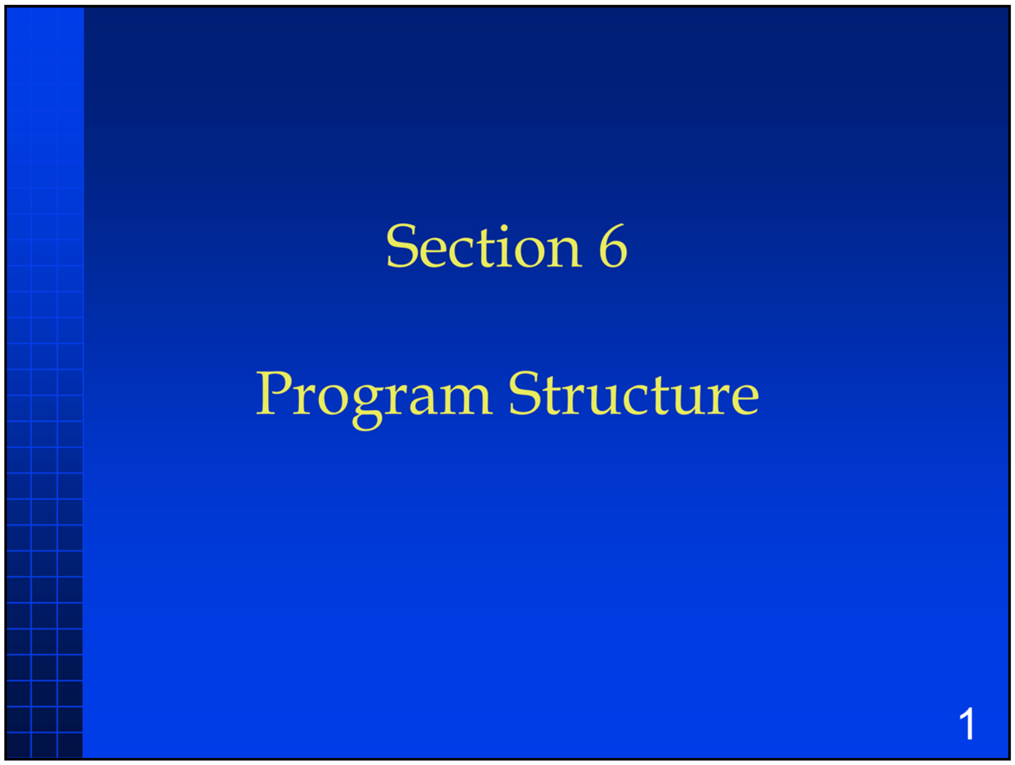


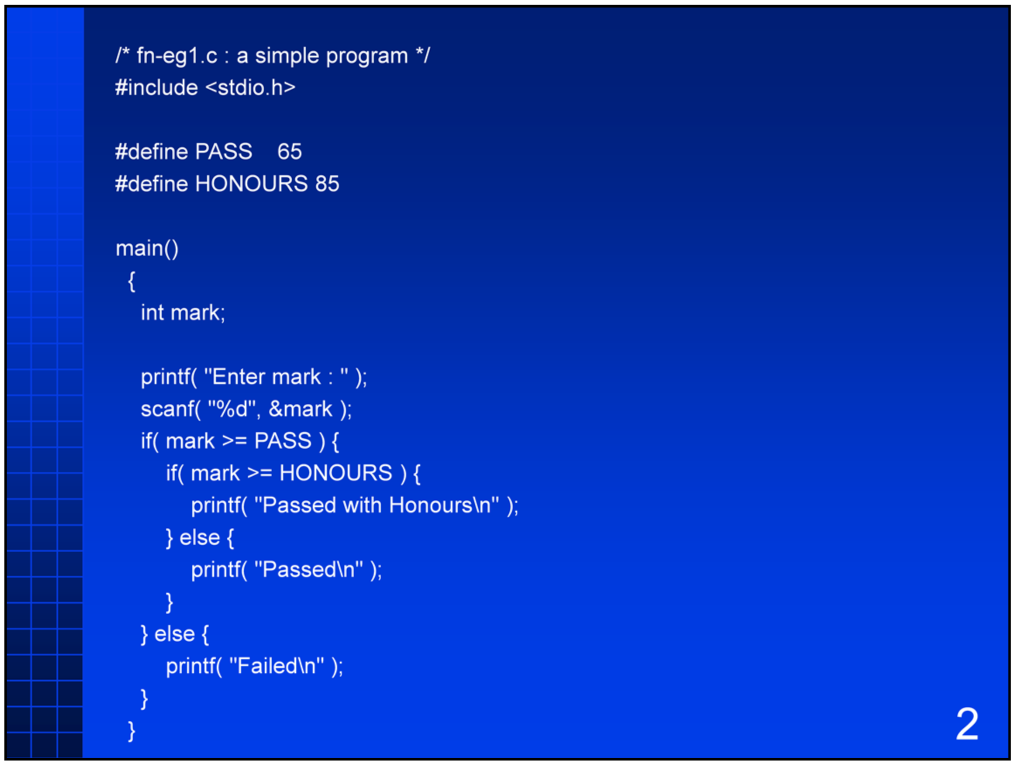
Section 6

Program Structure

in C, programs are collections of functions:

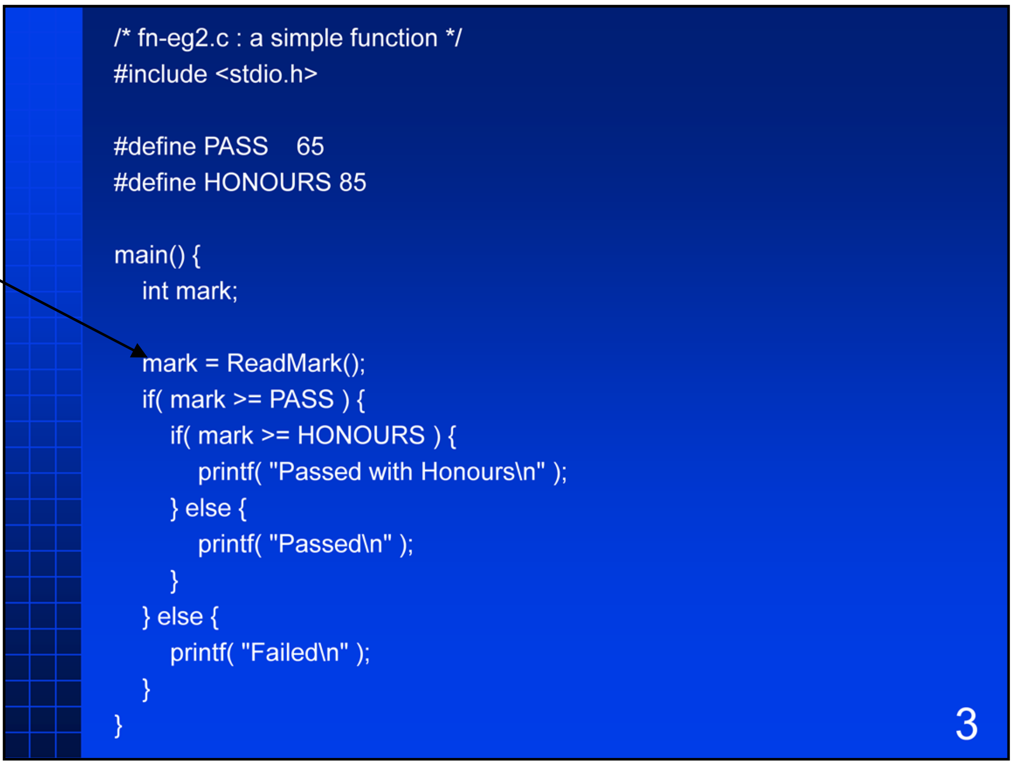
some functions we write, some are provided

some functions return values, some do not. even if they do, we can ignore/discard value



```
/* fn-eg1.c : a simple program */
#include <stdio.h>

#define PASS    65
#define HONOURS 85

main()
 {
   int mark;

   printf( "Enter mark : " );
   scanf( "%d", &mark );
   if( mark >= PASS ) {
       if( mark >= HONOURS ) {
           printf( "Passed with Honours\n" );
       } else {
           printf( "Passed\n" );
       }
   } else {
       printf( "Failed\n" );
   }
 }
```

typical program: reads a value, performs a computation
currently all contained within a function called main.
note ugly format -- saves space



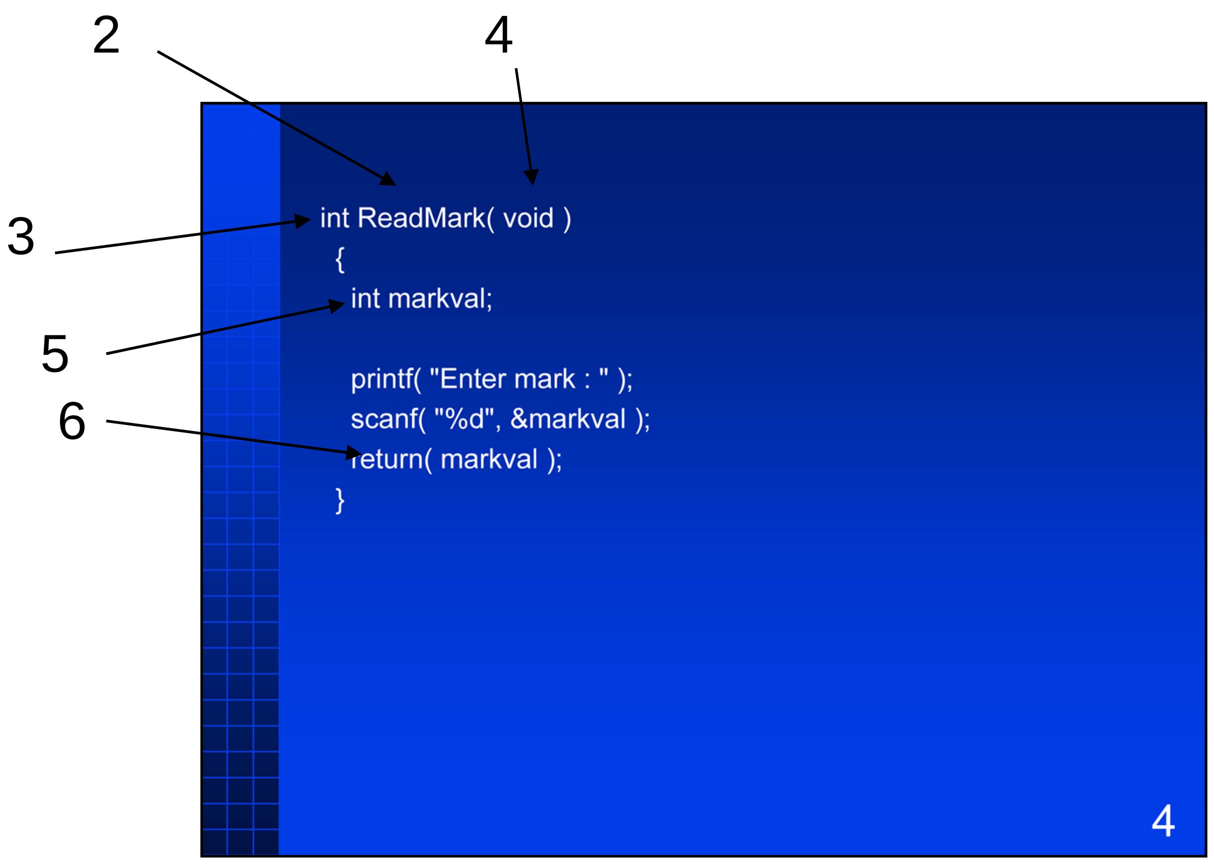
1

```
/* fn-eg2.c : a simple function */
#include <stdio.h>

#define PASS    65
#define HONOURS 85

main() {
   int mark;

   mark = ReadMark();
   if( mark >= PASS ) {
      if( mark >= HONOURS ) {
         printf( "Passed with Honours\n" );
      } else {
         printf( "Passed\n" );
      }
   } else {
      printf( "Failed\n" );
   }
}
```

3

now, isolate the reading operation: put into own function

1) imagine that we have a function “ReadMark” that will prompt for and read number, then return number. this is how we would invoke it



```
int ReadMark( void )
 {
  int markval;

  printf( "Enter mark : " );
  scanf( "%d", &markval );
  return( markval );
 }
```

2) definition of function. note similar structure to that of main

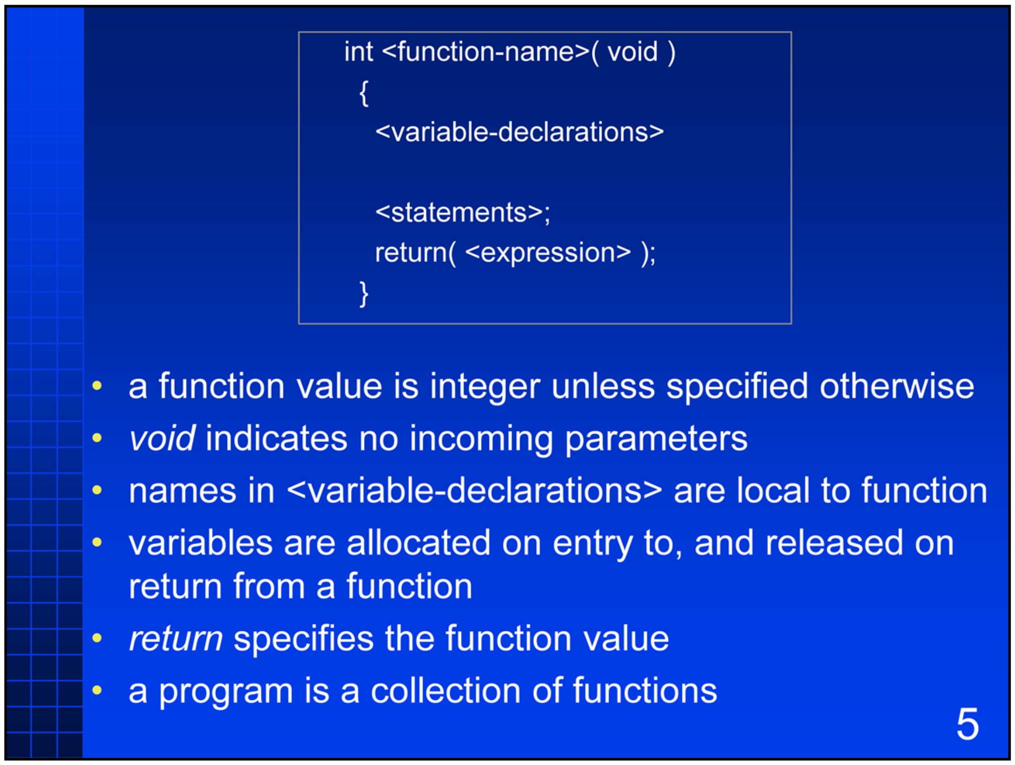
  - located in same source-file, two fns in same compilation unit (compiled together)

3) return type is integer. note similarity of function declaration to integer declaration.

4) incoming parameters. none in this case, so kwd “void”

5) variable definition. can be accessed only within this function.

6) return statement: executable, gives value to be returned



```
int <function-name>( void )
  {
    <variable-declarations>

    <statements>;
    return( <expression> );
  }
```

- a function value is integer unless specified otherwise
- *void* indicates no incoming parameters
- names in <variable-declarations> are local to function
- variables are allocated on entry to, and released on return from a function
- *return* specifies the function value
- a program is a collection of functions

generic view:

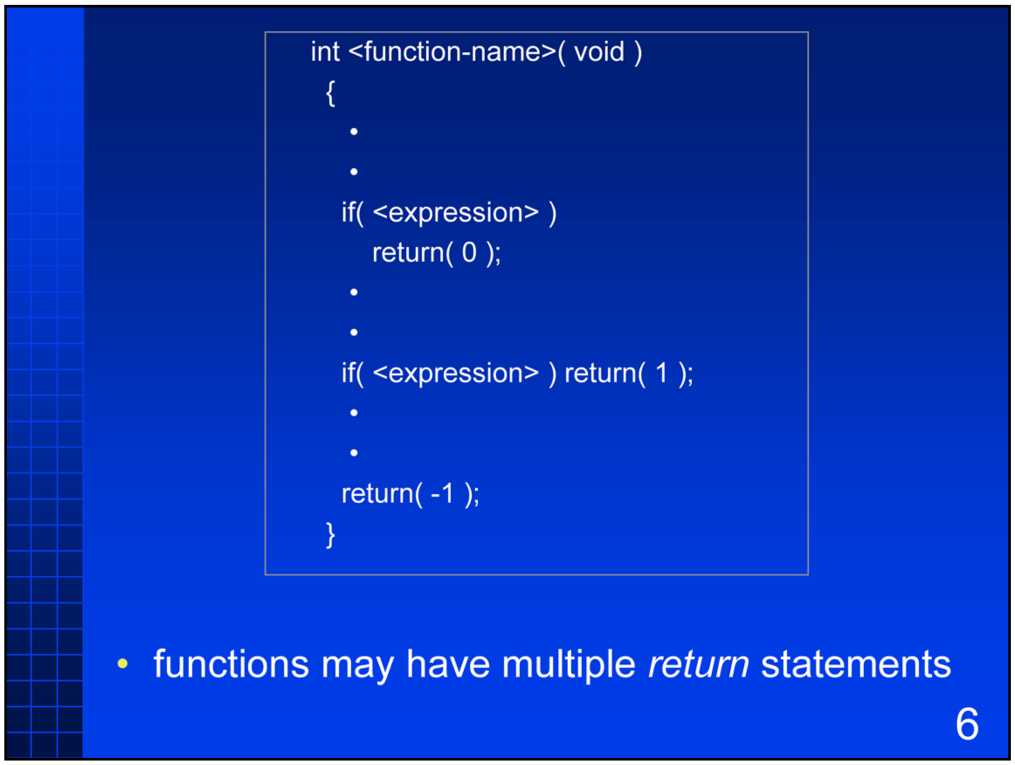
- int is the default type of a function, main returns an int.

- our definition of main should specify void --- acceptable for historical reasons

- vars are local: cannot refere to mark in readvmark, cannot refer to markval in main

- no history between invocations, created and destroyed each invocation

- every program has a main, defined as starting point.

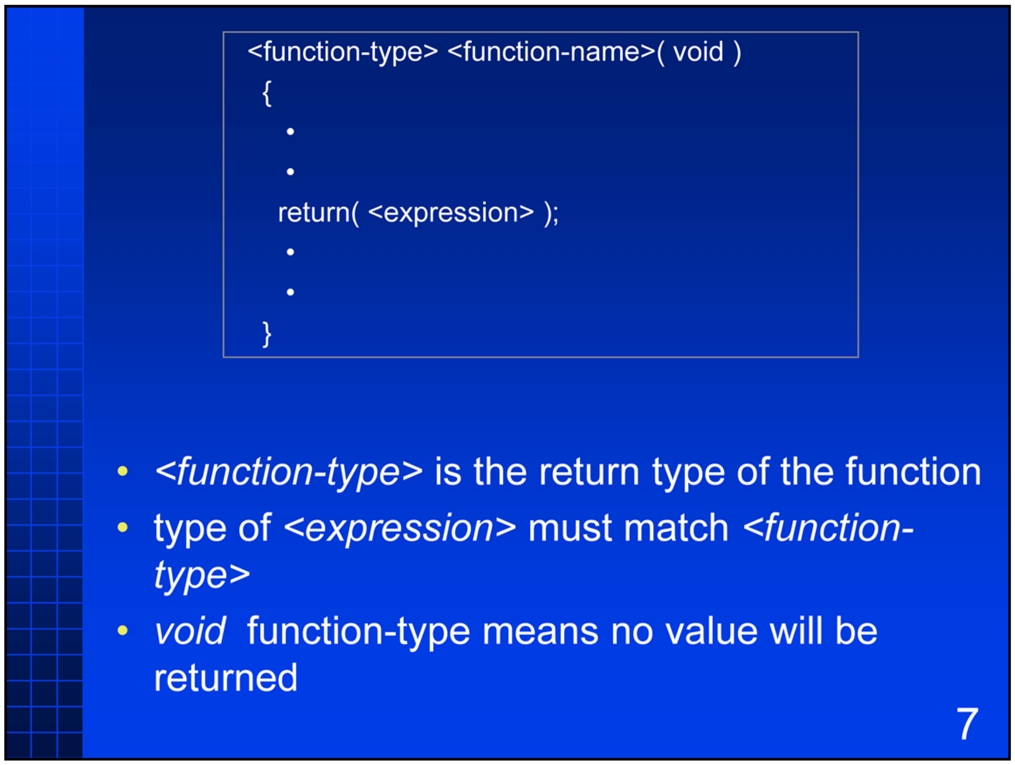


```
int <function-name>( void )
 {
   •
   •
   if( <expression> )
       return( 0 );
   •
   •
   if( <expression> ) return( 1 );
   •
   •
   return( -1 );
 }
```

- functions may have multiple *return* statements

6

- to emphasise that return is executable
- dubious engineering? useful for error-handling



```
<function-type> <function-name>( void )
 {
    •
    •
   return( <expression> );
    •
    •
 }
```

- *<function-type>* is the return type of the function
- type of *<expression>* must match *<function-type>*
- *void* function-type means no value will be returned

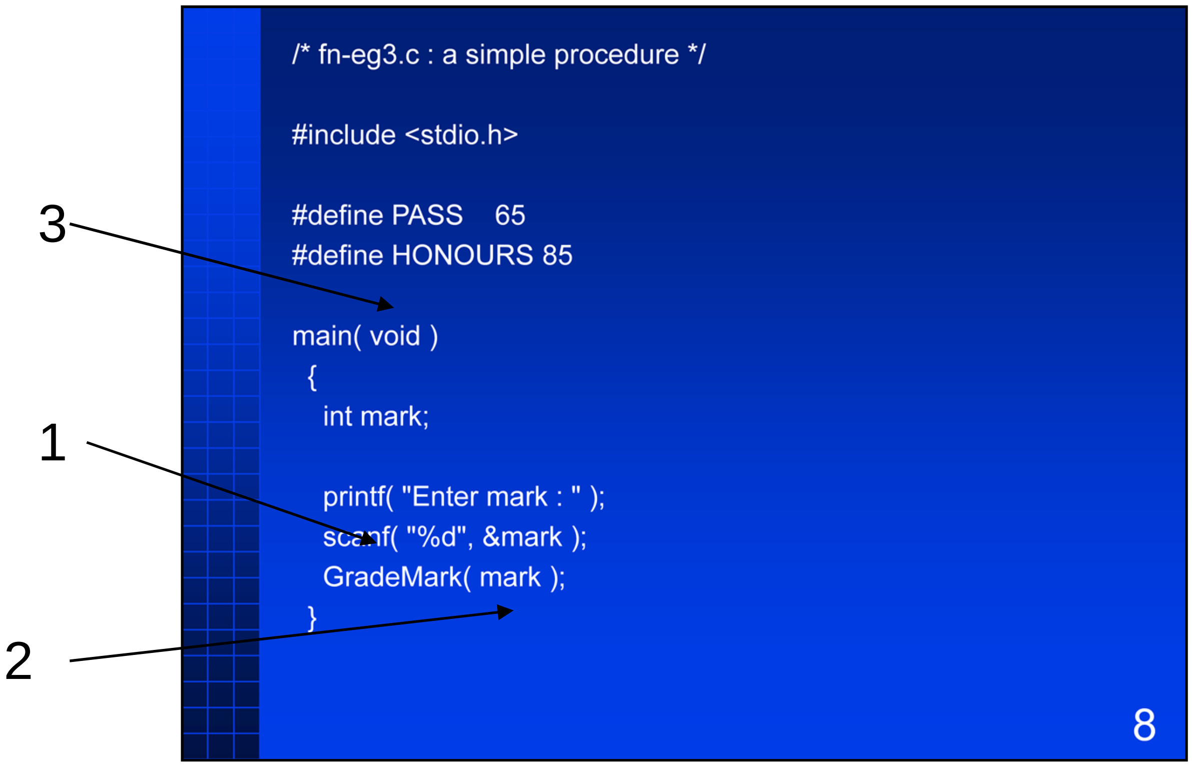
7

general for for parameterless functions

- function type can be any type: int char, unsigned long, double etc.

- compilers will not enforce type-match, may issue warnings in some cases (eg constants)

- can use void keyword for return-type to indicate no type returned, often called a “procedure”



```
/* fn-eg3.c : a simple procedure */

#include <stdio.h>

#define PASS    65
#define HONOURS 85

main( void )
 {
   int mark;

   printf( "Enter mark : " );
   scanf( "%d", &mark );
   GradeMark( mark );
 }
```

example of a procedure (function that returns nothing)

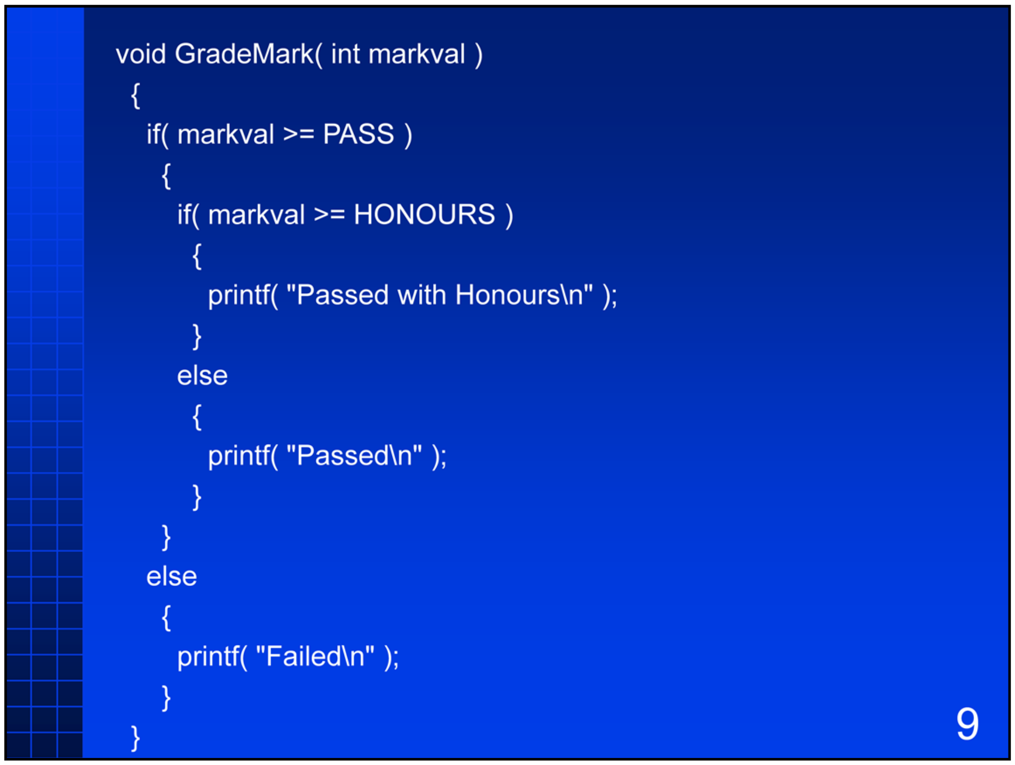
procedures useful for side-effects (since they don’t return a value).

in this case, procedure to grade a mark: mark will be passed as a parameter

1) invocation of procedure: like function invocation with discarded result: same as printf etc.

2) pass parameter

3) corrected version of definition of main: void parameter list

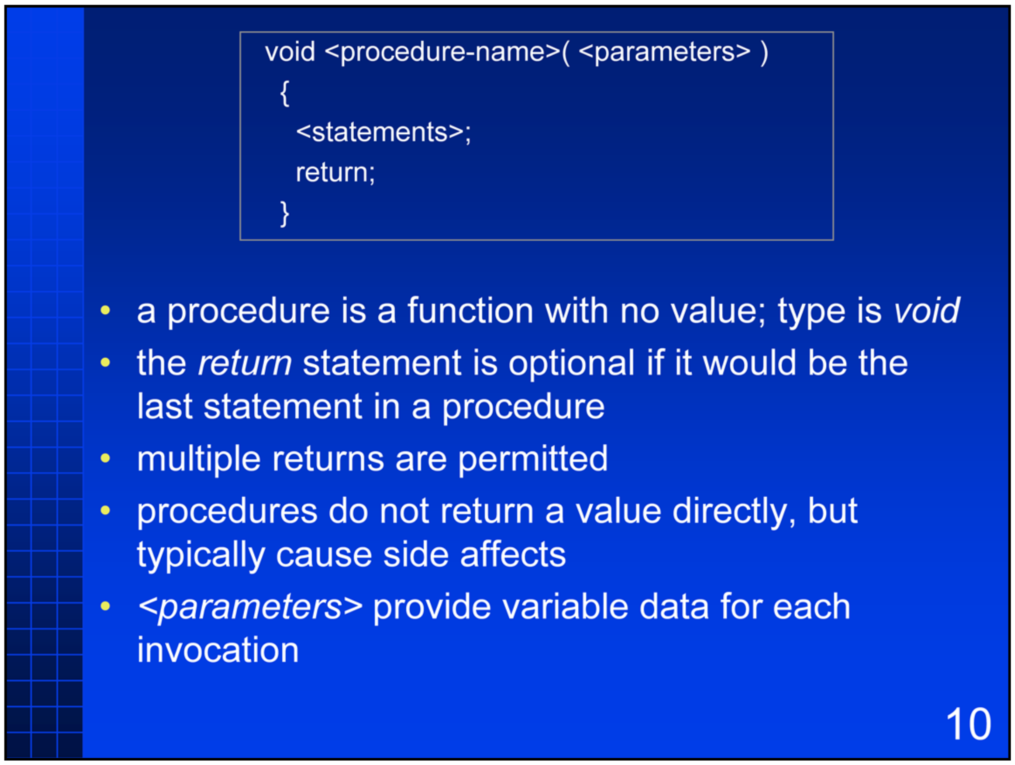


```
void GradeMark( int markval )
 {
  if( markval >= PASS )
   {
    if( markval >= HONOURS )
     {
      printf( "Passed with Honours\n" );
     }
    else
     {
      printf( "Passed\n" );
     }
   }
  else
   {
    printf( "Failed\n" );
   }
 }
```

procedure definition: return-type is void

parameter is one integer: “int markval” syntax like variable definition: in fact, behaves just like a variable that is initialized with value of parameter that was passed at point of invocation

parameter definition can be like any variable declaration, long, short unsigned, even const.



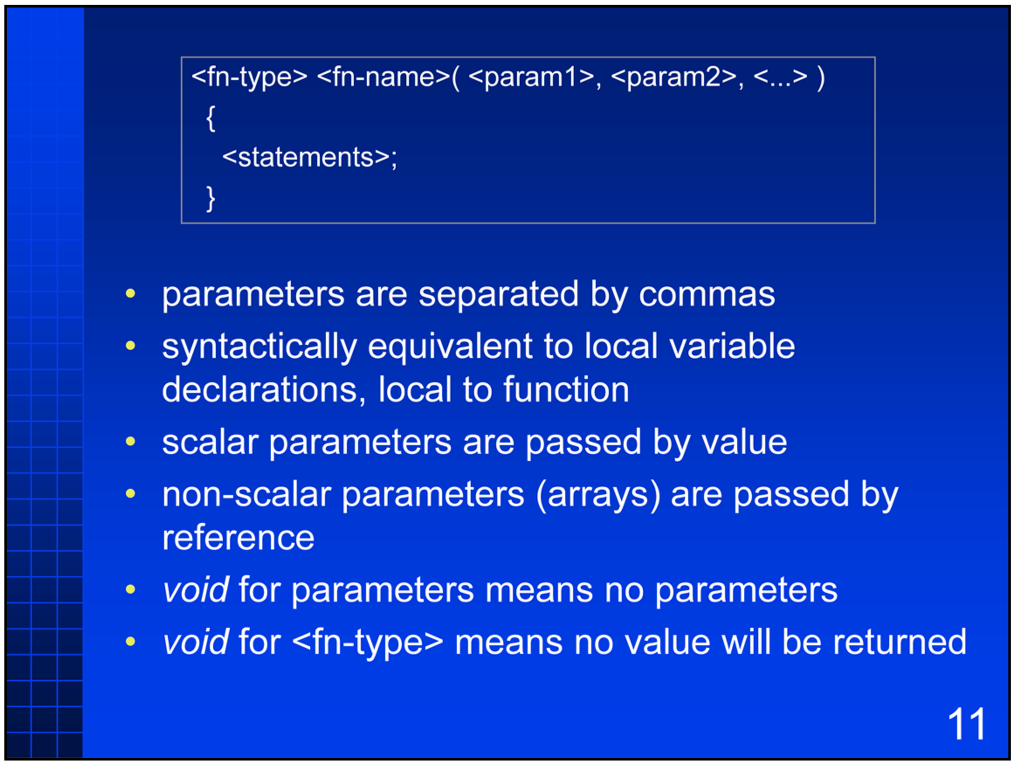
```
void <procedure-name>( <parameters> )
 {
   <statements>;
   return;
 }
```

- a procedure is a function with no value; type is *void*
- the *return* statement is optional if it would be the last statement in a procedure
- multiple returns are permitted
- procedures do not return a value directly, but typically cause side affects
- *<parameters>* provide variable data for each invocation

general form

return: since no value to return, not needed if procedure exit is at end of definition (single exit)

however, can use multiple returns (like functions) for control-flow purposes (eg error returns)



```
<fn-type> <fn-name>( <param1>, <param2>, <...> )
 {
   <statements>;
 }
```

- parameters are separated by commas
- syntactically equivalent to local variable declarations, local to function
- scalar parameters are passed by value
- non-scalar parameters (arrays) are passed by reference
- *void* for parameters means no parameters
- *void* for <fn-type> means no value will be returned

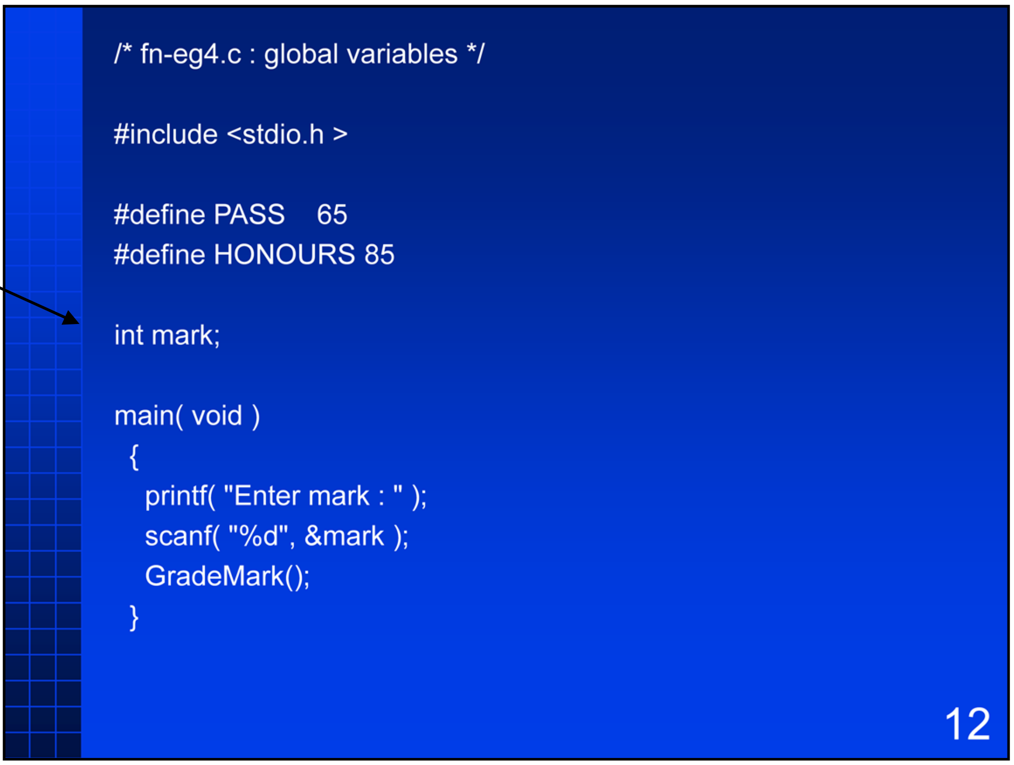
general form: function type (possibly void), function name and parameter list (void indicates none)

multiple parameters in a list, separated by commas

parameter linkage: scalars (all manner of integers; floats) passed by value, parameter behaves like an initialized local varibale

non-scalars (eg arrays) passed by reference (address of variable is passed): function can change value -- more on this later

1



```
/* fn-eg4.c : global variables */

#include <stdio.h >

#define PASS    65
#define HONOURS 85

int mark;

main( void )
 {
   printf( "Enter mark : " );
   scanf( "%d", &mark );
   GradeMark();
 }
```

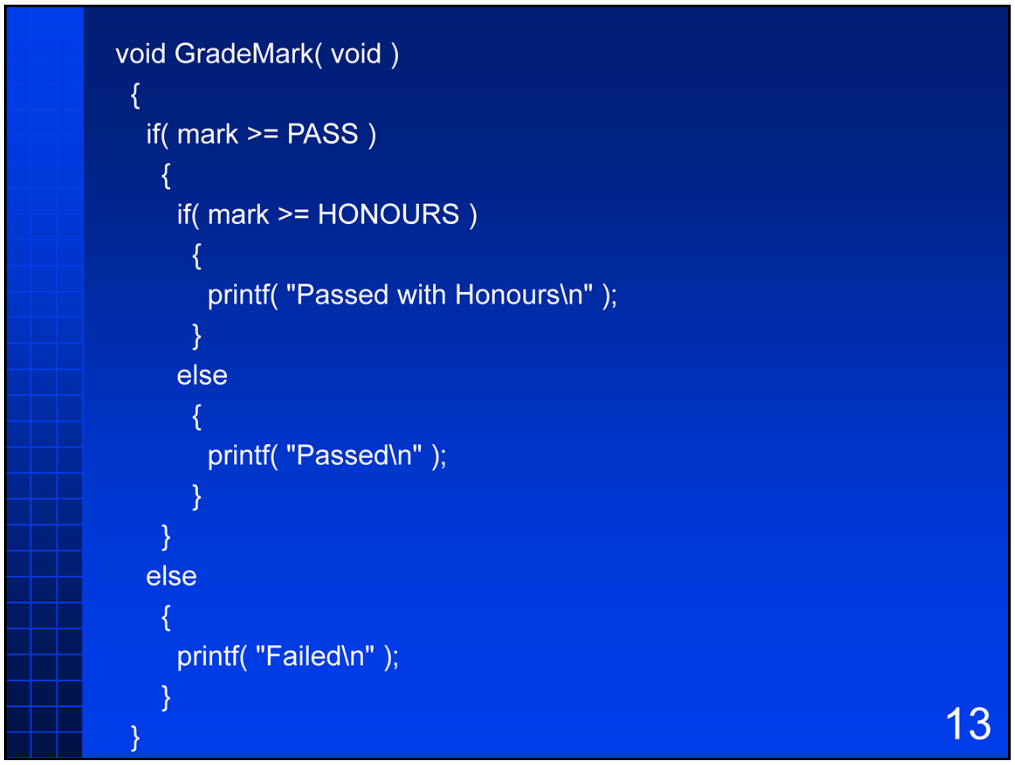
all previous examples, variable all local; no sharing of data between functions, except for parameters and return values

want to have variables accessible everywhere, in any function

called global variables

1) global variable “mark”, syntax same, defined outside any function

said to have “program scope” (accessible anywhere in program); what we’ve had up to now is “function scope” (accessible inside a function)

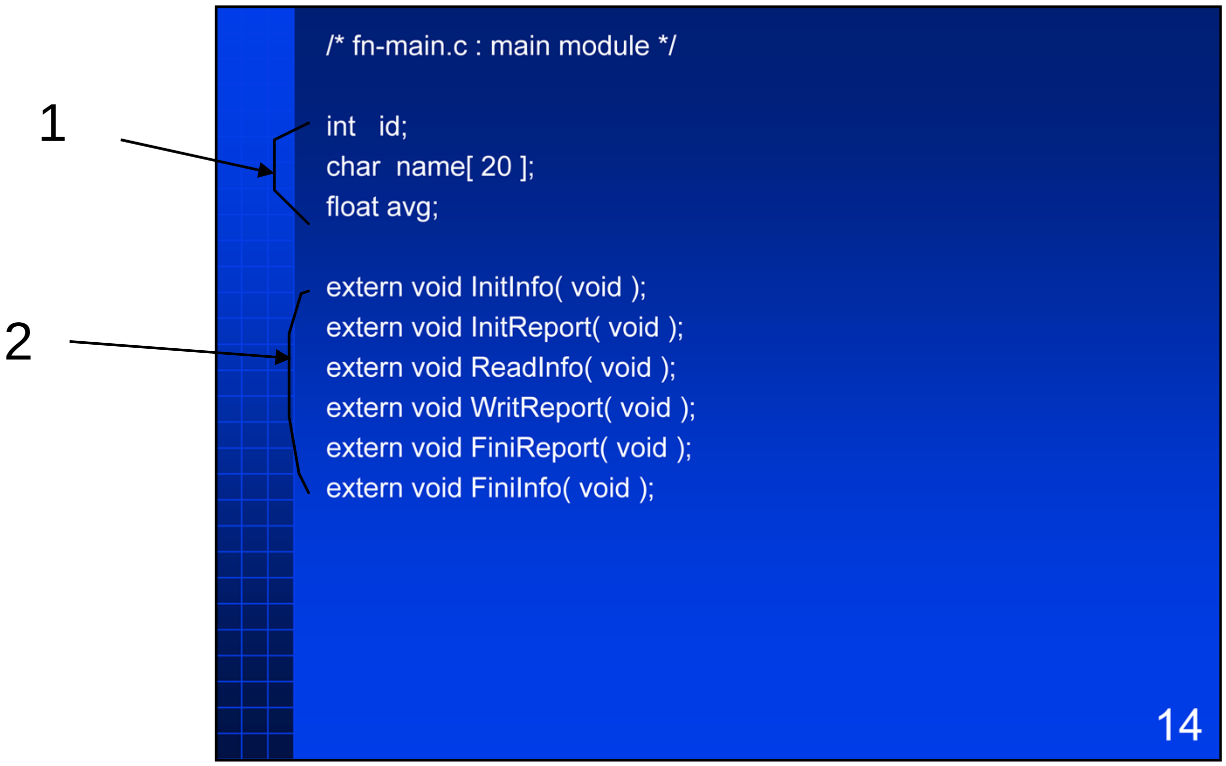


```
void GradeMark( void )
 {
   if( mark >= PASS )
     {
       if( mark >= HONOURS )
         {
           printf( "Passed with Honours\n" );
         }
       else
         {
           printf( "Passed\n" );
         }
     }
   else
     {
       printf( "Failed\n" );
     }
 }
```

definition of grademark: no parameters, no return value; access global variable

nothing but a big side-effect

something about order of declaration?



```
/* fn-main.c : main module */

int   id;
char  name[ 20 ];
float avg;

extern void InitInfo( void );
extern void InitReport( void );
extern void ReadInfo( void );
extern void WritReport( void );
extern void FiniReport( void );
extern void FiniInfo( void );
```

Up to now, all source contained in a single file.

In real world, programs composed of many files, often called modules. take a look at some of these issues.

Notes show a filename comment at top of source-files.

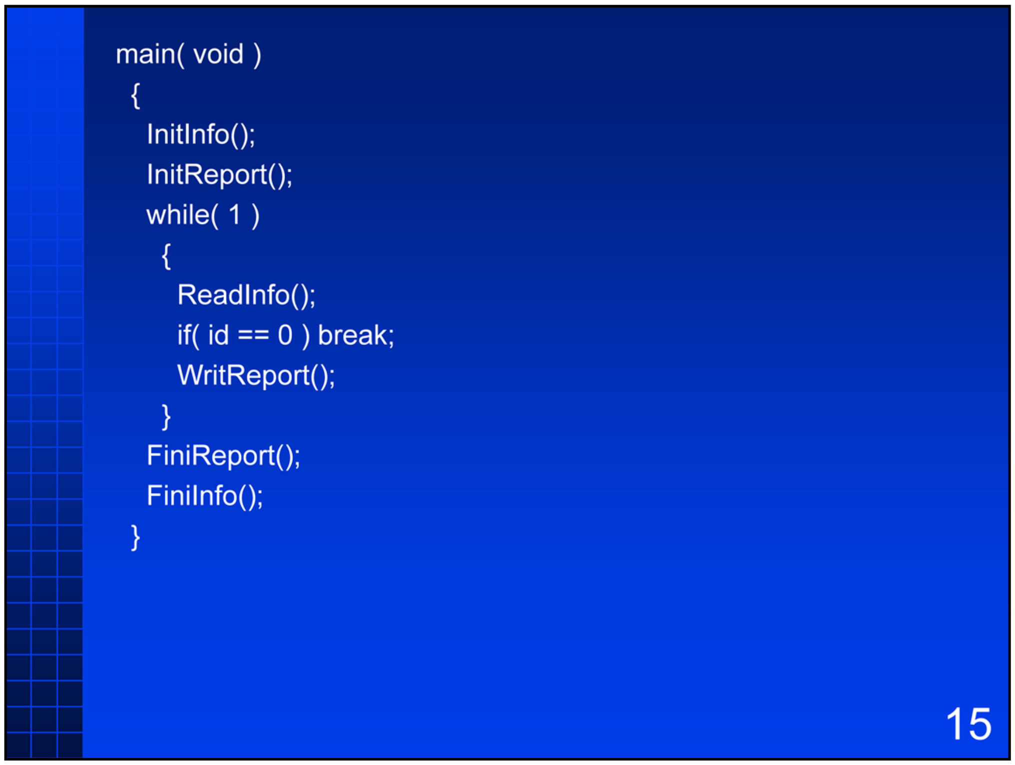
Big part of the issue of compulation-unit management is controlling the visibility of things, saying what can be seen where. Have to understand difference between definition and declaration.

Definition say what the thing is, what its scope is, and defines content (reserves space or lists statements).

Declaration says what it is and what its scope is, omits content. So:

1) definition of program-scope (global) vars. storage is reserved here. visible everywhere.

2) function declarations: extern means “globally visible”, but not statements given here, implies must be elsewhere

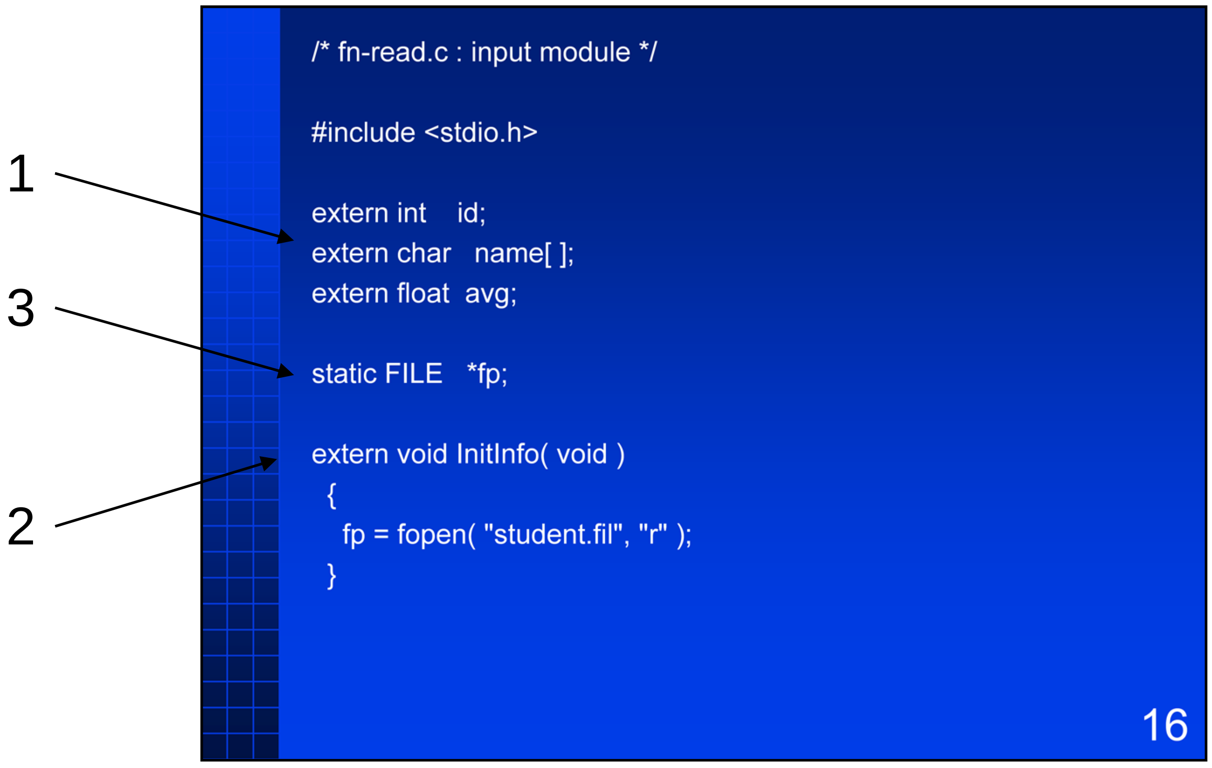


```
main( void )
 {
   InitInfo();
   InitReport();
   while( 1 )
    {
      ReadInfo();
      if( id == 0 ) break;
      WritReport();
    }
   FiniReport();
   FiniInfo();
 }
```

definition of main (same source file as previous slide)

roughly speaking, everything must be declared or defined before referenced. if not, int is assumed to be type -- if subsequent declaration or definition is different, problem.

Note btw, no standard header files in this module.



```
/* fn-read.c : input module */

#include <stdio.h>

extern int    id;
extern char   name[ ];
extern float  avg;

static FILE   *fp;

extern void InitInfo( void )
  {
    fp = fopen( "student.fil", "r" );
  }
```

16

next module (sourcefile)

1) declarations of variables: extern means global scope and defined elsewhere (slightly different meaning)

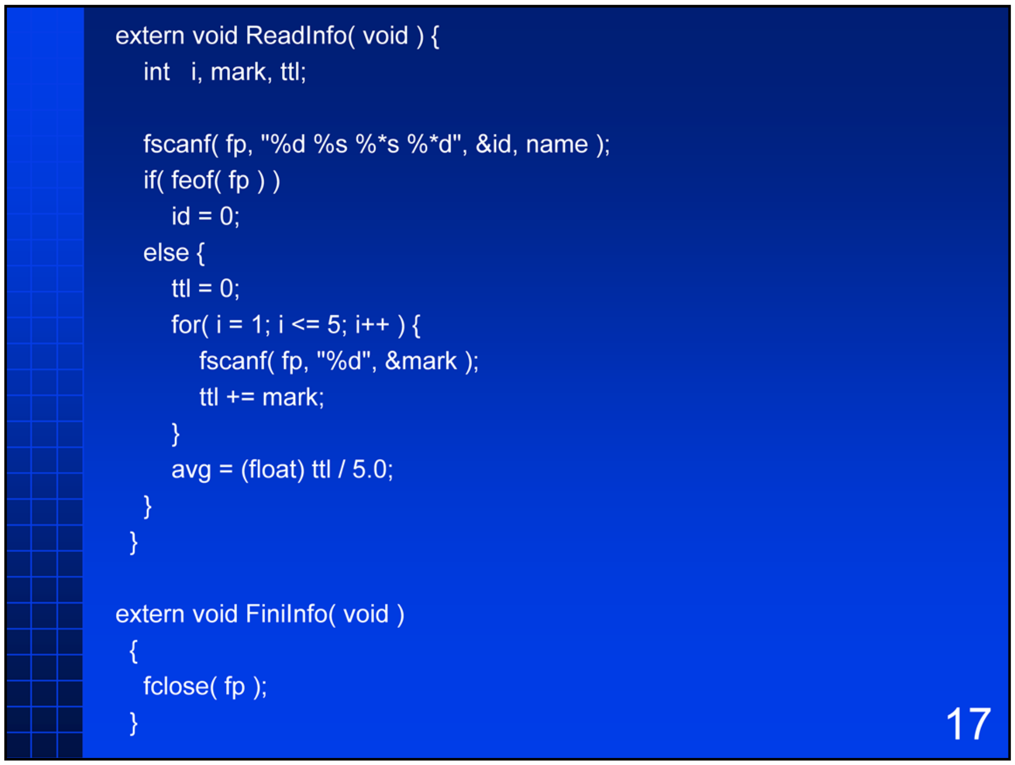
note array declaration omits size: note really needed, since storage elsewhere. just need to know its an array.

2) definition of a function that has program scope (global). extern says global, presence of statement block means that this is the definition.

Use of extern is different for variables and functions. For vars, extern means defined elsewhere, used only in declarations, not definitions.

For functions, extern means global scope. can be used in definitions and declarations (actually, its the default).

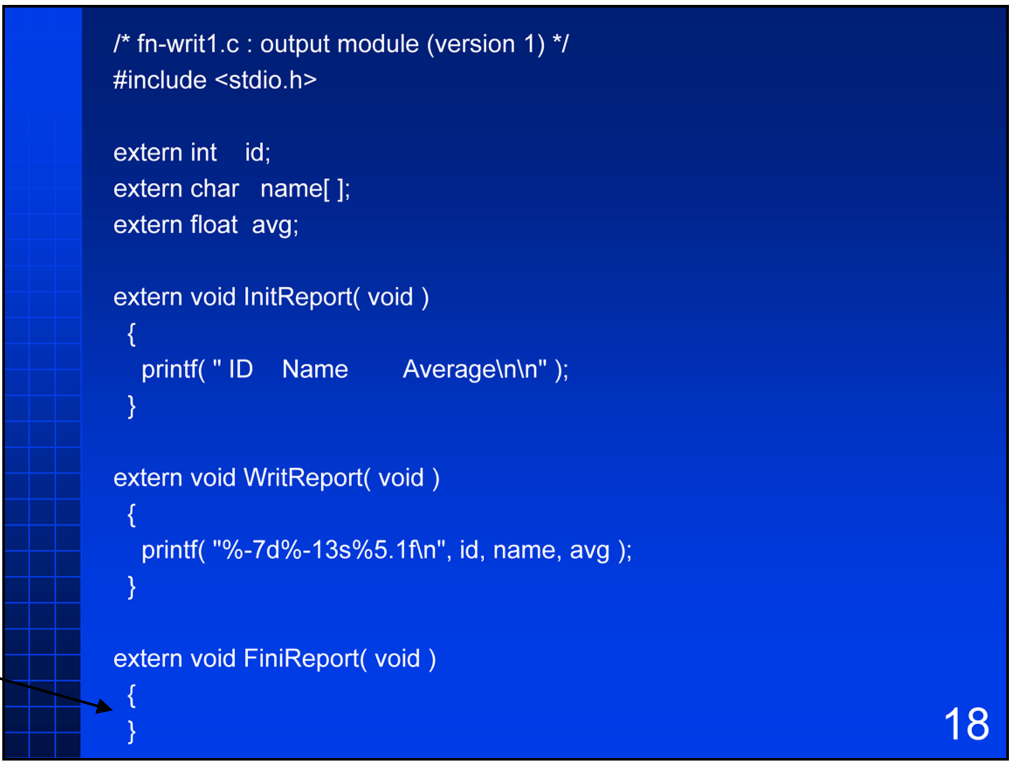
3) definition of a variable that has file scope: visible within any of the functions in the module, but not outside this module.



```
extern void ReadInfo( void ) {
    int   i, mark, ttl;

    fscanf( fp, "%d %s %*s %*d", &id, name );
    if( feof( fp ) )
        id = 0;
    else {
        ttl = 0;
        for( i = 1; i <= 5; i++ ) {
            fscanf( fp, "%d", &mark );
            ttl += mark;
        }
        avg = (float) ttl / 5.0;
    }
  }

extern void FiniInfo( void )
  {
    fclose( fp );
  }
```

more function definitions. note use of file scope variable and program scope variables.

personal coding convention: use of capitalized names etc to help distinguish scope



```
/* fn-writ1.c : output module (version 1) */
#include <stdio.h>

extern int    id;
extern char   name[ ];
extern float  avg;

extern void InitReport( void )
 {
   printf( " ID    Name        Average\n\n" );
 }

extern void WritReport( void )
 {
   printf( "%-7d%-13s%5.1f\n", id, name, avg );
 }

extern void FiniReport( void )
 {
 }
```

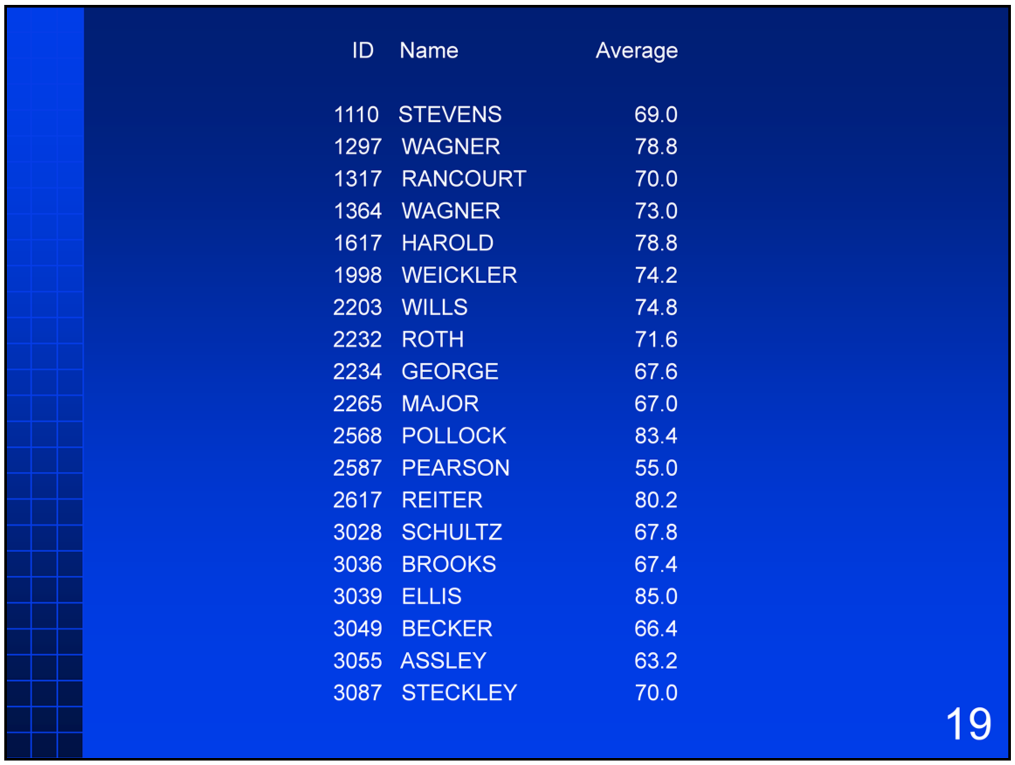
1

18

last module; provides definition of remaining functions.

another set of declarations for the global variables. Rule is, one definition and many declarations.

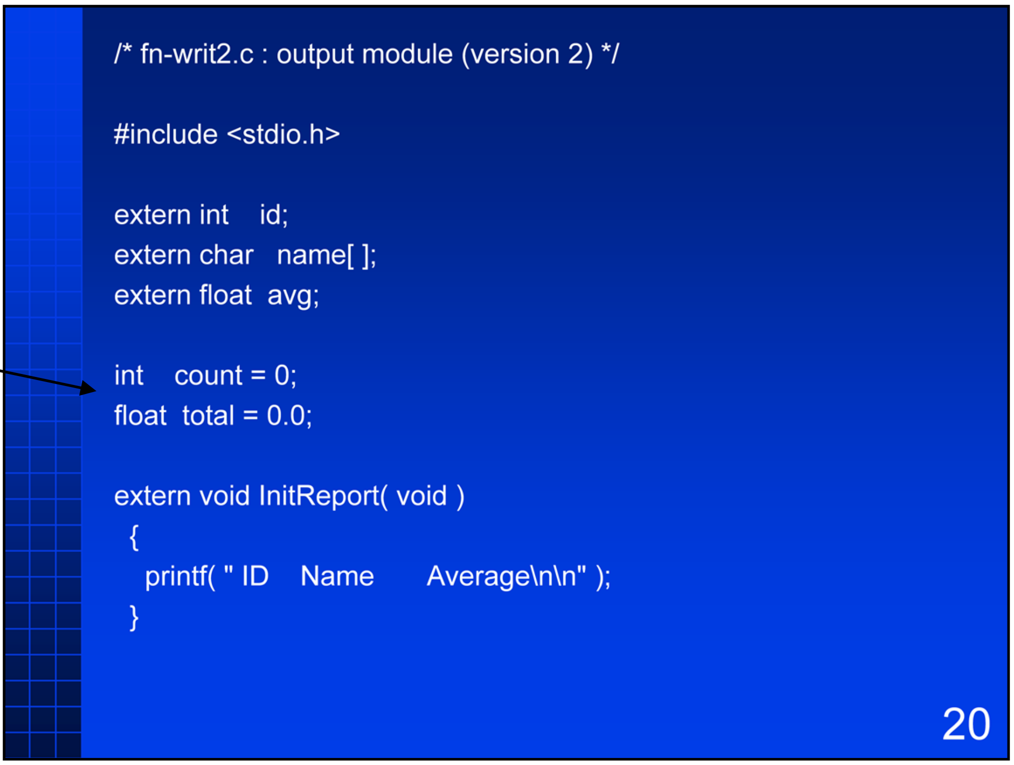
1) empty function


```
   ID   Name            Average

1110  STEVENS             69.0
1297  WAGNER              78.8
1317  RANCOURT            70.0
1364  WAGNER              73.0
1617  HAROLD              78.8
1998  WEICKLER            74.2
2203  WILLS               74.8
2232  ROTH                71.6
2234  GEORGE              67.6
2265  MAJOR               67.0
2568  POLLOCK             83.4
2587  PEARSON             55.0
2617  REITER              80.2
3028  SCHULTZ             67.8
3036  BROOKS              67.4
3039  ELLIS               85.0
3049  BECKER              66.4
3055  ASSLEY              63.2
3087  STECKLEY            70.0
```

The output to our program.

Remaining programs here are refinements of the last module. Program was organized in such a way that the output operations isolated into one module.

1,2

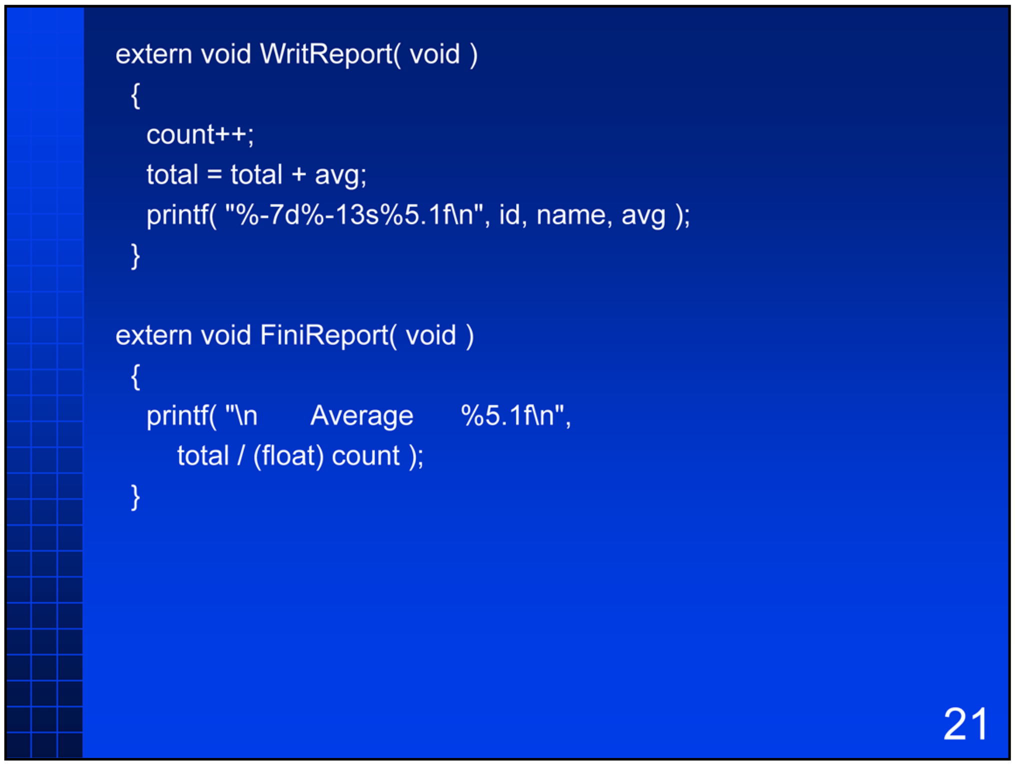


```
/* fn-writ2.c : output module (version 2) */

#include <stdio.h>

extern int    id;
extern char   name[ ];
extern float  avg;

int    count = 0;
float  total = 0.0;

extern void InitReport( void )
 {
   printf( " ID    Name       Average\n\n" );
 }
```

Refinement: compute class average. Need more variables to add up marks and count number in class.

1) definition of new variables. Definitions can occur along-side declarations, no issue.

2) these vars are initialized. initialization occurs only once in the life of the execution of a program, at load-time (as program prepared for execution).

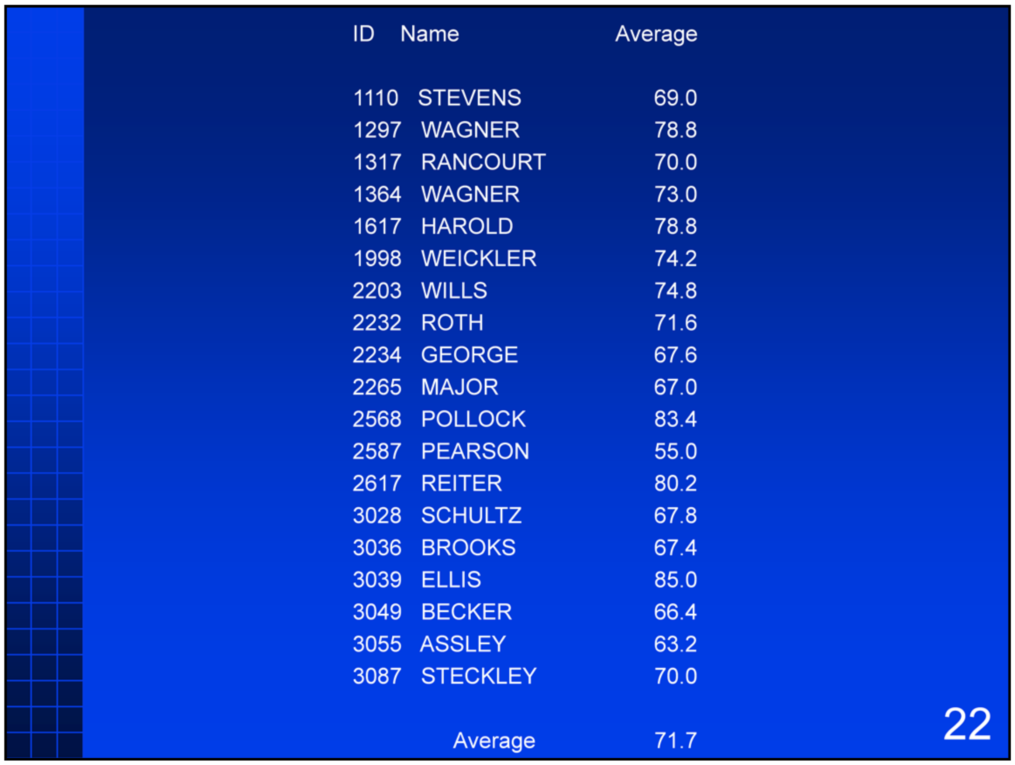


```
extern void WritReport( void )
 {
   count++;
   total = total + avg;
   printf( "%-7d%-13s%5.1f\n", id, name, avg );
 }

extern void FiniReport( void )
 {
   printf( "\n       Average       %5.1f\n",
       total / (float) count );
 }
```

references to variables.

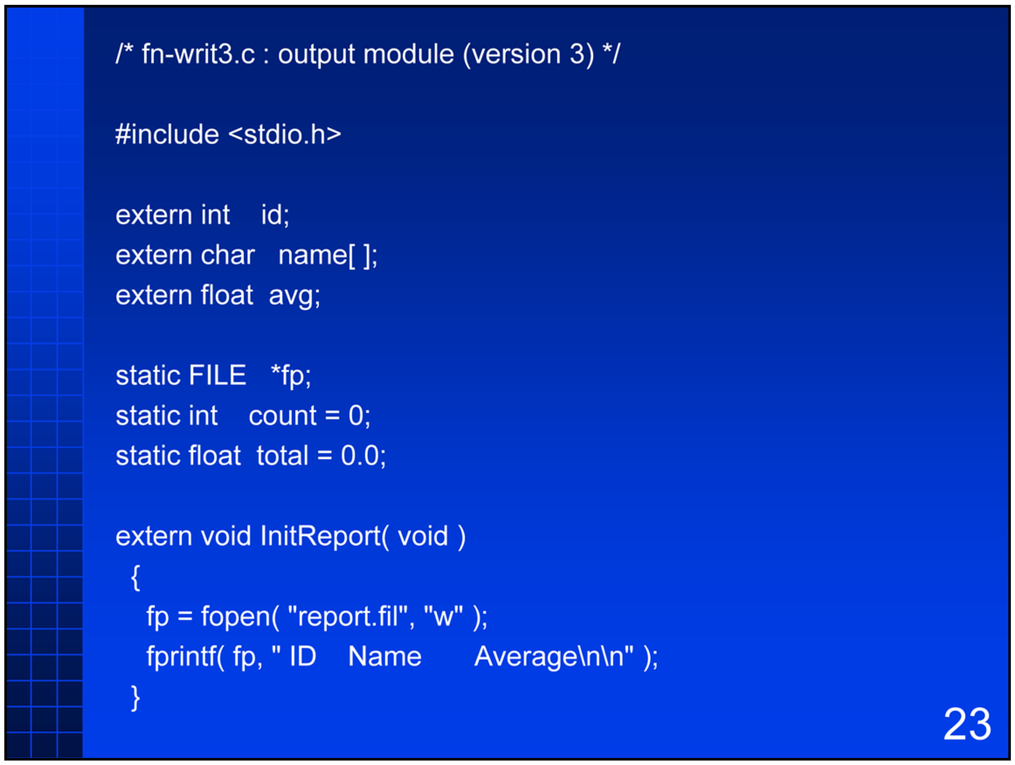
modular development: enhance functionality, replace empty function.

main program stays the same



```
ID    Name              Average

1110  STEVENS              69.0
1297  WAGNER               78.8
1317  RANCOURT             70.0
1364  WAGNER               73.0
1617  HAROLD               78.8
1998  WEICKLER             74.2
2203  WILLS                74.8
2232  ROTH                 71.6
2234  GEORGE               67.6
2265  MAJOR                67.0
2568  POLLOCK              83.4
2587  PEARSON              55.0
2617  REITER               80.2
3028  SCHULTZ              67.8
3036  BROOKS               67.4
3039  ELLIS                85.0
3049  BECKER               66.4
3055  ASSLEY               63.2
3087  STECKLEY             70.0

          Average          71.7
```

lovely new output



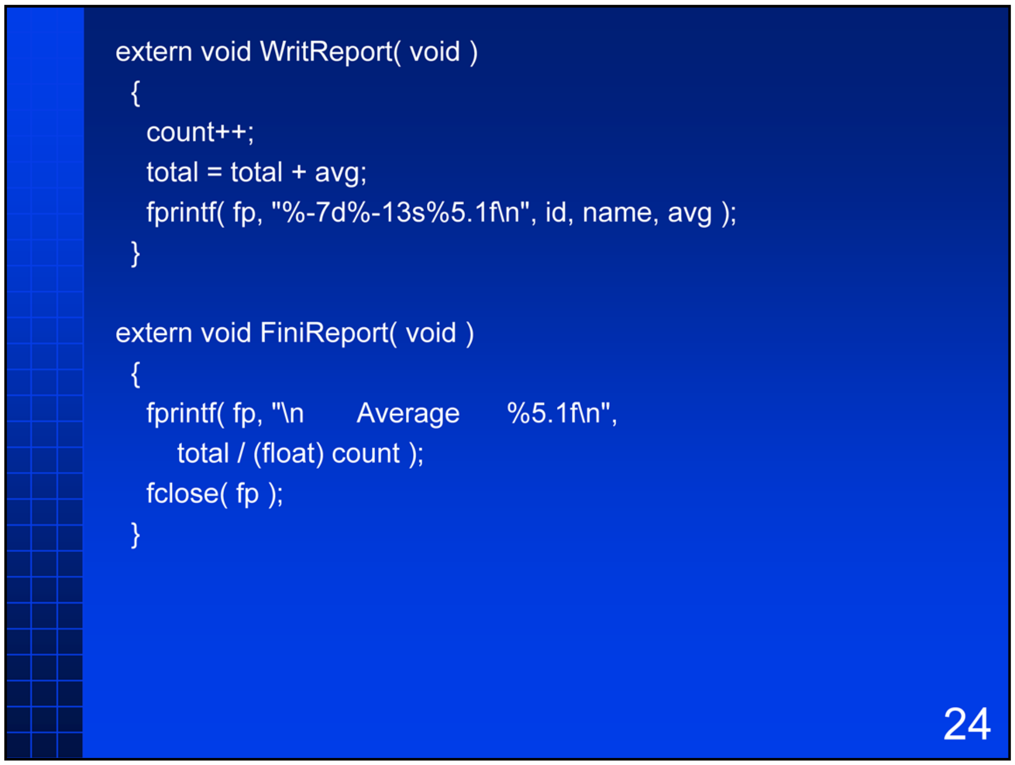
```
/* fn-writ3.c : output module (version 3) */

#include <stdio.h>

extern int    id;
extern char   name[ ];
extern float  avg;

static FILE   *fp;
static int    count = 0;
static float  total = 0.0;

extern void InitReport( void )
 {
   fp = fopen( "report.fil", "w" );
   fprintf( fp, " ID    Name       Average\n\n" );
 }
```

23

Next enhancement: write output to a file instead of stdout

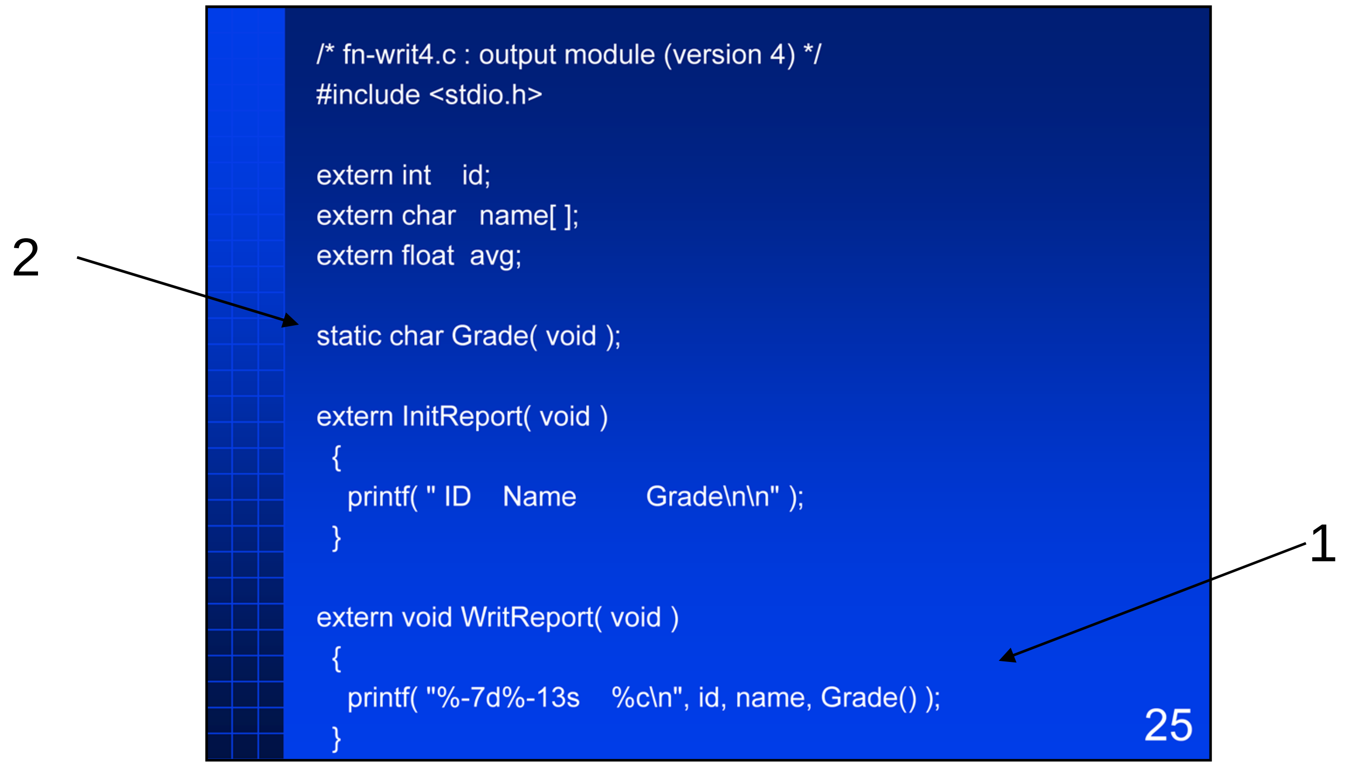
1) declare a file variable. static means that it is visible only within this module.

- example of data hiding, encapsulation. implementation of output module is independent of other modules, so restricted visibility is appropriate. change the average-computation variables so that they also are static.



```
extern void WritReport( void )
 {
   count++;
   total = total + avg;
   fprintf( fp, "%-7d%-13s%5.1f\n", id, name, avg );
 }

extern void FiniReport( void )
 {
   fprintf( fp, "\n       Average       %5.1f\n",
       total / (float) count );
   fclose( fp );
 }
```

more stuff for third revision



```
/* fn-writ4.c : output module (version 4) */
#include <stdio.h>

extern int    id;
extern char   name[ ];
extern float  avg;

static char Grade( void );

extern InitReport( void )
  {
    printf( " ID    Name         Grade\n\n" );
  }

extern void WritReport( void )
  {
    printf( "%-7d%-13s    %c\n", id, name, Grade() );
  }
```

25

Final revision: change original program to output letter grades instead of numbers. Need a function that converst value in global variable “avg” to a letter (represented by a character)

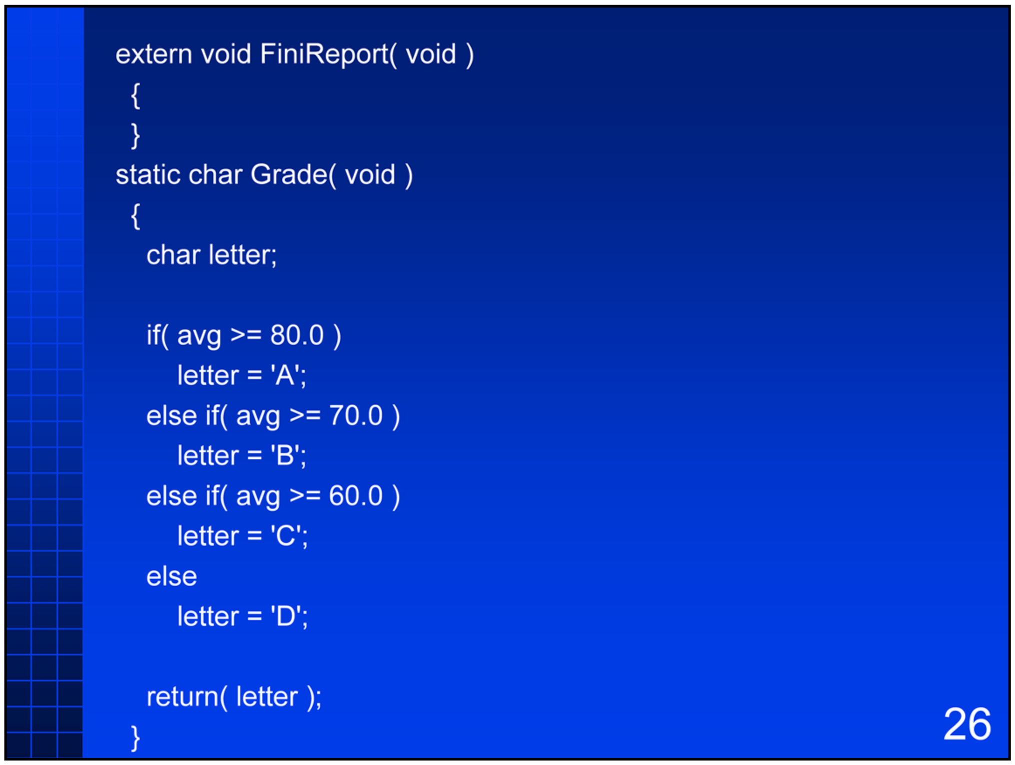
[flip forward slide]

1) usage: invoke grade in parm list, returns char, printf %c directive.

Problem. using Grade() before defined. C assumes integer, creates a “shadow” or “tentative” definition with integer return-type. When real definition occurs, error: turned out to be not an int.

2) So, need a forward declaration or function prototype to “define before use” . equivalent syntax to extern, but use static.

Can also prototype global functions. identical to external declaration -- since that’s all it is, really.

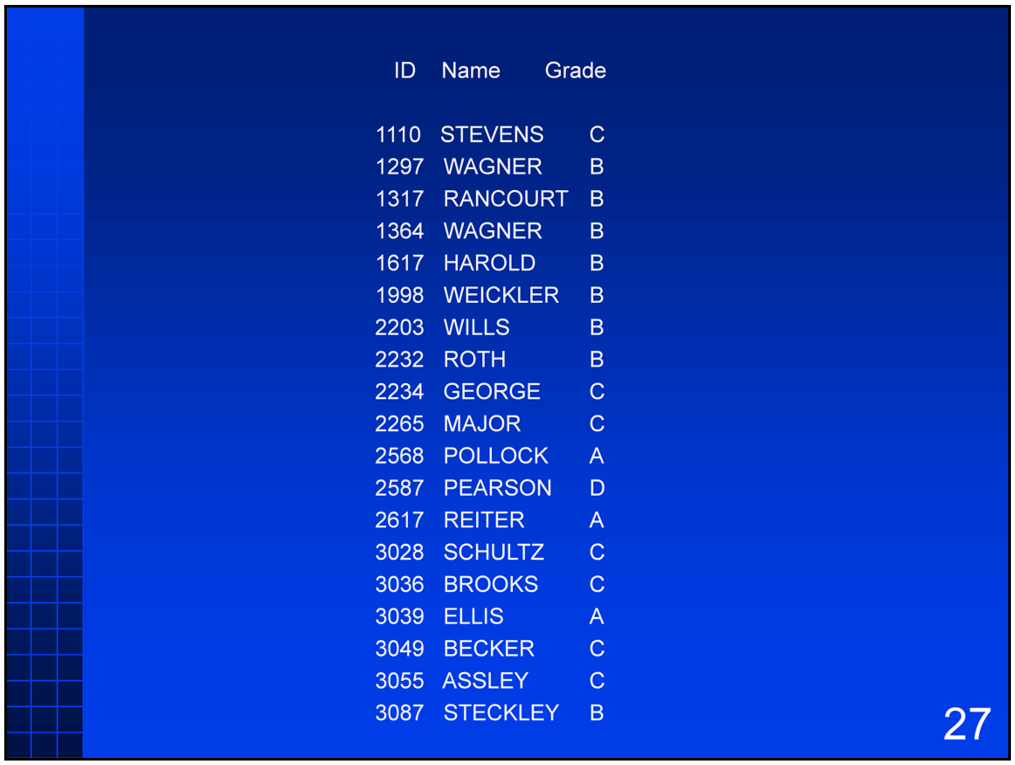


```
extern void FiniReport( void )
 {
 }
static char Grade( void )
 {
   char letter;

   if( avg >= 80.0 )
      letter = 'A';
   else if( avg >= 70.0 )
      letter = 'B';
   else if( avg >= 60.0 )
      letter = 'C';
   else
      letter = 'D';

   return( letter );
 }
```

So, function will return char, no parameters. Only used within this module.

Can use “static” in the definition of the function, same meaning as variables. only visible within this module

[go back and look at usage]



```
  ID   Name       Grade

1110  STEVENS     C
1297  WAGNER      B
1317  RANCOURT    B
1364  WAGNER      B
1617  HAROLD      B
1998  WEICKLER    B
2203  WILLS       B
2232  ROTH        B
2234  GEORGE      C
2265  MAJOR       C
2568  POLLOCK     A
2587  PEARSON     D
2617  REITER      A
3028  SCHULTZ     C
3036  BROOKS      C
3039  ELLIS       A
3049  BECKER      C
3055  ASSLEY      C
3087  STECKLEY    B
```

lovely new output