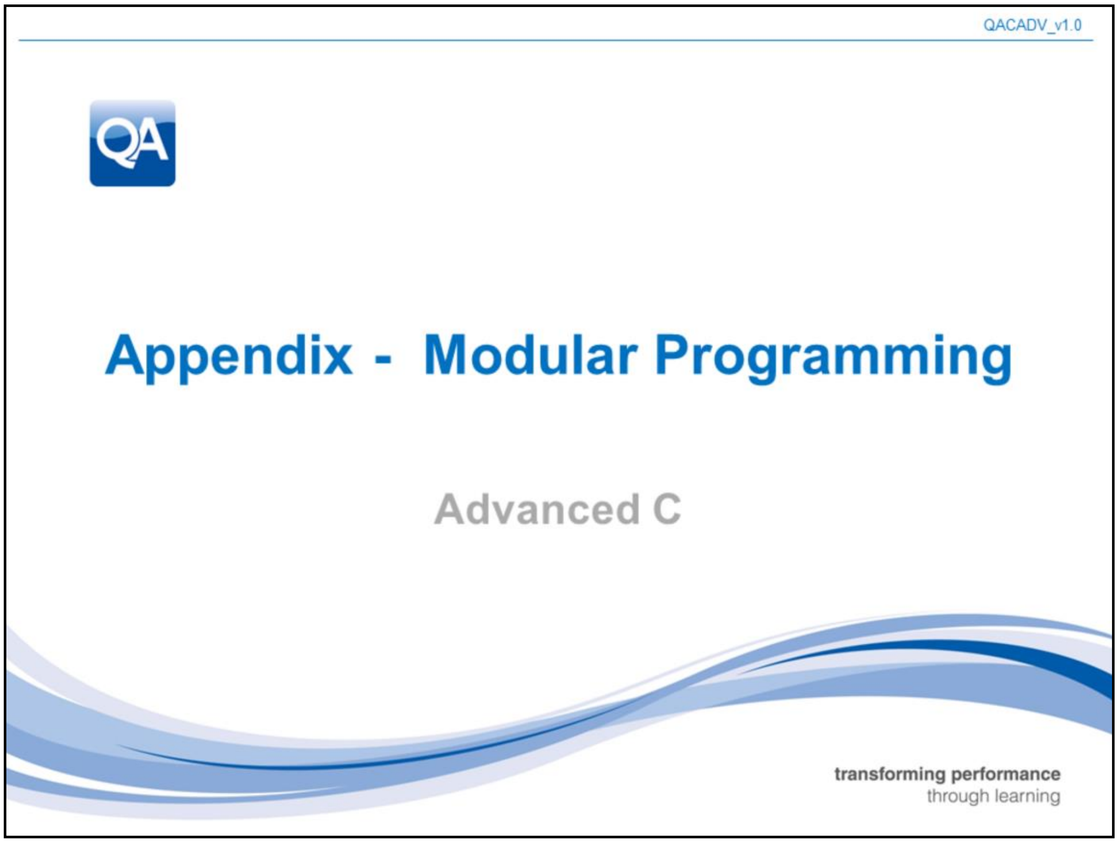
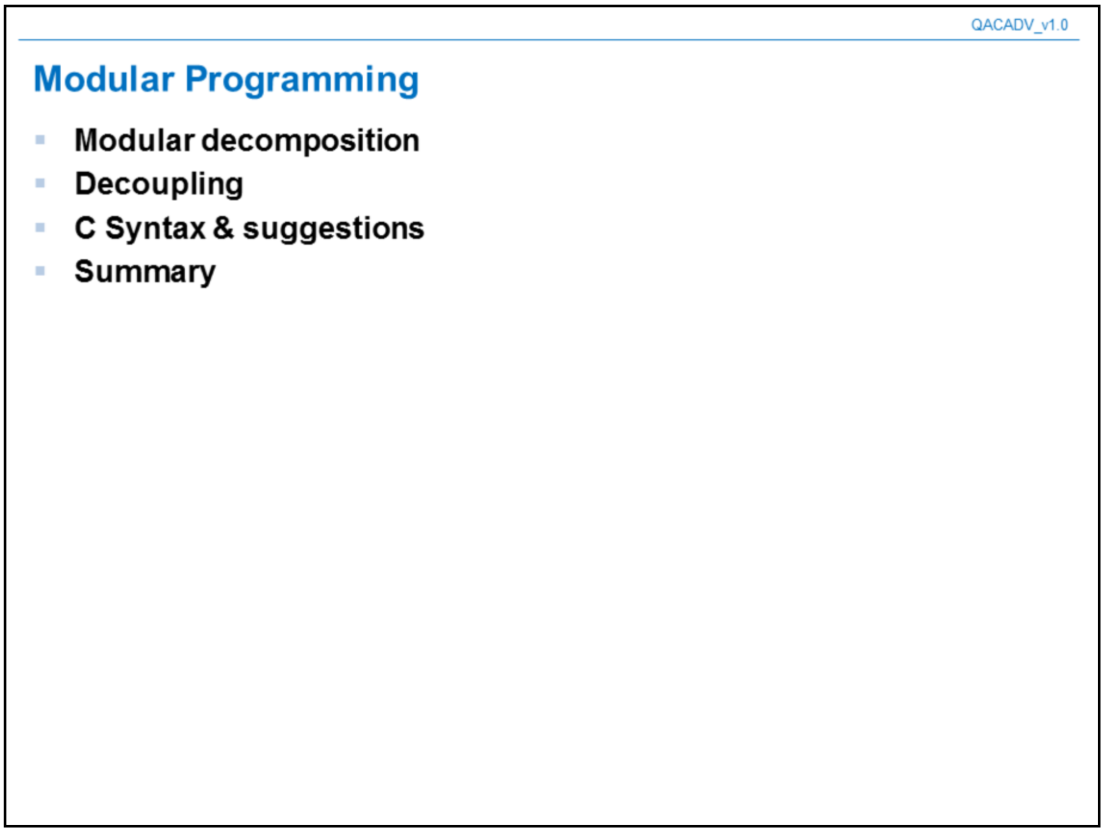


QACADV_v1.0

Appendix - Modular Programming

Advanced C

transforming performance
through learning

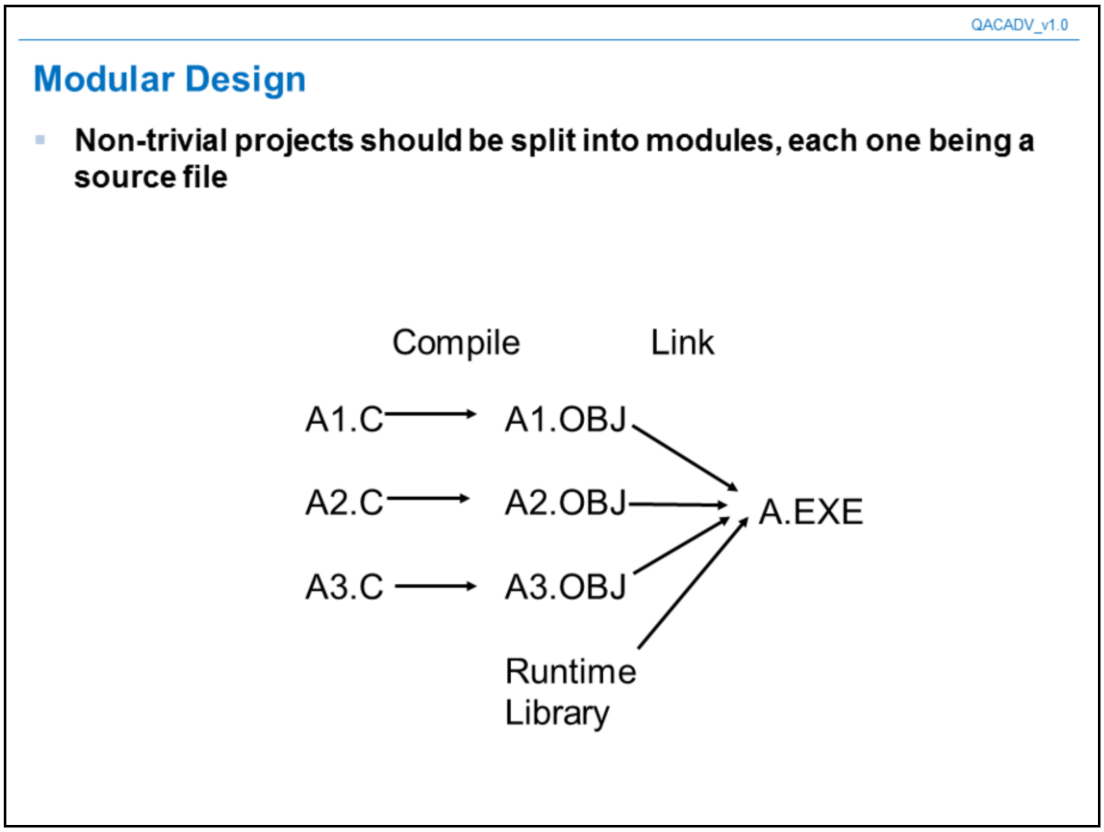


QACADV_v1.0

Modular Programming

- Modular decomposition
- Decoupling
- C Syntax & suggestions
- Summary

The objective of this chapter is to describe some techniques used for modular programming. Wherever possible, the power of the language is exploited.



QACADV_v1.0

Modular Design

- **Non-trivial projects should be split into modules, each one being a source file**

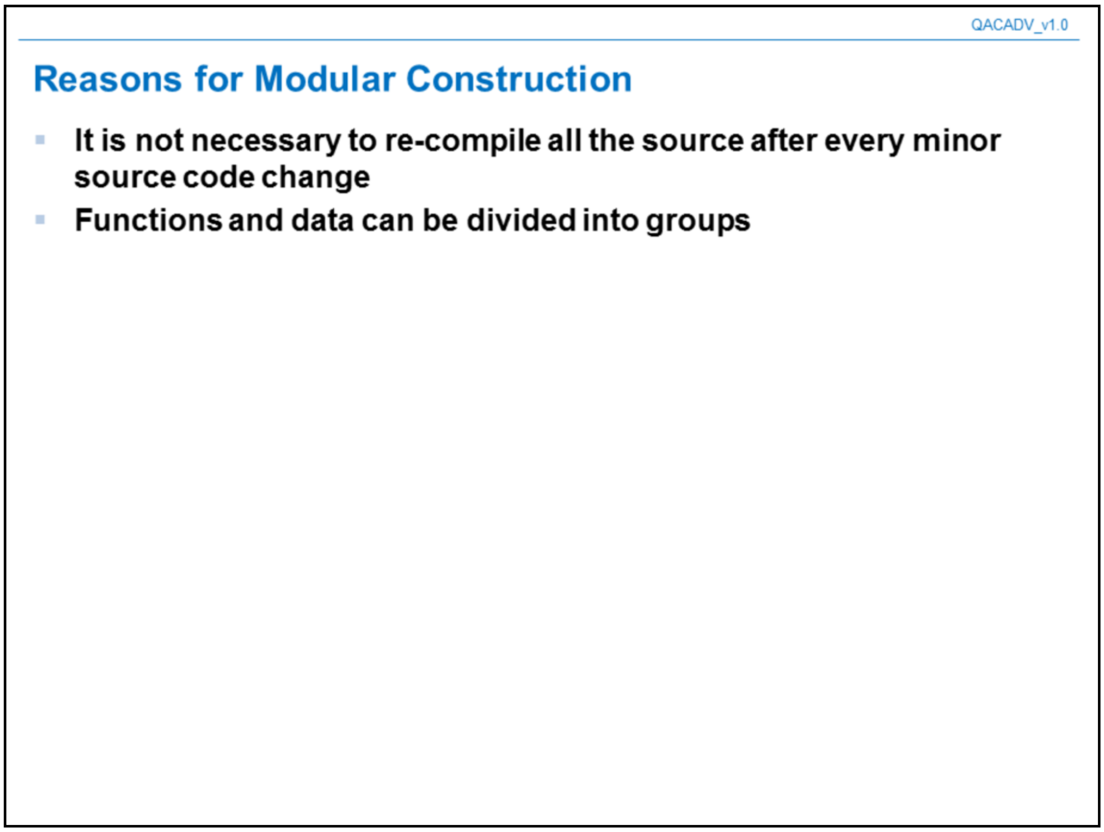
$$\begin{tikzcd}
 & \text{Compile} & \text{Link} \\
\text{A1.C} \arrow[r] & \text{A1.OBJ} \arrow[dr] & \\
\text{A2.C} \arrow[r] & \text{A2.OBJ} \arrow[r] & \text{A.EXE} \\
\text{A3.C} \arrow[r] & \text{A3.OBJ} \arrow[ur] & \\
 & \text{Runtime Library} \arrow[uur] &
\end{tikzcd}$$

We are reviewing here the compiling and linking stages. The diagram shown above illustrates a very common technique which is a "part and parcel" of the C environment; splitting the application into reasonably-sized C source files.

An individual module, i.e. a single C source file, has a very simple and lonely life cycle. It is created, compiled in isolation and tested by linking it with a test harness (a proven set of routines which invoke and are invoked by the routines of the module). Once it has satisfied the necessary testing at this level, the `.obj` (or `.o`) file is ready for the linking stage.

This procedure eases the task of unit development, but also introduces major problems during the integration and maintenance stages. Firstly, the entire set of object modules need to be in place for the initial build, and secondly, any sensible maintenance would require an intimate knowledge of the building process. These are the only real physical overheads introduced by a modular design approach.

Whether we like it or not, this method is used for static library linking (e.g. used by standard non-GUI applications). The library can be thought of as a set of object modules which get linked with the application modules, so that the resulting executable is "stand-alone", i.e. independent of any runtime support.

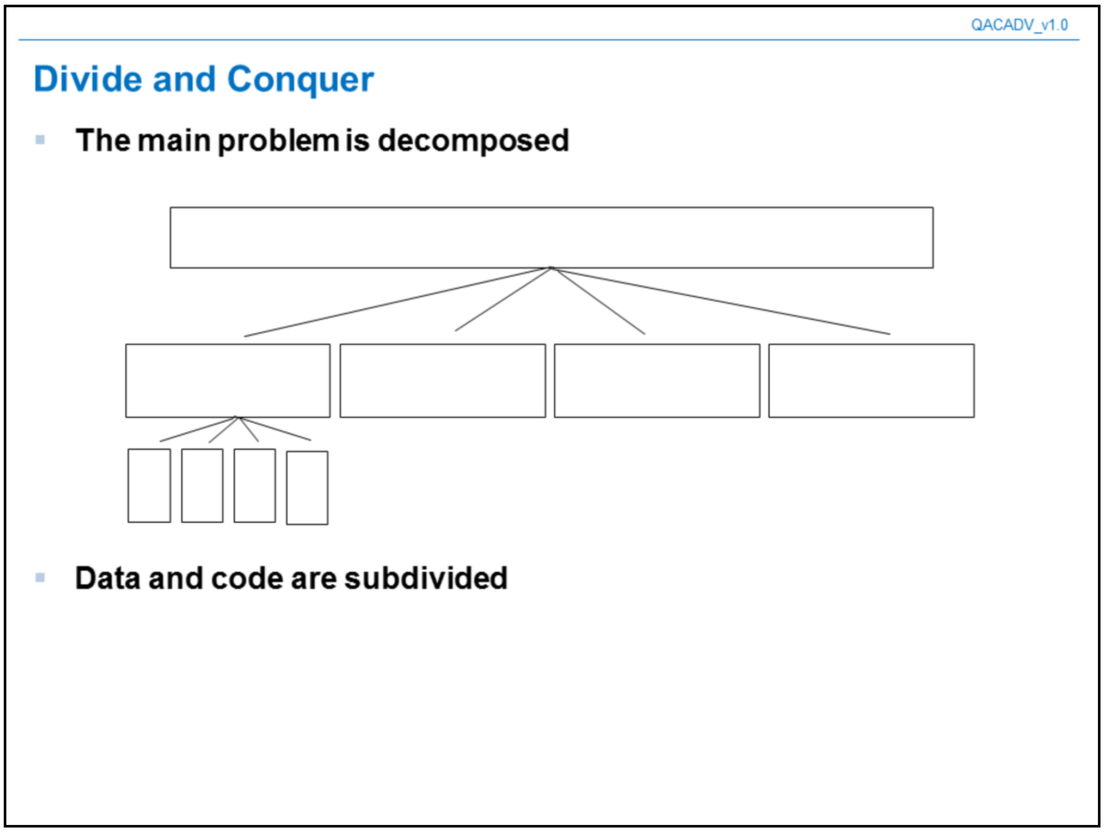

QACADV_v1.0

Reasons for Modular Construction

- It is not necessary to re-compile all the source after every minor source code change
- Functions and data can be divided into groups

Modular development is a tool for maintenance as well as for testing. The individual source file usually contains a set of routines which are logically connected, i.e. they invoke each other and use the same data structures, maybe even the same data.

Once the application has been integrated, tested and delivered, it is a relatively straightforward task to enhance and add features. Debugging is made easier by the very nature of the modularity, since the logic should be clearer and individual processes easier to isolate. This implies that once the maintenance requirements are defined, the individual module or set of modules can be selected and updated accordingly, and the remaining modules can be left well alone. Each updated module needs to be "unit tested", as described in the previous page, and then re-linked into the system. We are not making light of the very last procedure; this re-link or re-build could be a complicated process. However, it is usually automated.



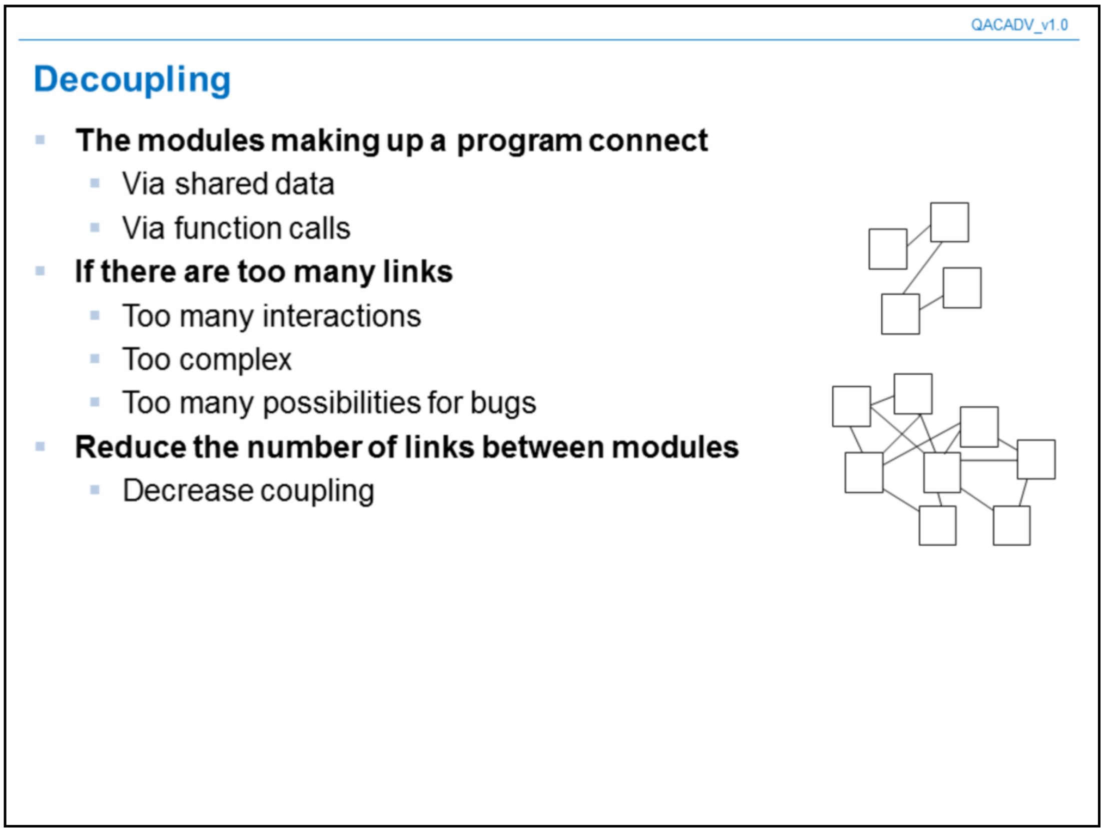
QACADV_v1.0

Divide and Conquer

- The main problem is decomposed
- Data and code are subdivided

So far, we have talked about the theory and behaviour of a modular system. How do we create one? The next few pages describe some of the problems and suggest tips which will ease the task.

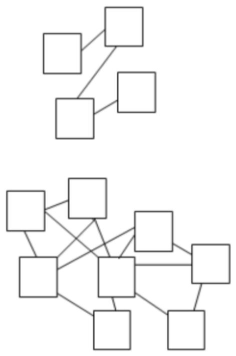
The "divide and conquer" decomposition illustrated above can be performed at any time during or just after the design stage, but must be done with much care. Unless a proper division is performed, a "domino effect" will occur when maintenance is carried out. This could happen if, for example, a data structure is shared across many code fragments. It is very necessary to divide data, as well as the routines, so that data is processed only by logically and physically "close" code. This attachment between data and code is referred to as *cohesion*, and the dependence between modules is referred to as *coupling*.



QACADV_v1.0

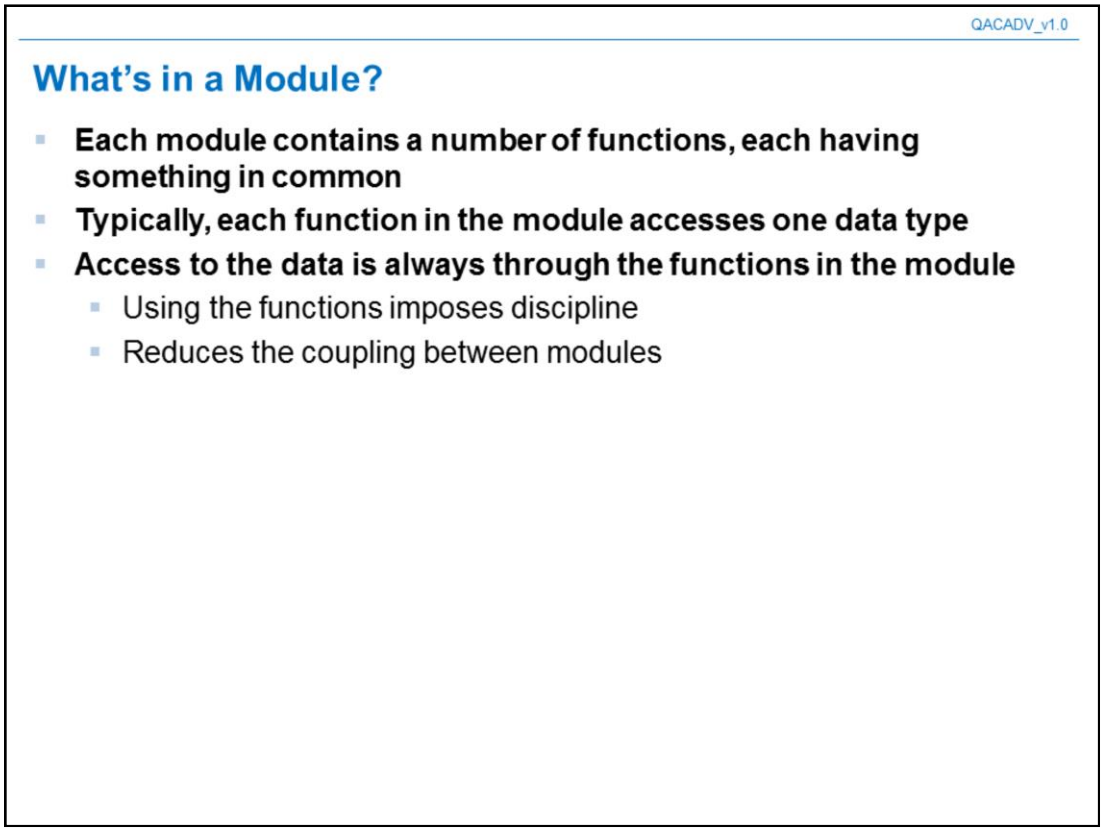
Decoupling

- **The modules making up a program connect**
  - Via shared data
  - Via function calls
- **If there are too many links**
  - Too many interactions
  - Too complex
  - Too many possibilities for bugs
- **Reduce the number of links between modules**
  - Decrease coupling



For a large or complex system, the decoupling of modules can be a very complicated process. Part of the system is dependent on another if it shares data or invokes or is invoked by its routines. This dependence may appear to be very tight and would be highlighted by the number of connections or links in the structure breakdown diagrams. This would occur if there were too many mutual invocations or access to common data. The complexity of such a network of links would mean a lack of comprehension and an increase in the possibility of bugs during implementation.

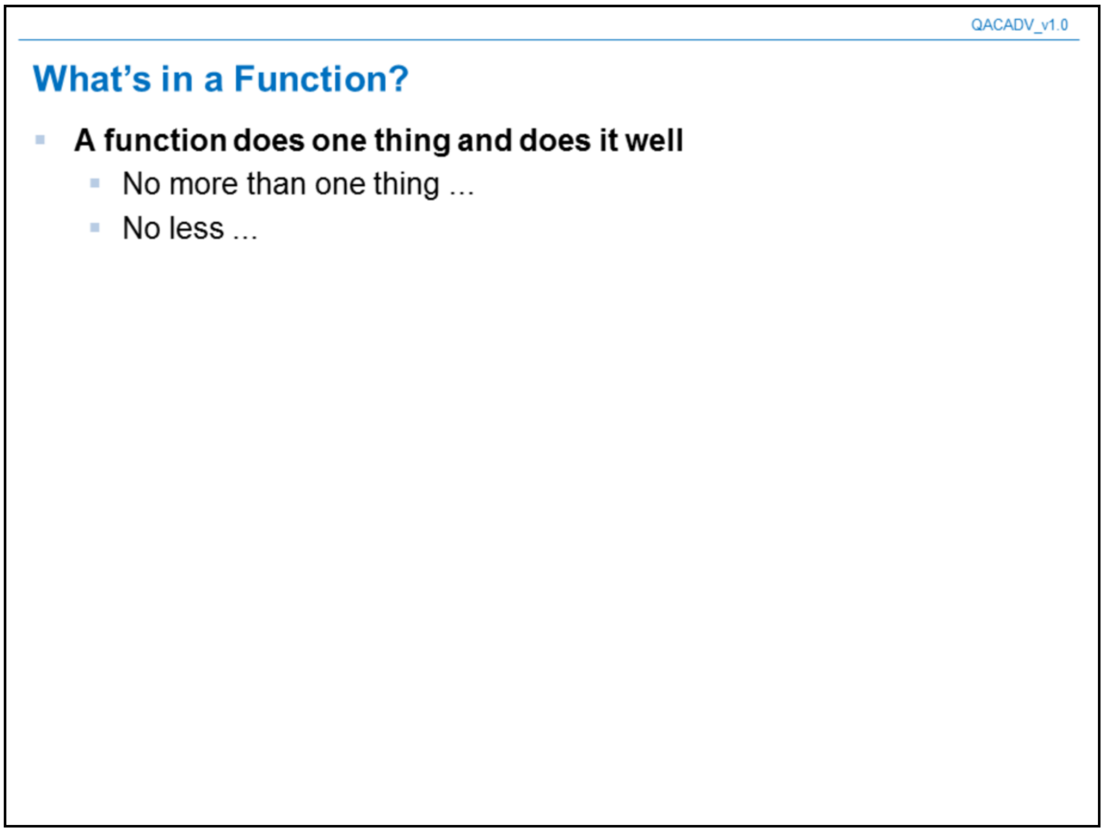
The answer is simple: reduce the number of links! Taking an existing design and trying to achieve this is not simple! However, there are some guidelines to ease the task at the design stage. We shall first look at the individual module.


QACADV_v1.0

What's in a Module?

- **Each module contains a number of functions, each having something in common**
- **Typically, each function in the module accesses one data type**
- **Access to the data is always through the functions in the module**
  - Using the functions imposes discipline
  - Reduces the coupling between modules

A module is defined as being a single C source file. Each module contains a number of logically connected routines, i.e. macros and functions. Each routine is responsible for a process which accesses and possibly affects a data item. The well-defined interface to these routines implies that the access to data items is easy to control. In fact, functions could be designed whose only purpose is to access data, i.e. "accessor" functions. The discipline involved with these techniques reduces the coupling, i.e. dependence between modules. So now, we look at an individual function.

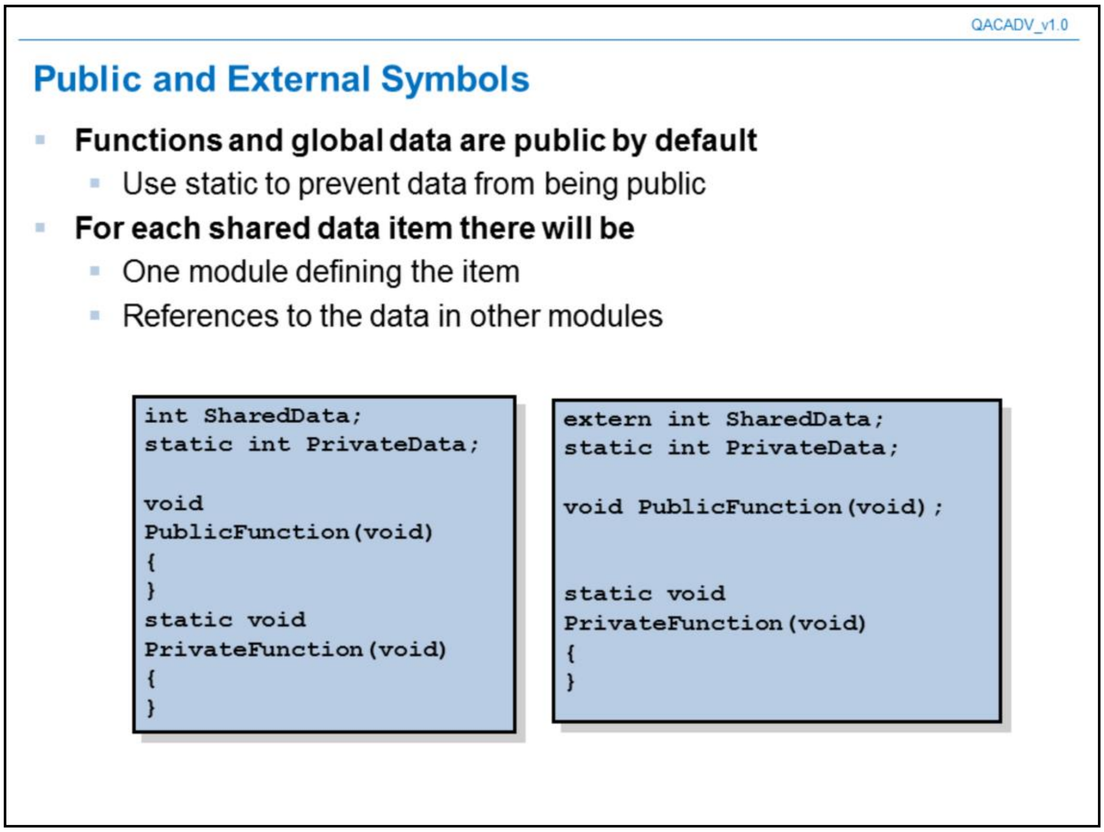

QACADV_v1.0

What's in a Function?

- **A function does one thing and does it well**
  - No more than one thing …
  - No less …

This definition of a function is well-deserved; its simplicity is the answer. As long as huge amounts of data are not being copied, function invocation is reasonably efficient. Any slight overhead involved is small when compared to the gains in security and ease of maintenance.

Clearly, the statement that "a function does one thing" is open to artistic licence. It could be taken as anything from incrementing a counter to sorting a data file. Information hiding is the important thing. The externally-perceived interface is what counts.



QACADV_v1.0

Public and External Symbols

- **Functions and global data are public by default**
  - Use static to prevent data from being public
- **For each shared data item there will be**
  - One module defining the item
  - References to the data in other modules

```
int SharedData;
static int PrivateData;

void
PublicFunction(void)
{
}
static void
PrivateFunction(void)
{
}
```

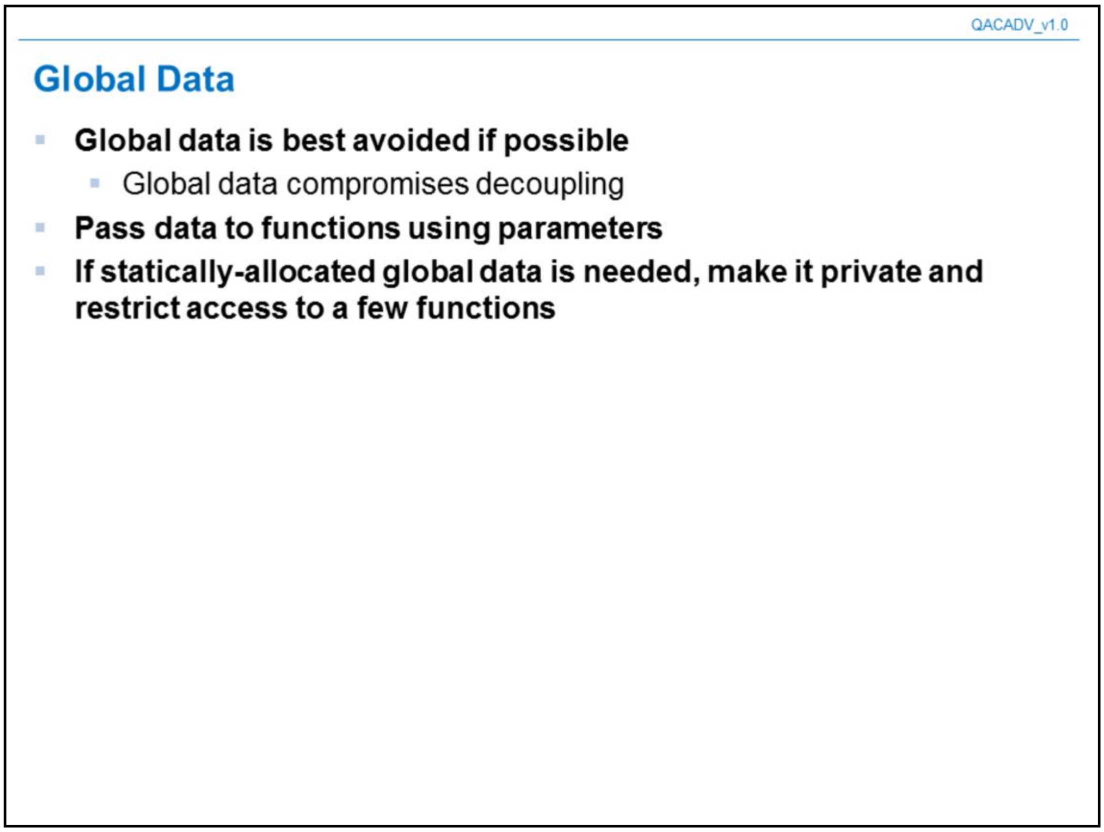
```
extern int SharedData;
static int PrivateData;

void PublicFunction(void);


static void
PrivateFunction(void)
{
}
```

C has its own rules for importing data and functions. It is also possible to make data and functions private at the module level. The rules and syntax are as follows:

> Unless declared as `static`, global data and functions are potentially accessible to other modules. This is achieved by specifying that the data is external data (using `extern`) and by publicising the function prototype in the "other" modules.
>
> If qualified as `static`, global data and functions can only be accessed/invoked by functions in the same module. Any request to use `extern` on these items or invoke them in other modules will be picked up as a link error.

Data and functions in the application scope are defined in one (and only one) module. The `extern` and prototype constructs are used in all other modules that require access.

Sharable functions are essential, since they provide the only means of access between modules. Whether or not global data is essential is debatable.



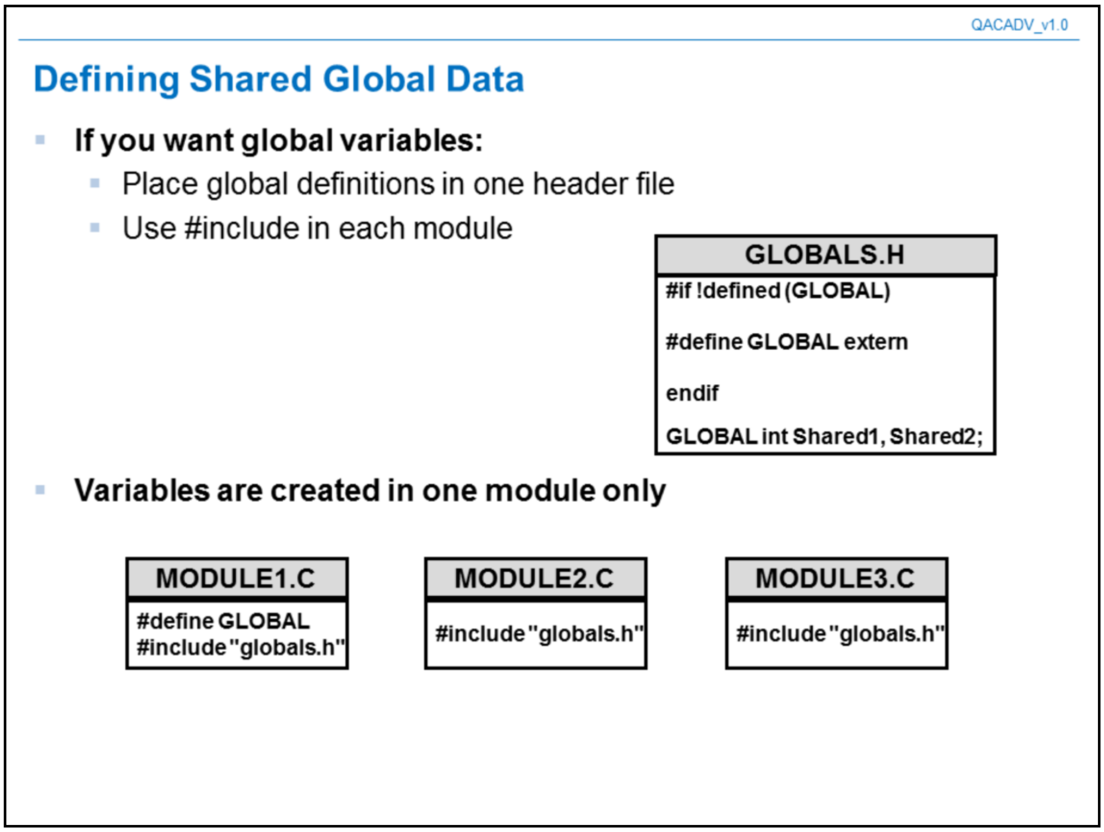
QACADV_v1.0

Global Data

- **Global data is best avoided if possible**
  - Global data compromises decoupling
- **Pass data to functions using parameters**
- **If statically-allocated global data is needed, make it private and restrict access to a few functions**

Global data is the only way to access data by name throughout a module and potentially throughout the entire application. This reduces the necessity to pass multiple copies around the system. These two reasons make global data attractive to some developers. There is also a third reason; it was also the only type of data available for the first couple of decades! These reasons, however, should be overshadowed by the dangers. The alternatives are much more secure and, in most cases, just as convenient and almost as efficient.

For the occasions when global data is a necessity, use `static`. For constant data, use the `const` keyword.



QACADV_v1.0

Defining Shared Global Data

- **If you want global variables:**
  - Place global definitions in one header file
  - Use #include in each module

GLOBALS.H

```
#if !defined (GLOBAL)

#define GLOBAL extern

endif

GLOBAL int Shared1, Shared2;
```

- **Variables are created in one module only**

MODULE1.C

```
#define GLOBAL
#include "globals.h"
```

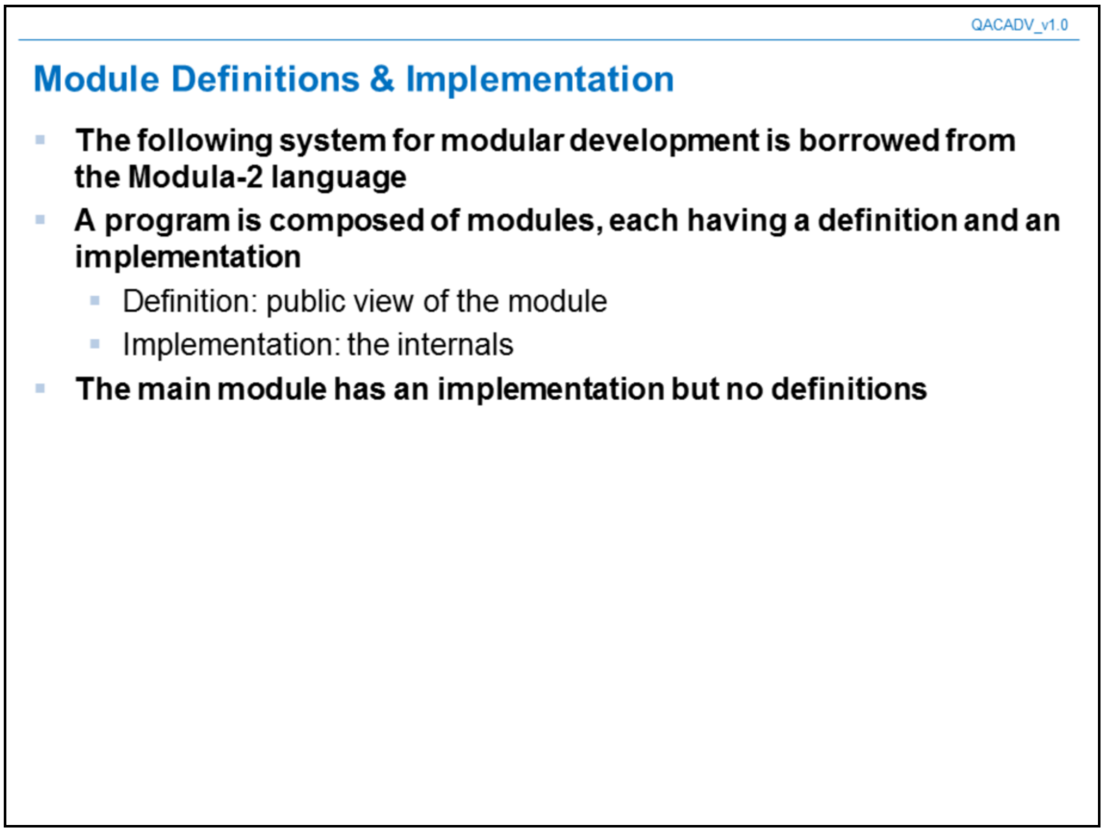
MODULE2.C

```
#include "globals.h"
```

MODULE3.C

```
#include "globals.h"
```

The example given above shows a clean method for using global items. The housing of all application-scope globals in a single header file is acceptable in terms of conventional C program organisation. The use of the preprocessor is neat and robust. Clearly, care must be taken to use the `#define GLOBAL` in one and only one module. This neat method must not fool the developer, since there are still all the potential dangers of global access underneath it all!



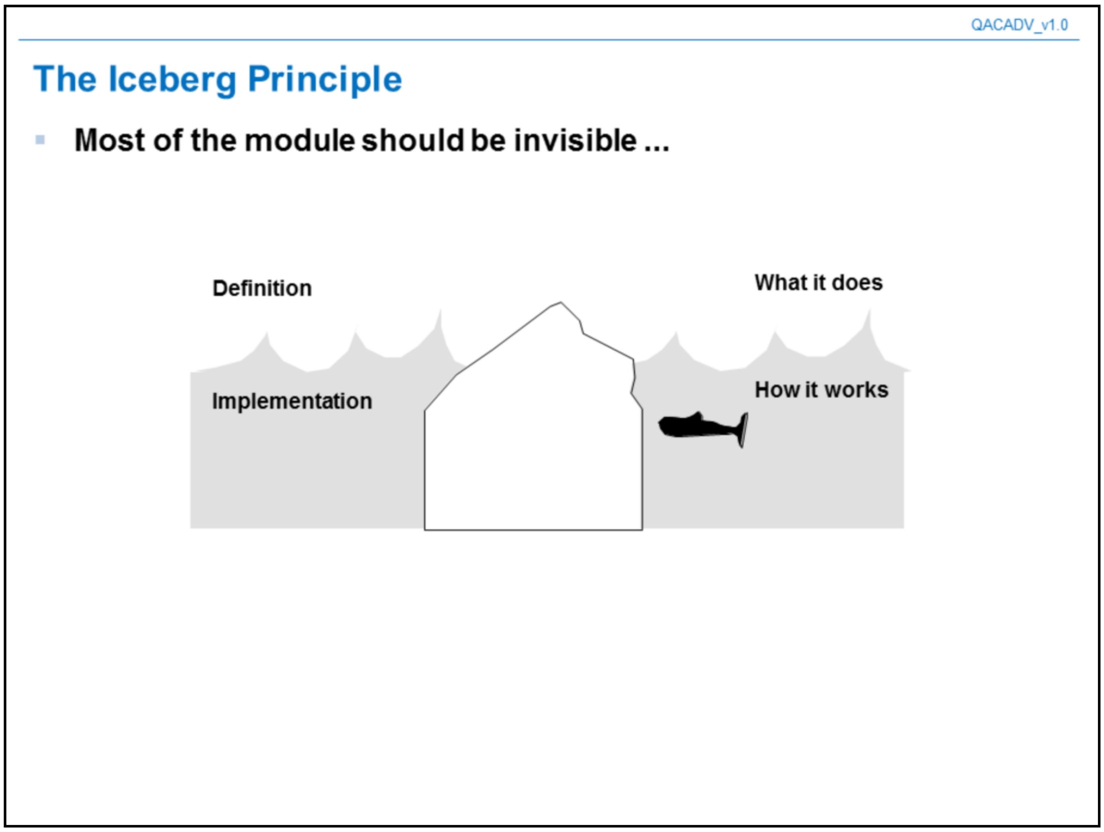
QACADV_v1.0

Module Definitions & Implementation

- **The following system for modular development is borrowed from the Modula-2 language**
- **A program is composed of modules, each having a definition and an implementation**
  - Definition: public view of the module
  - Implementation: the internals
- **The main module has an implementation but no definitions**

Modula-2, Ada and, to a certain extent, Pascal have this rather neat technique of interface/implementation specification built into the language. It is an attempt at information hiding; a technique employed by most object-oriented languages including C++ and Smalltalk.

It is a tool for re-use. The client or user programmer can make use of the module or set of modules. He or she is provided with the interface, can use the functionality, and is totally unaware of the how the processes are implemented internally.

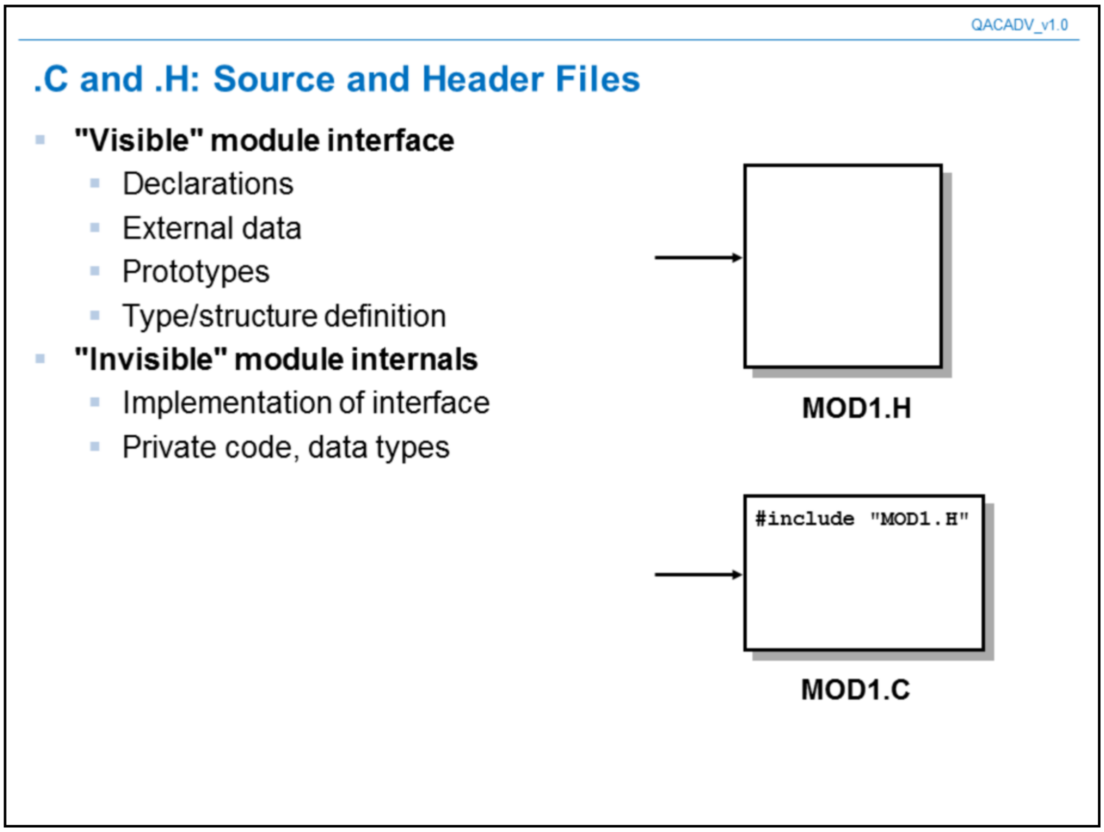

QACADV_v1.0

The Iceberg Principle

- **Most of the module should be invisible ...**

Definition
What it does
Implementation
How it works

The principle is simple: what the client doesn't know, the client probably doesn't need to know. For most of us, it's similar to using a car; we drive it, but we don't need to know how it works. Naturally, there are the car designers and car mechanics of the software industry!



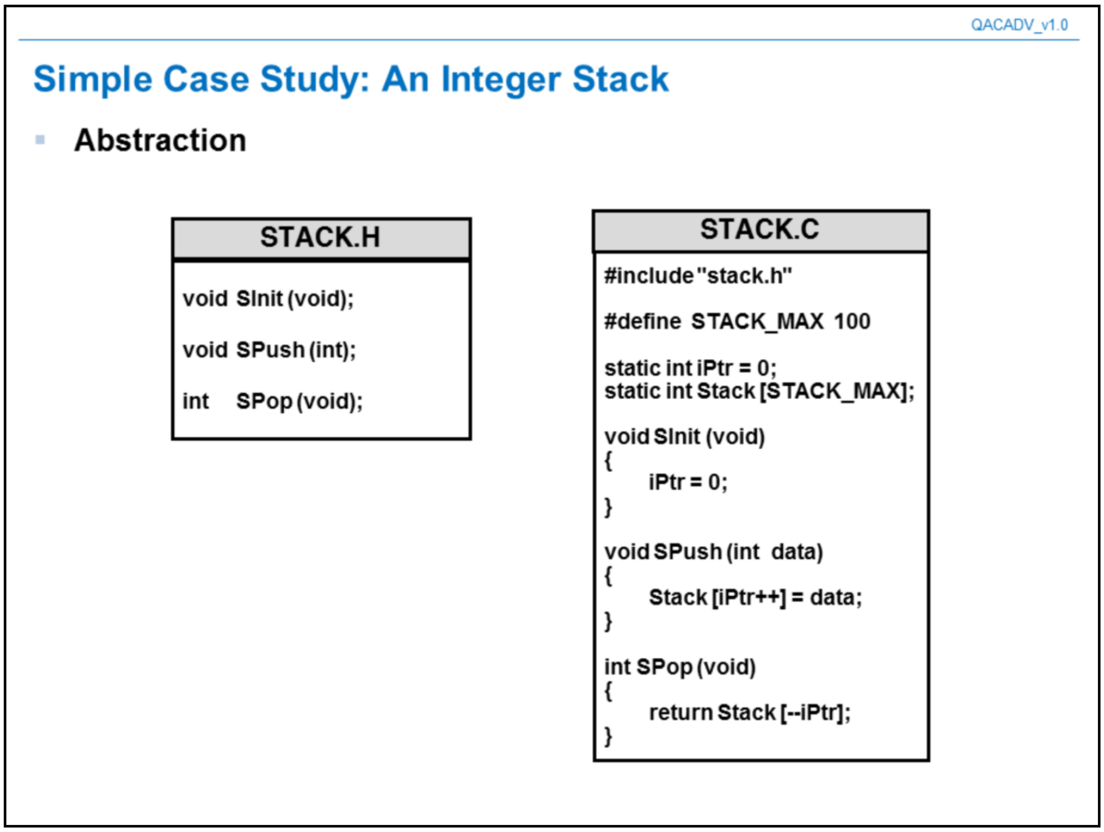
QACADV_v1.0

.C and .H: Source and Header Files

- **"Visible" module interface**
  - Declarations
  - External data
  - Prototypes
  - Type/structure definition
- **"Invisible" module internals**
  - Implementation of interface
  - Private code, data types

This is achieved in C through the use of header files. Even the novice C programmer has used the standard I/O functions, even if it is just to display "Hello World". The principle has been used for this simple process: the header file `stdio.h` holds all the goodies, and we can forget about the function implementations.

What the developer has to do is to emulate this process, i.e. to create a module which makes use of a header file. Put all the global declarations (the `extern`s), the function prototypes, the `typedef`s and the `struct`/`union` templates in the header, and place the global (possibly `static`) definitions and function bodies in the C file. A module consists of its interface in `module.h` and its implementation in `module.c`.



QACADV_v1.0

Simple Case Study: An Integer Stack

- **Abstraction**

STACK.H

```
void SInit (void);

void SPush (int);

int    SPop (void);
```

STACK.C

```
#include "stack.h"

#define  STACK_MAX  100

static int iPtr = 0;
static int Stack [STACK_MAX];

void SInit (void)
{
        iPtr = 0;
}

void SPush (int  data)
{
        Stack [iPtr++] = data;
}

int SPop (void)
{
        return Stack [--iPtr];
}
```

This process is often referred to as *abstraction*:

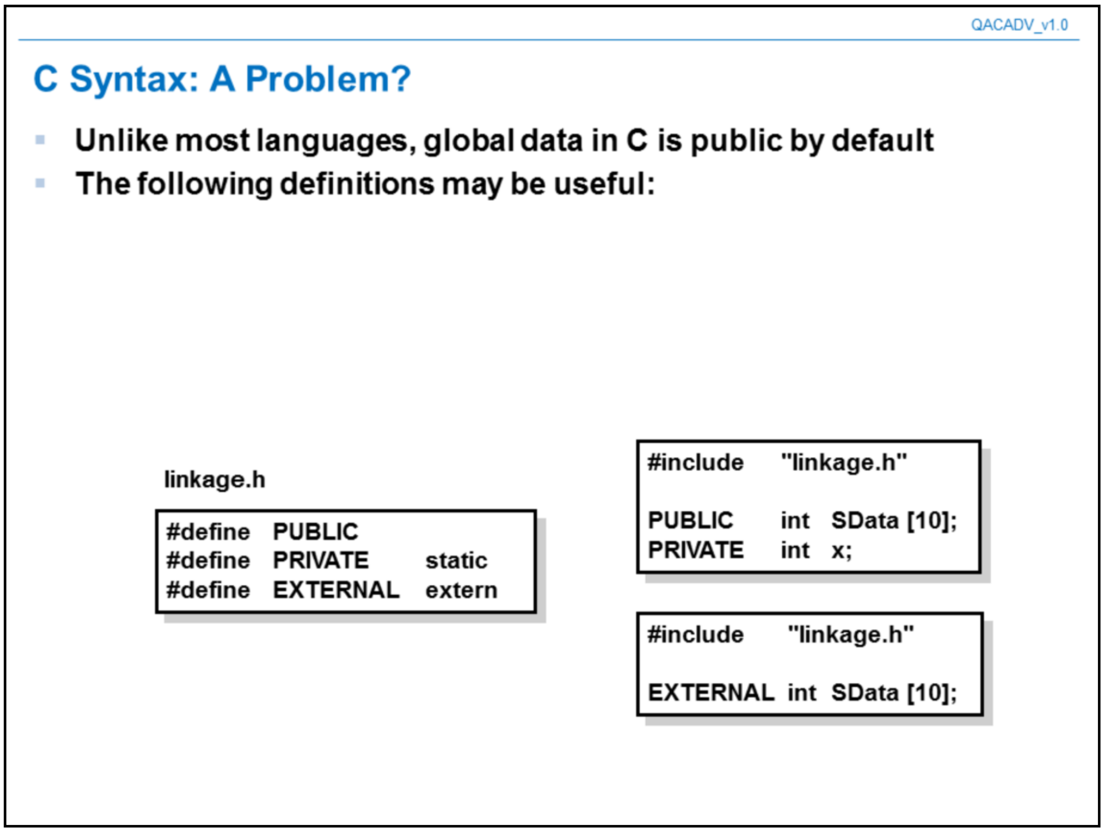
> "*The act of giving the user of an object only the details required. All other properties are hidden or ignored. The abstraction is usually context sensitive.*"

The example above illustrates a simple abstract data type - the *stack*. The interface is simply the prototypes of the functions which implement the two major behavioural properties of a stack, namely `SPush` and `SPop`, and a third function which initialises the stack.

The implementation of a stack contains the function definitions and the appropriate data. The `static` keyword is used to make the data private to the `stack.c` module - this implies that the client or user code cannot access this data directly, but must access it through the appropriate functions. This is not only for security reasons, but it is common sense, since the stack functions should control the stack data.

One of the great advantages of the *abstract data type* approach is that the internal implementation of the stack could be changed at some later date without causing the rest of the program to be modified.

For example, we might decide that the stack is best implemented as a linked list rather than as a fixed-size array. Obviously the stack implementation functions `SInit()`, `SPush()` and `SPop()` will have to change internally to use linked list logic rather than array indexing. However, the interface of these functions will not have to be modified, and the rest of the program will not have to be changed at all.



QACADV_v1.0

C Syntax: A Problem?

- Unlike most languages, global data in C is public by default
- The following definitions may be useful:

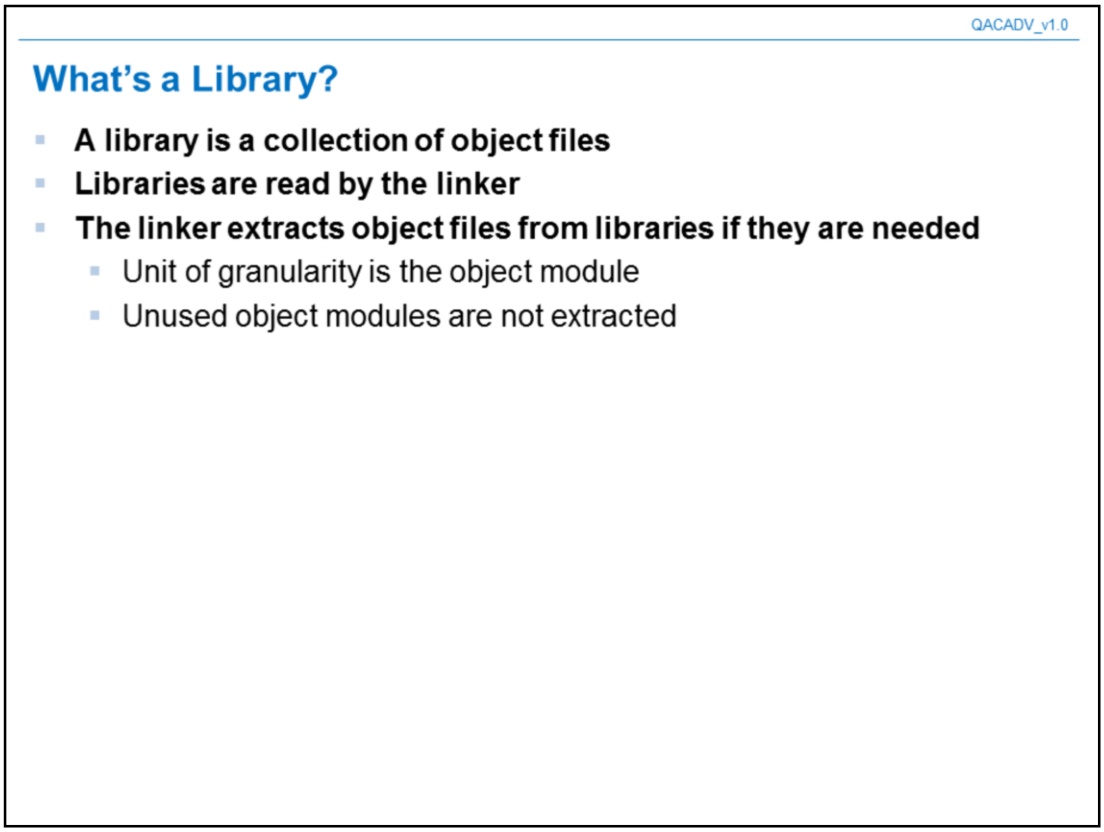
linkage.h

```
#define   PUBLIC
#define   PRIVATE      static
#define   EXTERNAL     extern
```

```
#include     "linkage.h"

PUBLIC       int   SData [10];
PRIVATE      int   x;
```

```
#include      "linkage.h"

EXTERNAL  int   SData [10];
```

Although abstraction can be achieved to a certain extent, the language does not support it very well. We have to use several tricks to achieve what is required, and even then it is a compromise. One problem is that the C global data item is public by default, since it can be imported into any other module using the `extern` keyword. The preprocessor can be used to clarify the intention. Using the `PUBLIC`, `PRIVATE` and `EXTERNAL` tokens, the story is clear.



QACADV_v1.0

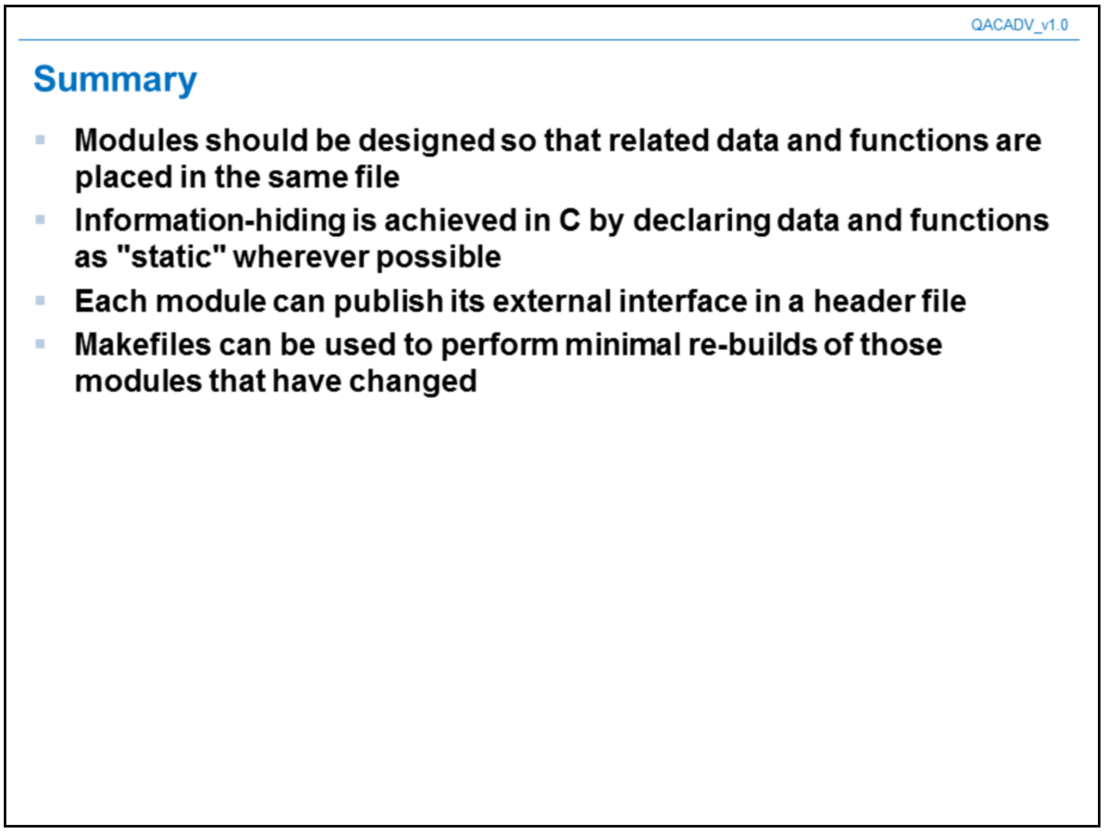
What's a Library?

- **A library is a collection of object files**
- **Libraries are read by the linker**
- **The linker extracts object files from libraries if they are needed**
  - Unit of granularity is the object module
  - Unused object modules are not extracted

A library is a set of object modules in a format which can be interpreted by the linker. Information about functions invoked by a user module is held in the user module's object file. The linker will use this information to extract the appropriate set of object modules from the library. These object modules from the library are then linked with the client module.

Clearly, the linker needs to have access to the appropriate library. This is achieved either by using default library-path settings, or by the client providing the name of the library or libraries at the link stage.

The linker extracts a complete module; it is unable simply to pull out individual function definitions. For example, if the application module invokes `printf`, the module containing `printf` is extracted. Any other functions in the module will also be included.


QACADV_v1.0

Summary

- Modules should be designed so that related data and functions are placed in the same file
- Information-hiding is achieved in C by declaring data and functions as "static" wherever possible
- Each module can publish its external interface in a header file
- Makefiles can be used to perform minimal re-builds of those modules that have changed

C provides us with a number of facilities and language constructs to achieve a modular programming approach, where related data and functions can be grouped together in a single file. This offers several advantages:

- Increased intellectual integrity of the system.
- Decoupled components, which helps to reduce the so-called *ripple effect* when code is changed.
- Reduction in compilation dependencies.

The `static` keyword should be used wherever possible to shield the internal details of a module from the "outside world". This makes it easier to change the internal implementation of these modules without having to modify the rest of the program.

The public interface can be published in a header file so that other modules can make use of these definitions. The temptation to have a single, all encompassing header file which contains all the prototypes and all the external data declarations for the whole system should be avoided if at all possible, since it introduces too many potential dependencies between different parts of the program.

The `MAKE` utility, provided with most modern C compilers, allows large systems to be built and modified in a controlled manner. The dependency lists in the makefile indicate which components of the program need to be re-built when files are changed. Version control systems can also be invaluable for keeping track of the various versions of modules during the development and maintenance life cycle of a product.