Functions

Function Definition

The definition of a function in C++ follows

⟨*return type*⟩
⟨*function name*⟩ (⟨*argument list*⟩)
⟨*block*⟩

When a function does not return a result, e.g. a procedure, then the return type is `void`.
To return a value from a function, C++ provides the keyword **return**.

```
double
square ( const double x )
{
    return x*x;
}
```

Function Call

A function is called by

⟨*function name*⟩ (⟨*argument1*⟩, ⟨*argument2*⟩, . . .);

Example:

```
double y = square( 4.3 );
```

One can not define a function in the body of another function:

```
double cube ( const double  x )
{
    // ERROR
    double square ( const double  y ) { return y*y; }

    return square( x ) * x;
}
```

Function Examples

Previous computation of factorial in functional form:

```
int
factorial ( const int  n )
{
    int  f = 1;

    for ( int i = 1; i <= n; i++ )
        f *= i;

    return f;
}
```

Coding Principle No. 10

Make all function arguments **const**, *except when changing value (see later).*

Function Examples (Cont.)

Power function with positive integer exponents:

```
double
power ( const double  x, const unsigned int  n )
{
    switch ( n )
    {
        case 0 : return 1;
        case 1 : return x;
        case 2 : return square( x );
        default:
        {
            double  f = x;
            for ( int i = 0; i < n; i++ ) f *= x;
            return f;
}   }   }
```

Coding Principle No. 11

Make sure, that a function has a call to **return** *in every execution path.*

Main Function

The `main` function is the first function called by the operating system in your program. Every program must have exactly one main function.
In principle, only code in `main` and functions called directly or indirectly from `main` will be executed.
The main function may be implemented without arguments and has a return type of `int`:

```
int
main ()
{
    ... // actual program code
    return 0;
}
```

The value returned from `main` is supplied to the operating system. As a standard, a value of 0 signals no error during program execution.

Call by Value

In previous examples, only the *value* of a variable (or constant) is used as the argument of a function, e.g. changing the value of the argument does not change the value of the original variable:

```
int
f ( int m )       // non const argument!
{
    m = 4;        // explicitly changing the value of argument m
    return m;
}

int  m = 5;
int  n = f( m ); // m is unchanged by f
```

This is known as call-by-value.

Remark

It is nevertheless advised to use **const** *arguments.*

Call by Reference

If the original variable should be changed in a function, a pointer or reference to this variable has to be supplied:

```
int
f ( int & m )     // reference argument
{
    m = 4;        // changing m, changes the variable pointed to
    return m;
}

int  m = 5;
int  n = f( m ); // m is changed by f to 4
```

This is known as call-by-reference.

Call by Reference

The same function with pointers:

```
int
f ( int * m )       // reference argument
{
    *m = 4;         // changing m, changes the variable pointed to
    return *m;
}

int  m = 5;
int  n = f( & m ); // m is changed by f to 4
```

Call by Reference

When using references to *constant* variables, the value can not be changed:

```
int
f ( const int & m )
{
    m = 4;           // ERROR: m is constant
    return m;
}
```

Therefore, this is (almost) equivalent to call-by-value and needed for advanced datatypes (see later).
For basic datatypes, using call-by-reference, even with **const**, is usually not advisable, except when changing the original variable.

Call by Reference

Example for multiple return values

```
void
min_max ( const int n1, const int n2, const int n3,
          int & min, int & max )
{
    if ( n1 < n2 )
        if ( n1 < n3 )
        {
            min = n1;

            if ( n2 < n3 ) max = n3;
            else           max = n2;
        }
        else
        {
            min = n3;
            max = n2;
        }
    else
        ...
}
```

Recursion

Calling the same function from inside the function body, e.g. a *recursive* function call, is allowed in C++:

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

Remark

The recursion depth, i.e. the number of recursive calls, is limited by the size of the stack, *a special part of the memory. In practise however, this should be of no concern.*

Recursion

It is also possible to perform recursive calls multiple times in a function:

```
unsigned int
fibonacci ( const unsigned int  n )
{
    unsigned int  retval;

    if ( n <= 1 ) retval = n;
    else          retval = fibonacci( n-1 ) +
                           fibonacci( n-2 );

    return retval;
}
```

Remark

Remember, that variables belong to a specific block and each function call has it's own block. Therefore, variables, e.g. `retval`*, are specific to a specific function call.*

Function Naming

A function in C++ is identified by it's name and the number and type of it's arguments. Hence, the same name can be used for different argument types:

```
int      square ( const int    x ) { return x*x; }
float    square ( const float  x ) { return x*x; }
double   square ( const double x ) { return x*x; }
```

Coding Principle No. 12

Functions implementing the same algorithm on different types should be named equal.

This can significantly reduce the number of different functions you'll have to remember und simplifies programming.

Function Naming

If only the return type is different between functions, they are identified as equal:

```
float  f ( int x ) { ... }
double f ( int x ) { ... }  // ERROR: "f" already defined
```

Default Arguments

Arguments for a function can have default arguments, which then can be omitted at calling the function:

```
void
f ( int n, int m = 10 )
{
    // ...
}

{
    f( 5 );     // equivalent to f( 5, 10 )
    f( 5, 8 );
}
```

Only limitation: after the first default value, all arguments must have default values:

```
void g1 ( int n, int m = 10, int k );       // ERROR
void g2 ( int n, int m = 10, int k = 20 ); // Ok
```

Default Arguments and Function Names

Two functions with the same name must differ by their arguments without default values:

```
void f ( int n1, int n2, int n3 = 1 ) { ... }
void f ( int n1, int n2 )             { ... }

...

{
    f( 1, 2, 3 );   // Ok: call to f(int, int, int)
    f( 1, 2 );      // Error: call of "f(int, int)" is ambiguous
}
```

Function Name Scope

A function can only be called, if it was previously implemented:

```
void f ( int x )
{
    g( x );          // ERROR: function "g" unknown
}

void g ( int y )
{
    f( y );          // Ok: function "f" already defined
}
```

or *declared*, i.e. definition of function without function body:

```
void g ( int y );  // forward declaration

void f ( int x )
{
    g();             // Ok: "g" is declared
}
```

This is known as forward declaration.

Function Name Scope

Of course, every function with a forward declaration has to be implemented eventually:

```
void g ( int y );  // forward declaration

void f ( int x )
{
    g();
}

...

void g ( int y )   // implementation
{
    f( y );
}
```

Inline Functions

Calling a function involves some overhead. For small functions, this overhead might exceed the actual computation:

```
double  square ( const double x ) { return x*x; }

{
    double  f = 0;

    for ( int i = 0; i < 100; i++ )
        f += square( double(x) );
}
```

Here, simply calling `square` takes a significant part of the runtime. Some compilers automatically replace the function call by the function body:

```
...
    for ( int i = 0; i < 100; i++ )
        f += double(x) * double(x);
...
```

Inline Functions

Replacing the function call by the function body is called inlining. To help the compiler with such decisions, functions can be marked to be inlined by the keyword **inline**:

```
inline double
square ( const double x )
{
    return x*x;
}
```

Especially for small functions, this often dramatically increases program performance.

Remark

If the function body is too large, inlining can blow up the program since too much code is compiled, e.g. every occurence of the function, and therefore decreases performance!

Function Pointers

A function, like a variable, is stored somewhere in the memory and therefore, also has an address. Hence, a pointer can be aquired for it. For a function

⟨*return type*⟩ ⟨*function name*⟩ (⟨*argument list*⟩);

a pointer is defined by

⟨*return type*⟩ (* ⟨*variable name*⟩) (⟨*argument list*⟩);

Example:

```
int f ( const int  n, int &  r );

{
    int ( * pf ) ( const int n, int & r );  // function ptr named "pf"

    pf = f;
}
```

Function Pointers

A variable holding the address of a function can be used as a function by itself:

```
int  n = 0;

pf = f;       // pf holds address to f
pf( 2, n );   // call to f
```

Since function pointers are normal variables, they can be supplied as function arguments:

```
double  f1 ( const double x )              { return  x*x; }

double  f2 ( double ( * func ) ( const double x ),
             const double x )              { return func( x ); }

int main ()
{
    f2( f1, 2.0 );  // returns f1( 2.0 )
}
```

Function Pointers

Example: apply Simpson rule to various functions

```
double
simpson_quad ( const double a, const double b,
               double ( * func ) ( const double ) )
{
    return (b-a) / 6.0 * ( func(a) +
                           4 * func( (a+b) / 2.0 ) +
                           func(b) );
}

double f1  ( const double  x ) { return x*x; }
double f2  ( const double  x ) { return x*x*x; }

int main ()
{
    cout << simpson_quad( -1, 2, f1 ) << endl;
    cout << simpson_quad( -1, 2, f2 ) << endl;
}
```

Functions and Minimal Evaluation

As discussed, C++ uses minimal evaluation when looking at logical expressions, e.g. only evalutates until results is known. If functions are used in the expressions, it can imply, that they are not called at all:

```
double f ( const double  x ) { ... }

...
    // f is not called if x >= 0.0
    if (( x < 0.0 ) && ( f( x ) > 0.0 ))
    {
        ...
    }
```

For the programmer this means:

Coding Principle No. 13

Never rely on a function call in a logical expression.

Functions and static Variables

In contrast to standard variables in a function, which are specific to a specific function call, for **static** variables in all function calls the same instance, e.g. memory position, is referenced:

```
double
f ( const double x, long & cnt )
{
    static long  counter = 0;     // allocated and initialised
                                  // once per program
    cnt = ++counter;

    return 2.0*x*x - x;
}

int main ()
{
    long  cnt = 0;

    for ( double x = -10; x <= 10.0; x += 0.1 )
        f( x, cnt );

    cout << cnt << endl;          // print number of func. calls
}
```