



Large Programs

- **Objective**
 - To design, implement and maintain larger applications
- **Contents**
 - Separate module compilation
 - Sharing data across modules
 - Sharing functions across modules
 - Keeping data private
 - Keeping functions private
 - Standard C runtime library
- **Summary**

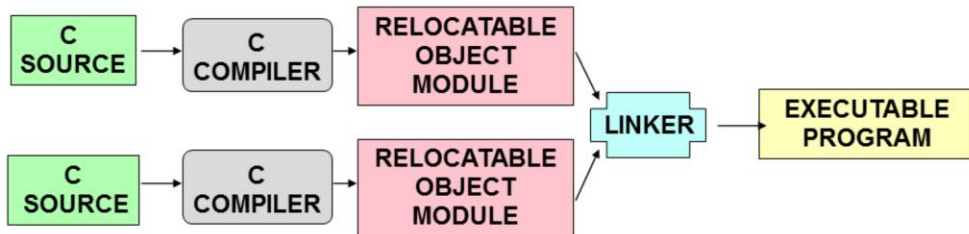


The objective of this chapter is to describe how a large application can be built, starting from several source files. The source file, referred to as a module, is the basic unit from which the application is built. The chapter deals with data and function communication and scope between modules.

The chapter concludes with a review of the standard runtime library with some examples from some of the more commonly used routines.

Separate Module Compilation

- A C program need not all be compiled at the same time
- The source of a program may be spread across several files



- What are the advantages of modular programming?

An individual module is a compilable unit. Its contents are sufficient to satisfy the needs of the compiler. This infers that all the syntax is correct and complete.

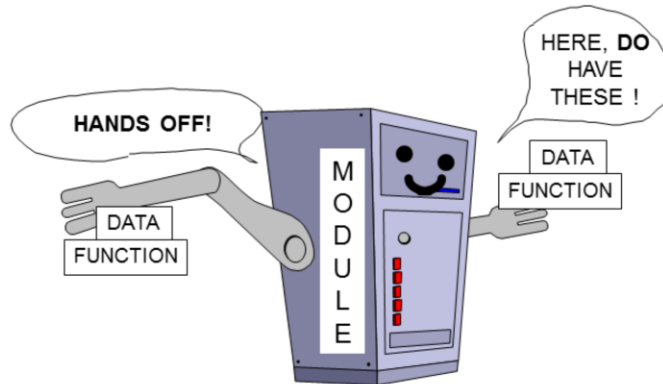
A module is a combination of global data and function definitions, together with preprocessor directives, `typedef` statements and structure/union templates. The last three components are usually found in a header file.

A successful compilation will result in a single relocatable object module, which is usually a `.obj` or `.o` file. This contains the machine-code instructions that have been translated purely from your module code.

A group of object files will form the major input to the linker.

Communicating Between Modules

- It is necessary to appreciate C's scoping rules for single source files
- A modular program needs to...
 - Share data across module boundaries
 - Call functions across module boundaries
 - Protect data and functions from other modules



An individual module contains statements and directives that will be interpreted by the compiler as requests to make data and functions potentially available to the entire application.

By default, global data has the scope of the module. Unless otherwise specified, other modules can effectively 'import' this data to their own scope using the `extern` keyword.

By default, function definitions have application scope, unless otherwise specified.

If declared outside a block, `typedefs` and `structure/union` templates have file scope.

Sharing Data Across Modules

```
FILE1.C
int count;

int main(void)
{
    ...
    count = 5;
    ...
}

FILE2.C
#include <stdio.h>
extern int count;

void funca(void)
{
    int i = count;
    ...
}

void funcb(void)
{
    ...
    count = 10;
    ...
}

FILE3.C
#include <stdio.h>
extern int count;

void funcc(void)
{
    ...
    ...
}

void funcd(void)
{
    int j = count;
    ...
}
```

This example illustrates the extending of global data from module to application scope. The `extern` keyword effectively imports a global item from another object module.

Care must be taken when using `extern`. The rule is:

Define the data item in one file only and 'extern' it in any file that requires access. Leave all other modules alone!

An easier way to do this is to 'extern' it in every source file by placing it in a header file that is `#included` in all files. The existence of an `extern` and a definition in the same file does not cause any trouble. The compiler effectively throws away the `extern` declaration.

The breaking of the rule will cause a linker error.

Sharing Functions Across Modules

- Functions can be called simply by providing the correct prototypes, as recommended by the ISO standard
- Failure to provide a prototype could cause problems, since the compiler assumes an integer return

```
double func(void);  
  
int main(void)  
{  
    ...  
    double y = func();  
    ...  
}
```

FILE1.C

```
double func(void)  
{  
    ...  
    ...  
    ...  
}
```

FILE2.C

Function scope is far simpler. The presence of a function prototype ensures that the function can be called. It is the task of the caller to supply a prototype to ensure that the call is correct. Absence of prototypes results in problems.

Keeping Data Private

```
static int count;
```

```
int main(void)
{
    ...
    count = 5 ;
    ...
}
```

FILE1.C

A static global variable can only be used in the module in which it is defined

```
#include <stdio.h>
```

```
extern int count;
```

```
void funca(void)
{
    int i = count;
    ...
}
```

```
void funcb(void)
{
    int j = count;
    ...
}
```

FILE2.C

**LINK
ERROR !**

A global data item can enforce module scope using the `static` keyword. This ensures that no other module can access the data. Any request by a module to `extern` it will result in a linker error.

`static` is the only C keyword that has two meanings. There is no compiler ambiguity; the scoping rules of data should ensure it. `static` used with local data implies 'static' or permanent storage. `static` used with global data implies keeping the scope to that of the module, i.e. making it private.

Keeping Functions Private

The diagram shows two source files, FILE1.C and FILE2.C, illustrating the effect of the `static` keyword on function visibility.

FILE2.C (Left): Contains a function definition `static double funca(void)`. The `static` keyword is circled in red. Below it is another function `void funcb(void)` which calls `funca()`. A label `FILE2.C` is at the bottom right of this block.

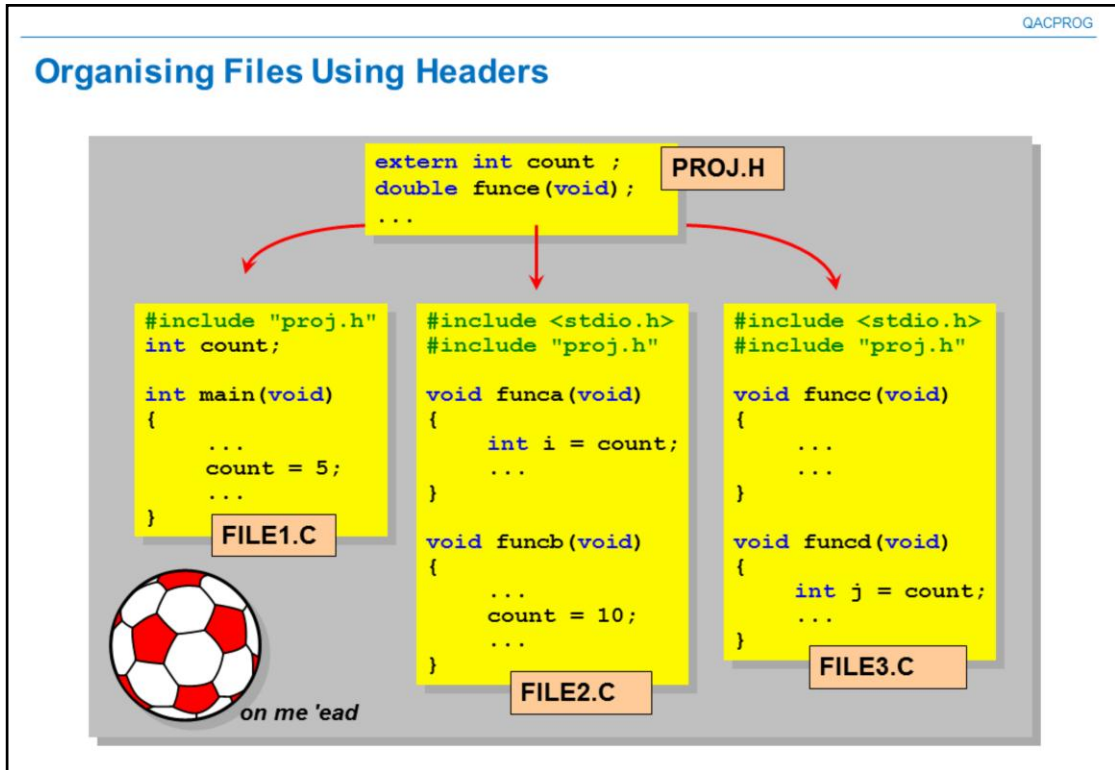
FILE1.C (Right): Contains a function declaration `double funca(void);` and a `main` function that calls `funca()`. A label `FILE1.C` is at the bottom right of this block.

A red arrow points from the `funca()` call in FILE1.C to a box labeled **LINK ERROR !**, indicating that the linker cannot find the definition of `funca` because it is static and only visible within FILE2.C.

Functions defined as static can only be called within the module in which they are defined

The `static` keyword used with function definitions has a similar effect. The function cannot be called from another source file. A function call would be picked up by the linker as an error.

This is a useful mechanism for library writers. It means that a module could contain definitions of 'low-level' utility functions, which are used only by functions within the same module. Clients or users of the library could not call them.



This example leads on from the general discussion of data and function communication across modules. Header files can improve organisation and maintenance. The majority of program developers would be lost without them.

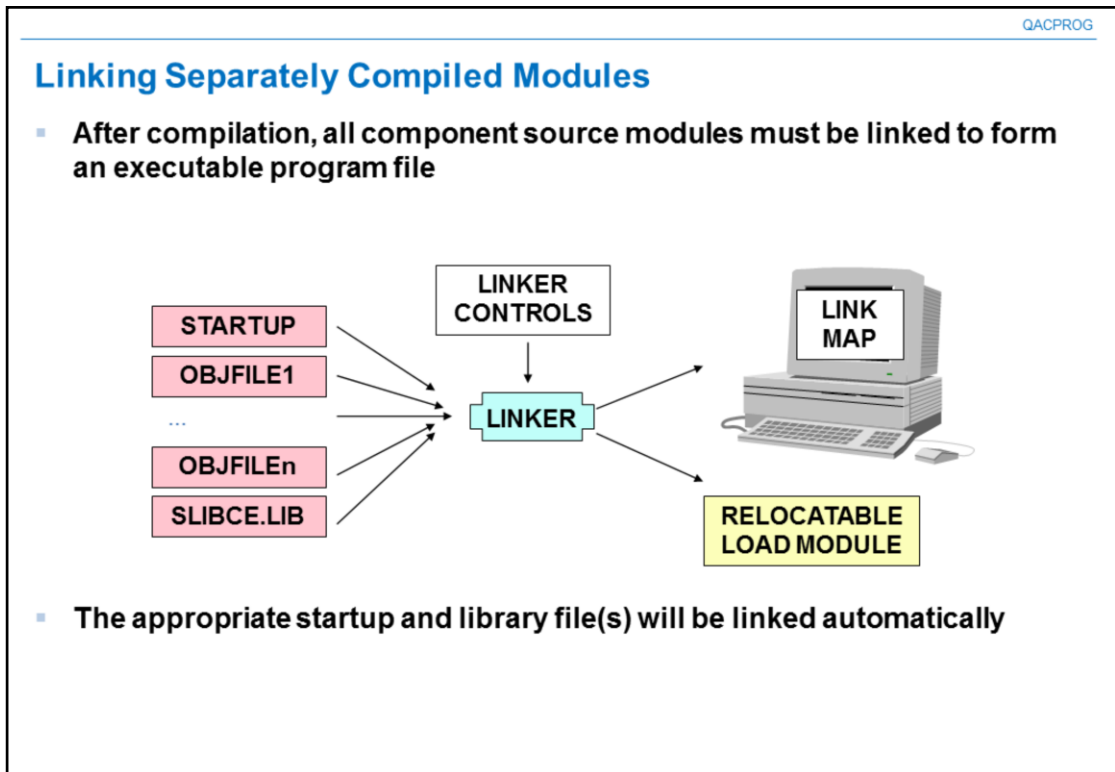
There are five things to put in header files:

- #includes (requiring #if protection)
- #defines (and other preprocessor directives)
- function prototypes
- structure/union templates
- typedef statements
- extern data declarations

There are three things that should not be to put in header files:

- global variable definitions (consts are acceptable)
- function definitions

The organisation of modules and their header files is best managed using a building tool such as make.



Source files can be compiled. This is achieved in two ways:

1. From the command line:

Individually using the appropriate compiler/option combination, e.g. `CL /C file1.c`, or `gcc -c file1.c` or in groups.

2. From within an Integrated Development Environment (IDE) like Visual C++, KDevelop, and so on.

Once you have the complete set of object modules, the linking can be carried out. The linker requires *everything* before it can perform its task correctly. This includes startup code and libraries. These are usually set up as defaults, but are changeable during communication with the linker. Again, you have two choices:

From the command line:

Interactively with a link program. You would need to supply the names of the object files, then answer questions concerning default names, libraries and option, etc.

From within an environment or from the command line:

Using a make file.

The make file is a text file that contains information in a special 'make' language syntax. The information is sufficient for the make program to compile and link only what is necessary.

Linking is not usually seen as a separate step, the linker is normally called automatically from the compiler.

The Runtime Library

- **Set of predefined routines that C programmers may call from their program**
- **Provides fast and efficient basic functions not provided by the language**
- **To use the routine...**
 - Read all about it in the compiler's runtime library reference manual
 - `#include` the specified file that contains any declarations and definitions that you may require
 - Call it. You will need to provide the correct arguments and cater for the appropriate return value



An ISO C compiler must provide the standard library routines in a library or libraries.

All standard compilers provide these routines, but they also provide other routines alongside them in the same library. Strictly, the resulting library is not standard. So, be warned! It is not in the compiler manufacturer's interest to publicise this fact.

What is Available?

- **The standard library contains many useful functions**
 - Worth learning if you are serious about C
 - Good language/library references are listed in the bibliography



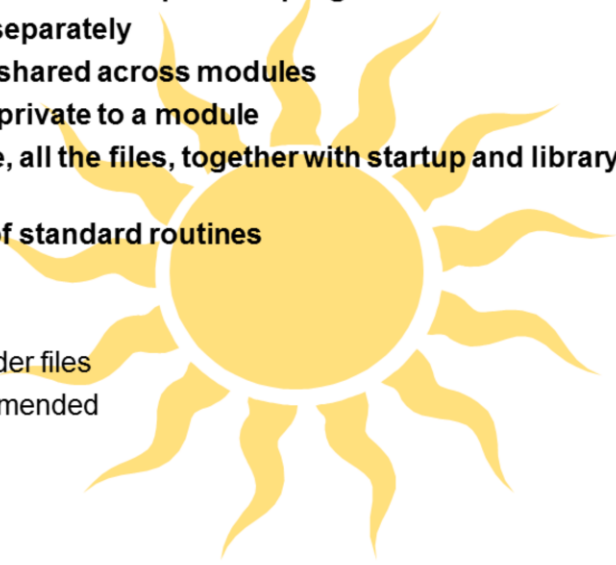
<assert.h> <locale.h> <stdio.h>
<ctype.h> <math.h> <stddef.h>
<errno.h> <setjmp.h> <stdlib.h>
<float.h> <signal.h> <string.h>
<limits.h> <stdarg.h> <time.h>

<iso646.h>
<wchar.h>
<wctype.h>

There are currently 15 sections (see above) to the ANSI 1989 Standard C. Three more were added in the Amendment (NA1 1995) and a further 6 for C99. The latter addition is not discussed in this course (as mentioned earlier, C99 is covered in QA's Advanced C Course).

The library in all its glory could warrant a complete course in its own right. We will cover a few of the more popular areas, i.e. time and date management, variable argument lists, maths support, a miscellaneous set and error handling. Note that strings were covered in some detail in the arrays chapter and I/O had an entire chapter dedicated to it.

Summary

- **A module is a source file that contains part of a program**
 - **Modules can be compiled separately**
 - **Data and functions can be shared across modules**
 - **Data and functions can be private to a module**
 - **To create an executable file, all the files, together with startup and library files, are linked together**
 - **The library contains a set of standard routines**
 - **To use any routine**
 - Read about it
 - `#include` appropriate header files
 - Call it precisely as recommended
- 

It is inevitable that the techniques discussed in this chapter will be used. It is worthwhile investing time and energy to become familiar with make utilities, especially now that make is available throughout most of the C development environments.

Intentionally left blank