

The role of functions

So far, all of the code examples have been fragments inserted into the main program. Some large C++ applications have 100,000s of lines of code, so clearly there needs to be a way of breaking the code down into small modules that can be independently tested and then assembled into the larger system. In a sense a function is a kind of contract, where the input and output behavior of the function is guaranteed according to a set of requirements. This approach to design makes it possible to use the function without concern for the internal details.

An example of a function definition in C++ is as follows.

```
unsigned factorial(unsigned n) { //Definition of the function
    unsigned result = 1;
    for(unsigned i=1; i<=n; ++i)
        result *= i;
    return result;
}

int main() {
    unsigned fact = factorial(5); // "calling" the function
}
```

There are a number of aspects to notice about this example. First is the form of declaring the function. The first element is a data type that specifies the type of value that will be returned by the function. In this case the return type is **unsigned**.

The next element is the name of the function, in this case **factorial**. It is good programming practice to give a function a name that provides insight into the computation performed by the function. Some programmers like to use very short function names, e.g.

unsigned f(unsigned n).

However, this approach leads to code that is difficult to read and understand. There is little or no penalty against using descriptive function names.

The arguments that are provided to the function are a comma-separated list of type declarations enclosed in the **()** expression. In this case there is only one declaration, **unsigned n**. The function is computed inside a scope as indicated by the **{}** pair. At any point in the function computation, a value may be output by the **return** statement.

The result of the **return** statement is that the value of the function is assigned to the left side of the equation in the main program. In this case the variable **fact** receives the

value 120. When the function is encountered in the main program, the flow of execution is transferred to the first statement within the function scope. After the **return**, the flow returns to the statement calling the function.

A more complicated example follows.

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int a = 5, b = 6;
    swap(&a, &b);
    cout << "a " << a << "  b " << b << endl;
}
```

The output is

a 6 b 5

In this case the function is not meant to return a value. Indeed, if a **return** is called inside the **swap** the function will be exited at that point and no value will be returned. However, a compiler error will result in any attempt to return any other type such as **int**. In C++, the fact that a function is not meant to return a value is indicated by the **void** return type. The compiler interprets this type as nothing to return.

In this example, the function **swap** uses its arguments as both inputs and outputs. The values of **a** and **b** can be changed because the function is passed pointers to the variables. Inside **swap**, the pointers are dereferenced to get at the values of **a** and **b** in order to swap them.

Suppose the program is changed as follows.

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int a = 5, b = 6;
    swap(a, b);
    cout << "a " << a << "  b " << b << endl;
}
```

The output of the program is now

a 5 b 6

The variables are no longer being swapped! However if the values of **a** and **b** inside the scope of the swap function are examined after the last statement but before the closing **}** bracket, they will be **a==6** and **b==5**. This fact can be verified by examining the values in a debugger. This behavior often leads to mysterious bugs when programming with functions that are difficult to detect.

What is happening is that the inputs to a function are copied when assigned to the arguments of the function. To make this more clear, consider a slightly changed main program.

```
int main() {  
    int u = 5, w = 6;  
    swap(u, w);  
    cout << "u " << u << "   w " << w << endl;  
}
```

The inputs, **a** and **b** of the **swap** function are assigned the values of **u** and **w** but are not themselves the actual variables, **u** and **w**. The internal swap variables **a** and **b** are at different memory locations and represent completely independent declarations. The confusion arises when the variable names outside the scope of the function are the same as those inside. The programmer easily forgets that there is no connection between them. This problem provides another reason to use descriptive variable and function names.

The reason that the first version of the **swap** function with pointer type arguments was successful is that it was handed the memory locations for the variables outside its scope and so could access and mutate their values.

The reference

The first form of **swap** using pointers is classic C-style programming. It has several advantages: 1) The variable inputs to a function are not copied in order to assign them to an argument; 2) arguments can act as both inputs and outputs.

However, passing pointers around between functions can be very dangerous. For example, a function could call **delete** on a pointer passed into it from another function. The calling function would then have a mess on its hands. On the other hand, if someone doesn't delete the pointer then a memory leak will be incurred.

The C++ language provides an alternative between fully copying a variable's value to assign it to a function argument and passing the pointer to the variables address. This alternative is called the *reference* and is indicated by the **&** symbol. To illustrate, again consider the **swap** function, this time recoded to use reference arguments.

```

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int u = 5, w = 6;
    swap(u, w);
    cout << "u " << u << "  w " << w << endl;
}

```

The output of the program will now be,
a 6 b 5

The declaration `int& a` indicates that the argument has direct access to the value of `a`, in this case the variable `u`, but without having to copy `u`. Indeed, the reference symbol `a` can be thought of as just an alias for `u`, where any value assigned to `a` will also be assigned to `u`. The advantage of the reference is that local variables can be passed safely without having to worry about accidental attempts to delete them, as might be the case when passed as pointers.

To make the alias property a little more obvious, consider the following code.

```

int i = 10;
int& j = i;
j = 7;
// at this point i == 7;

```



Once `j` is declared to be a reference to `i`, both variables are just different names for the same value.

Note that it is nonsense to declare a reference without providing a value. For example,

```
int& j;
```

will cause a compiler error, since `j` does not refer to anything.

Another problem arises in attempting to assign a constant to a reference. For example,

```
int& j=1;
```

causes the following compiler error,

Error 1 error C2440: 'initializing': cannot convert from 'int' to 'int &'

In order to understand this problem, it is necessary to understand the difference between a constant expression and a variable. A constant is not something that one can take the address of. It is only defined on the *right hand side* of an expression. A variable is on the *left hand side* of an expression and is assigned a memory location that can be accessed. Thus it is nonsensical for a reference to be an alias for a constant value. If this were possible then the constant could be changed, which is a contradiction. One might want the compiler error to be more directly descriptive of the problem, but that is always the case.

The **const** keyword

C++ defines the **const** keyword in order to allow constants to be assigned to references, and otherwise to insure that the value of a reference is not changed when the programmer wants to prevent called functions from changing it. For the previous example the revised statement,

```
const int& j=1;
```

passes through the compiler with ease. The **const** insures that **j** can never be changed. Indeed any attempt to do so will cause a new compiler error. For example,

```
const int& j=1;  
j = 5;
```

will produce the following compilation error.

Error 1 error C3892: 'j' : you cannot assign to a variable that is const

The **const** declaration is often used to indicate the type of input and output arguments of a function. For example,

```
#include <math.h>  
bool safe_sqrt(const double& x, double& y) {  
    if(x<0) {  
        y=0;  
        return false;  
    }  
    y = sqrt(x);  
    return true;  
}
```

In this program, the input **x** is tested to see if it is negative. If so, the function evaluation is invalid as indicated by returning the **bool** result **false**. It is clear from the **const** declaration of the first function argument that **x** cannot be changed within the function. The output, **y**, is just an alias for the variable assigned to that argument by the calling program.

Consider the following examples of calls to the function.

```
int main() {
    double x1 = 3;
    double y1;
    bool good = safe_sqrt(x1, y1); //A proper call to the function
    good = safe_sqrt(x1, 5); //will cause a compiler error
    safe_sqrt(4, y1); //ok
}
```

In the first call to **safe_sqrt**, the variable **y1** is a *left side* quantity that can be assigned a reference, and the function successfully competes with **y1** taking on the value of 3 . The second call attempts to assign a *right side constant quantity to a non-const reference*. A compiler error will result. The last call is ok since the first argument of **safe_sqrt** is a **const** reference. Note that the return value of a function can be ignored if desired. The last call to **safe_sqrt** does not assign the return value to anything.

More complexities of const

There are many subtle aspects of using the **const** key word. However the benefits are great, since the compiler can help detect programming errors when functions attempt to violate **const** correctness. The best way to master the issues with **const** is to review a series of examples of increasing complexity.

```
double const& x = 1.5;
const double& y = 1.5;
```

Is there any difference in these two declarations? The effect is the same that neither **x** or **y** can be changed. There will be no detectable difference in the behavior of **x** and **y**, both are references to **const double** variables.

It is possible to declare a pointer to be **const**. Consider the difference in the following two statements.

```
const int* bp;
int* const cp;
```

The best way to read such declarations is backwards. Reading the first example from right to left, **bp** is a pointer to a **const int**. In this case, the pointer can be changed to point to something else, but it can't be dereferenced to change the value of the data being pointed to. Reading the second example backwards, **cp** is a **const** pointer to a non-**const int**. Here the value of the integer can be changed, but the address corresponding to the pointer **cp** is fixed. Any attempt to change the pointer address will result in a compiler error.

Just to show how perverse the syntax of **const** can be, consider the following.

```
const int* v1;
int const * v1;
```

These statements produce identical results for the type of **v1**. Both of these read that **v1** is a pointer to a **const int**. Perhaps a way to reconcile this seemingly arbitrary ordering is again to read from right to left. In the first statement we see a ***** pointer declaration that is to a **const int**, so far so good. The second case reads, a pointer to an **int** data type that is **const**, i.e., pretty much the same thing. It can be argued that the last form is more consistent since **const** occurs before **int**, reading backwards.

Ratcheting up the complexity it is also possible to declare a **const** pointer to a **const** variable type.

```
const int* const v2;
int const * const v2;
```

These statements produce the same result. Now neither the pointer address location or the value pointed at can be changed. Again the second form might be considered more consistent with right to left interpretation, but the first form is used almost exclusively.

The use of **const** is watched closely by the compiler. Consider the following sequence of assignments

```
const int v = 1;
int* w = &v;
```

Here the attempt to assign the pointer to a **const int** to an **int** in the second line produces the following compiler error.

*Error 1 error C2440: 'initializing' : cannot convert from 'const int *' to 'int *'*
If the second statement were allowed then the constant value of **v** could be changed.

The const_cast statement

Sometimes it is necessary to explicitly violate the **const** nature of a variable. The programmer after all is the boss, not the compiler. The compiler helps the programmer realize that mistakes have been made, but **const** can be overruled if necessary. The appropriate approach is the **const_cast** statement.

The removal of **const** restrictions is illustrated by the following example.

```
int j = 3;
const int* v1=&j;
int* v2 = const_cast<int*>(v1);
*v2 = 5;
```



At the end of this sequence of assignments, **j** will have the value 5, and no compiler errors will occur. Note that the **const_cast** expression only applies to pointer or reference types, and to cast away **const**.

Recursive functions

C++ supports recursive function calls, i.e., functions that call themselves. The classic example of this style of programming is the **factorial** function encountered in previous lectures. Here is a recursive form of the function.

```
unsigned int factorial(int n) {  
    if (n>=1)  
        return n * factorial(n-1);  
    else  
        return 1;  
}
```

Note that the function is called recursively until the value of **n==0**. This is the base state where the answer is trivial.


The general recursion approach to programming proceeds by defining a base state and then what considers what happens in general when the state is reduced by one recursion level. In the example, the general case is to compute factorial of **n** by multiplying **n** by **factorial** of **n-1**. Ultimately, factorial of 0 is defined as 1 and so the recursion can terminate at this base state.

Another example of the same recursive pattern is the following function to measure the length of a string.

```
unsigned length(char *s) {  
    if (*s == '\0') return 0;  
    else return(1 + length(s+1));  
}
```

Here the base state is ***s == '\0'** and the recursion step is moving the pointer one step forward in the string and adding 1 to the length.

A drawback of recursive programming is that the code is more difficult to understand and debug. For example, the factorial example has the following call stack for **n=5**.

	lec5.exe!factorial2(int n=0) Line 45
	lec5.exe!factorial2(int n=1) Line 43 + 0xc bytes
	lec5.exe!factorial2(int n=2) Line 43 + 0xc bytes
	lec5.exe!factorial2(int n=3) Line 43 + 0xc bytes
	lec5.exe!factorial2(int n=4) Line 43 + 0xc bytes
	lec5.exe!factorial2(int n=5) Line 43 + 0xc bytes
	lec5.exe!main() Line 79 + 0x7 bytes

So the function passes through itself repeatedly with decreasing values of **n**. A subtle problem could arise at some level of recursion, but it is not obvious how to determine which level, since each call is of identical form. Also space is taken up on the stack with the local variables of the function which are re-allocated on each call, and the stack could overflow with large problems. With straight iteration the local variables are removed and replaced each time through the loop.

Nevertheless, there are some programming tasks that are difficult to implement without using recursion, particularly in dealing with graph and tree data structures. For this reason, many programming solutions to problems in artificial intelligence involve recursion. Consider the following example,

```
class tree_cell {
public:
    tree_cell* left_child_;
    tree_cell* right_child_;
};

void node_count(tree_cell* node, unsigned& count) {
    count++;
    if((*node).left_child_)
        node_count((*node).left_child_, count);
    if((*node).right_child_)
        node_count((*node).right_child_, count);
}

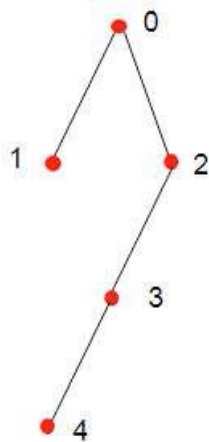
int main() {
    tree_cell tree[5];
    tree[0].left_child_ = tree + 1;
    tree[0].right_child_ = tree + 2;
    tree[1].left_child_ = 0;
    tree[1].right_child_ = 0;
    tree[2].left_child_ = tree + 3;
    tree[2].right_child_ = 0;
    tree[3].left_child_ = tree + 4;
    tree[3].right_child_ = 0;
    tree[4].left_child_ = 0;
    tree[4].right_child_ = 0;
    unsigned count = 0;
    node_count(tree, count);
    cout << " count = " << count << endl;
}
```



The tree structure is shown in Figure 1, along with the memory layout of the **tree** array. In this example the main program prints out,

```
count = 5
```

which is correct, since there are five nodes in the tree. The function **node_count** is called recursively as the tree is traversed in a depth first manner. Note that the programming style is neat and uncomplicated, regardless of the tree structure.



tree	value
tree	0x00d9ff3c {left_child_=0x00d9ff44 right_child_=0x00d9ff4c }
[0]	{left_child_=0x00d9ff44 right_child_=0x00d9ff4c }
[1]	{left_child_=0x00000000 right_child_=0x00000000 }
[2]	{left_child_=0x00d9ff54 right_child_=0x00000000 }
[3]	{left_child_=0x00d9ff5c right_child_=0x00000000 }
[4]	{left_child_=0x00000000 right_child_=0x00000000 }

Figure 1 The tree data structure.

static variables

Sometimes the programmer wants the value of a variable inside the scope of a function to persist between function calls. A typical reason is that something special should happen the first time a function is called. For example, a function might be designed to keep track of how many times it is called to aid in determining bottlenecks or inefficiencies in programming. The following is a function that counts its accesses.



```
unsigned call_me() {
    static unsigned count = 0;
    return ++count;
}
```

Each time this function is called the return value will increase by one. As seen, a static variable can be initialized at compile time. That value can be changed by the function and persist across repeated function calls, even though the function exits the scope of the variable.

Another example of the use of the static variable within a function is to avoid recomputing something that is expensive to compute each time a function is called. For example,

```
//this function will be called only once
double* compute_pi_array(double* adr, const unsigned & max_n) {
    double pi = 3.141593;
    for(unsigned i=0; i<max_n; ++i)
        adr[i] = pi/(i+1);
    return adr;
}
```

```
double pi_over_n(const unsigned& n) {
    static double parray[1000];
    //static variables are only initialized once on the first call
    static double* adr = compute_pi_array(&parray[0], 1000);
    if(n==0)
        return 0;
    else if (n>1000)
        return 3.141593/n;
    else
        return parray[n-1];
}
```

The goal of the second function is to divide a 360 degree pie up into integral slices. The programmer plans to call this function many times and decides that it will be faster to do a table lookup than compute the double precision division. The programmer also knows that the most popular number of pie slices is no greater than 1000. The problem is how to avoid recomputing the array of integral pi fractions. One solution is to use two **static** variables, the array of pi fractions, **parray**, and the address of the beginning of this array **adr**.

Partitioning function definitions and implementations into files

So far in the examples it has been possible to do everything in the main program. This approach will quickly get out of hand. In order to keep the size of file manageable and meaningful the program is broken up into files with extension **.h** and **.cxx** (or **.cpp**). The **.h** file usually contains the definition of the interface of the function, i.e. return type and argument types. The **.cxx** file usually contains the actual implementation of the function. Implementation is typically kept out of the **.h** file so that the compiler doesn't have to copy all the implementation code into each other file that includes the **.h** file.

As an illustration, consider the previous example for computing fractions of pi. The two functions will be declared and implemented in **a separate functions.h and functions.cxx file**, and called from the main program.

The contents of the **functions.h** file is,

```
#ifndef functions_h_
#define functions_h_

extern double* compute_pi_array(double* , const unsigned & );

extern double pi_over_n(const unsigned& );

#endif // functions_h_
```

There are several new things to notice about this file. The first two lines of the file define a *guard*, which prevents a .h file from being loaded every time another file includes a file that includes functions.h. The guard is accomplished by checking if a variable, **functions_h_** is defined. If it is not defined then the code up to the closing **#endif** is copied into the file that includes **function.h**. If the variable **functions_h_** is already defined, i.e. the file has already been included, then the copying will be skipped and no action is taken. Note that statements beginning with **#** are *compiler directives*. These statements are not C++ code but are used by the compiler to include or modify code.

The next new key word is **extern**, which is essentially a promise to the compiler that the following function declaration will be honored by the programmer and the linker will be able to locate the function somewhere else in the code. When declaring functions or variables in a .h file, C++ assumes that they are declared **extern** by default¹. Note that declaration of the function does not need to have variable names in order to be defined. Indeed, the names are only symbols that represent the values of the defined types. From the function's point of view, only the argument types matter. However, it is good programming practice to specify meaningful variable names to help other programmers (and yourself) to understand your code.

Important note – In C++, if two functions have the same name and argument signatures (declared types) but different return types, then they are considered the same function. Any attempt to declare both will lead to a compiler error. So for example,

```
extern int round(double);  
extern short round(double);
```

are seen as the same function. One way to see why they must be treated the same is when they could be called without assigning the return to a variable. The following statement would be ambiguous if both definitions above are allowed.

```
round(2.1);
```

-----End Note-----

The implementation of the functions defined in **functions.h** can be separated away in an appropriately named **functions.cxx** file. The contents of this file is below.

```
#include "functions.h"  
double* compute_pi_array(double* adr, const unsigned & max_n) {  
    double pi = 3.141593;  
    for(unsigned i=0; i<max_n; ++i)  
        adr[i] = pi/(i+1);  
    return adr;  
}
```

¹ **extern** is used mainly when linking to old C implementations of a function, that are called from C++. In this case the declaration that a function's implementation is in C is indicated by **extern "C"**.

```
double pi_over_n(const unsigned& n) {
    static double parray[1000];
    static double* adr = compute_pi_array(&parray[0], 1000);
    if(n==0)
        return 0;
    else if(n>1000)
        return 3.141593/n;
    else
        return parray[n-1];
}
```

Note that `functions.h` has to be included at the top of the `.cxx` file. In order for other parts of the program to access these functions, such as `main`, it is only necessary to include the `functions.h` file, which declares their signatures. For example the main program that uses these functions is defined in a file `function_main.cxx` as below.

```
#include "functions.h"

int main() {
    double p3 = pi_over_n(3);
    double p4 = pi_over_n(4);
    double pgt1000 = pi_over_n(1001);
    double peq0 = pi_over_n(0);
}
```

The linker is smart enough to realize that the functions used in the main program and declared in `functions.h` are implemented in `functions.cxx`. This example requires a slight increase in understanding CMake expressions in order to build the executable. The appropriate `CMakeLists.txt` file is shown below.

```
SET(my_sources
    functions.cxx    functions.h
    function_main.cxx
)
ADD_EXECUTABLE( ftest ${my_sources} )
```

The first statement is assigning a variable, `my_sources`, to a list of file names. These files are essential to build the executable. The second statement is similar to that used for the simple “hello world” example. The name of the executable is `ftest.exe` on windows and `ftest` on Unix. The syntax `${my_sources}` represents list of file names bound to the variable. The `SET` statement can be eliminated if the second statement is written out as,

```
ADD_EXECUTABLE(ftest functions.cxx functions.h function_main.cxx)
```

, but it is much neater to keep the source files in a separate CMake variable.

File scope and **static**

Suppose that it is desired to share the value of **pi** across the previous two routines as shown below.

```
double pi = 3.141593;

double* compute_pi_array(double* adr, const unsigned & max_n) {
    for(unsigned i=0; i<max_n; ++i)
        adr[i] = pi/(i+1);
    return adr;
}

double pi_over_n(const unsigned& n) {
    static double parray[1000];
    static double* adr = compute_pi_array(&parray[0], 1000);
    if(n==0)
        return 0;
    else if(n>1000)
        return pi/n;
    else
        return parray[n-1];
}
```

This declaration can be a problem since it is often the case that programmers will tend to use common names for constants (or functions) and the danger of a clash is very high, such as this example. Suppose that there is another file that declares the official value of pi, called `math_constants.h`

```
extern double pi;
```

and given a value in `math_constants.cxx`.

```
double pi = 3.14159265358979323846;
```

This declaration will trigger a linker error as follows.

```
math_constants.obj : error LNK2005: "double pi" (?pi@@3NA)
already defined in functions.obj
2>Debug\ftest.exe : fatal error LNK1169: one or more multiply
defined symbols found
```

It is necessary to use the key word **static** in order to restrict the local definition of pi to within the file `functions.cxx`. The local declaration there should be changed to

```
static double pi = 3.141593;
```

Now, this value of the variable name **pi** is invisible elsewhere outside the file. Note that the **static** declaration can be applied to function declarations as well. It is always sound programming to use the **static** declaration for local constants and functions that you don't want to be visible outside the file in which they are being used².

Default Arguments

It is often the case that a function will be designed with some degree of flexibility, but one doesn't want to make the programmer specify all the arguments of a function for common cases that occur 99% of the time. For example consider a function for testing if two quantities are nearly equal.

```
bool near_equal(double x, double y, double tolerance=1.0e-10);
```

The arguments that are to be always specified come first in the signature declaration. The default arguments are at the end. The function can be called as follows.

```
if( near_equal(x, y) ) {  
...  
}
```

The function will be passed the value **1.0e-10** for the tolerance. Note that the specification of default arguments has to be contiguous. For example, given the definition of a function,

```
void f(int a, int b, int c=1, int d = 2, int e =3);
```

the following call will result in **z** being interpreted as a value for **c** and **w** as a value for argument **d**.

```
int x=0, y=5, z=10, w = 13;  
f(x, y, z, w);
```

² Note that C++ also supports the concept of "namespaces." File scope can also be achieved by an "anonymous" namespace designation. Discussion of this advanced topic will be deferred for now.