



Valued Functions and Void Functions

- Functions with single return value

```
double total(int number, double price)
{
    double cost;
    cost = number * price;
    return (cost * 1.05);
}
```

A function that returns no value: **void** function

```
void horizontal_line()
{
    cout << "\n-----";
    return;
}
```

- Difference from value-returning functions
 - **void** in place of return value type: no value is returned
 - **return** statement does not contain an expression



A Simple void Function Call

```
int main()
{
    cout << "\nName:";
    horizontal_line();
    cout << "\nAddress:";
    horizontal_line();
    cout << "\nCity, State, Zip:";
    horizontal_line();
    cout << "\nPhone Number:";
    horizontal_line();
    return 0;
}
```

Output of the program with *horizontal_line* function

Name:

Address:

City, State, Zip:

Phone Number:



Return Statement of a Void Function

For functions that return a value

- **return** statement specifies the value returned

For a **void** function

- **return** statement simply ends the function call
- optional (not required)
- need it if you want to return before the end of the function, e.g.:
- **void** pizza_division(**int** number, **double**

```
total) {
```

```
    double portion;  
    if (number == 0)  
    {  
        return;  
    }  
    portion = total/number;  
    cout << "Each one will get "  
        << portion << " of whole pizza.\n";  
} // note - missing return statement
```



Functions and Types of Parameters

TWO kinds of functions:

- functions with types
 - return a function value
- **void** functions
 - do not return a function value

TWO kinds of function parameters:

- **value** parameter: call-by-value
 - actual parameter in a function call can be a variable, an expression, or a constant value
 - only the value of the argument in the call is substituted for the formal parameter
 - the function only *sees* the value of the actual parameter
- **reference** parameter: call-by-reference
 - function references the variable specified in the call, i.e. it can alter the actual variable itself
 - actual parameter in a function call must be a variable



Reference Parameters

Parameter declaration:

-

is of the form "type&" followed by the name of the formal parameter

e.g. `void foo(char& letter,)`
`{ }`

where `letter` is a reference parameter

```
void swap(int& var1, int& var2);
```

Reference
Parameters



Reference Parameter Example

```
void swap(int& x, int& y)
// exchange the current values of the
// two integer parameters
{
    int temp;

    temp = x;
    x = y;
    y = temp;
    return;
}

// somewhere in a calling program

swap(var_1, var_2);
```

The occurrence of the ampersand, **&**, in the parameter list of the function definition establishes that

- the parameters are called by reference
- the parameter variables **x** and **y** in the procedure swap, when it is executing, will be the variables **var_1** and **var_2** of the calling program, **NOT** their values



Call-by-Reference Parameters

```
#include <iostream.h>
void get_numbers(int& input1, int& input2);
void swap_values(int& var1, int& var2);

int main()
{
    int first, second;
    get_numbers(first, second);
    swap_values(first, second);
    cout << "In reverse order the numbers are"
         << first << " " << second << endl;
    return 0;
}

void get_numbers(int& input1, int& input2)
{
    cout << "Enter two integers: ";
    cin >> input1;
        >> input2;
}

void swap_values(int& var1, int& var2)
{
    int temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```



Another Reference Parameter Example

Suppose we want a function that will square the values of two int variables.

e.g. we have:

```
int main()
{
    int one, two;
    .....
    square_them(one, two);
    .....
    return 0;
}

-----

void square_them (int& p1, int& p2)
{
    p1 = p1 * p1;
    p2 = p2 * p2;

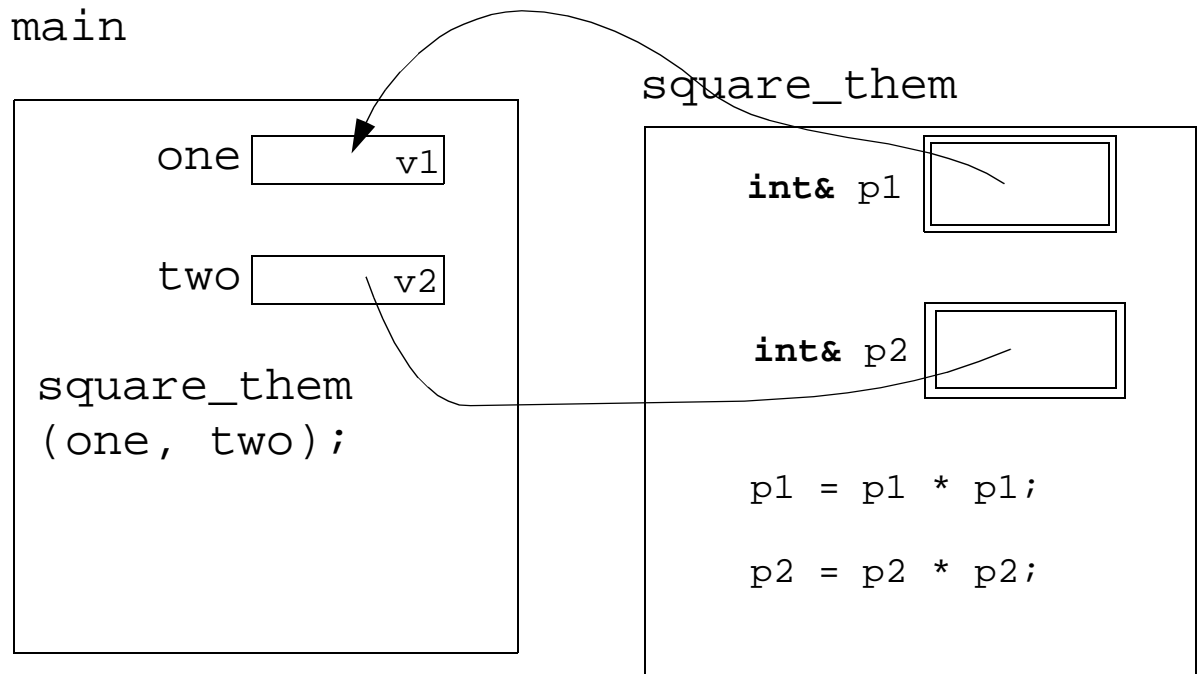
    //      no need for a return
    //      because it is a void function

}
```

(PICTURE)



Picture showing Reference Variables



Note that `p1` and `p2` don't actually exist as data storage locations - they simply refer to the places where the data values can be found.



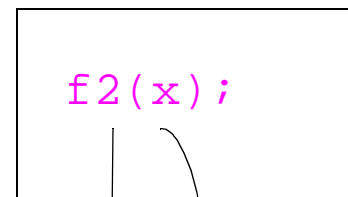
Functions Calling Functions

A function can call another function:

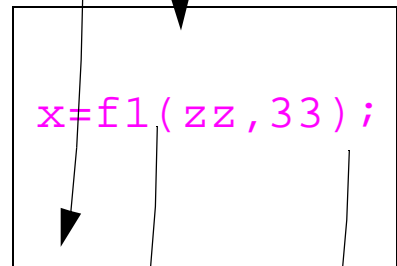
- We already know this: `int main()` can contain a function call - see all the examples of the past weeks!

```
int f1(double x, int y);
void f2(int& x);
int main()
{
    int x = 1;
    double y = 4.5;
    f2(x);
    return 0;
}
void f2(int& x)
{
    double zz=77.7;
    x = f1(zz, 33);
    return;
}
```

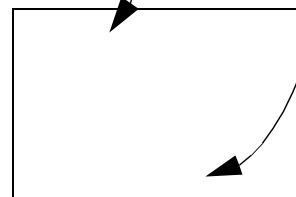
main



f2



f1





Problem Solving

- Is an art
not a science
- You must teach yourselves
we can only give guidelines
- It requires practice
like any art form
- The following list of "guidelines" are valid
they have been accumulated through years of experience
they will sound like platitudes
until you incorporate them into your own problem
solving skills



Problem Solving

- One big problem is much harder to solve than many little problems!
- Understand the problem: WHAT is wanted

write *on paper* WHAT results are expected from the program

WHAT functions (procedures) will be required to generate these results

WHAT parameters will these functions require

(write the function headers?)

write the function descriptions before you consider even a single line of code

these describe WHAT the function does (should do?)

- **None of this requires any knowledge of C++!**



Writing C++ Code

- C++ syntax obscures understanding!

write out your solutions intuitively — on paper

flow charts

pseudocode

translate to C++ syntax — on paper

- NEVER compose code at the terminal
- Very difficult to create a perfect solution (to any problem) on the first try!

begin with a partial solution that runs, then make it better
iterative refinement/enhancement



Testing and Debugging C++ Code

- Syntax errors

these are found for you by the compiler

syntax errors often cascade; correcting the first error may eliminate several following errors

correction through guessing seldom works — it does waste time

the actual error often occurs in the preceding line

- Logical errors

are *hard* to find

- Test code!

look at the output: is it what you wanted?

does it really work?

--- debugging: how to find a bug

- Key points

display intermediate results (*debug printout*)

test each function separately (*stubs* and *drivers*)