Large Programs - Chapter Summary

1 Separate Module Compilation

Large C programs are usually written as a series of separate modules, which are only compiled and linked to form the final executable program when the project is nearing completion.

There are many practical advantages of this approach.

It may take many days to compile the source code for a really large program, and if there are any bugs in the finished product, they may be very hard to trace in such a large volume of code. When they have been tracked down and the appropriate section of code has been changed, the lengthy process of compilation begins all over again. The "write-compile-test" cycle is simply too time consuming and unwieldy if a large program is written and compiled as a single unit.

It is far more efficient to write, compile and test program modules individually, only linking them when each module performs as it should. The "write-compile-test" cycle is more manageable with modular programs, and it is much easier to isolate bugs within smaller units of code, each one of which is responsible for performing only a single programming task. Once a module has been completed and compiled, there is then no need to compile it again.

In a large project, each separate module may be written by a different programmer or team of programmers. These programmers must work in parallel and in consultation with each other, sharing a common coding standard, but it is not a trivial benefit that a modular approach to programming will allow them the freedom to use their own skills and initiative to design and develop their particular part of the application.

Modular programming also saves time in the longer term by making individual program modules available to be used over again in many different applications.

The course notes illustrate the process of compiling separate program files to produce relocatable object modules, and of linking these modules with each other and code from the standard libraries to produce the final executable program.

2 The Scope of Data in Modular Programs

Data items that need to be protected from changes made elsewhere in the program need to be local in scope, whereas data items that need to be widely shared between many different functions should be global.

It has been shown in the Function and Program Structure chapter that the scope of a data item within a program file depends on where it is declared, whether inside or

outside a function, and the storage class to which it is allocated, whether it is an `auto`, `register` or `static` item.

A modular program needs to share some data not only between functions in the same program file, but between functions in different files altogether. It also needs to allow functions defined in one module to be called from another module.

Conversely, it is often necessary to protect data in one module from being unintentionally changed by operations carried out elsewhere in the program.

3 Sharing Data between Modules: storage Class extern

Data to be used in several different program modules must clearly be global.

Global data can be defined only once in a program, however, and this raises potential problems when a large program is split up amongst several files that are compiled independently and then linked. If a global item is encountered in a file in which it is not declared, the compiler will throw up an error message when the file is compiled. If, on the other hand, a global item is declared in every file in which it is used, potential ambiguity problems arise, with the linker trying to match the data item with its definition.

These problems are avoided if all global data is defined in one file and declared in other files with the `extern` specifier. A declaration that an item is classified as `extern` alerts the compiler that there is no need to create storage space for it because provision for assigning this has already been made elsewhere.

4 Sharing Functions between Modules

A function defined in one module can only be called by a function in another module if the modules have both been compiled and linked.

C allows a module that contains a function call defined in another module to be compiled successfully. To achieve this, the function prototype must be present in the calling module. If a prototype is missing, the compiler will assume that the called function returns an integer. The compiler is also not able to check the arguments used in the call. Luckily, most compilers issue warnings if prototypes are missing.

Robust programs require the presence of the prototype of every called function, as recommended by ISO C.

5 Protecting Data

Global data provides a mechanism for sharing data between functions and, using the `extern`, between modules without the overhead of argument/parameter passing. There is a drawback, however; data is vulnerable to be changed inadvertently by operations carried out anywhere in a program.

Global data items are defined outside functions, but by classifying them as `static` ensures that the data may only be accessed by functions defined in the same module. For this reason, the `static` global data item is safer than the 'true' global data item, as it cannot be accidentally affected by functions defined in other modules throughout the application, as any attempt to use `extern` on one will produce a linker error.

This does allow the programmer the possibility of using the same identifier for global data items in several modules. To ensure data integrity, therefore, it is good practice to isolate `static` global items and the functions that use them from the rest of the program, and to keep them in a single file of their own.

6 Protecting Functions

By default, functions are assumed by the compiler to be of storage class `extern`, and hence global in their scope. It is possible, however, to limit the scope of a function by declaring it to be of storage class `static`. A `static` function can only be called from within the module in which it is defined. If a linker fails to find the definition of a function, whose prototype (declaration) says it is static, in the same module as a call to it, it will issue an error.

Some functions may have unhelpful side-effects if applied to data of the correct type in the wrong context. Declaring them to be `static` functions and isolating the code that uses them in the file in which they are defined ensures that they are not inadvertently called from elsewhere in the program.

It is conceivable, for instance, that an applications program devised for an international building contractor would use one set of functions for estimating costs in countries where inflation is in single figures, and a very different set for estimating costs in nations suffering from the problems of hyper-inflation. In one instance, costs might calculated over the length of a project's lifetime by applying a simple compounding routine to take account of the effects of inflation. In the other case, a more complex function might be required, incorporating the rate of change of the level of inflation in addition to the compounding effect itself. The "hyper-inflation" functions could be declared `static` functions and placed in a separate module away from the rest of the program so that they could not be accidentally invoked in the cases where they would provide inappropriately-inflated cost estimates.

7 Linking Separately-Compiled Modules

When all the separate modules in a program have been compiled, the linker must be used to link the object code files so-produced with each other, and with an appropriate start-up file and library file, to produce a final executable file.

Details of the commands needed and the options available when using the linker will be given in the compiler manual.

Current versions of the Microsoft C/C++ and IBM C/C++ compilers place information on the start-up and library files required in the object files during compilation. The linker supplied with these compilers also provides appropriate prompts for all the various command-line arguments required to implement the linking process.

8 The C Runtime Library

Although many applications, such as input/output routines, are not part of the C language itself, the C runtime library provides a vast range of fast and efficient basic functions that programmers can call on to perform a wide variety of programming tasks.

In many cases, routines are available in both macro and function form. The `putc` and `getc` routines, for instance, are macro versions of the `fputc` and `fgetc` functions respectively.

A macro creates in-line code, which executes faster than an equivalent call, but produces larger programs, because the same piece of code is inserted into the program every time the routine is used. A function makes more efficient use of storage space, because the function code is only defined once, however many times the function is called. The macro version of the routine should only be used if speed of execution is critical, whereas the function version should be used in all other circumstances.

Full details of the library routines available in a C implementation will be given in the appropriate section of the compiler manual, often under the heading "Runtime Routines by Category", and familiarity with the contents of this section will ensure that valuable time is not wasted recreating functions that already exist. However, care must be taken if a program is to be ported across to other platforms. This is because most compilers include several, perhaps hundreds, of other non-standard routines, and the compiler manufacturers do not consider it of major importance to publicise which routines are standard and which are not. It is essential that the programmer understands the consequences of using non-standard routines. *A complete list of prototypes of all the standard routines may be found after this chapter summary.*

It is possible to buy additional libraries of C routines to perform certain kinds of tasks; typically complex mathematical or graphics routines.

A function from the runtime library may be called in the user's program by invoking its name and supplying the appropriate number of arguments of the correct type, as detailed in the manual dedicated to the runtime library. This has to be accompanied by including the appropriate header file that contains the function's prototype or, in the case of a macro, the definition. Using the prototype will ensure that full type checking of arguments and return values is implemented when the program is compiled.

The appropriate runtime library must be included in the linking process when the object code files are linked. The linker will search the library file and copy into the final executable file the compiled code for the library functions used in the program.