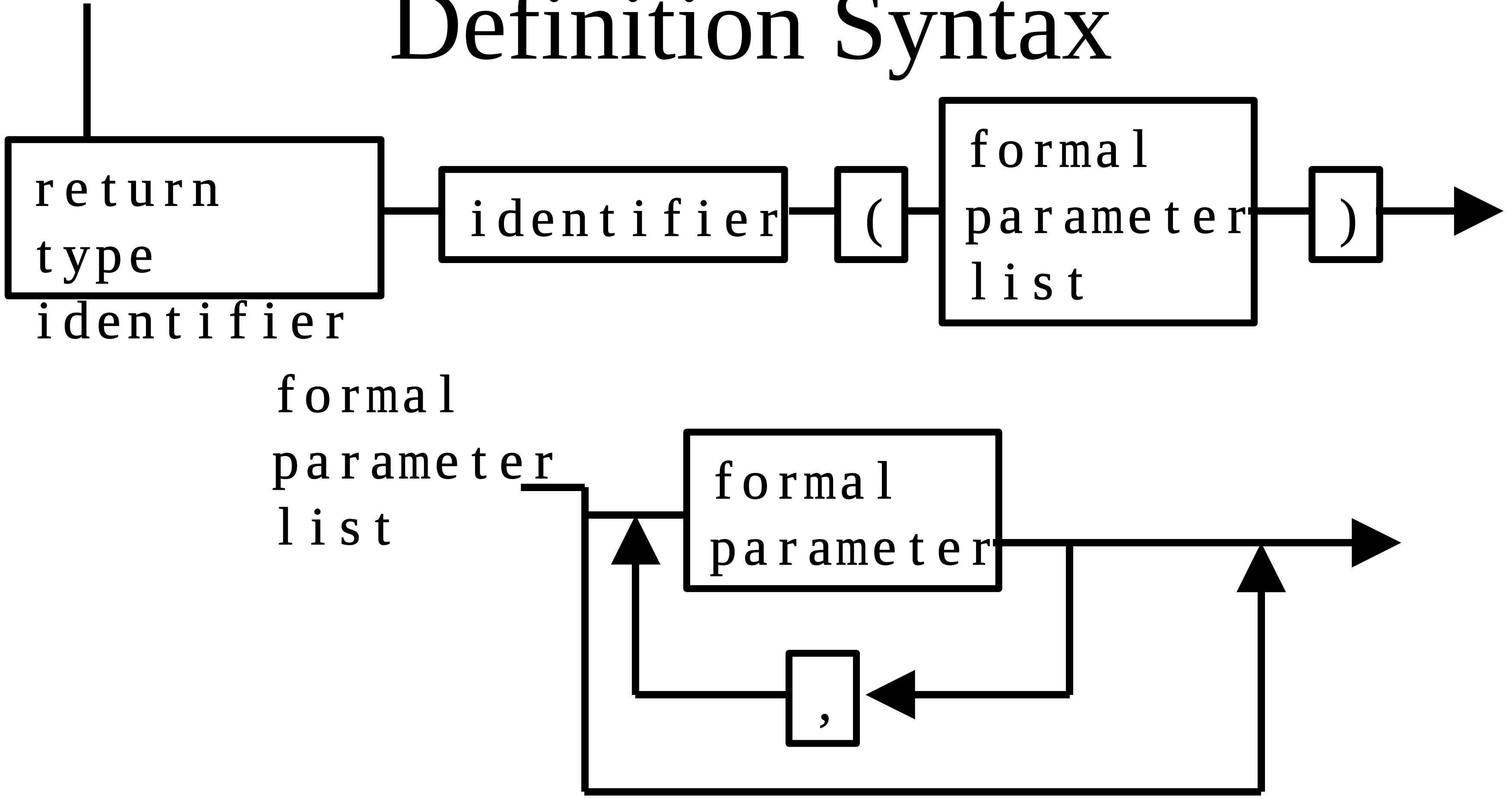
  - Variables

```
        sqrt( x );
```

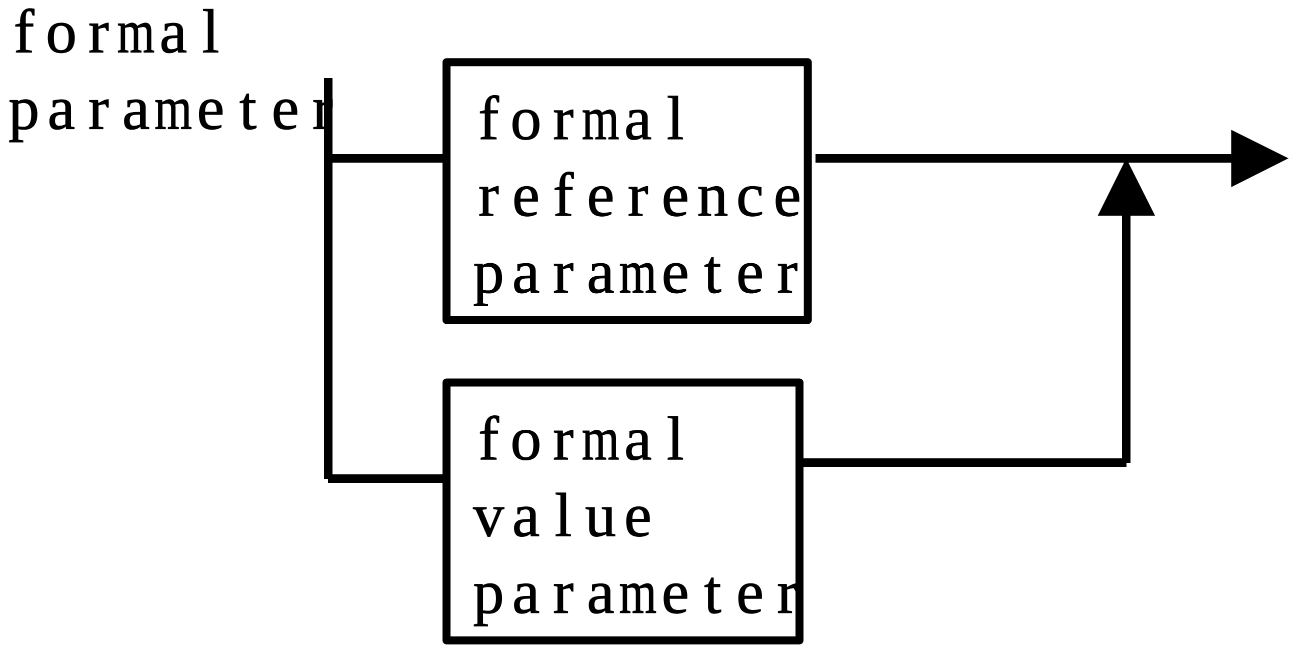
  - Expressions

```
        sqrt( sqrt( x ) );
        sqrt( 3 - 6x );
```

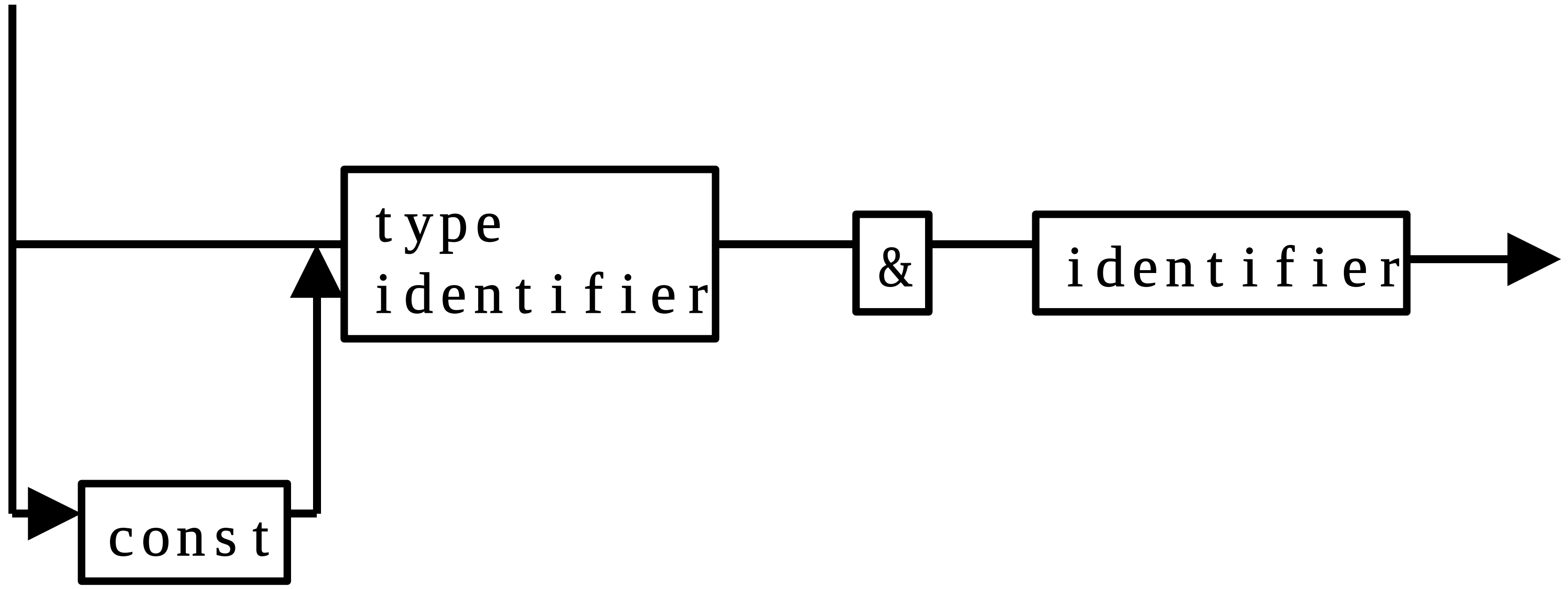
Function (with parameters or arguments) Definition Syntax



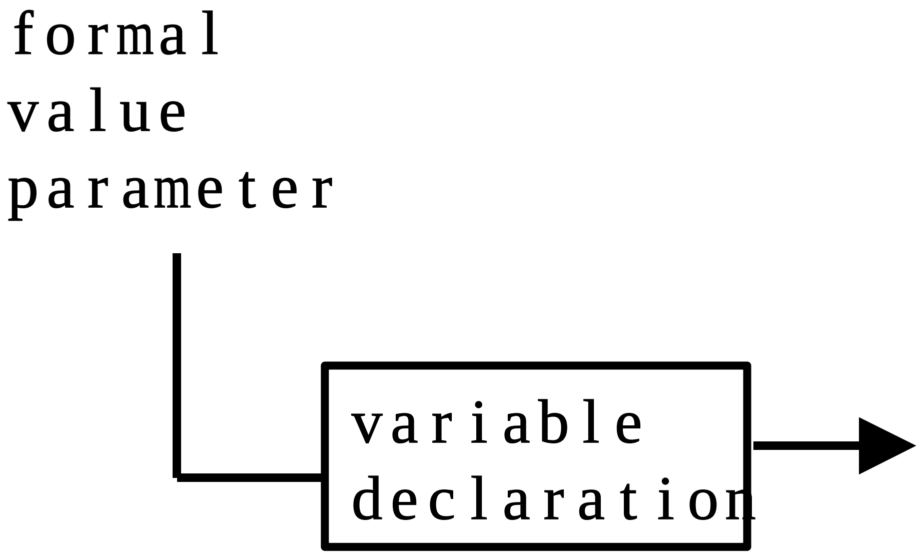
Parameter Syntax



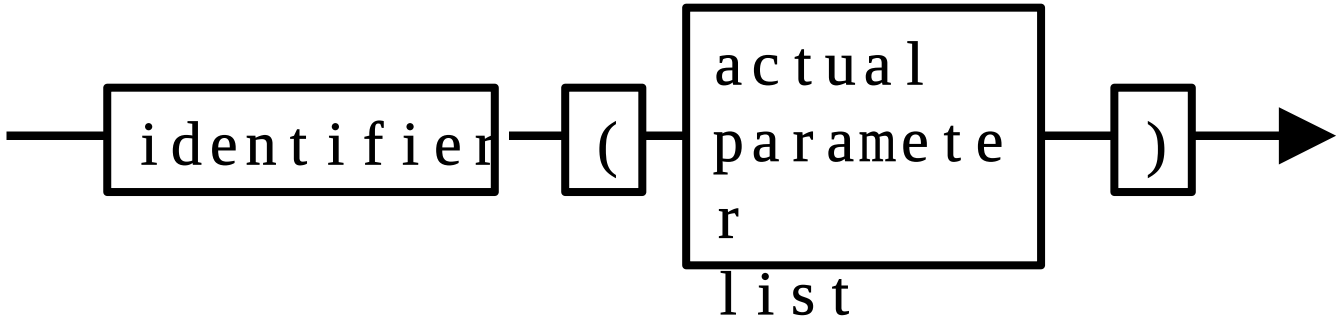
Formal Reference Parameter

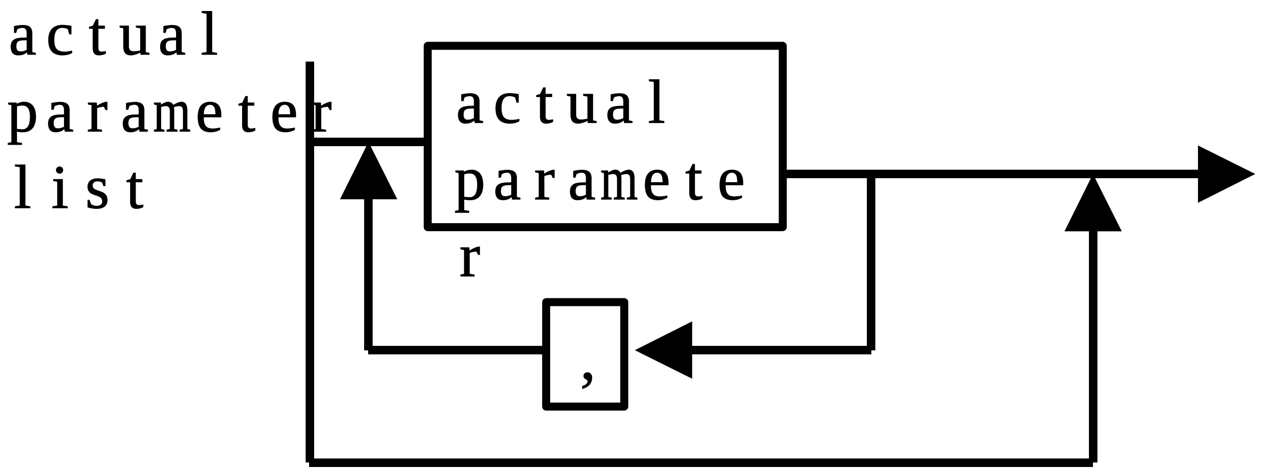


Formal Value Parameter



Function (with parameters) Call Syntax





Function Definitions

- **Prototype** tells compiler what to expect to see in the code and defers implementation until later

```
// Creating and using a programmer-defined function
#include <iostream.h>
int square( int );   // function prototype
int main()
{
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";  //function call
   cout << endl;
   return 0;
}
// Function definition
int square( int y )
{
   return y * y;
}
```

Terminology

- Make sure that you *understand* and *use* the following terminology
  - Prototype
  - user defined function
  - invoke
  - parameter
  - return, return type

Function Definitions with Return

- Formal definition (syntax) below
- Also need prototypes
- Calling convention required

Formal Syntax

```
Return-type function-name(parameter-list)
{
   declarations
   statements
}
```

Example Function with Return()

```
/* A programmer-defined square function */
#include <iostream.h>
int square(int);   /* function prototype */
void main( ){
  int x;
  for (x = 1; x <= 10; x++)
    cout<< square(x); //function call
  cout<<endl;
  return 0;
}
/* Function definition */
int square(int y)
{
  return y * y;
}
```

Function Interactions Through Parameters

- Functions may accept data from where they are called.
- Functions may return data to where they were called.
- Functions may have local data not accessible to the rest of the program.

Function with Parameters by Value

```
// Fig. 3.4: fig03_04.cpp
// Finding the maximum of three integers
#include <iostream.h>
int maximum( int, int, int );   // function prototype
int main()
{
  int a, b, c, mymax;

  cout << "Enter three integers: ";
  cin >> a >> b >> c;      // a, b and c below are arguments to  the maximum function c
  cout << "Maximum is: " << maximum( a, b, c ) << endl;

  mymax = maximum(a, b, c); // my code to demonstrate a value returning
  return 0;
}
```

```
// Function maximum definition
// x, y and z below are parameters to
// the maximum function definition
int maximum( int x, int y, int z )
{
  int max = x;
  if ( y > max )
    max = y;
  if ( z > max )
    max = z;
  return max;
}
```

Parameters by Value

- Local copy of parameter made
- Actual parameter unaffected by assignments to local copy
- **Automatic Variable** created off the stack
- Question
  - what is the effect of replacing
    - mymax = maximum(a, b, c); //with
    - maximum(a, b, c);   // ?

Another Function with Parameters by Value

```
/* Cube a variable using call by value */
#include <iostream.h>
int cubeByValue(int  );
main(){
   int number = 5;
   int return_num = 0;
   cout<<"The original value of number is \n”<< number;
   return_num = cubeByValue(number);
   cout<<"The new value of number is \n”<< return_num;
    cout<<"The old value of number is \n”<< number;
   return 0;
}
int cubeByValue(int n)
{   return n * n * n;   /* cube local variable n */}
```

For Practice Only

- Do questions (they are mostly very small)
  - (3.12), (3.13), (3.14), (3.18), (3.20), (3.48 graded)
- Harder ones include the following
  - (3.41), (3.42)
- Ask me for the code for all of these, will put it on the network later !

Graded Exercises I

- Ex. 3.48 (use return) Deitel & Deitel 3rd Ed.
- Write two separate functions using value parameters and a *return* statement to calculate each of the following…..
  - area of a circle
  - circumference of a circle
  - a rectangle
  - a triangle
  - volume of a sphere (4/3*Pi* r^3)

Graded Exercises II

- Write a program which a builder will use to calculate the area of garden left on a site after building a house. The house is a rectangle, as is the site. You must subtract the area of the house from the area of the site. Make sure that the house fits into the site, check the lengths and sides of each. Use the function for area of a rectangle from the previous question.

Parameters by Reference I

- If more than one value has to be returned from a function, then we cannot simply use the `return ()` statement.
- We can send in the memory location of the variables we want to change.
- We use '&' in the formal parameter list to tell the compiler that we want to change these values.

```
/* A rewritten square function with reference parameter */
#include <iostream.h>
void square(int &);   /* function prototype with reference parameter*/
void main( ){
   int x;
   for (x = 1; x <= 10; x++){
       square(x);                          //function call
       cout<< x;}
   cout<<endl;
}

void square(int & y)   // no return statement here, note the '&'
{
   y *=  y;
}
```

Notes on `void square(int &);`

- The return type is **void**, no return statement will appear in the body of the function
- The prototype includes a '**&**' in the parameter list, this denotes a *reference parameter*.
- The function call passes '**x**' as a parameter, this appears as '**y**' within the body of the function, **y** is **x**, not a local copy, we have passed the *memory location* of '**x**' for use as '**y**' !

Function with Parameters by Reference II

```
/* Cube a variable using call by reference */
#include <stdio.h>
void cubeByReference(int &); //note the void return type here and the ‘&’

void main( ){
   int number = 5;
   cout<<"The original value of number is \n”<< number;
   cubeByReference(number);        // no need to assign value from function
   cout<<"The new value of number is \n”<< number;
   return 0;
}
void cubeByReference(int &nRef) // used the ‘&’ again
{nRef = nRef * nRef * nRef;  /* cube number in main */} // no need for return()
```

Reference Parameters in Code

- We do not use the return statement in the function
- For the prototype we use
  - `void cubeByReference(int &);`
- To invoke the function we use
  - `cubeByReference(number);`
- It looks then same here as before, just examine the formal parameters to distinguish value or reference

Function with Parameters by Reference III

```
int squareByValue(int a){
   return a *= a;
// caller's argument not modified
}

void squareByReference(int &cRef)
{
   cRef *= cRef;
 // caller's argument modified
}
```

```
#include <iostream.h>
int squareByValue(int);
void squareByReference(int &);
main(){
   int x = 2, z = 4;
   cout << "x = " << x << " before squareByValue" << endl
        << "Value returned by squareByValue: "
        << squareByValue(x) << endl
        << "x = " << x << " after squareByValue" << endl << endl;
   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference(z);
   cout << "z = " << z << " after squareByReference" << endl;
   return 0;
}
```

Storage

- Several issues exist
  - Scope
  - Persistence
- We will be revisiting this later with classes
- For now lets look at variables...

Storage Classes

- Storage class specifiers
  - Storage class
    - Where object exists in memory
  - Scope
    - Where object is referenced in program
  - Linkage
    - Where an identifier is known
- Automatic storage
  - Object created and destroyed within its block
  - `auto`
    - Default for local variables.
    - Example:

      ```
      auto float x, y;
      ```
  - `register`
    - Tries to put variables into high-speed registers
  - Can only be used with local variables and parameters

3.10 Storage Classes

- Static storage
  - Variables exist for entire program execution
  - **`static`**
    - Local variables defined in functions
    - Keep value after function ends
    - Only known in their own function
  - **`Extern`**
    - Default for global variables and functions.
    - Known in any function

Identifier Scope Rules

- File scope
  - Defined outside a function, known in all functions
  - Examples include, global variables, function definitions and functions prototypes
- Function scope
  - Can only be referenced inside a function body
  - Only labels (**`start:`**, **`case:`**, etc.)
- Block scope
  - Declared inside a block. Begins at declaration, ends at }
  - Variables, function parameters (local variables of function)
  - Outer blocks “hidden” from inner blocks if same variable name
- Function prototype scope
  - Identifiers in parameter list
  - Names in function prototype optional, and can be used anywhere

```cpp
// Fig. 3.12: fig03_12.cpp
// A scoping example
#include <iostream>

using std::cout;
using std::endl;

void a( void );   // function prototype
void b( void );   // function prototype
void c( void );   // functio

int x = 1;      // global va

int main()
{
   int x = 5;   // local variable to main

   cout << "local x in outer scope of main is " << x << endl;

   {            // start new scope
      int x = 7;

      cout << "local x in inner scope of main is " << x <<
   }            // end new scope

   cout << "local x in outer scope of main is " << x << endl;

   a();         // a has automatic local x
   b();         // b has static local x
   c();         // c uses global x
   a();         // a reinitializes a
   b();         // static local x re
   c();         // global x also ret

```

x is different inside and outside the block.

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5
```

1. Function prototypes

1.1 Initialize global variable

1.2 Initialize local variable

1.3 Initialize local variable in block

2. Call functions

3. Output results

```cpp
   cout << "local x in main is " << x << endl;

   return 0;
}

void a( void )
{
   int x = 25;  // initialized each time a is called

   cout << endl << "local x in a is " << x
        << " after entering a" << endl;
   ++x;
   cout << "local x in a is " << x
        << " before exiting a" << endl;
}

void b( void )
{
    static int x = 50; // Static initialization on
                       // first time b is called.
    cout << endl << "local static x is " << x
         << " on entering b" << endl;
    ++x;
    cout << "local static x is " << x
         << " on exiting b" << endl;
}

void c( void )
{
   cout << endl << "global x is " << x
        << " on entering c" << endl;
   x *= 10;
   cout << "global x is " << x << " on exiting c" << endl;
}
```

Local automatic variables are created and destroyed each time **a** is called.

```
local x in a is 25 after entering a
local x in a is 26 before exiting a
```

Local static variables are not destroyed when the function ends.

```
local static x is 50 on entering b
local static x is 51 on exiting b
```

Global variables are always accessible. Function **c** references the global **x**.

```
global x is 1 on entering c
global x is 10 on exiting c
```

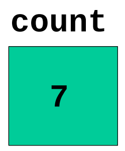
```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

Program Output

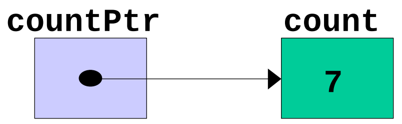
Introducing Pointers

Direct access to memory is forbidden in Java, but is common and even necessary in C and C++

Pointer Variable Declarations and Initialization



- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)
  - Pointers contain the address of a variable that has a specific value (indirect reference)



- Indirection
  - Referencing a pointer value
- Pointer declarations
  - **`*`** indicates variable is a pointer

```
int *myPtr;
```

declares a pointer to an **`int`**, a pointer of type **`int *`**

  - Multiple pointers require multiple asterisks

```
int *myPtr1, *myPtr2;
```

Using Pointers

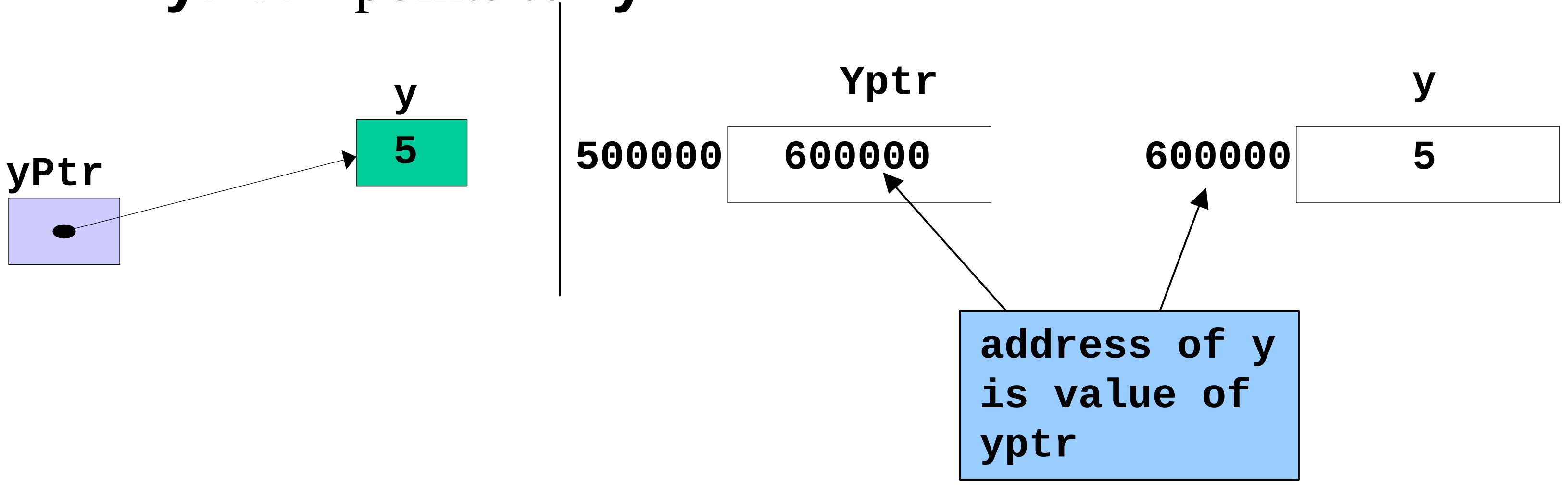
- Can declare pointers to any data type
- Pointers initialization
  - Initialized to **`0`**, **`NULL`**, or an address
    - **`0`** or **`NULL`** points to nothing

Pointer Operators

- **&** (address operator)
  - Returns the address of its operand
  - Example

```
int y = 5;
int *yPtr;
yPtr = &y;    // yPtr gets address of y
```

  - **yPtr** “points to” **y**



Pointer Operators...

- **`*`** (indirection/dereferencing operator)
  - Returns the value of what its operand points to
  - **`*yPtr`** returns **`y`** (because **`yPtr`** points to **`y`**).
  - **`*`** can be used to assign a value to a location in memory

```
*yptr = 7;       // changes y to 7
```

  - Dereferenced pointer (operand of **`*`**) must be an lvalue (no constants)
- **`*`** and **`&`** are inverses
  - Cancel each other out

```
*&myVar == myVar
```

and

```
&*yPtr == yPtr
```

```cpp
// Fig. 5.4: fig05_04.cpp
// Using the & and * operators
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   int a;         // a is an integer
   int *aPtr;     // aPtr is a pointer to an integer

   a = 7;
   aPtr = &a;     // aPtr set to address of a

   cout << "The address of a is " << &a
        << "\nThe value of aPtr is " << aPtr;

   cout << "\n\nThe value of a is " << a
        << "\nThe value of *aPtr is " << *aPtr;

   cout << "\n\nShowing that * and & are inverses of
        << "each other.\n&*aPtr = " << &*aPtr
        << "\n*&aPtr = " << *&aPtr << endl;
   return 0;
}
```

The address of **a** is the value of **aPtr.**

The ***** operator returns an alias to what its operand points to. **aPtr** points to **a**, so ***aPtr** returns **a**.

This is called **dereferencing**

Notice how ***** and **&** are inverses

```
The address of a is 006AFDF4
The value of aPtr is 006AFDF4
The value of a is 7
The value of *aPtr is 7
Showing that * and & are inverses of each other.
&*aPtr = 006AFDF4
*&aPtr = 006AFDF4
```

- 1. Declare variables
- ze
- 3. Print Program Output

Calling Functions by Reference

- Call by reference with pointer arguments
  - Pass address of argument using **&** operator
  - Allows you to change actual location in memory
  - ***** operator used as alias/nickname for variable inside of function

```
void doubleNum( int *number )
{
    *number = 2 * ( *number );
 }
```

  - ***number** used as nickname for the variable passed in
  - When the function is called, must be passed an address

```
 doubleNum( &myNum );
```

```
// Fig. 5.7: fig05_07.cpp
// Cube a variable using call-by-reference
// with a pointer argument
#include <iostream>

using std::cout;
using std::endl;

void cubeByReference( int * );    // pr

int main()
{
   int number = 5;

   cout << "The original value of number is " << number;
   cubeByReference( &number );
   cout << "\nThe new value of number is " << number << endl;
   return 0;
}

void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  // cube number in main
}
```

Notice how the address of **`number`** is given - **`cubeByReference`** expects a pointer (an address of a variable).

Inside **`cubeByReference`**, **`*nPtr`** is used (**`*nPtr`** is **`number`**). This is **dereferencing** again

```
The original value of number is 5
The new value of number is 125
```

- 1. Function prototype - takes a pointer to an `int`.
- 1.1 Initialize variables
- 2. Call function
- . Define function
- Program Output

We will Revisit Pointers

- Arrays make extensive use of pointers
- Pointers to user defined data structures
  - Arrays and Strings
  - Implementing stacks and queues, linked lists and trees
  - Classes and Structs
  - Pointers to Functions even!
- Pointers are used in File I\O