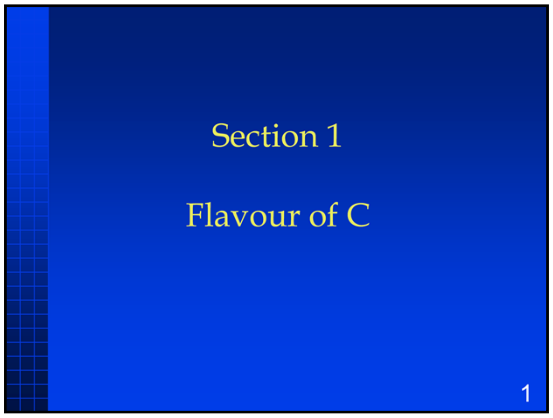


Section 1

Flavour of C

1

Opening remarks: Purpose is to give overview of essential constructs in language and libraries. Crash course.

Will get “reading knowledge” of C: be able to recognize what a pgm does, make modifications (most applicable!)

Presumed background: programmer, comfortable with general programming techniques. [ask about experience]

My background:member of CSG, research group, focus on prog. lang design and implementation, sw eng. I am not a C lawyer - cannot quote chapter and verse on standard [show and tell standards docs]

Presentation style: workshop, encourage questions and discussion. Some demonstrations. Present overviews and then refine -- need to omit details sometimes (too overwhelming, but ask if confused). Development environment available in lab for trying out. All source-code made available. [lab machines slow - encouraged to copy demos to diskettes and take away, bring C code in for trial]

Will be looking at pure ANSI C, will be system independent. Will not discuss implementation techniques for any specific systems (esp. not Windows). Lab uses PC version under Windows (Watcom C), but we’ll be minimalist.

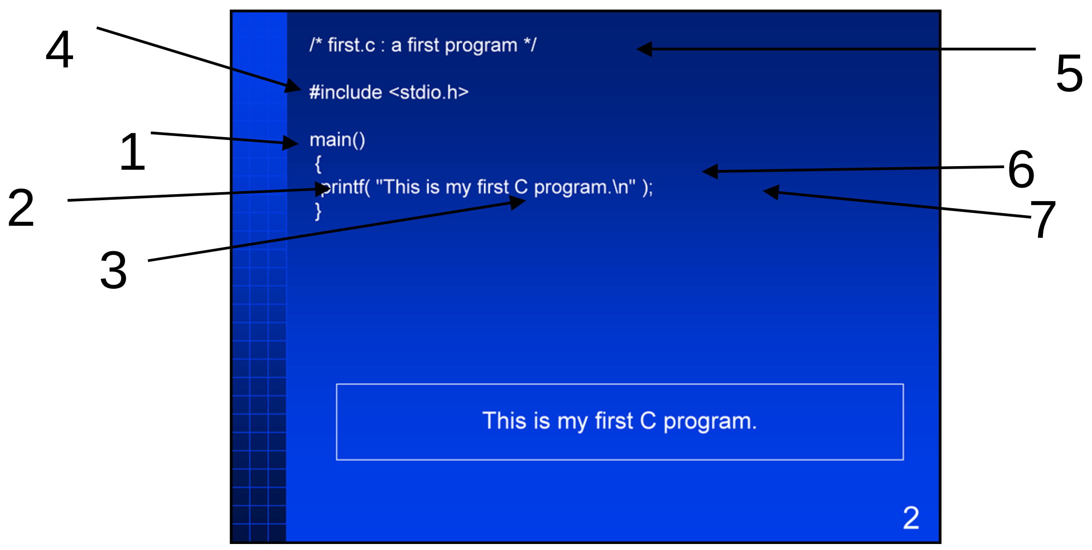
History of C: Developed in Bell Labs (now ATT) late ‘60s and early ‘70s. Developed in conjunction with early UNIX systems on PDP-11 and other machines. Linguistic roots in languages BCPL and B (both Bell Labs research languages; B used in first UNIX implementation on PDP-7). Also Fortran (predominant scientific language in North America, not COBOL, not ALGOL).

Language style is lean and mean (low-level, provides access to hardware). Language is minimal, standard libraries provide much of the functionality, instead of building stuff into the language. Language is designed to make function calling efficient, program structuring easy.

Language intended to be replacement for assembler for systems implementation (cheaper to develop and easier to maintain). Want to be within 10-15% of efficiency of hand-written assembler. Early versions of C on PDP-11, Honeywell 6000, s/370; UNIX implemented in C on pdp-11, interdata 8/32. Early claims of 95% portability (but small samples: 13000 lines of code!)

Initial popularity in academia mid to late ‘70s. Bell labs strange licencing: give away source to universities, exorbitant fee for commercial ($50K source licence, no support (gas $.65/gal)). ATT breakup let ATT enter computing, started to licence more freely. As PCs developed through 1980s, C became feasible. Portability became especially important with new hardware all the time. Workstation-class machines (eg Sun, HP) chose UNIX and C as standard software. Available just about everywhere [comment about Mac, s/390]

Current popularity: [ask audience] portability less important, development and maintenance cost key features. Assembler code way too expensive, esp for RISC machines that are difficult to program (100s of instructions for simple operations).



```
/* first.c : a first program */

#include <stdio.h>

main()
 {
 printf( "This is my first C program.\n" );
 }
```

```
This is my first C program.
```

A first program, gives a broad introduction. Displays or prints a line of text somewhere. “Print” from old days of teletype terminals. Today, displays line of text on screen. Every implementation has a standard (default) place. Details later. Several important points:

1) main() -- starting point (entry point). provides connection to environment (sysdep). actually a function definition: braces.

2) printf() -- another function, reference in this case (invoking the function, not defining it). defined “elsewhere”. This is an example of a statement in C: in this case, statement invokes a function. Terminology of function, procedure, routine, etc.

3) argument or parameter to function [ask about concept]. defines the string that we want printed. Quoted string, literal string, characters appear exactly as shown (white lie for #6)

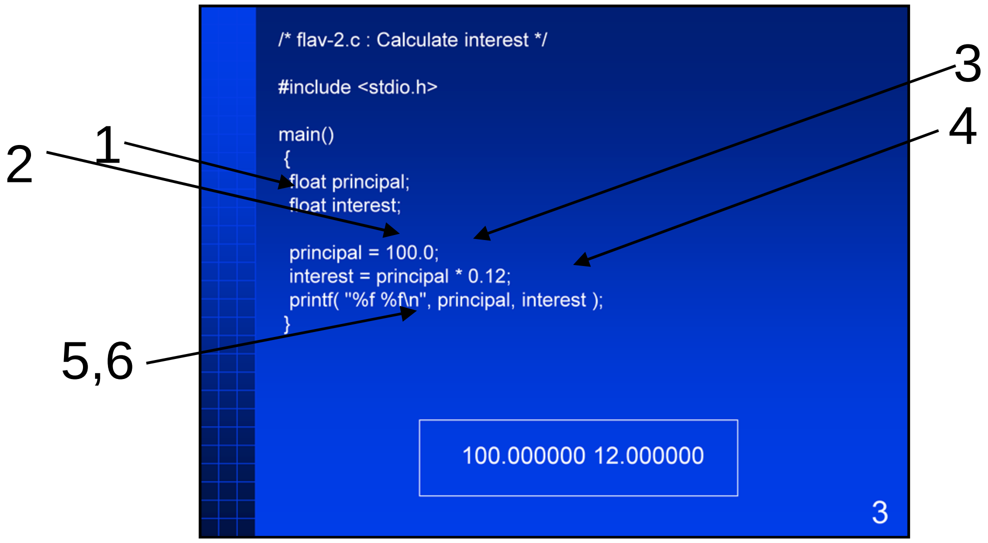
4) “Elsewhere”: call “include directive” one example of features of “pre-processor” or “macro” language (used to modify source code prior to compilation; macro idea -- gather stuff together and label, then use label). Include: copy stuff from named file and pretend that we typed it ourselves. Contains lots of definitions and ofher useful stuff. Name of file enclosed with angles <> (format of name is system dependent). <> means this is a standard entity (defines where to look for file). can also use ““ to mean a private file (different search rules).

5) comments: can be multi-line; do not nest

6) an escape character. “\” means next character has special meaning (“newline” in this case). Converted at “compile-time” (during compilation): compiler recognises \ and makes substitution. Others include \t for tab character, \f formfeed and \0 for null character (character code 0). Note that how printf deals with whese escaped characters is a property of the function, not the language. Typically system-dependent -- newline representations differ (eg unix, dos), or none at all (mainframes with EBCDIC). (concept of NL char from UNIX file-system model).

7) semicolon is statement terminator (not separator like Pascal eg.). Most of the time, OK to put in semicolons wherever you want (some exceptions to be seen).

8) presentation is free-form, spacing and line irrelevant (except in literal string). \ is continuation character.



```
/* flav-2.c : Calculate interest */

#include <stdio.h>

main()
 {
  float principal;
  float interest;

  principal = 100.0;
  interest = principal * 0.12;
  printf( "%f %f\n", principal, interest );
 }
```

```
100.000000 12.000000
```

Another program, some more details.

1) variable declaration: reserves space. “float” is type, say how much space, gives meaning/ interpretation of that space. Float is binary floating-point, other details are system-dependent. Typical PC: 2 bytes IEEE standard +-e38, 6 digits. s/390 4 byte e+-75, 8 digits

name of variable: case sensitive, usual rules (some details to be discussed later)

2) assignment operator

3) numeric constant -- floating-point

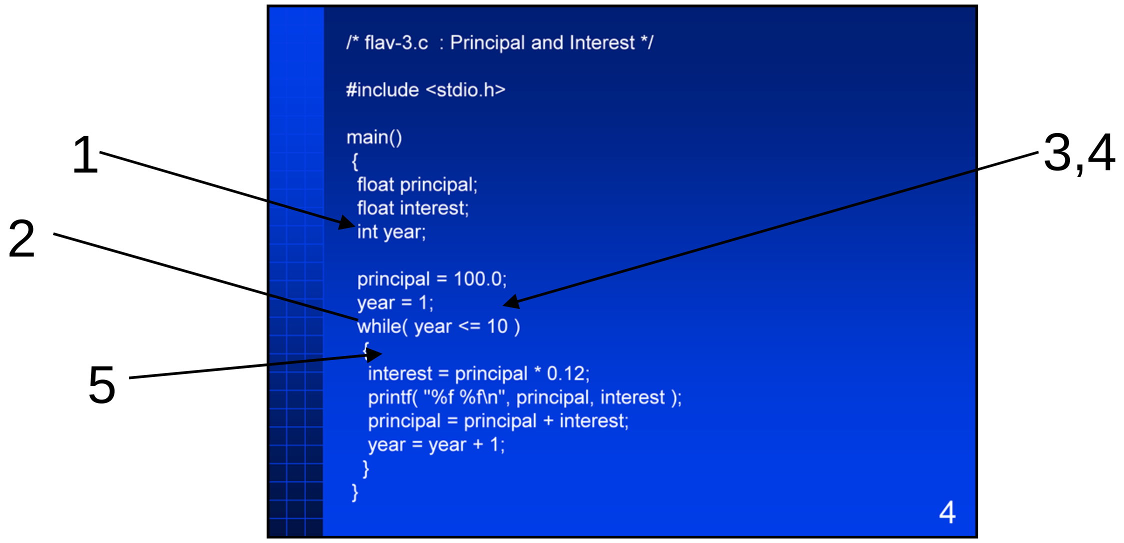
4) arithmetic operator -- multiplication

5) control string in printf: not displayed literally. % means substitution item, sub values given as subsequent parameters to printf.

6) for eg. %f means get next floating-point value, substitute and format it (somehow, more details later) and display. substitution occurs at run-time, not compile time. can say control string is interpreted. equivalent to fortran format strings. programmer’s responsibility to get things right (match up directives with values), provide correct number. A directive without a value not detected; generally produces garbage.

7) be clear distinction between %f (function action at run-time) and \n (compile-time replacement).   when program is running %’s take time, \’s don’t.

8) points out that valiable number of parameters is acceptable.



```
/* flav-3.c  : Principal and Interest */

#include <stdio.h>

main()
 {
  float principal;
  float interest;
  int year;

  principal = 100.0;
  year = 1;
  while( year <= 10 )
   {
    interest = principal * 0.12;
    printf( "%f %f\n", principal, interest );
    principal = principal + interest;
    year = year + 1;
   }
 }
```

Some more:

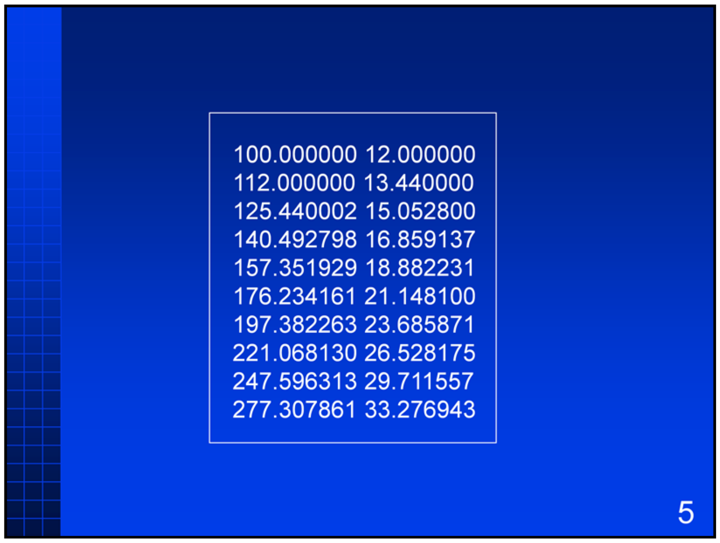
1) integer variables: big variation in implementations, PC 16-bit vs 32-bit (opsys dependence); integer constants

2) looping construct. repeat block of statements some number of times. This one is called a “while” statement (more of these later).   repetition controlled by ...

3) conditional expression (yields true or false). loop repeats as long as true. contains a ...

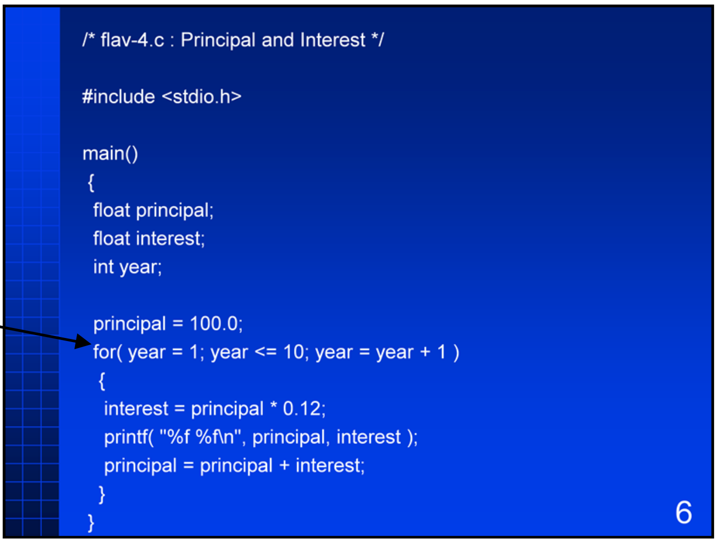
4) relational operator. compares operands and is true or false.

5) compound statement. treats sequence of statements as a single unit. this is a requirement of the “while”: it repeats a single statement, so use compound to join together. repeated statement is referred to as the “object” statement



```
100.000000 12.000000
112.000000 13.440000
125.440002 15.052800
140.492798 16.859137
157.351929 18.882231
176.234161 21.148100
197.382263 23.685871
221.068130 26.528175
247.596313 29.711557
277.307861 33.276943
```

output from previous program. note fairly ugly appearance. will fix this soon.



```
/* flav-4.c : Principal and Interest */

#include <stdio.h>

main()
 {
  float principal;
  float interest;
  int year;

  principal = 100.0;
  for( year = 1; year <= 10; year = year + 1 )
   {
    interest = principal * 0.12;
    printf( "%f %f\n", principal, interest );
    principal = principal + interest;
   }
 }
```

slight refinement:

1) for statement instead of while. same ideas, looping construct, repeat some statement (compound in this case) some number of times. three parts:

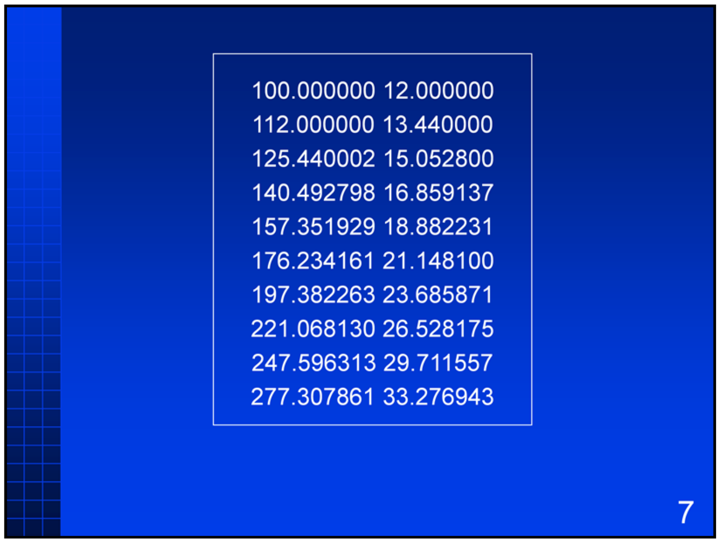
initialization: what to do before first time through object statement. only done once ever.

termination condition: keep doing the object statement as long as this condition is true.

loop incrementor: do this statement after the object statement (every time through)

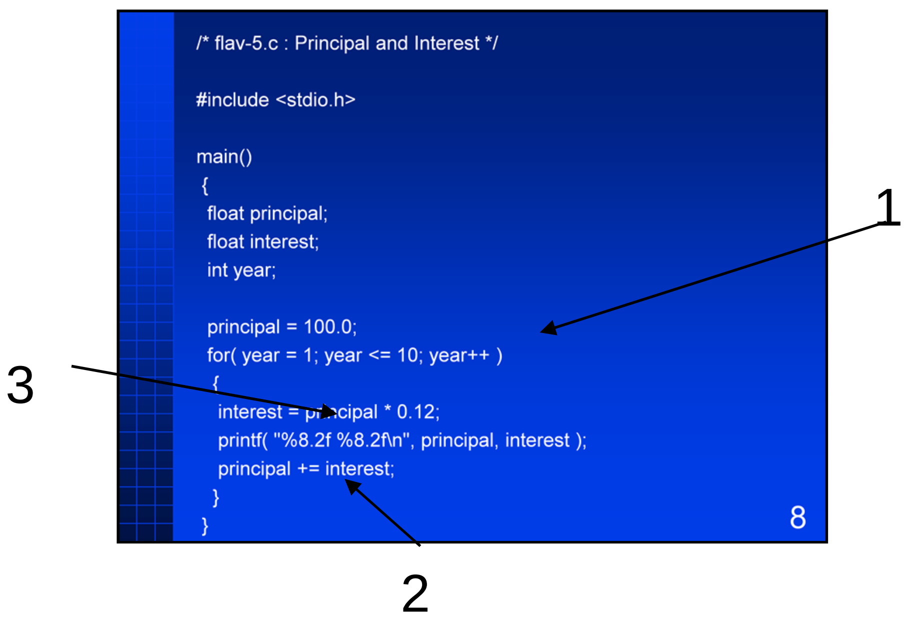
This pgm is equivalent to previous, more succinct -- less typing (C programmers don’t like to type)

Succintness is guiding principle in C, will see many examples of this (as in next)



```
100.000000 12.000000
112.000000 13.440000
125.440002 15.052800
140.492798 16.859137
157.351929 18.882231
176.234161 21.148100
197.382263 23.685871
221.068130 26.528175
247.596313 29.711557
277.307861 33.276943
```

same ugly output



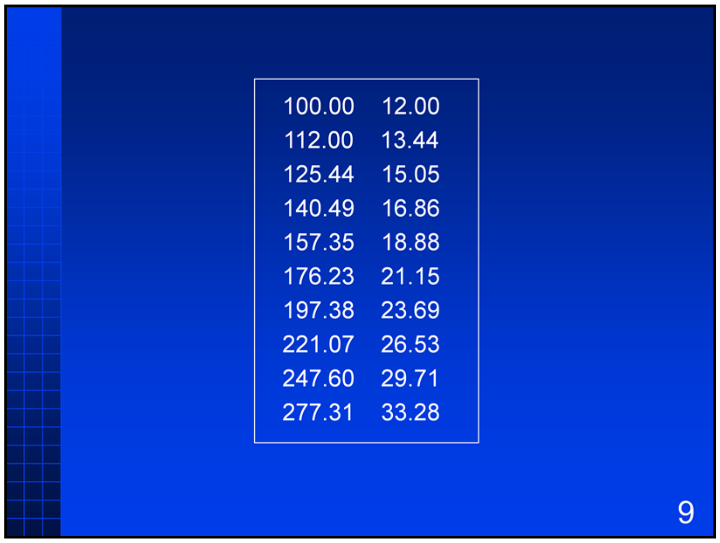
```
/* flav-5.c : Principal and Interest */

#include <stdio.h>

main()
 {
  float principal;
  float interest;
  int year;

  principal = 100.0;
  for( year = 1; year <= 10; year++ )
   {
    interest = principal * 0.12;
    printf( "%8.2f %8.2f\n", principal, interest );
    principal += interest;
   }
 }
```

more short-cuts and succinct expressions:

1) automatic incement operator, “auto-increment” (post-increment in this case). Add 1 to the variable. exactly same as x=x+1. PDP-11 instruction set concept.

2) special assignment “plus gets” or “plus equal”: add value of the expresison on right to the variable on the left. same as x = x + expression

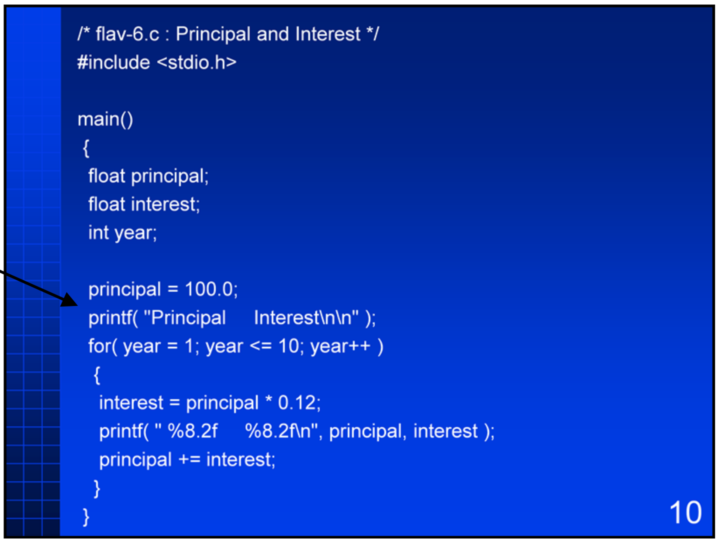
3) improve appearance of output, %f8.2 says field-width of 8, 2 decimals. lots of details to be discussed later.



100.00	12.00
112.00	13.44
125.44	15.05
140.49	16.86
157.35	18.88
176.23	21.15
197.38	23.69
221.07	26.53
247.60	29.71
277.31	33.28

nicer appearance. note rounding etc.

1

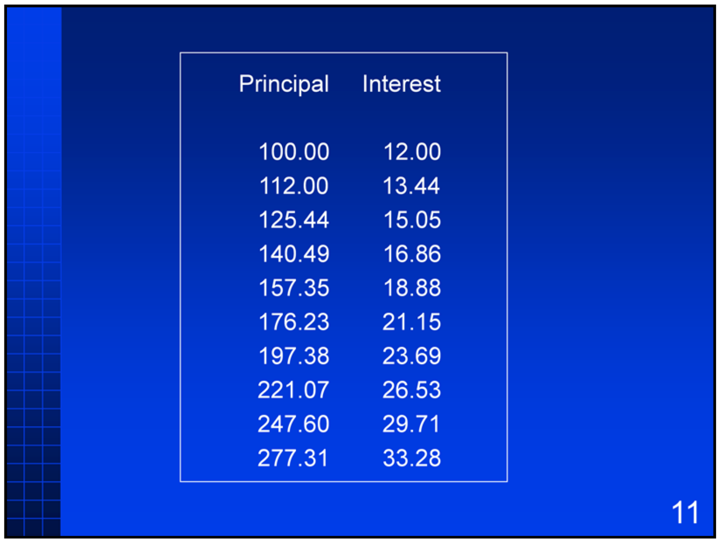


```
/* flav-6.c : Principal and Interest */
#include <stdio.h>

main()
 {
  float principal;
  float interest;
  int year;

  principal = 100.0;
  printf( "Principal     Interest\n\n" );
  for( year = 1; year <= 10; year++ )
   {
    interest = principal * 0.12;
    printf( " %8.2f     %8.2f\n", principal, interest );
    principal += interest;
   }
 }
```

10

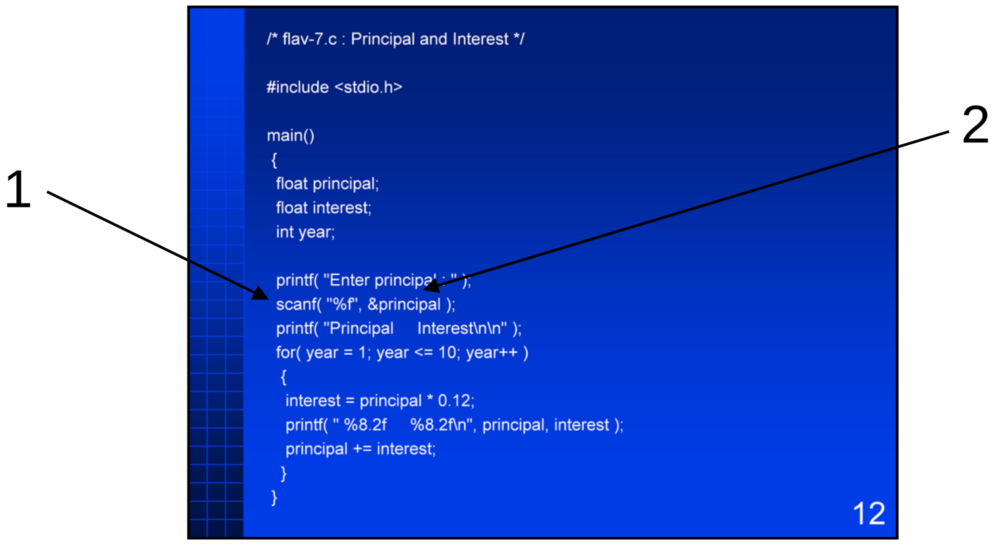
another improvement

1) printf of string literal to produce title. spacing done hard way (counting). use double newlines to produce blank line



Principal	Interest
100.00	12.00
112.00	13.44
125.44	15.05
140.49	16.86
157.35	18.88
176.23	21.15
197.38	23.69
221.07	26.53
247.60	29.71
277.31	33.28

looks nice...



```
/* flav-7.c : Principal and Interest */

#include <stdio.h>

main()
 {
  float principal;
  float interest;
  int year;

  printf( "Enter principal : " );
  scanf( "%f", &principal );
  printf( "Principal     Interest\n\n" );
  for( year = 1; year <= 10; year++ )
   {
    interest = principal * 0.12;
    printf( " %8.2f     %8.2f\n", principal, interest );
    principal += interest;
   }
 }
```

simple input

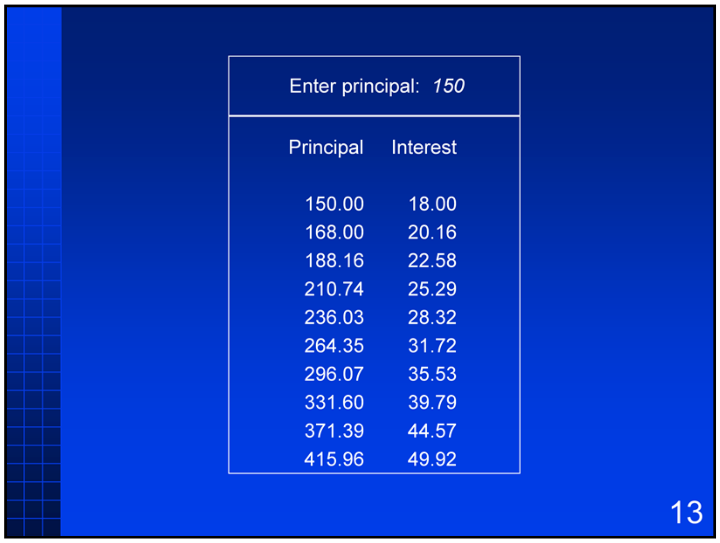
1) scanf: used to read stuff from somewhere. system-dependent: typically via keyboard. like printf, implementations define standard place. Scanf is “high-level”: reads characters and converts to floating-pt number. skips white-space, newlines etc.

works as opposite to printf. control strings dictate what is expected in input stream, parameters indicate where to place results.

Error-handling is bad. if item not found, get 0, no way to determine what’s there. hence scanf is of limited applicability for “industrial-strength” software.

2) note the & preceding the name of the variable. it says that instead of value of variable (like printf), we want to modify the variable; we want a reference to the var. Will discuss this in more detail later.

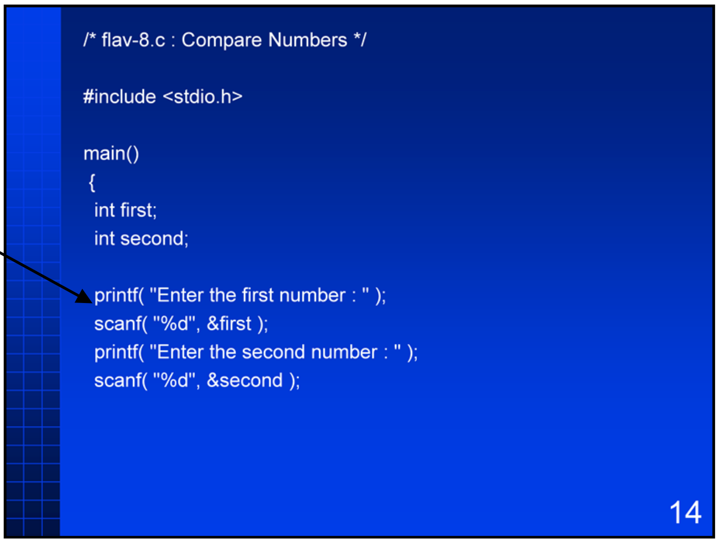
Note: no \n in first printf. system-dependent behaviour, but typically a prompt to allow input to be typed beside output.



```
Enter principal:  150

Principal    Interest

  150.00      18.00
  168.00      20.16
  188.16      22.58
  210.74      25.29
  236.03      28.32
  264.35      31.72
  296.07      35.53
  331.60      39.79
  371.39      44.57
  415.96      49.92
```

nice output

1

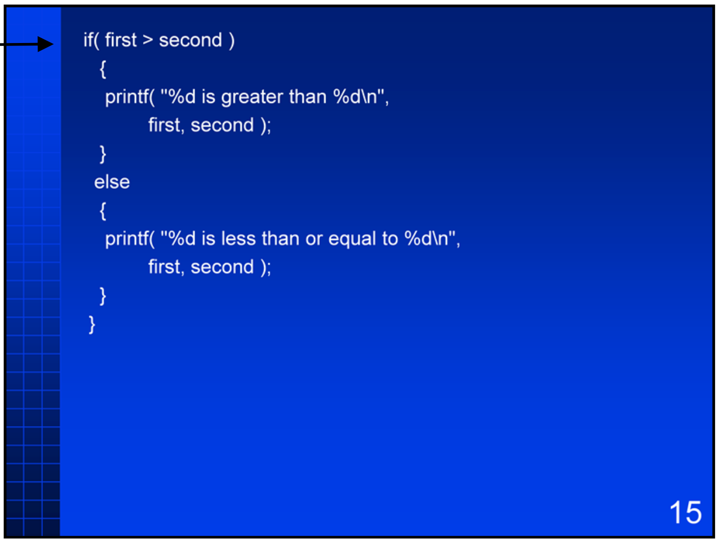


```
/* flav-8.c : Compare Numbers */

#include <stdio.h>

main()
 {
  int first;
  int second;

  printf( "Enter the first number : " );
  scanf( "%d", &first );
  printf( "Enter the second number : " );
  scanf( "%d", &second );
```

Some new stuff:

1) scanf directive %d for integer (decimal, not s/370 packed decimal). remember & means reference (will modify variable, not use value).

2



```
if( first > second )
  {
   printf( "%d is greater than %d\n",
           first, second );
  }
 else
  {
   printf( "%d is less than or equal to %d\n",
           first, second );
  }
}
```

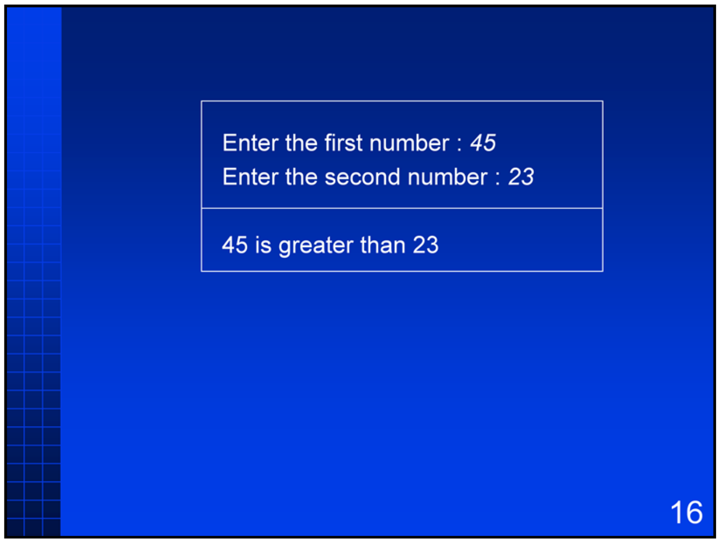
15

continued:

2) if statement: choose between two statements depending on value of conditional expression. (relop). if true, do first, otherwise do second.

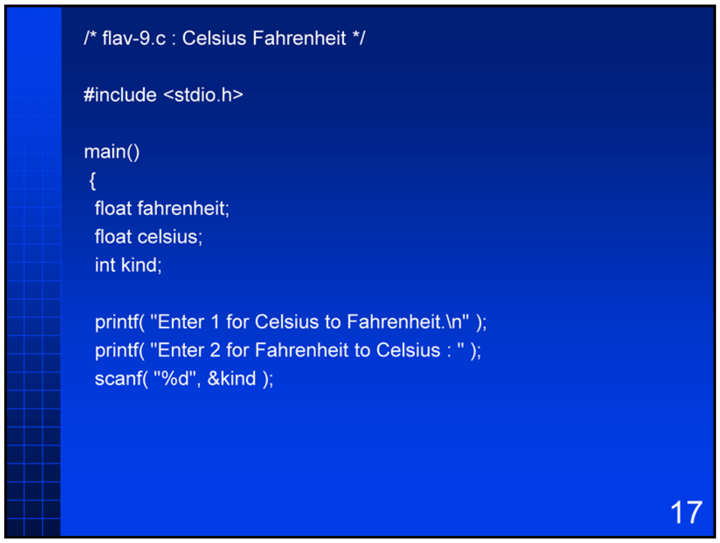
Else is optional, if false and no else, nothing happens

Object statemments are single statements: use braces brackets to form compound statements. Not strictly necessary here, but cann stress strongly enough to use them when in doubt (will see kinds of problems later)



```
Enter the first number : 45
Enter the second number : 23

45 is greater than 23
```

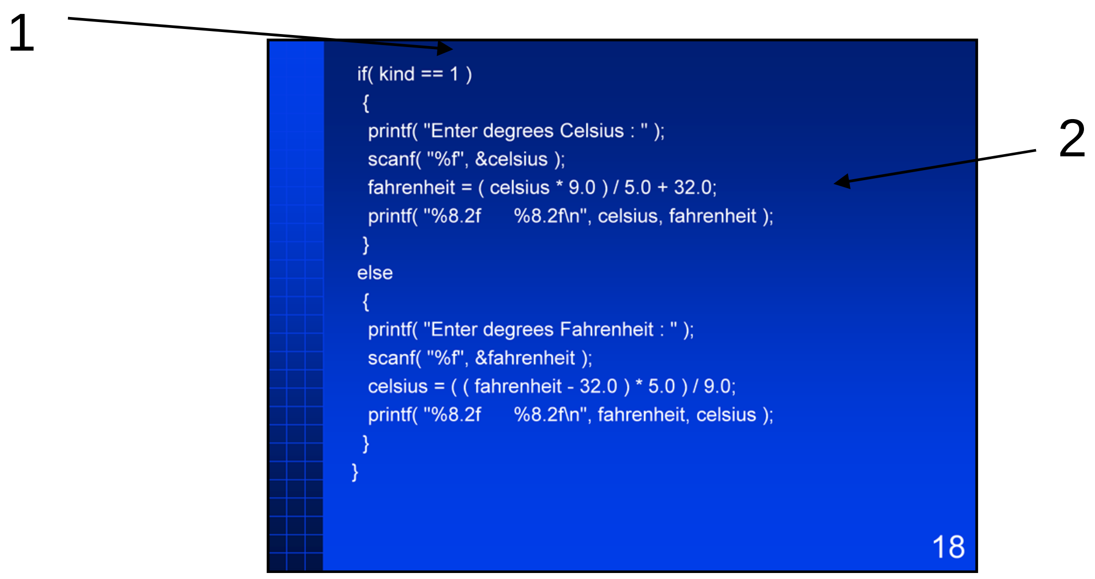
output of program



```
/* flav-9.c : Celsius Fahrenheit */

#include <stdio.h>

main()
 {
  float fahrenheit;
  float celsius;
  int kind;

  printf( "Enter 1 for Celsius to Fahrenheit.\n" );
  printf( "Enter 2 for Fahrenheit to Celsius : " );
  scanf( "%d", &kind );
```

17

another program. reads an integer to determine how to proceed, then converts

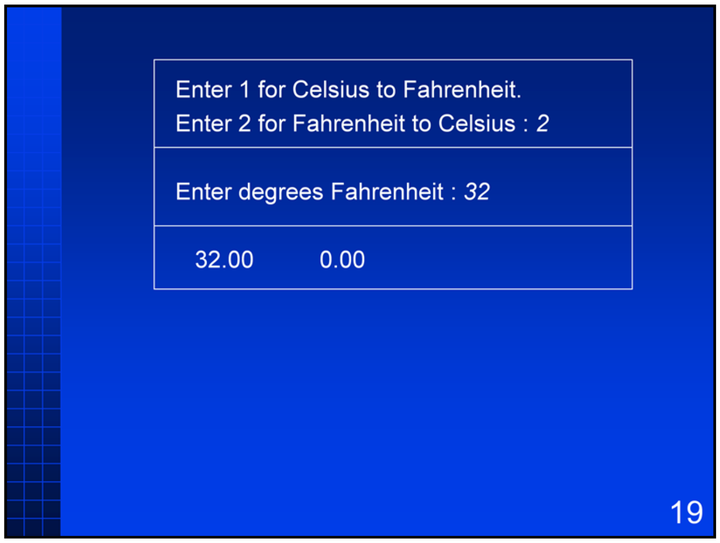


```
 if( kind == 1 )
  {
   printf( "Enter degrees Celsius : " );
   scanf( "%f", &celsius );
   fahrenheit = ( celsius * 9.0 ) / 5.0 + 32.0;
   printf( "%8.2f      %8.2f\n", celsius, fahrenheit );
  }
 else
  {
   printf( "Enter degrees Fahrenheit : " );
   scanf( "%f", &fahrenheit );
   celsius = ( ( fahrenheit - 32.0 ) * 5.0 ) / 9.0;
   printf( "%8.2f      %8.2f\n", fahrenheit, celsius );
  }
}
```

1) == is the relation operator for equality. note that using = by mistake is really bad.

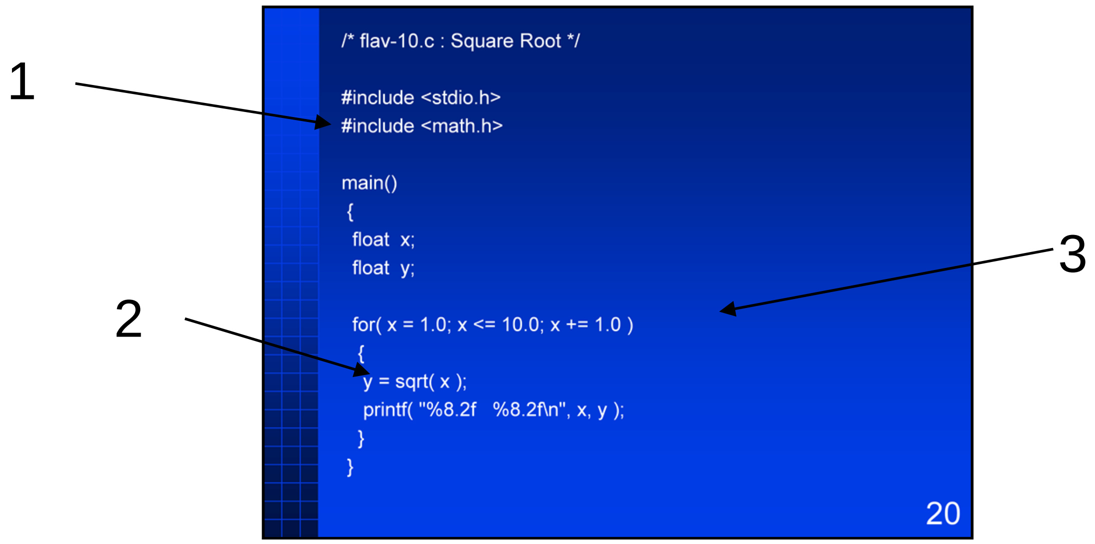
2) an arithmetic expression. follows all the typical rules of algebra for priorities. use parens for clarity and to change order or operation.

3) braces are mandatory here, since object actions are more than a single statement.



```
Enter 1 for Celsius to Fahrenheit.
Enter 2 for Fahrenheit to Celsius : 2

Enter degrees Fahrenheit : 32

  32.00        0.00
```

output of previous



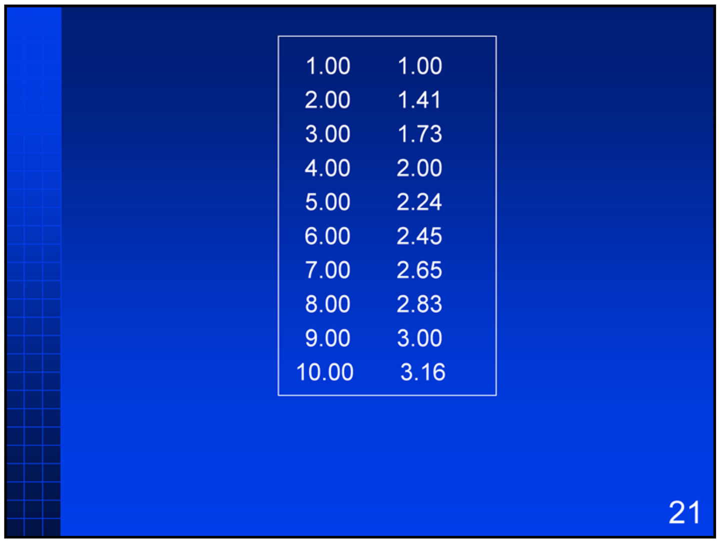
```
/* flav-10.c : Square Root */

#include <stdio.h>
#include <math.h>

main()
 {
  float  x;
  float  y;

  for( x = 1.0; x <= 10.0; x += 1.0 )
   {
    y = sqrt( x );
    printf( "%8.2f   %8.2f\n", x, y );
   }
 }
```

another example:

1) another header file. contains definitions related to mathematical functions.

2) use sqrt function. example of a function that returns a value.

3) for statement: controlled with fp numbers, not integer (demonstrates equivalence to while loop). detail: use x += 1.0 instead of x++, explanation later.



1.00	1.00
2.00	1.41
3.00	1.73
4.00	2.00
5.00	2.24
6.00	2.45
7.00	2.65
8.00	2.83
9.00	3.00
10.00	3.16