


QACPROG

Functions

- **Objective**
 - To implement a block-structured design
- **Contents**
 - Function prototypes
 - Calling functions
 - Defining functions
 - Function argument passing
 - The *return* Statement
 - Passing arguments by copy
 - Scope of variables and storage classes
- **Summary**
- **Practical**
 - Function design and implementation



The objective of this chapter is to introduce the most important block-structured mechanism, i.e. the function. A function is a named compound statement that can handle information coming in and going out, as well as having local data of its own.

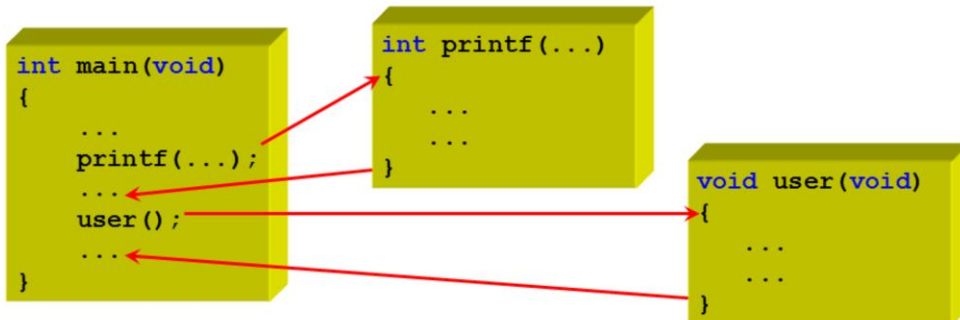
Since the standardisation of C by the ISO committee, function implementation is more robust, since the compiler is able to perform more integrity checks.

This chapter introduces call-by-value communication between functions. Call by reference is covered in the *Pointers and Functions* chapter.

Functions by their very nature define a scope for their data. All argument and local identifiers defined within the head and body of a function are said to have local scope. The issue of lifetime and storage details are another matter.

Program Structure

- **Typical programs consist of numerous small functions**
 - Why?
- **The function called `main()` is compulsory**
 - C program execution starts at `main()` and continues by
 - Calling standard C library functions
 - Calling user-written functions



A C program consists of a set of functions that communicate with each other. So far, we have written programs using the one compulsory function, i.e. `main`. The execution of the program begins with `main` and continues sequentially through `main`'s code statements, which invoke functions as they appear. Unless otherwise indicated, execution ceases at the end of `main`.

In the examples and practical sessions, programs have communicated with at least two library functions, i.e. `printf` and `scanf`. Using these two functions has been fairly intuitive, and communication has been fairly straightforward. Invoking our own functions is achieved by using the same technique, i.e. simply by calling the function by name and by supplying appropriate information within the parentheses.

This chapter concentrates on how we can design, code and use our own functions.

QACPROG

A Simple Function

*Function
Prototype* →

*Function
Call* →

*Function
Definition* {

```
#include <stdio.h>

void cube_of_4(void);

int main(void)
{
    cube_of_4();
    return 0;
}

void cube_of_4(void)
{
    int i, result = 1;
    for (i = 1; i <= 3; i++)
    {
        result *= 4;
    }
    printf("4 to the 3rd power = %d\n", result);
}
```

Before a function is called a function prototype is specified to describe the calling interface of the function

The example shown above is one of the simplest examples we could give. Although it is a little contrived, it indicates clearly what is required, i.e.:

- Declare the function details to the compiler.
- Call the function within a code block.
- Define the function implementation.

The declaration provides the details of your function before the compiler sees the first invocation. Name, input and output details are supplied. ISO C introduced the full version of the declaration and called it a prototype.

The invocation or call of the function is done by naming the function (and supplying appropriate input) within an expression that would be capable of making use of any output from the function.

The function definition is very similar to `main`. It has a name, brackets and a compound statement with data and code statements.

The example shown above has no input and output requirements. The `void` keyword in brackets indicates to the compiler the absence of input. The `void` keyword before the function name indicates the absence of output.

Note: ISO C supports a return type of `int` only for `main`. We have suddenly introduced the return statement in the `main` function, a topic for later on in the chapter.

Passing Multiple Arguments

- `n_to_power_p`
 - a more useful function
 - raise `n` to the power `p`

First call

Second call

```
void n_to_power_p(int n, int p);  
  
int main(void)  
{  
    int x = 2, y = 5;  
    n_to_power_p(4, 3);  
    n_to_power_p(x, y);  
    return 0;  
}  
  
void n_to_power_p(int n, int p)  
{  
    int i, result = 1;  
    for (i = 1; i <= p ; i++)  
    {  
        result *= n;  
    }  
    printf("%d to the power %d = %d\n",  
        n, p, result);  
}
```

*Note the
Prototype*

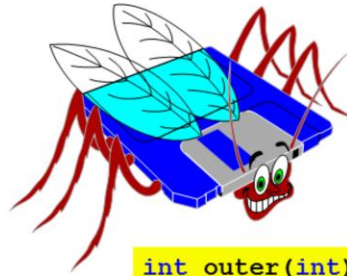
In this example, we concentrate on *input* information. As indicated in the prototype, our function takes two ints as input, but provides no output.

The two calls to the function each supply two `int` values, often referred to as arguments, which will be passed into the function. These could be any expression that yields an integer result.

The function definition holds the code that will do the processing. Its first task is to handle the two ints coming in. It places them, in order, into two local ints called `n` and `p`, which are often referred to as parameters. The process is automatic. `n` and `p` are declared at this point in the code (within the brackets) and are initialised with the values coming in from the call. These values, held in `n` and `p`, can be used within the function to start up the process. Note the presence of two other ints, `i` and `result`. These are temporary local variables that are used to help the process.

More on Functions

- **If function prototypes are incorrect, or not used, C will *not* check**
 - That arguments to functions are in the correct order
 - That arguments to functions have the correct type
 - That the correct number of arguments are passed!
- **Function definitions cannot be nested**
 - But can be defined in any order



```
int outer(int)
{
    int inner(int)
    {
    }
}
```



Function prototyping is essential in a C program

So far, we have seen two examples. These two are all that is necessary to illustrate most of the techniques. The rules can be stated at this point.

Functions cannot be defined within another function. Function definitions are effectively external items; they are the only kind of block that obeys this rule. This means that a function is potentially available to the entire program because it is not hidden within another. We shall see in the *Working with Large Programs* chapter that there are ways of making the use of functions private.

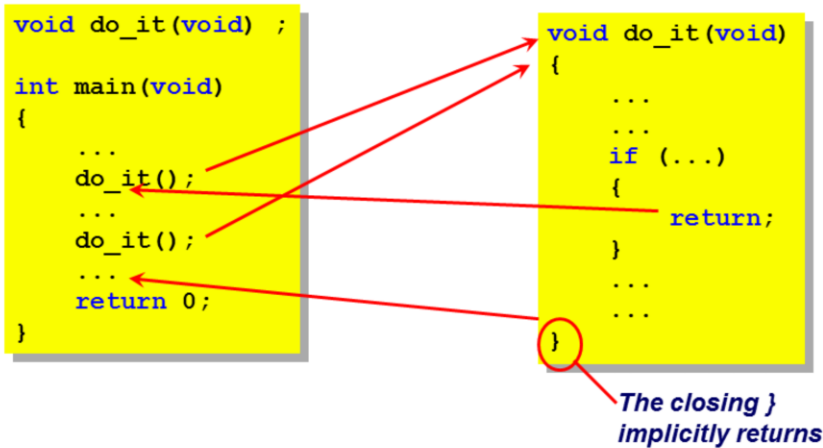
The physical position of a function is immaterial, as long as the compiler sees the prototype and the linker has access to the object file containing the definition.

If prototypes are not used, you pay for it! The full rules on this come later in this chapter.

Output from a function is covered on the next few pages.

return

- **Explicitly terminates execution of the statements in a function and returns control to the calling function**



The `return` statement sends control back to the statement that invoked the function. In the example, the `return` is executed after a decision. This is a usual practice and should not interfere with the natural flow. If there is no `return` statement, control is automatically returned when the closing `}` is reached.

The return from `main` can now be officially described. The return value must be a `int` which is used as the process return code left in the calling process, often the operating system, by the program on completion.

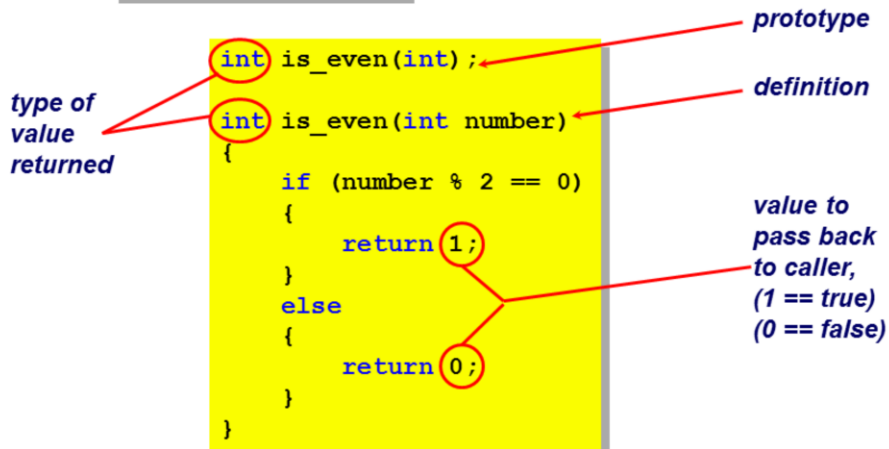
With `void` functions it is good practice to put the `return` statement in, even if it is just before the `}`.

A function can possess as many return statements as it likes. This is very useful if using a `switch` statement.

The `return` Statement

- Functions can return a *value* back to the calling function
 - The general form of the return statement is:

```
return expression;
```



Output information is given in the prototype in the same way as input information, i.e. a type name is used. However, only one item can be returned. The output information is simply a single type name. In the example shown above, it is `int`.

This has been what `main` has been doing when returning 0 (although confusingly returning zero from `main` indicates success).

This is also placed in the function definition heading. The actual value to be returned from the function is placed after the `return` keyword. This value must be the same type as that indicated in both the prototype and the function definition.

In the slide example we return 1 or 0 to signify true or false if the expression in the `if` statement condition evaluates to true or false respectively. However, it is generally considered poor practice to have multiple return statements in a single function. We could refactor the code by introducing an identifier (e.g. `int result`), but in this case it is simpler to just return the value of the conditional expression:

```
return (number % 2 == 0);
```


Exercise: Communication

- Because `is_even()` returns an `int` value, it can be called anywhere in the code where an `int` value is expected

```

/* Prototype */
int is_even(int);

int main(void)
{
    int x, y = 19;
    ...
    x = is_even(y);
    ...
    if (is_even(42))
        printf("Even\n");
    ...
    printf("%d\n", is_even(y+1));
    ...
}

```

Fill in the values in the boxes

What is the value of `x` ?

What do the `printf`'s display ?

```

/* Definition */
int is_even(int num)
{
    if (num % 2 == 0)
        return 1;
    else
        return 0 ;
}

```

A function with a `return` value must be called in the appropriate way if that value is to be used. It means that the function should be called from within an expression or subexpression. For example, a financial program has a function that returns the current interest for its different accounts. Its prototype is:

```

int rate(char);      /* account type is character coded,      */
                      /* rate returned in hundredths of % */

```

Examples of invocation could be:

```

interest *= rate('v');
...
if (rate(ch) > 1125)
    ...
bonus = base_val + calc_bonus(rate(ch), years_emp);
...

```

Note: `calc_bonus` has the prototype:

```

int calc_bonus(int, int);

```

Solution:

From top to bottom, the boxes should contain 19, 0, 42, 1, 20, 1.
The value of `x` is zero. The `printf`'s display Even and then 1.

Function Prototypes - A Closer Look

- Function prototypes have the following form...

```
return_type name ( parameter_list );
```

- If `parameter_list` is omitted, no argument type checking is performed and default argument conversions occur!

```
int vague();
```

```
int precise(void);
```

Use `void` to indicate
'no parameters required'

- If `return_type` is omitted, `int` is assumed



`explicit` is a
C++ keyword

```
/*int*/ implicit(void);
```

```
void explicit(void);
```

Use `void` to indicate
'no value returned'

The three parts to the prototype are:

Output Information	Name	Input Information
--------------------	------	-------------------

These should all be correct, if you want the compiler to be on your side!

The full story is this:

If there is no input information, i.e. a *blank* (not `void`), then the compiler will not perform any integrity checks on arguments. Automatic conversion will take place, i.e. the arguments will get converted to the parameter types as defined in the *Expressions* chapter.

If there is input information and the number of arguments match, the values in the call are used to initialise the corresponding parameters.

If the number of arguments supplied does not match the parameters, the effect is undefined.

If there is no output information, it is as if you have placed `int` as the return value.

If there is no prototype at all, you can assume that the compiler will treat the function as:

```
int fname();
```

Function Prototypes - Some Examples

```
/* No parameters, float returned */  
float random_number(void);  
  
/* No parameters, nothing returned */  
void clear_screen(void);  
  
/* Three int parameters, long returned */  
long mins_sofar(int, int, int);  
  
/* identifiers may be included */  
long secs_sofar(int day, int month, int year);  
  
/* Beware - assumes int return types ! */  
sqr_root(double value);  
  
/* Beware - no argument checking ! */  
float add_2_floats();
```



The only extra information supplied here is that it is permissible to supply dummy argument names in the prototype. As long as they are legal identifiers, the compiler will ignore them.

It may be worthwhile to adopt a style that uses the names for documentation purposes. This is encouraged if the function uses a type more than once. The user of the function will then get a clear idea of the order of values. In the `secs_so_far` function, it is clearly indicated in which order the `day`, `month` and `year` values are to be entered.

QACPROG

Writing Functions

Prototypes

1

Function Calls

2

Function Definitions


3

```
long calc_total(int, long);
void draw(char);
void do_it(void);

int main(void)
{
    long lnum;
    lnum = calc_total(5, 125L);
    draw('x');
    ...
}

long calc_total(int rate, long num)
{
    ...
}

void draw(char c)
{
    ...
}
```



Our advice is as follows:

Declare

Call

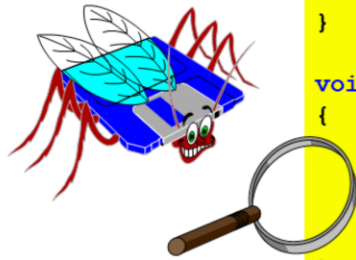
Define

specify the input and output information precisely, and you will not have communication problems.

Does This swap() Function Work?

*What is output
at each of the
printf statements ?*

*What does
this imply?*



```
void swap(int, int);  
...  
int main(void)  
{  
    int x = 7, y = 5;  
    swap(x, y);  
    printf("x now = %d\n", x);  
    return 0;  
}  
  
void swap(int lhs, int rhs)  
{  
    int temp = lhs;  
    lhs = rhs;  
    rhs = temp;  
    printf("lhs now = %d\n", lhs);  
}
```

The program indicates the possible restrictions on the method used to input information into a function. The input information is passed by value, i.e. each item is a 'photocopy' of the original. The parameters lhs and rhs are given copies; they are not handles to the original data items x and y. This is why the swap function does not work.

Copied Parameters

- **Function arguments are passed by copy**
 - The called function receives a private temporary copy of each argument (not its address)
 - It cannot alter the original argument in the calling function
- **'Pointers' are needed to achieve 'call by reference' in C**
 - A later chapter discusses this topic

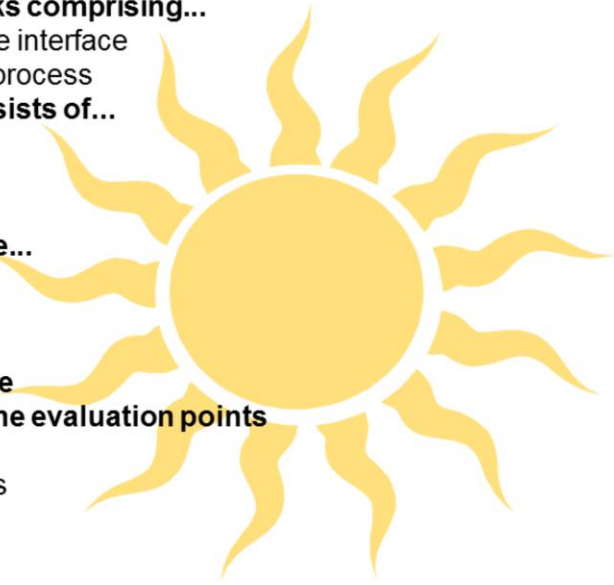


Local parameters are usually declared as data items, i.e. variables or constants. Variables can be changed within the function body. If this is done, they do not affect anything outside the function. However, if the parameter variable is passed a memory location instead of data, it may be possible to access the data at that address. To do this, the parameter must be declared to contain an address. Variables capable of doing this are called pointers, which are covered later in the course.

The remainder of the chapter defines which part of the program has access to the data and how it is stored in memory, i.e. scope and storage class.

Summary

- **Functions are named blocks comprising...**
 - A header that contains the interface
 - A body that contains the process
- **The function interface consists of...**
 - Return information
 - A name
 - A list of parameters
- **Functions require a precise...**
 - Declaration (prototype)
 - Call
 - Definition
- **Functions may be recursive**
- **Function calls/returns define evaluation points**
- **Functions define scope**
 - Visibility of data identifiers



Function invocation means:

Each expression in the argument list is evaluated.

The value of the expression is assigned to its corresponding formal parameter at the beginning of the body of the function. The expression is converted to the type of the formal parameter (if necessary and possible).

The body of the function is executed.

If a return statement is executed, control passes back to the calling environment.

If the return statement includes an expression, the value of the expression is converted (if necessary) to the type given to the type specifier of the function, and that value is passed back to the calling environment.

If no return statement is present, control is passed back to the calling environment when the end of the body of the function is reached.

If a return statement without an expression is executed, or no return statement is present, no useful value is returned to the calling environment.

All arguments are passed with 'call by value'. The called function receives a private temporary copy of each argument (not its address), so it cannot alter the original argument in the calling function.

The 'call by value' mechanism is in contrast to 'call by reference'. A later chapter will explain how to accomplish the effect of 'call by reference'. It is a way to pass addresses (references) of variables to a function that then allows the body of the function to make changes to the values of variables in the calling environment.

All arguments are evaluated before being passed into and out of functions.

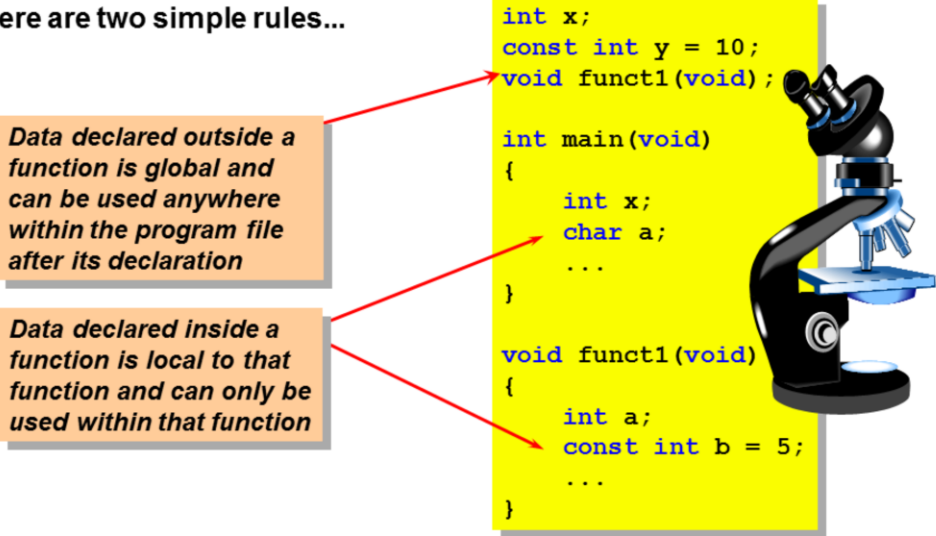
Scoping, Data Storage and data lifetime is a fundamental issue when designing functions. Knowledge of the stack is essential if we are to make our functions efficient and robust.

Scope of Data

- The scope of a data item is that part of a program in which it can be referenced
- There are two simple rules...

Data declared outside a function is global and can be used anywhere within the program file after its declaration

Data declared inside a function is local to that function and can only be used within that function



```
int x;  
const int y = 10;  
void funct1(void);  
  
int main(void)  
{  
    int x;  
    char a;  
    ...  
}  
  
void funct1(void)  
{  
    int a;  
    const int b = 5;  
    ...  
}
```

Scope refers to the accessibility of data. There are two types of scope: local and global.

Local scope defines access constrained to a block. Data, whether variable or constant, defined inside any block has local scope. The inference is that only statements within the block are able to access that data directly by name.

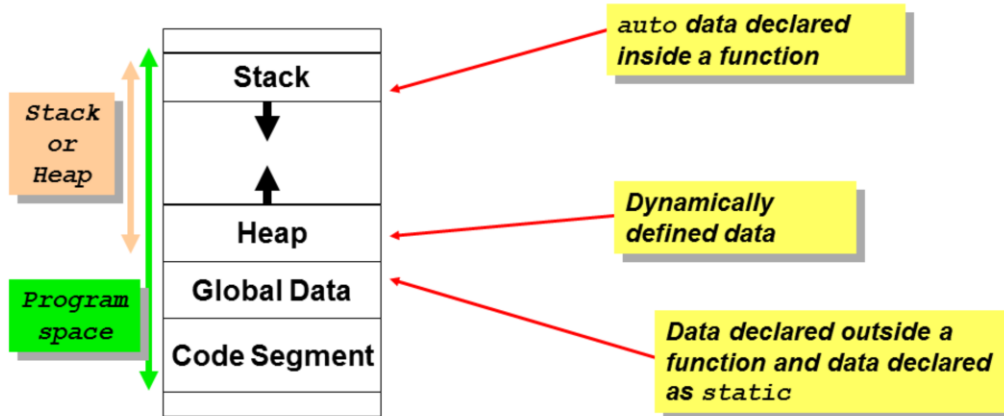
Global scope defines a wider scope. Data declared outside function blocks is potentially accessible to the entire program. By default, all statements in the source file below the declaration have access. Scope beyond that needs to be defined explicitly. This is covered in the Working with Large Programs chapter.

An identifier that names a local data item can be used anywhere else in the program, without destroying the access rules for that local data item, e.g. the data named `a` in the code shown above. If it shares its name with a global data item, the global data item cannot be accessed within the scope of the local data, e.g. the data item `x` in the code shown above.

It is worthwhile developing or using standards that define a naming convention, so that you can distinguish between local and global identifiers.

Memory Layout

- C Programs all have similar layout in memory
- Location of data depicts its behaviour



The diagram above illustrates a typical layout of a C program in memory. Our focus of attention are the three blocks described in detail.

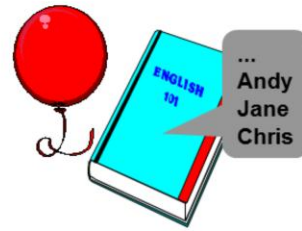
The stack is a shrinking/expanding block of memory which is used by the runtime to process functions. So data declared locally are allocated in this block (although see `statics` below). This data is recycled as soon as a function has been exited from.

The heap is also a shrinking/expanding block of memory. This area of memory is reserved for dynamically allocated data and does not get re-cycled by the system during the lifetime of the program. Both allocation and de-allocation is handled by programmers using standard library function calls – a topic of later chapters on Pointers. It means that the data may stretch over the lifetime of many function calls. The stack and the heap 'fight' for the same 'super' block, but there is never any overlap.

Global data is located in a fixed block of memory. This will also house data locally defined in functions which are marked `static`. A topic covered further on in this chapter.

The Storage Class of Data

- **Data items are treated in different ways depending on...**
 - Where they are declared
 - Their storage class
- **Storage class refers to the manner in which memory is allocated for data**
- **Types of storage class**
 - *auto*
 - *static*
 - *register*
 - *extern*
- **extern is covered in a later chapter**



It is necessary for all programmers to be aware of these storage classes, especially *static* and *extern*.

With the exception of *extern*, they are described in the following pages. *extern* is used to extend the scope of global data beyond the default, i.e. that of file scope. It is described in the *Working with Large Programs* chapter.

Storage Class *auto*

- ***auto* can be used before locally-declared data**
 - It is often omitted (the compiler assumes it by default)
 - *auto* data items exist only during the execution of the function in which they were defined
 - They are automatically created on function invocation and disappear automatically on function termination

Automatic Local

```
int add(int a, int b)
{
    int total;

    total = a + b;
    return total;
}
```

What value does an auto variable have on creation?

How do we deal with auto constants?



auto is probably the least-used C keyword. It is the default for all data declared locally. The word conjures up the idea that ‘something occurs’ without any programmer intervention. This is an apt description of the process, since the runtime environment automatically grabs memory from its pool of locations dedicated for runtime processing. The memory is released as soon as the data goes out of scope, i.e. as soon as the function return or `}` is reached. The value on ‘creation’ is garbage, and its access on destruction is illegal.

auto data can be variable or constant. An `auto const` must be initialised, e.g.:

```
const double pi = 3.14159;
```

Storage Class *static*

- **static data items...**
 - Are created on first entry to the function in which they are defined and retain their values through successive function calls
 - Remain statically allocated throughout program execution
 - Have a default initialisation to zero

```
int add_to_total(int c)
{
    static int total;

    total += c;
    return total;
}
```

Variables declared outside a function have static storage by default

```
/*static*/ int count;

int main(void)
{
    ...
}
```

What about static consts?



The *static* storage class describes the property of data that exists throughout the lifetime of the program execution. There are two types of *static* data: global data and specially-designated local data.

The local data needs to be designated by using the *static* keyword in the declaration. A local *static* exists for 'all time'. It is managed by the compiler, which allocates special 'static' storage for it before the program is executed. However, this storage is only directly accessible by statements within the function block. The statements can access and change the data just like any other data in scope. However, the value of the *static* item remains firmly in place until it is next accessed.

Global data is *static* by default. Be warned! Do not use the *static* keyword on global data unless you know the whole story. Used on global data, *static* means something very different. The item is still *static*, but its access is no longer 'fully' global. 'Static' globals are covered in the *Working with Large Programs* chapter.

Because the compiler is responsible for allocating storage, the initial value of variables is zero. 'Static' data can be initialised to a different value, if so desired. Local *statics* will receive the initial value once and once only. The runtime environment ignores the initialisation, e.g.:

```
static int total = 1 ;
```

This guarantees a value of one for the first invocation of the function. This is the method used to initialise local *const statics*.

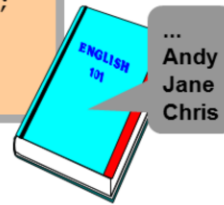
Storage Class *register*

- The ***register*** variable prefix advises the compiler of heavy usage
 - Machine registers may be used if available
 - Can prefix automatic variables and function parameters
 - Not generally recommended with modern compilers

```
int parameter(register int used_lots)
{
    ...
    ...
}

int counter(int x)
{
    register int master_count;
    ...
    ...
}
```

See notes on *volatile*



‘Register’ variables have to be local automatics. This includes function parameters. If no register is available, the data is stored as a true *auto*, i.e. by allocating memory from the runtime stack.

Nowadays, compilers are better than most programmers at deciding how CPU registers should be used. Fine-tuning using this technique is best left to the experts. Indeed, the *volatile* keyword can be used to quell any over-keen compiler, since it overrides any thought of placing a variable in a *register*.

Like the other storage specifiers, *volatile* is used to qualify the definition of data. It does not specify a storage class as such it simply informs the compiler that the data could potentially be accessed by another process. It is almost the opposite of *register* in that the compiler should not place it in one during any optimisation. It is rarely used nowadays except in multi-threaded code, or for data holding memory addresses that are to be accessed or potentially modified by hardware.

Note that both *const* and *volatile* can be used together.

Scope, Storage Class and Lifetime

- **Scope refers to...**
 - Where the data can be referenced by name
- **Storage class refers to...**
 - How and where the data is defined
- **Local autos allocated on the stack**
 - 'Recycled' post function return
 - Lifetime of function run
- **Local statics and globals allocated in 'global' space**
 - Always 'live'
 - Lifetime of program
- **All other data allocated on the heap**
 - Via pointers – see later
 - Lifetime under programmer control

```
int a, b, c;
double calc(int, int);

int main(void)
{
    int a, d;
    double e;
    e = calc(a, d);
    ...
}

double calc(int p, int q)
{
    static int sum;
    double temp;
    register int i;
    ...
}
```

Global data items, both variables and constants, are visible from their definition onwards within the file. This is sometimes referred to as file scope or module scope. The `extern` keyword is used to extend the scope beyond the file/module – a topic covered in a later chapter.

It is also possible to force the compiler to keep the scope of global data to that of the file/module.

Storage classes, scope and initialisation are summarised in the language summary appendix.

In brief, the lifetime of data is dependent on its storage class.

All local `auto` data is allocated on the stack, which is an section of recycled memory dedicated to functions that are currently executing or waiting in the call chain.

Global or local `statics` are allocated in static storage which is an area of memory which is always 'alive' and as appropriate, made available to parts of the program as required.

The last storage area is that of the heap which is used for allocating data dynamically at runtime. The compiler has no knowledge of this data's requirements which are totally governed by dedicated library functions that use pointers to locate and process the data. This is covered in later chapters.