

FUNCTIONS - INTRODUCTION

- ❑ Consider the algorithm to draw two 4x4 squares at certain positions

- // draw the first at position (2,2)

move to (2,2)

draw by (4,0)

draw by (0,4)

draw by (-4,0)

draw by (0,-4)

- // draw the second at position (4,4)

move to (4,4)

draw by (4,0)

draw by (0,4)

draw by (-4,0)

draw by (0,-4)

INTRODUCTION

- ❑ Suppose we want to draw 100 squares ? Surely we don't want to write out the square drawing instructions 100 times

- ❑ **Functions** allow a sequence of statement to be referred to by a **name** and **parameterized**

- ❑ Calling a function causes statement sequence to be executed and a value may be returned when the function terminates

Parameterization allows the same function to be applied to many different values without changing the statements

- ❑ sqrt is function that computes square root of number, sqrt(4) computes square root of 4, sqrt(9) computer square root of 9! Same sequence of statements inside function, with different parameter returns different value

A SOLUTION TO THE SQUARES DRAWING PROBLEM

- // algorithm to draw a unique square (draw_square)
draw by (4,0)
draw by (0,4)
draw by (-4,0)
draw by (0,-4)
- // algorithm to draw the two squares
move to (2,2)
draw_square
move to (4,4)
draw_square

A SECOND SOLUTION TO THE SQUARES DRAWING PROBLEM

- // algorithm to draw a unique square (draw_square) at a position (x,y)
move to (x,y)
draw by (4,0)
draw by (0,4)
draw by (-4,0)
draw by (0,-4)
- // algorithm to draw the two squares
draw_square at (2,2)
draw_square at (4,4)

FUNCTIONAL ABSTRACTION

Also known as procedural abstraction

- ❑ Allows program to be composed of collection of functions
- ❑ Allows common statement sequences to be written once
- ❑ Each function names a sequence of logically connected statements
- ❑ Functions may call other functions

A FUNCTION PROGRAM EXAMPLE

```
#include "pen.h"           // include file
Pen p(10);                // declare variable
void draw_square()        // function to draw a 4x4 square
{
    p.draw_by(4,0);
    p.draw_by(0,4);
    p.draw_by(-4,0);
    p.draw_by(0,-4);
}

int main()                // main function
{
    p.move_to(2,2);
    draw_square();
    p.move_to(4,4);
    draw_square();
    return 0;
}
```

TOP-DOWN PROGRAM DESIGN

- Decompose a problem into smaller manageable problems

Suppose we want a program to draw 3x3 stars



- Decompose the problem to draw a row of three stars

TOP-DOWN PROGRAM DESIGN

- To draw the whole pattern
 - Move the pen to the position of the bottom row
 - Draw a row of three stars
 - Move to the start of the second row
 - Draw a row of three stars
 - Move to the start of the third row
 - Draw a row of three stars
- To draw a row of stars
 - Draw a star
 - Move the pen three units to the right
 - Draw a star
 - Move the pen three units to the right
 - Draw a star
- The smallest unit of the problem is now to draw a star

FUNCTION DECLARATION

- ❑ Introduces the **function name**, **function return type**, and **function parameters** to the program

- The function body (statements) is **not** part of the declaration
- A function must be declared before it is used

Format: *return_type* **function_name** (*parameter_list*);

parameter_list: type param1, type param2, type param3, etc

- ❑ Some function declarations:

- `double sqrt(double);`
- `int func1();`
- `void func2(char, int);`

- ❑ Header files typically contain function declarations, e.g. `math.h` contains: `double sqrt(double)`

FUNCTION DEFINITION

- ❑ Introduces the function name, function return type, and function parameters as well as the **function body** to the program

- Function body (the implementation) is a compound statement

```
return_type function_name(parameter_list)
```

```
{  
    // Function body  
}
```

- Function Definition Example:

```
void print_func()  
{  
    cout << "hello world" << endl;  
}
```

- A function can be defined in any part of the program text or within a library

FUNCTIONS AND PROGRAM STRUCTURE

- ❑ A Program typically consists of functions

Each function average 20-30 lines

- ❑ Functions must be declared before they are used, e.g:

```
//preprocessor statements
define function 1
define function 2
define function 3
    { call function 2}
main()
{
    call function 1
    call function 3
}
```

FUNCTIONS AND PROGRAM STRUCTURE

- ❑ Using declarations allows the order to be flexible:

```
//preprocessor statements
declare function 2
define function 3
    { call function 2}
define function 1
define function 2
main
{
    call function 1
    call function 3
}
```

VOID FUNCTIONS

- ❑ These are functions that do not return a value
They can still cause a **side effect** by modifying a global variable

- ❑ Example:

```
int num = 1;
void print_func()
{
    cout << "hello world" << num << endl;
    num = num + 1;
}
```

- ❑ **Note:** It is not possible to declare variables of type void because type void has no values, i.e. it has no structure

PARAMETERIZED FUNCTIONS

- ❑ A function **declaration** must include the list of parameter types (with an optional name):

– `int func(int)` or `int func(int x)`

The optional name is to improve readability

- ❑ Each parameter type specifies the type of the value that should appear as a function **argument** when the function is called
 - Therefore the function **func** takes one parameter which is an integer
func can be called using: `int x = func(42);`
 - The function name is followed by a bracketed list of function arguments
 - Note: A function definition also requires a parameter list
 - Within the function body the parameters are accessible as variables which are named in the parameter list with their types:

PARAMETERIZED FUNCTIONS

- `int func(int x, int y)`
`{`
 // place function body here
 // variables x and y can be used here
`}`
x and y are the **parameter variables** that can be used in the function body
- **Note** that the **types** of the parameters must be the **same** in the function **declaration** and **definition**

- **Formal parameters:** the parameters used in the function definition, e.g. x and y in

```
int func(int x, int y) ...
```

PARAMETERIZED FUNCTIONS

- **Actual parameters or arguments:** the parameters used in the function call, e.g. 10 and 20 in

```
int x = func(10, 20);
```

- **Note** that the actual parameters are evaluated **before** being passed to the function, e.g.

```
int a = 10, b = 20;  
int x = func(a/2, a+b);
```

- The actual parameters passed to `func` are **5** and **30**
- The order of evaluation of arguments is undefined

FUNCTIONS AND SCOPE

- Functions may only be referred to within the **scope** of the declaration
- Declarations made in a function body or compound statement are **local** to that scope
- Function **parameters** are variables in the scope of the function body
- A **new** variable is created for each parameter whenever the function is called
- Each parameter variable is **destroyed** when the function terminates

Hence **parameters are local** to the function they are part of

RETURNING FROM A FUNCTION

- A function that returns a value (i.e. not void) must contain at least one return statement

```
int times_two(int x)
{
    return x*2;
}
```

The return statement evaluates the following expression and returns its value as the result of the function

- A function terminates as soon as a return value is executed
- The value returned by a return statement must be of the type specified in the function declaration
- A function can have several return statements but only one of them can be executed

RETURNING FROM A FUNCTION

- A return statement can also be used without a return expression in a void function

```
void func()
{
    // do some stuff here
    if (a == b)
        return;           // return early
    // otherwise carry on
}
```

When return is executed the function will immediately terminate

DEFAULT PARAMETERS

- Function parameters may be given default values

```
int func(int x = 10);
```

- If func is called with an empty parameter list

```
func();
```

then the parameter variable x will be initialized to 10

- If func is called with a parameter

```
func(15);
```

then the default is **NOT** used and the parameter variable x is initialized to the actual parameter value, i.e. 15

DEFAULT PARAMETERS

- Note that default parameters must appear at the **END** of the parameter list

```
int func1(int x = 1);           // this is ok
int func2(int x, int y = 1);   // this is ok
int func3(int x = 1, int y);    // this is not ok
```

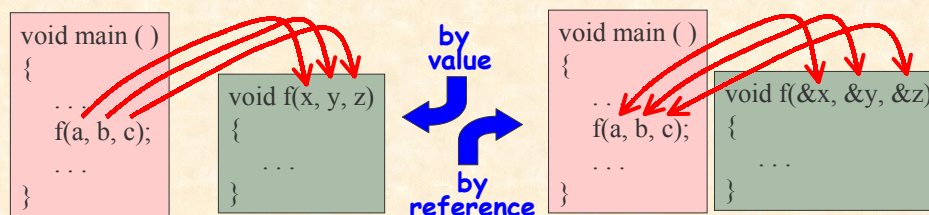
- Also several default parameters can be given

```
int func4(char c, int x = 1, int y = 1);    this is ok
```

PARAMETER PASSING IN FUNCTIONS

- Arguments/Parameters are a way to share data between two functions.
- In all the programs we have seen so far, the arguments in a function call provide data needed by the called function
 - they pass data into the function
- There are times when a function wants to use its parameters to pass data back to the calling program.
 - examples are when functions must return more than one value or must modify the arguments passed to them

We need two types of parameter passing to handle these two situations.



CALL BY VALUE

- ❑ C++ uses **call by value** to pass information to functions. **Given:**

```
void func(int i)
{
    i = 10;
}

int main()
{
    int i = 2;
    func(i);
    // i still has the value 2 here
}
```

CALL BY VALUE

- ❑ Call by value means the **value** of `i` is passed as a parameter to `func`
 - The **local** variable `i` of `func` is initialized to that value
 - The **local** variable `i` of `main` that appears in the call is left untouched - only a **copy of its value** is passed to the function `func`

CALL BY REFERENCE

- ❑ We have already seen that C++ uses **call by value** to pass information to a function (using parameters)

- ❑ References allow a second parameter passing mechanism to be used - **call by reference**

- ❑ If a **formal parameter** type is a reference type

```
int func(int &x)
```

Then the parameter variable x will be a **reference** to the **actual parameter**

CALL BY REFERENCE

```
int y = 1;
```

```
func(y);
```

x will be initialized to a reference to y when the function func is called

So x and y will both denote the **SAME** object

Contrast this to call by value where x will be initialized to a copy of the value of y

- ❑ Note that it is an error to write

```
func(1);
```

USES OF CALL BY REFERENCE

- For Example:
- Given that we want a function that takes two variables and swaps their values around, we can write

```
void swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Therefore

```
int a = 1, b = 2;
// at this point a equals 1 and b equals 2
swap(a, b);
// at this point a equals 2 and b equals 1
```

USES OF CALL BY REFERENCE

- Call by reference has several uses
 - for efficiency - large objects do not need to be copied
 - as a mechanism that allows actual parameters to be changed by a function
 - it provides a mechanism for a function to "return" more than one value. Data can be calculated in a function and passed back to the caller via reference parameters

```
int divide(int x, int y, int &q, int &r)
{
    if (y == 0) return 0;           // return fail
    q = x / y; r = x % y; return 1; // return all ok
}
```

USES OF CALL BY REFERENCE

This can be used as follows:

```
int how_many, rem;  
if (divide(17, 3, how_many, rem)) {  
    // do something using how_many and rem  
}  
else {  
    // do something else since divide failed  
}
```

- **Note:** Functions that does not return value are sometimes called procedures. For example

```
void clear(int from, int to);  
is declaration of procedure.
```