

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.

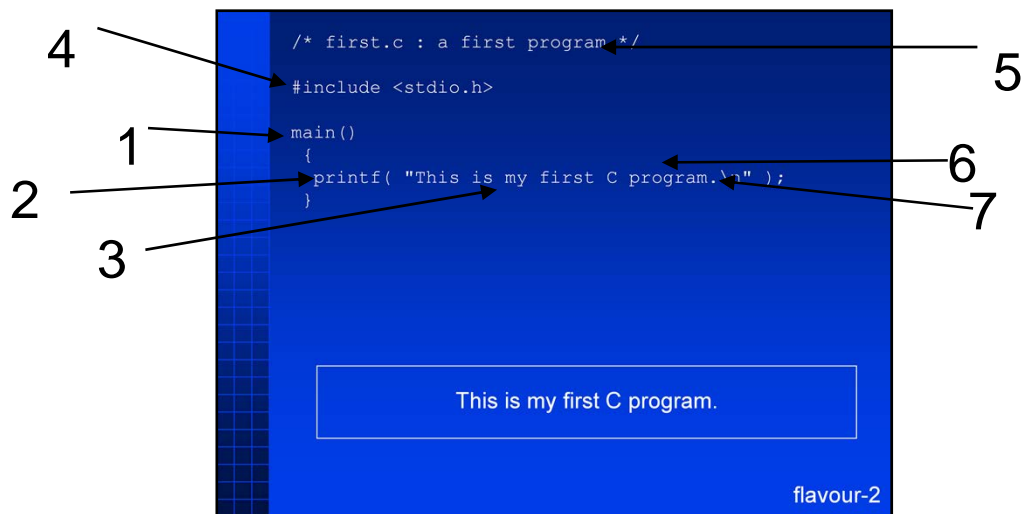
History of C: Developed in Bell Labs (now ATT) late '60s and early '70s. Developed in conjunction with early UNIX systems on PDP-11 and other machines. Linguistic roots in languages BCPL and B (both Bell Labs research languages; B used in first UNIX implementation on PDP-7). Also Fortran (predominant 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!)

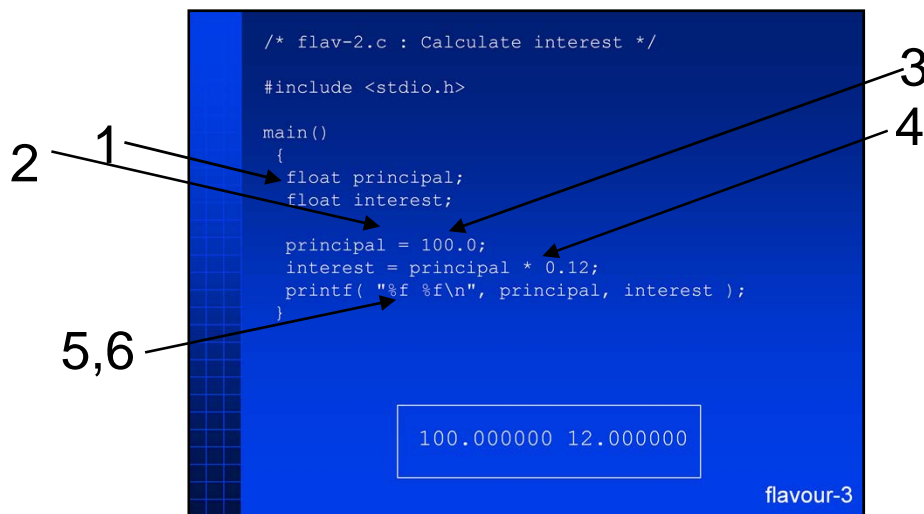
Initial popularity in academia mid to late '70s. Bell labs strange licencing: give away source to universities, exorbitant fee for commercial (\$50K source licence, no support (gas \$.65/gal)). ATT breakup let ATT enter computing, started to licence more freely. As PCs developed through 1980s, C became 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).



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 other 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 these 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.



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

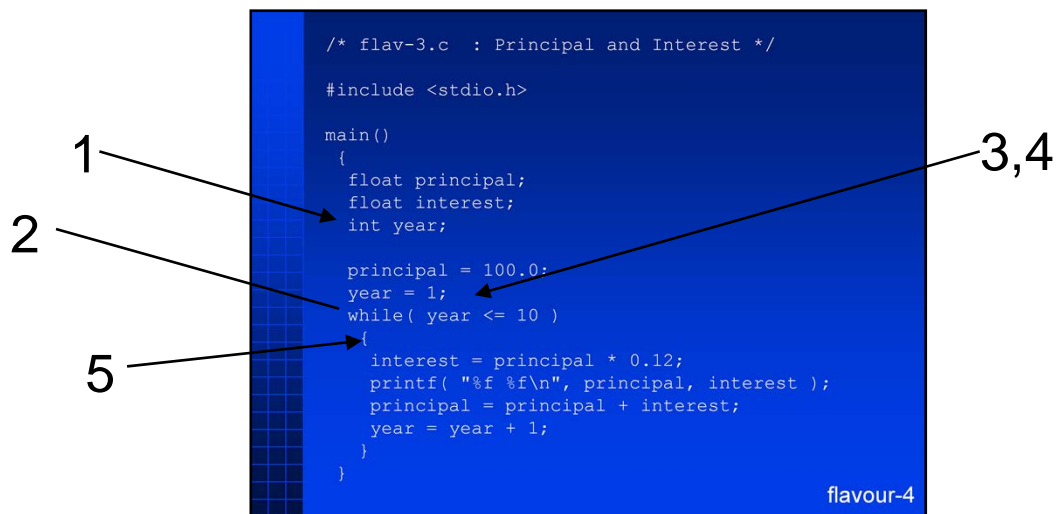
4) arithmetic operator -- multiplication

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.

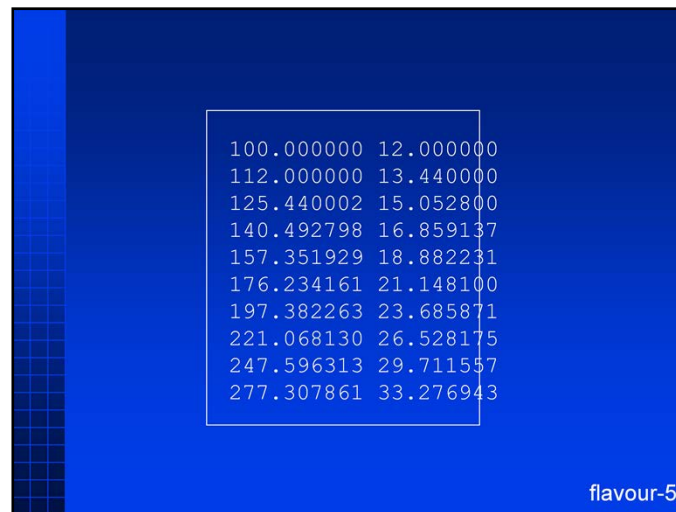
7) be clear distinction between %f (function action at run-time) and \n (compile-time replacement). when program is running %'s take time, \n's don't.

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



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



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

1

```
/* 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;  
    }  
}
```

flavour-6

slight refinement:

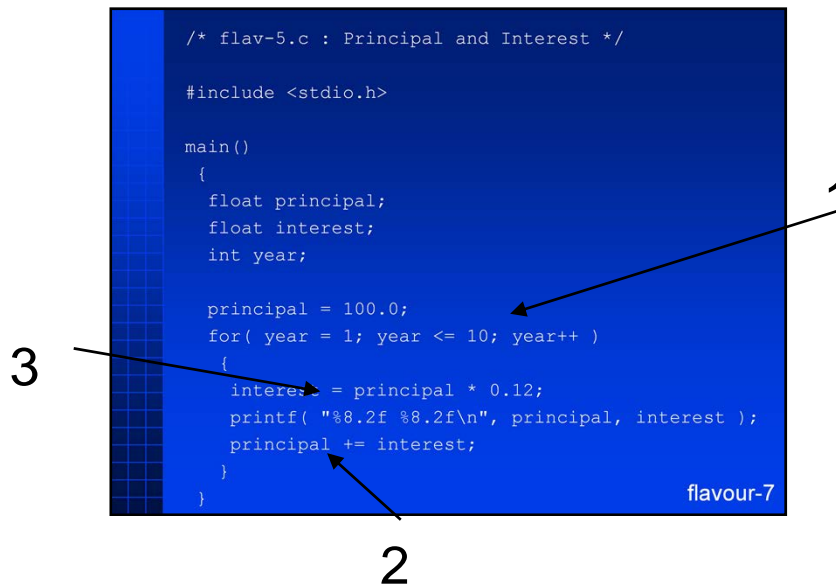
1) for statement instead of while. same ideas, looping construct, repeat some statement (compound in this case) some number of times. three parts:

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.

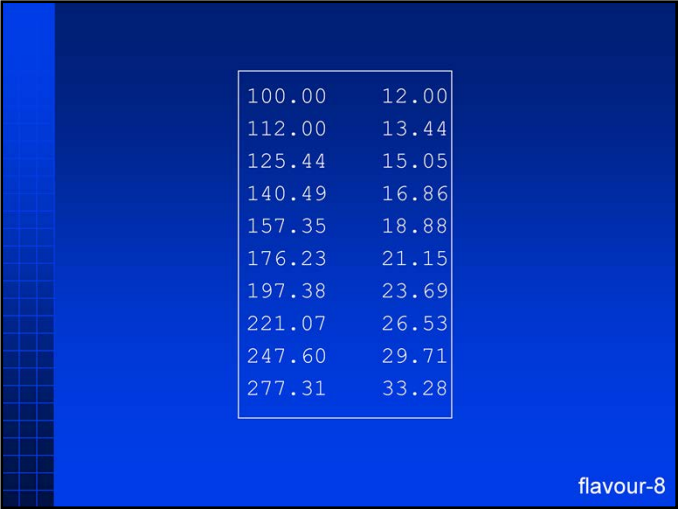
loop incrementor: do this statement after the object statement (every time through)

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)



more short-cuts and succinct expressions:

- 1) automatic increment 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 expression 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

flavour-8

nicer appearance. note rounding etc.

1

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

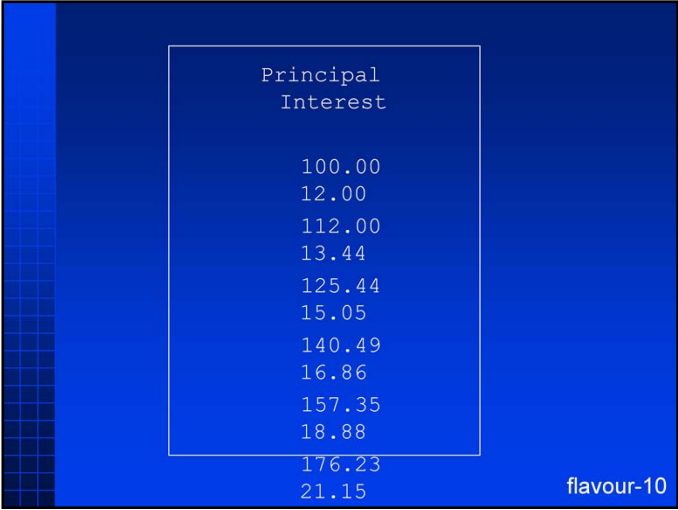
main()
{
    float principal;
    float interest;
    int year;

    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;
    }
}
```

flavour-9

another improvement

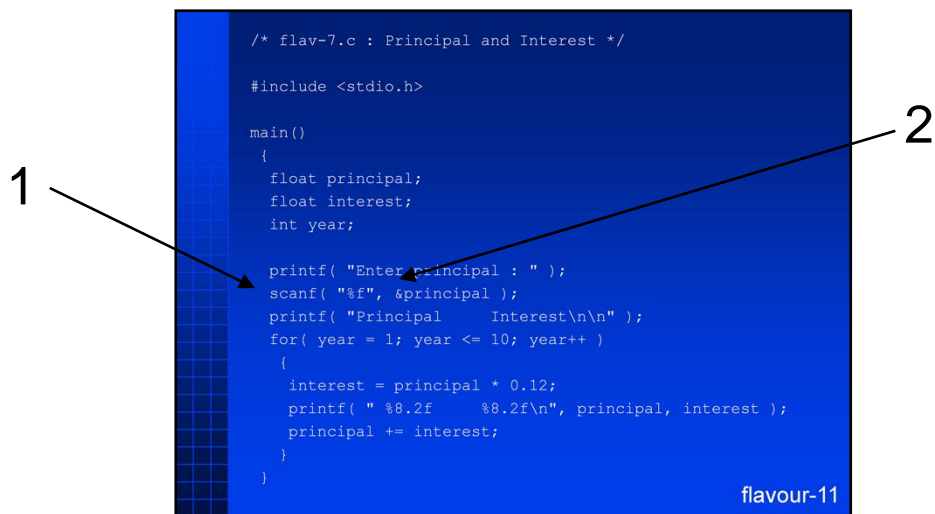
1) printf of string literal to produce title. spacing done hard way (counting). use double newlines to produce blank line



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	

flavour-10

looks nice...



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

flavour-12

nice output

1

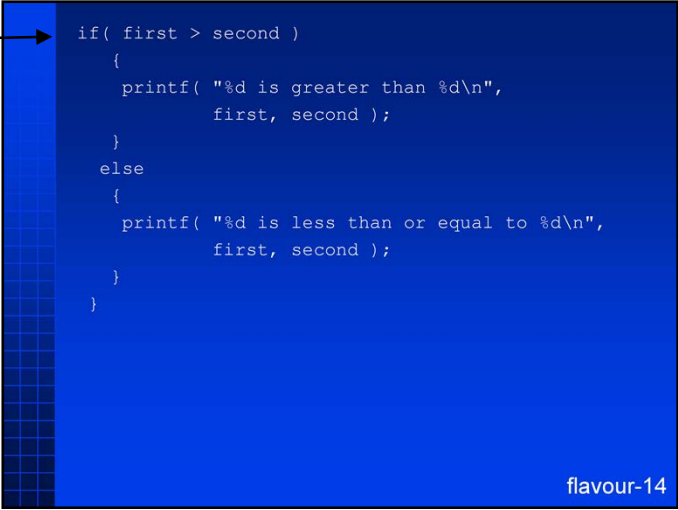
```
/* flav-8.c : Compare Numbers */  
  
#include <stdio.h>  
  
main()  
{  
    int first;  
    int second;  
  
    printf( "Enter the first number : " );  
    scanf( "%d", &first );  
    printf( "Enter the second number : " );  
    scanf( "%d", &second );  
}
```

flavour-13

Some new stuff:

1) scanf directive %d for integer (decimal for base-10, not hex or octal or binary. definitely 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 );
}
```

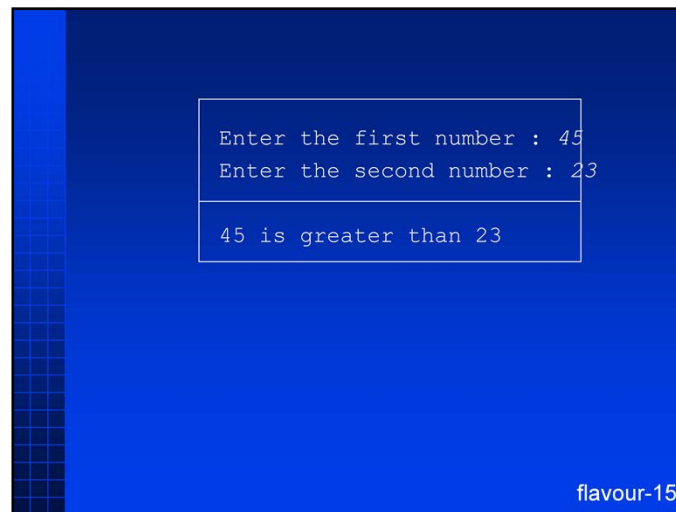
flavour-14

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



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 );
```

flavour-16

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

1

```
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 );
}
}
```

flavour-17

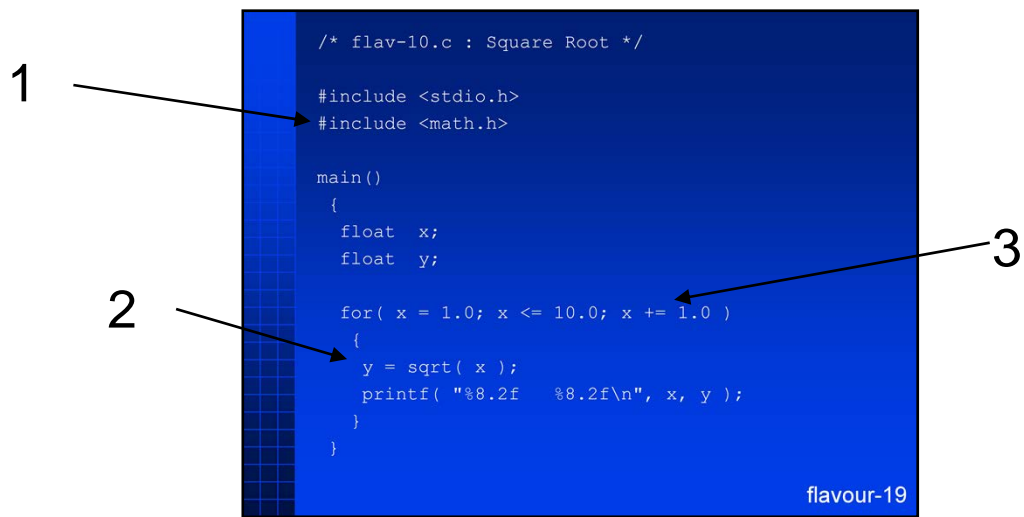
2

- 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

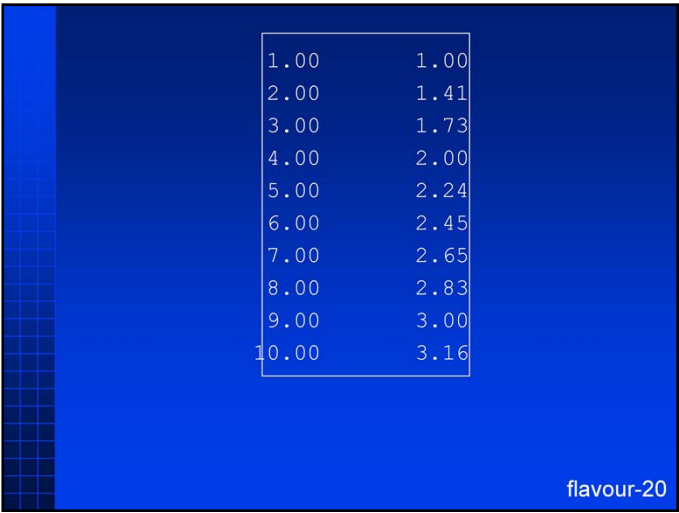
flavour-18

output of previous



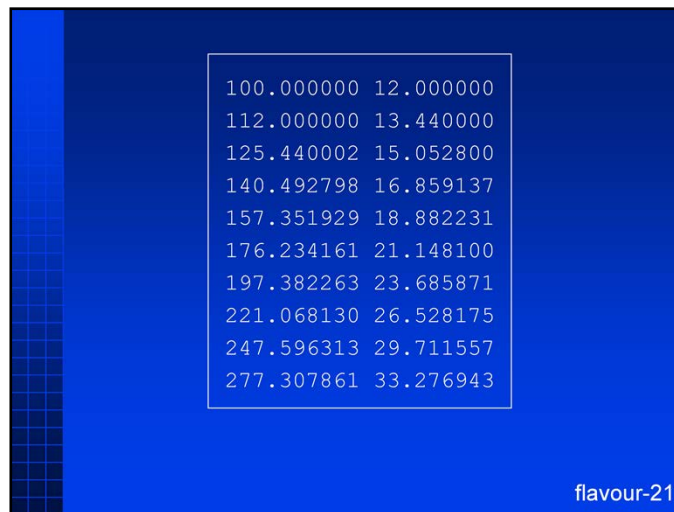
another example:

- 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

flavour-20



output for flav-4

same ugly output

Arithmetic Operators

ops-1

C has lots of operators -- originally, an attempt to model instructions sets of hardware (esp PDP-11)

Lots of them (powerful, terse, overwhelming at times)

Before discussing operators, need to have a quick look at arithmetic types and declarations (specify ranges of values and storage):

Basically, most things are integers (ints), considered to be equivalent to machine word. Historically, a word was the smallest addressible unit of storage (PDP-11, might even be true for x86, who knows?).

Character (char) is also considered to be an “arithmetic” type; it's just a very small integer (can only hold values from 0 to 255 or -128 to +127). Generally assume that a char is the smallest unit of data (corresponds to a byte of storage).

- integers of varying sizes and "signedness"

- size: long or short or not specified
- sign: signed or unsigned (default is signed)

- examples of integer types:

```
char
signed char
unsigned char
int
signed int
unsigned int
```

ops-2

declarations: what the type is and what the name of the variable is.

aside: rules for identifiers as usual, remember case sensitive.

integers: lots of different modifiers that can be applied

size: how much storage, system-dependent unspecified means system default (for PC, 16 or 32 bits depending on OS), long and short a relative to system default.

sign: should the number be considered signed or unsigned. may or may not be of concern (affects things like relative comparisons, overflow conditions)

- examples of integer types: (continued)

```
short  
short int  
short signed int  
short unsigned  
short unsigned int
```

```
long  
long int  
long signed int  
long unsigned  
long unsigned int
```

ops-3

lots of ints

- reals (floating-point) of varying precision and range:

```
float  
double  
long double
```

- example declarations:

```
int          i;  
double       xval1;  
char         first_initial;  
unsigned short NameLength;  
long int     status_word;
```

ops-4

floats are all system-dependent: PC uses IEEE 2, 4, 8 bytes, etc

declarations: pick the appropriate type to model the data being represented.

Literal constants

25	int (decimal)†
25U	unsigned int
25L	long int
25UL	unsigned long int
25LU	
0x00ff	int (hexadecimal) †
0377	int (octal) †
'A'	int (character value)
12.3	double
12.3e-2	double
.1	double
12.3F	float
12.3L	long double

† the type of an unsuffixed integer constant varies with machine architecture and the value of the constant

ops-5

can force constants to acquire specific types (controls amount of storage).

note single character constant (not a string)

Symbolic constants

```
#define WIDTH 100
main() {
    unsigned int size;

    if( size > WIDTH )
        ...
}

-----

main() {
    const unsigned int WIDTH = 100;
    unsigned int size;

    if( size > WIDTH )
        ...
}
```

ops-6

dealing with constants:

constants are a fact of life, good programming practice to use symbolic constants

eg compare against 100 as literal const. better engineering to use constant.

1) use #define: another one of the preprocessor directives:

two parts, item and replacement. compiler will replace item with replacement wherever it occurs (very simple macro). substitution occurs at compile-time (referred to as a lexical replacement)

replacement can be arbitrary; can take time to compile

type of constant can change depending on context (eg in this example would be unsigned, since compared to unsigned; if changes to signed variable, constant would be considered signed).

2) different kind of constant, use a “storage class” in a variable declaration.

says that value of variable does not change (and provides its initial value)

type cannot change, but compiler could optimize storage (eg assembler literals instead of actual storage).

Basic Arithmetic

=	assignment
+	add, unary plus
-	subtract, unary minus
*	multiply
/	divide
%	mod
()	parentheses

For example:

```
x = y + 3 - 6 / (3 + z) ;  
x = y = 0 ;
```

ops-7

arith ops - do arithmetic

mod == remainder (integer only)

traditional priority of ops.

assignment is a binary operator, its “value” is the lhs.

In C, speak of “assignment expression”, not “assignment statement”

the act of assigning is almost like a side-effect, so in a simple assignment statement the value is discarded and we use the side effect.

so, statements like `a=b=c` work. (equiv to `a = (b=c)`)

but note:

if (`a = b`)

is probably not what you want.

Relational

==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

ops-8

relational ops - compare operands, yield true/false
highlight ! for not

used in control statements like if, while

remember == for equality test

For example:

```
if( x > y )
{
    if( x < z )
    {
        ....

    x = (y == z);

    TRUE  == 1  /* any non-zero */
    FALSE == 0
```

ops-9

result of a relational op is an integer value that is 0 for false and not 0 for true (typically 1 or -1)

this is why

```
if( a=b)
```

is so much trouble

Logical Connectives

&&	and
	or
!	not

Examples:

```
if( (x > y) && (x < z) )
```

```
if( (x <= 10) && (A[ x ] == 0) )
```

```
x = ( (y==0) || (z==5) );
```

ops-10

logical vs bitwise: logical combines true/false; bitwise works on bit representations (often equivalent)

yield integers with same meanings as relational

true && true is true, else false

false || false is false, else true

Bitwise

&	and
	or
^	exclusive or
~	not
>>	shift right
<<	shift left

ops-11

for &|^~ bitwise: do operations between respective bits in values
eg & does a bit-by-bit and of each bit in value

$\text{XOR} == (a \ \& \ \sim b) \ | \ (b \ \& \ \sim a)$

for << and >>: left operand is value, right operand is # of bits to shift

For example:

```
#define MASK 0x00000040    /* bit 25 */
unsigned int status;

status = status | MASK;    /* set mask */

if( status & MASK )        /* test mask */

status = status & ~MASK;   /* clear mask */

status = status ^ MASK;    /* toggle mask */
```

ops-12

these examples bit-twiddling

define a hex constant that has exactly one bit on: mask for the bit

shifting eg:

w = w >> 2 ; right-shift w by 2 bit, fill depends on sign (unsigned fills 0, signed fills according to top bit)

Auto Increment and Decrement

++	increment
--	decrement

For example:

```
z = --a;    /* Pre decrement */
```

```
x = ++a;    /* Pre increment */
```

```
z = a--;    /* Post decrement */
```

```
x = a++;    /* Post increment */
```

ops-13

unary operators that act on single operand

operation is to add or subtract one from operand

pre/post refers to when operation takes place wrt use of operand

only for integers, not for floating-point

Special Assignment

`+=` plus assign

`-=` subtract

`*=` multiply

`%=` mod

`/=` divide

`&=` and

`|=` or

`^=` exclusive or

`>>=` right shift

`<<=` left shift

ops-14

combination of arithmetic and assignment operations: do operation between two operands, then assign result to left op

For example:

```
y += 5;      /* equivalent y = y + 5 */
```

```
y += z--;    /* y = y + z; z--; */
```

```
status ^= MASK;
```

ops-15

shifting eg:

```
w >>= 2;
```


Conditional Expression

`<expr1> ? <expr2> : <expr3>`

ops-16

ternary operator:

```
evaluate expr1;  
if true, then do expr2  
    else do expr3
```

a little wierd, but useful once you get the hang of them

For example:

```
min = ( a < b ) ? a : b;
```

```
if( a < b )  
{  
    min = a;  
}  
else  
{  
    min = b;  
}
```

ops-17

like a “choose” operator sort of

another

```
pct = (b==0) ? 0 : (a/b)*100;
```

Comma Operator

<expr1> , <expr2>

For example:

```
c = ( a , b );
```

```
a;
```

```
c = b;
```

ops-18

result is 2nd expr (right)

left is evaluated, then discarded

associates left-to-right: (a,b,c) == ((a,b),c)

useful where need more than one expression, but only one is allowed
eg for stmt initialization, increment

sort of a sequence operator

=====

aside about statements, expressions: statements are things like if, while, var declarations; everything else is an expresison

Function invocation is an expression, assignment is an expression, often use expressions as statements for side-effect value eg auto inc/dec. conditional expression is an if-stmt in expressions clothing

Operators and Associativity in decreasing precedence

()	left to right	parentheses
!	right to left	logical not
~		bitwise not (1's complement)
++ --		auto ince., decr. (pre, post)
+ -		unary plus, minus
size of		get storage size
* &		dereference, address of
(type)	right to left	force type (typecast)
* / %	left to right	multiply, divide, modulus
+ -	left to right	plus, minus
>> <<	left to right	shift bits right, left
< <=	left to right	less than, ..or equal
> >=		greater than, ..or equal

ops-19

expressions have lots of rules

- precedence: which to do first
- associativity: how to treat sequence of operators that have no parentheses.

this stuff is for reference, don't intend to go into details

Operators and Associativity in decreasing precedence (continued)

== !=	left to right	equal, not equal
&	left to right	bit-wise “and”
^	left to right	bit-wise “exclusive or”
	left to right	bit-wise “or”
&&	left to right	connective “and”
	left to right	connective “or”
?:	right to left	conditional expr
= op=	right to left	assignment
,	left to right	comma

ops-20

more rules

Conversion Rules

- Hierarchy of conversions:

signed-char < unsigned-char < short <
unsigned-short < int < unsigned-int <
long-int < unsigned-long-int < float <
double < long-double

- Promote smaller types to int

ops-21

operators can operate between values of differing types.

rules about how to do this.

basic rule: preserve values (avoid data-loss), eg short+short converts each to an int first, then adds (prevents loss of overflow)

anything smaller than int is converted to int, combinations between int and “bigger” types as shown next

Conversion Rules

Op1	Op2	Result
-any-	long double	long double
-any-	double	double
-any-	float	float
-any-	unsigned long	unsigned long
-any-	long int	long int †
-any-	unsigned int	unsigned int
-any-	int	int

† only if long-int is really bigger than int

ops-22

always convert lesser type to bigger type first, then do operation. doesn't guarantee value-preserving, but tries.

Basic Arithmetic Types

Type	Size	Notes
char	>= 8-bits	signed or unsigned
signed char	>= 8-bits	-127 .. 127
unsigned char	>= 8-bits	0 .. 255
short int	>= 16-bits	-32,767 .. 32,767
unsigned short	>= 16-bits	0 .. 65,535
int	>= 16-bits	machine dependent
unsigned int	>= 16-bits	unsigned version
long int	>= 32-bits	-2,147,483,647 .. 2,147,483,647
unsigned long	>= 32-bits	
float	32-bits	real numbers
double	64-bits	

ops-23

type definitions are sys-dep, standard specifies only the minimums as shown

char signed or unsigned -- sysdep. std says "characters in standard printing char set" will never be negative, but what is "std char set"?

1

```
/* oper-1.c : Monthly Payment Schedule */  
  
#include <stdio.h>  
  
main()  
{  
    float balance, principal, interest;  
    int month;  
  
    balance = 10000.00;  
    month = 1;  
    printf( " month  balance" );  
    printf( " interest principal\n\n" );  
    interest = balance * 0.01;  
    principal = 750.00 - interest;
```

ops-24

example programs

1) syntactic shorthand

```

while( balance > principal )
{
    printf( "%6d%9.2f%9.2f%10.2f\n",
            month, balance,
            interest, principal );
    balance = balance - principal;
    interest = balance * 0.01;
    principal = 750.00 - interest;
    month++;
}
printf( "%6d%9.2f%9.2f%10.2f\n\n",
        month, balance,
        interest, principal );
printf( "number of months to repay is %d\n",
        month );
}

```

ops-25

2) expression side-effect statement: discard result of expression, just interested in side-effect of operator.

in this case

month++;

same as

++month;

month	balance	interest	principal
1	10000.00	100.00	650.00
2	9350.00	93.50	656.50
3	8693.50	86.93	663.07
4	8030.44	80.30	669.70
5	7360.74	73.61	676.39
6	6684.35	66.84	683.16
7	6001.19	60.01	689.99
8	5311.20	53.11	696.89
9	4614.31	46.14	703.86
10	3910.46	39.10	710.90
11	3199.56	32.00	718.00
12	2481.56	24.82	725.18
13	1756.37	17.56	732.44
14	1023.94	10.24	739.76
15	284.18	2.84	747.16
number of months to repay is 15			

ops-26

sample output

```
/* oper-2.c : Monthly Payment Schedule */
```

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

```
#define RATE    0.01
```

```
#define INITIAL 10000.00
```

```
#define PAYMENT 750.00
```

```
main()
```

```
{
```

```
    float balance, principal, interest;
```

```
    int month;
```

```
    balance = INITIAL;
```

```
    month = 1;
```

```
    printf( " month  balance" );
```

```
    printf( " interest principal\n\n" );
```

```
    interest = balance * RATE;
```

ops-27

symbolic constants

```
principal = PAYMENT - interest;
while( balance > principal )
{
    printf( "%6d%9.2f%9.2f%10.2f\n",
            month, balance,
            interest, principal);
    balance = balance - principal;
    interest = balance * RATE;
    principal = PAYMENT - interest;
    month++;
}
printf( "%6d%9.2f%9.2f%10.2f\n\n",
        month, balance,
        interest, principal );
printf( "number of months to repay is %d\n",
        month );
}
```

ops-28

Strings

strings-1

strings are versatile

not just visible character strings like literals, but foundation for lowlevel access of memory

C has no built-in, native string type like other langs -- therefore no operators, no varying-length. everything done with function-calls

strictly, strings are a composite array type (array of characters), however for introduction, we can consider as a monolithic type.

look at constants first, then variables.

```
/* str-1.c : string storage representation */

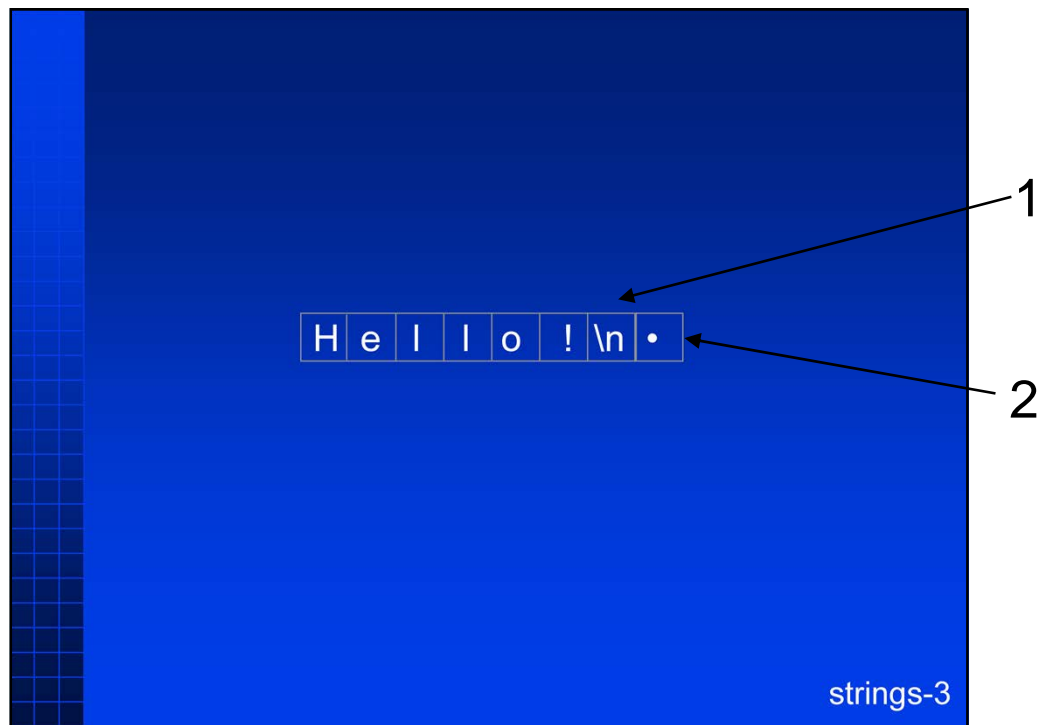
#include <stdio.h>

main()
{
    printf( "Hello!\n" );
}

Hello!
```

strings-2

Like the very first program, displays a string literal.
But how is the string represented in memory?



string takes 8 characters/bytes of storage: 6 visible, one for newline and one funny thing at end:

1) used to represent newline char that is escaped at compile-time. actually might be more, but we'll treat as one.

2) null character (character with character-set code of 0). nullchar is used to mark the end of the string (will elaborate on this later).

Note length is not stored anywhere. length is implicit -- end marked by null, have to count every char from beginning to nullchar. have to do this every time -- computing length is non-trivial operation


```
/* str-la.c : string constants */
```

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

```
#define GREETING "Hello!\n"
```

```
main()
```

```
{
```

```
    printf( GREETING );
```

```
}
```

Hello!

H	e	l	l	o	!	\n	•
---	---	---	---	---	---	----	---

strings-4

Same program, printf string is defined as a constant.
Note that the \n is in the string -- *it's just a character*

```
/* str-lb.c : string constants */  
  
#include <stdio.h>  
  
#define GREETING "Hello!"  
  
main()  
{  
    printf( "%s\n", GREETING );  
}
```

Hello!

H	e	l	l	o	!	•
---	---	---	---	---	---	---

strings-5

Slightly different -- the constant has no `\n`.

Instead, use a `printf` control-string with a string directive `%s`

like other `%` directives, substitutes values from parameter list; for strings, copy character-for-character

separates message content from control information

```
/* str-1c.c : string constants */

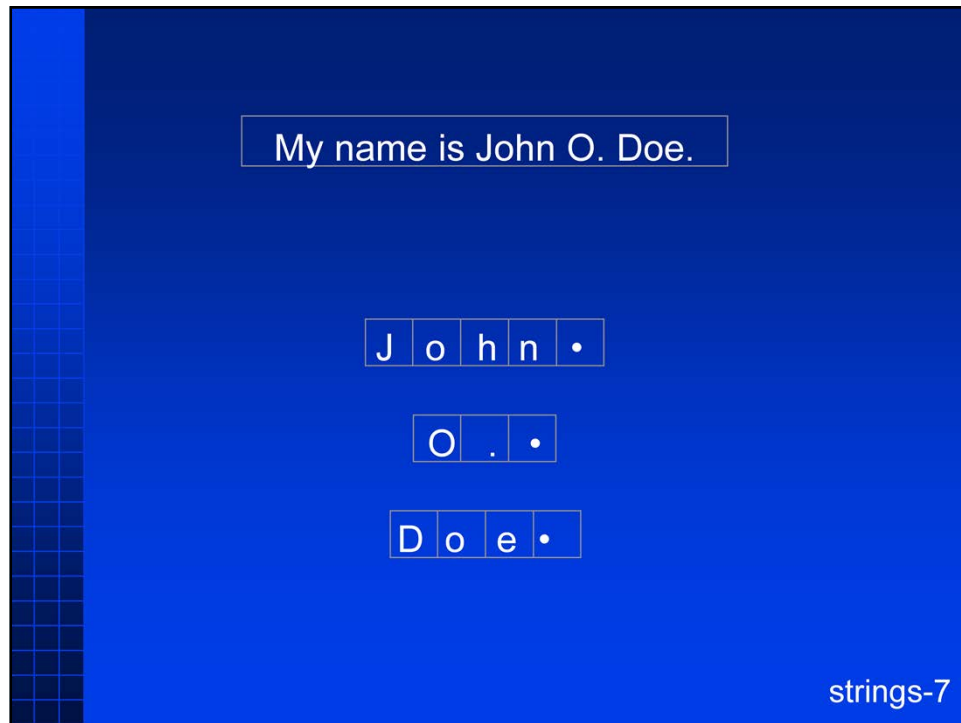
#include <stdio.h>

#define FIRST    "John"
#define INITIAL  "O."
#define LAST     "Doe"

main()
{
    printf( "My name is %s %s %s.\n",
           FIRST, INITIAL, LAST );
}
```

strings-6

some more string literal manipulation



Output of proceeding, storage representation

```
/* str-2.c : string concatenation */

#include <stdio.h>
#include <string.h>

#define FIRST    "John"
#define INITIAL  "O."
#define LAST     "Doe"
```

strings-8

1

(flip forward and back, two-slide sequence)

Now, more stuff: string variables and operations.

1) standard header file <string.h>

- defines standard functions for string manipulation
- as noted, use libraries rather than built-in (this eg is simple string concatenation) [C designed to be efficient at calling]
- null char is significant -- string functions generally treat null as end-of-string

```
main()
{
    char name[ 50 ];
    strcpy( name, FIRST );
    strcat( name, " " );
    strcat( name, INITIAL );
    strcat( name, " " );
    strcat( name, LAST );
    printf( "My name is %s.\\n", name );
}
```

My Name is John O. Doe.

strings-9

2) declaration of a string variable: informally, name is a 50-character string

- can hold string from 0 to 50 characters

3) strcpy “string copy” -- make a copy of a string (destination, must be a variable; source, can be a variable or a constant or literal)

- method: character-by-character copy until null char is encountered. null char is copied.

4) strcat “string concatenate” -- append the source string (2nd parameter) to the destination (first parameter)

- method: find end of destination by finding nullchar, then char-by-char copy. nullchar end of first is removed (overwritten), then new nullchar is placed at end of concatenation.

Values of the character array

												...	
J	o	h	n	
J	o	h	n		
J	o	h	n		O	
J	o	h	n		O	
J	o	h	n		O	.			D	o	e	.	

strings-10

build up string in pieces

note initial state is completely undefined, no null char.

can't use string functions on this, since it hasn't been properly initialized.

```
/* str-3.c : scanning string input */
#include <stdio.h>
#include <string.h>

main()
{
    char name[ 50 ];
    char first[ 20 ], initial[ 3 ], last[ 20 ];

    scanf( "%s %s %s", first, initial, last );
    strcpy( name, first );
    strcat( name, " " );
    strcat( name, initial );
    strcat( name, " " );
    strcat( name, last );
    printf( "My name is %s.\n", name );
}

strings-11
```

Using scanf to read string values

1) same idea as numbers; %s directive for strings

2) no & here:

- easy explanation: its a rule (strings have no &)
- hard explanation: & creates a reference, array names are already a reference [the expression value of an integer variable name is the contents of the variable, the expression value of an array variable name is the address of the array]

reading strings skips whitespace (blanks, tabs, newlines) => no blank-embedded strings

<i>John O. Doe</i>	
My name is John O. Doe.	

first:	J	o	h	n	.														
initial:	O	.																	
last:	D	o	e	.															

strings-12

sample input

```
/* str-4.c : string comparison */

#include <stdio.h>
#include <string.h>

main()
{
    char str1[ 50 ];
    char str2[ 50 ];

    scanf( "%s %s", str1, str2 );
```

strings-13

C (library really) supports concept of relational comparisons between strings (strings can be equal or not; relation can be ordered: one string can be less than another)

Ordering based on binary values of character-set -- ordering is system-dependent. these days, code pages, internationalization, dbcs make character-string ordering more complicated than it used to be.

Program here reads two string with scanf and then compares them

1

```
if( strcmp( str1, str2 ) == 0 )
{
    printf( "The strings %s %s are equal.\n", str1, str2 );
}
else
{
    printf( "The strings %s %s are not equal.\n", str1, str2 );
}
```

- Also: strncmp, strcoll

strings-14

In keeping with C philosophy, not part of language, no operators.

1) library function "strcmp":

returns 0 if equal

returns negative if lhs < rhs

returns positive if lhs > rhs

note that this is not a "string equal" function, does not return true/false

for relations, shorter is lesser (null char is less than anything else)

strncmp: restrict number of characters in strings

strcoll: compare according to locale collating sequence

this this

The strings this this are equal.

this that

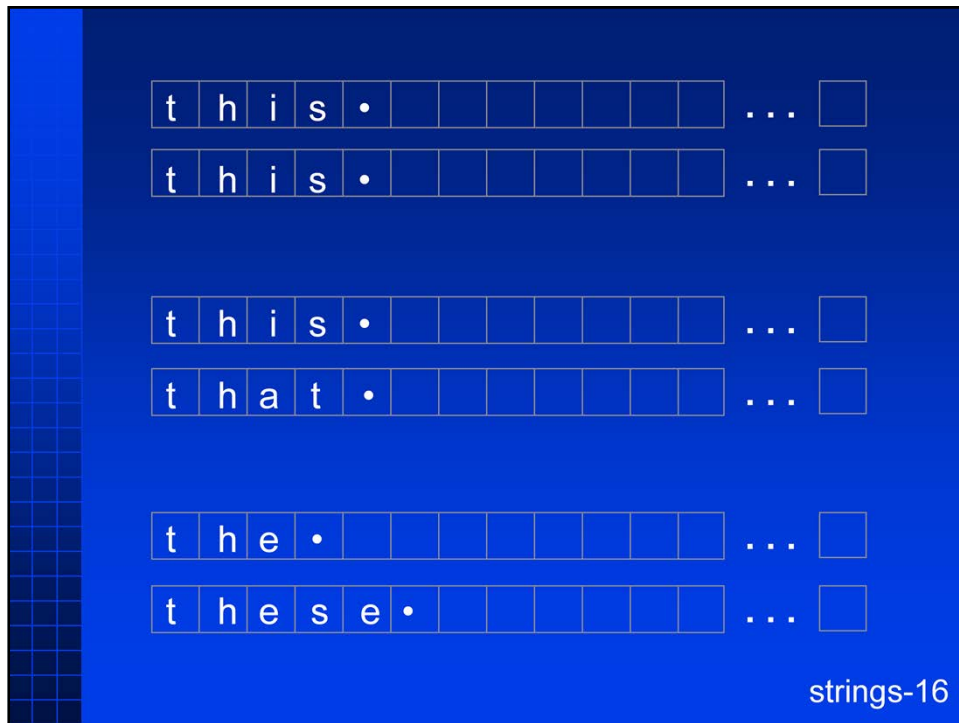
The strings this that are not equal.

the these

The strings the these are not equal.

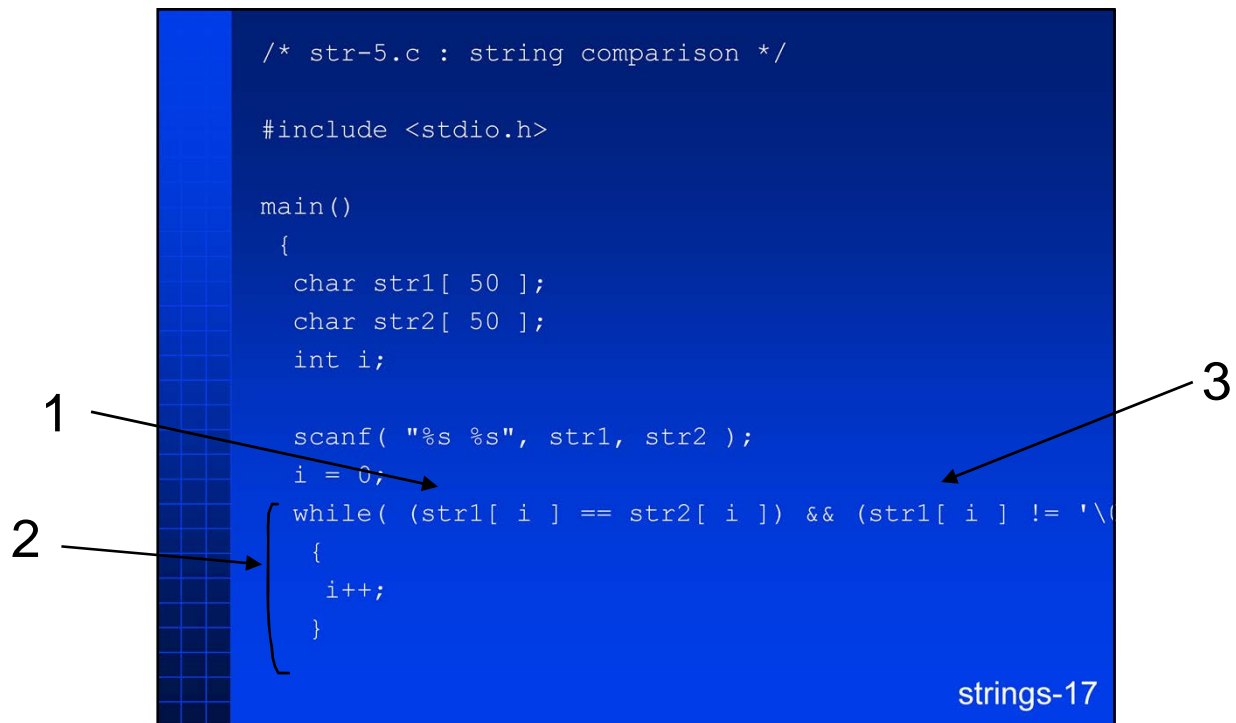
strings-15

samples, look at storage following



1. equal
2. not equal (this > that, returns +ve, i > a)
3. not equal (the < these, returns -ve, nullchar < s)

strcmp family compares character-by-character, whitespace is significant (eg leading, trailing blanks)



An example of character-by-character manipulation: determine if two strings are equal (subset of what `strcmp` does).

A bit of a look-ahead to array processing, since strings are really arrays of characters, but

1) array subscripting operation: selects *i*-th element from string.

note *i* starts at 0

2) loop structure: as long as same characters and not end-of-string, advance to next character (increment *i*)

3) null character constant: apostrophes (single-quotes) instead of doubles

at the end of this loop, strings are not equal; or end-of-string 1

if end-of-string 2, then we quit because not equal (unless also eos1)

```
if( str1[ i ] == str2[ i ] )
{
    printf( "The strings %s %s are equal.\n",
           str1, str2 );
}
else
{
    printf( "The strings %s %s are not equal.\n",
           str1, str2 );
}
}
```

strings-18

we've stopped advancing through the loop, either

- end of string1
- two strings not equal

check to see which.

if eos1, then if equal then also eos2 and strings are equal

if str2 is shorter, would have quit because not equal

Lexical concatenation

```
/* str-6.c : lexical concatenation */
#include <stdio.h>
#include <string.h>

main()
{
    char long_const[ 200 ];

    strcpy( long_const, "Here is a string constant
that "
Often, "
to "
);
    "will be very very long.
"such strings are difficult
"enter with text editors."
strings-19
```

1

long constants hard to enter in some text-editors (less of a concern these days in windowed systems)

compile-time concatenation of constants -- adjacent string literals

1) strlen -- part of standard string library, returns number of characters from beginning up to but not including null char, number of "visible" characters (misnomer?)



Long_const is 117 characters long

strings-20

output

String Initialization

```
/* str-7.c : string initialization */
#include <stdio.h>
#include <string.h>

main()
{
    const char prompt[ ] = "Name [%s]? ";
    char name[ 50 ] = "unknown";

    printf( prompt, name );
    scanf( "%s", name );
}
```

strings-21

can specify initial values for strings at compile time
called an initializer:

- 1) first example, string length unspecified, compiler will compute. most applicable to constant
- 2) second example specifies length and causes that much storage to be reserved. initial value must fit within that space (but need not be that long)

Input / Output

Printf, Scanf and
textfile processing

I/O-1

More about printf and scanf;

Quick look at textfile I/O (reading and writing files that contain text,
suitable for use with a text editor like notepad)

```
/* hello.c : a first program */  
  
#include <stdio.h>  
  
main()  
{  
    printf( "Hello, world.\n" );  
}
```

Hello, world.

I/O-2

The classic c program, prints/displays string somewhere.

Where? C adopts from UNIX and elsewhere notion of standard files: in C's case there are three:

stdout, where output goes (eg printf),

stdin, where input comes from (eg scanf),

stderr, where error-messages go

These entities are defined in <stdio.h> and automatically initialized.

Meaning and behaviour is system-dependent. On line-mode systems like old-style UNIX & DOS, stdout = screen, stdin = keyboard, stderr = screen.

In current windowing systems, unclear clear. Concept not really supported in these environments. Lab software, watcom c, creates a window that behaves like line-mode screen.

```
/* io-decimal.c : example decimal format */
```

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

```
#define FIRST 0
```

```
#define LAST 20
```

```
#define STEP 4
```

```
main()
```

```
{
```

```
    int i;
```

```
    printf( "Squares of i\n\n" );
```

```
    i = FIRST;
```

```
    while( i <= LAST )
```

```
    {
```

```
        printf( "%d %d\n", i, i * i );
```

```
        i = i + STEP;
```

```
    }
```

```
}
```

Squares of i

0 0

4 16

8 64

12 144

16 256

20 400

I/O-3

Review:

printf, control string, formatting directives, values contained in subsequent parameters, programmer's responsibility to get these correct.

using %d for integers here. note left aligned

```
printf( control, arg1, arg2, . . . );
```

d, i	decimal
o	unsigned octal
x, X	unsigned hexadecimal
u	unsigned decimal
c	single character
s	character string
e, E	exponential
f	real
g, G	real / exponential
%	%

I/O-4

there are lots of directives, this is most (missing %p for pointer, %n for target output length (written to parameter))

d,i: decimal integers [i for compatibility with scanf]

o: converts to octal

x,X: converts to hex, case controls output case

.
.
.

e,E: exponential (scientific: d.ddd eii), case controls output case

f: real (fixed-point)

g,G: decides for itself between e and f

?: prints a %

Again: control string defines types of parameters and how to process, therefore what to pass; user's responsibility

1

```
/* io-3decimal.c : example field width
format */
```

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

```
#define FIRST 0
```

```
#define LAST 20
```

```
#define STEP 4
```

```
main()
```

```
{
    int i;
```

```
    printf( "Squares of i\n\n" );
```

```
    i = FIRST;
```

```
    while( i <= LAST )
```

```
    {
```

```
        printf( "%3d %3d\n", i, i * i );
```

```
        i = i + STEP;
```

```
    }
```

```
}
```

Squares of i

0 0

4 16

8 64

12 144

16 256

20 400

I/O-5

- 1) minimum field width, will be enlarged if necessary. note right alignment of output
- minus sign inserted if necessary

1

```
/* io-zero.c : example ZERO fill format */
```

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

```
#define FIRST 0
```

```
#define LAST 20
```

```
#define STEP 4
```

```
main()
```

```
{  
    int i;
```

```
    printf( "Squares of i\n\n" );
```

```
    i = FIRST;
```

```
    while( i <= LAST )
```

```
    {
```

```
        printf( "%03d %03d\n", i, i * i );
```

```
        i = i + STEP;
```

```
    }
```

```
}
```

Squares of i

000 000

004 016

008 064

012 144

016 256

020 400

I/O-6

1) same, leading zeroes

1

```
/* io-left.c : example left justify format */
```

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

```
#define FIRST 0
```

```
#define LAST 20
```

```
#define STEP 4
```

```
main()
```

```
{
```

```
    int i;
```

```
    printf( "Squares of i\n\n" );
```

```
    i = FIRST;
```

```
    while( i <= LAST )
```

```
    {
```

```
        printf( "%-3d %-3d\n", i, i * i );
```

```
        i = i + STEP;
```

```
    }
```

```
}
```

Squares of i	
0	0
4	16
8	64
12	144
16	256
20	400

I/O-7

1) negative field-width: get rid of zeroes, left-align in fixed-width columns

```
/* io-float.c : example floating point format */
```

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

```
#define FIRST 0.0
```

```
#define LAST 2.0
```

```
#define STEP 0.4
```

```
main()
```

```
{  
    float i;
```

```
    printf( "Squares of i\n\n" );
```

```
    i = FIRST;
```

```
    while( i <= LAST )
```

```
    {
```

```
        printf( "%4.1f %5.2f\n", i, i * i );
```

```
        i = i + STEP;
```

```
    }
```

```
}
```

Squares of i

0.0	0.00
0.4	0.16
0.8	0.64
1.2	1.44
1.6	2.56
2.0	4.00

I/O-8

floating-point example, nothing new here

The image shows a C program named `io-exponent.c` that prints the squares of integers from 0.0 to 2.0 in exponential format. The program is annotated with two numbered arrows:

- Arrow 1:** Points to the `printf` statement `printf("%8.2E %8.2E\n", i, i * i);` inside the `while` loop.
- Arrow 2:** Points to the output of the program, which is a list of squares of integers from 0.0 to 2.0, formatted in exponential notation.

The output of the program is as follows:

Squares of i	
0.00E+00	
0.00E+00	
4.00E-01	1.60E-01
8.00E-01	6.40E-01
1.20E+00	
1.44E+00	
1.60E+00	
2.56E+00	
2.00E+00	
4.00E+00	

The program also prints a final line: `\n(%13.7e)\n`, which outputs `I/O-9`.

- 1) exponential/scientific format: mantissa and exponent; precise format is system-dependent and usually coordinated with system datatypes.
- 2) upper and lower case "e"

```

/* io-string.c : example string format */
#include <stdio.h>
#include <string.h>

main() {
    char message[ 100 ];

    strcpy( message, "Hello World" );
    printf( "%s\n", message );
    printf( "%10s\n", message );
    printf( "%-10s\n", message );
    printf( "%20s\n", message );
    printf( "%-20s\n", message );
    printf( "%20.10s\n", message );
    printf( "%-20.10s\n", message );
    printf( "%.10s\n", message );
}

```

I/O-10

string has 11 visible characters, not use of singles to show fields

- 1) print whole string
- 2) field length is a minimum, so no-op in this case
- 3) left-aligned, still a no-op
- 4) pads with blanks on the left (right-aligned by default)
- 5) left-aligned with minus, pads on right with blanks
- 6) format: width.maximum, used to truncate part of string that is displayed; right-aligned by default
- 7) same as above, left aligned
- 8) no width, but truncation: common use to display first “n” characters

```
/* io-table.c : Monthly Payment Schedule */
```

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

```
#define RATE    0.01
```

```
#define INITIAL 10000.00
```

```
#define PAYMENT 750.00
```

```
main()
```

```
{
```

```
    float balance, principal, interest;
```

```
    int month;
```

```
    balance = INITIAL;
```

```
    month = 1;
```

I/O-11

sample program that incorporates many of the ideas. employs a standard technique to guarantee columns with proper spacing.

1

```
printf( "%6s%9s%9s%10s\n\n", "month", "balance",
        "interest", "principal" );
interest = balance * RATE;
principal = PAYMENT - interest;
while( balance > principal )
{
    printf( "%6d%9.2f%9.2f%10.2f\n", month, balance,
            interest, principal);
    balance = balance - principal;
    interest = balance * RATE;
    principal = PAYMENT - interest;
    month++;
}
printf( "%6d%9.2f%9.2f%10.2f\n\n",
        month, balance, interest, principal );
printf( "number of months to repay is %d\n", month
)
```

I/O-12

1) note use of equal field width in both places to ensure columns, in particular, substituting constants in field widths.

month	balance	interest
principal		

1	10000.00	100.00
---	----------	--------

650.00		
--------	--	--

2	9350.00	93.50
---	---------	-------

656.50		
--------	--	--

3	8693.50	86.93
---	---------	-------

663.07		
--------	--	--

4	8030.43	80.30
---	---------	-------

669.70		
--------	--	--

5	7360.73	73.61
---	---------	-------

676.39		
--------	--	--

6	6684.34	66.84
---	---------	-------

683.16		
--------	--	--

7	6001.18	60.01
---	---------	-------

689.99		
--------	--	--

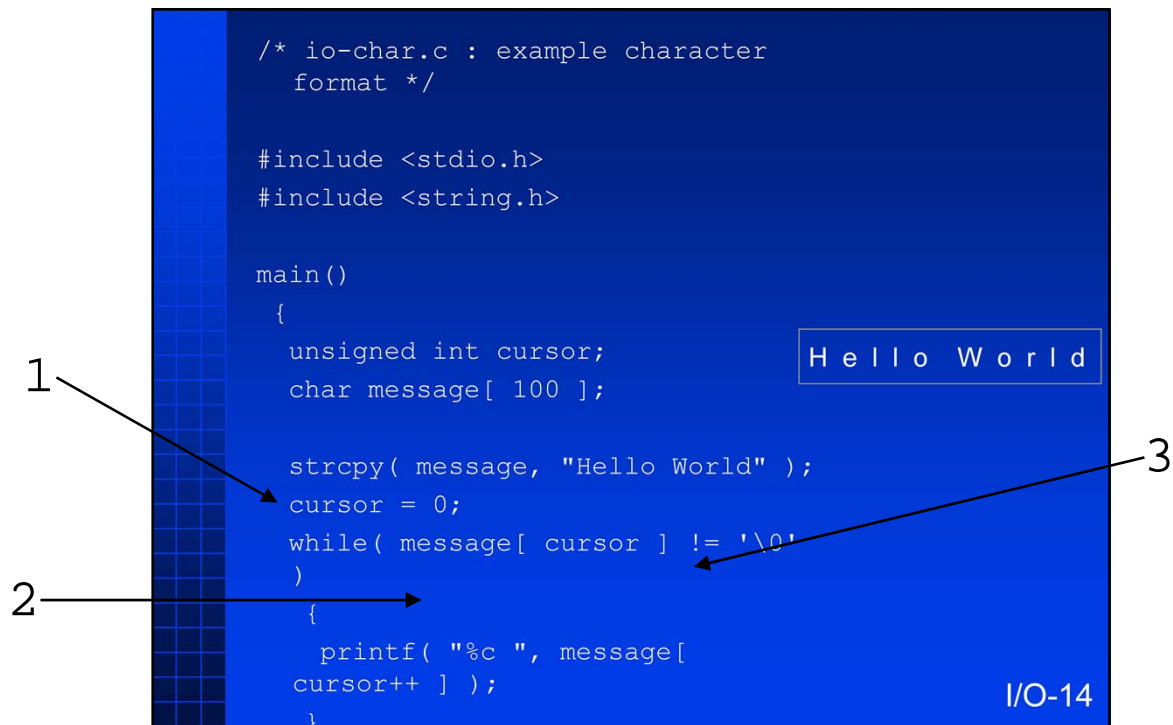
8	5311.19	53.11
---	---------	-------

696.89		
--------	--	--

9	4614.30	46.14
---	---------	-------

703.86		
--------	--	--

I/O-13



printing character strings on character at a time: more array preview

- 1) loop starts at first character and proceeds until nullchar encountered.
- 2) uses `%c` formatting directive to display single character
- 3) note use of post auto increment


```
scanf( control, arg1, arg2, . . . );
```

d	decimal
u	unsigned decimal
o	unsigned octal
x, X	unsigned hexadecimal
i	generalized integer
c	single character
s	character string
f	real / exponential
e, E	
g, G	
%	%

1

2

I/O-15

scanf directives, much the same idea as printf ones.

1) %d for decimal (base 10) integers; %i for integral values of any base (in which case, must follow rules for numbers)

2) no distinction in case (both allowed for completeness)

The Student File

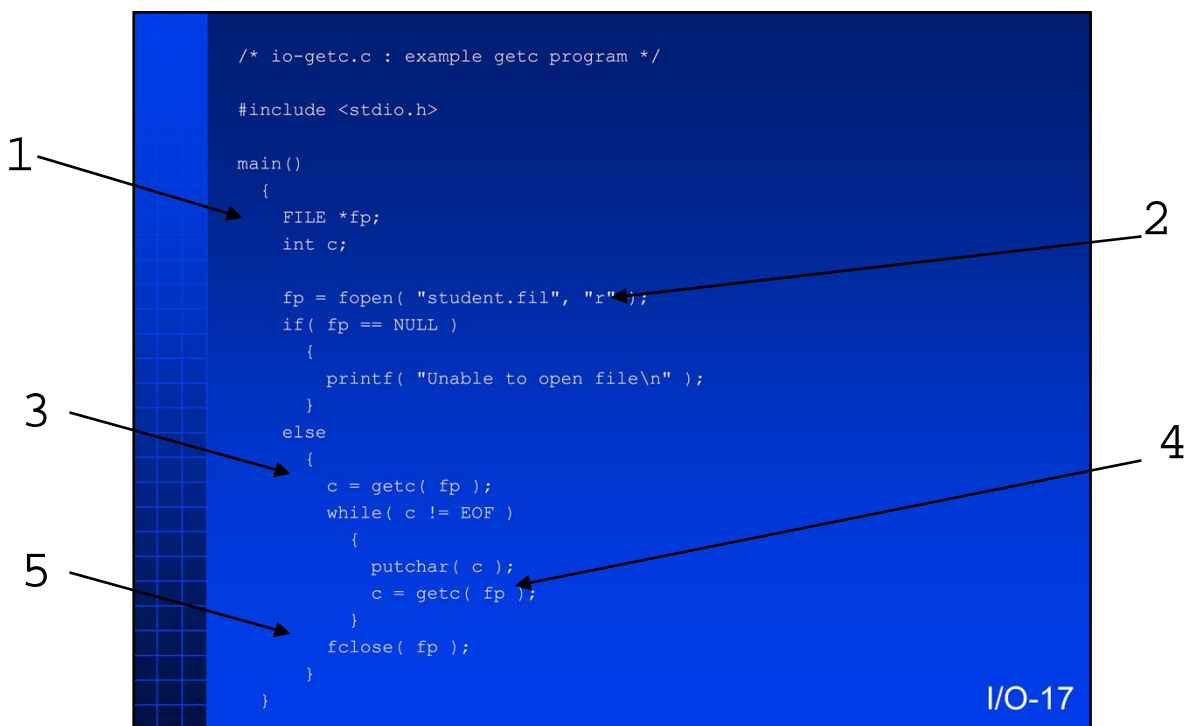
"student.fil"

1110	STEVENS	M	17	65	63	85	56	76
1297	WAGNER	M	15	65	86	85	84	74
1317	RANCOURT	F	16	75	72	70	68	65
1364	WAGNER	M	16	70	58	90	64	83
1617	HAROLD	M	17	85	80	80	75	74
1998	WEICKLER	M	16	72	74	75	75	75
2203	WILLS	F	16	73	72	72	73	84
2232	ROTH	M	17	72	70	70	74	72
2234	GEORGE	M	18	70	70	71	58	69
2265	MAJOR	M	16	65	65	68	68	69
2568	POLLOCK	M	17	89	88	85	92	63
2587	PEARSON	F	15	55	50	49	61	60
2617	REITER	M	17	100	68	69	75	89
3028	SCHULTZ	M	18	69	68	75	74	53
3036	BROOKS	M	18	65	68	69	70	65
3039	ELLIS	M	17	85	85	85	85	85
3049	BECKER	F	15	65	65	65	68	69
3055	ASSLEY	M	16	65	63	60	63	65
3087	STECKLEY	M	15	56	53	85	84	72

I/O-16

Standard files are OK, move on now to look at permanent-file (disk-file) processing. Will be looking at text-files eg the student file. lines of text arranged into columns or fields. Processing will be sequential (start at beginning of file, move forward until end-of-file).

other file-access methods possible, use a different library (eg binary, random-access)



Read student-file and copy verbatim onto standard output (display file)

1) magic variable declaration (pointer)

2) stdio function fopen: open the named file (name is sysdep) for read "r". returns constant NULL (defined in stdio) if failure, some magical value otherwise (don't care what)

3) get a character from the file and put into variable c. if there is no character available, put character constant EOF (defined in stdio) into var c.

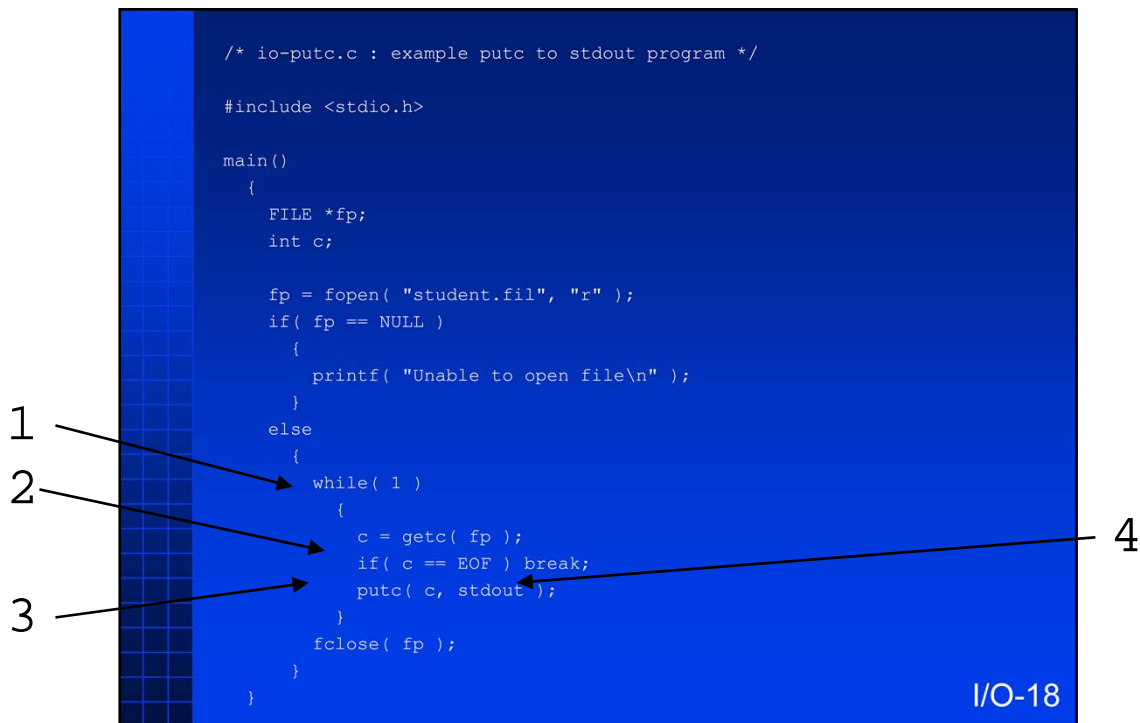
- note that c is declared as a int. allows arbitrary character codes. In particular, allows EOF to be defined as a value that is not any character

4) putchar: put a character on stdout

5) close the file

Note we pay no attention for line structure, no \n processing, just copy char-for-char. if \n encountered, treated and normal char and written. equivalent to printf("\n");

This is a UNIX-ism, view file as sequence of characters with no particular structure, newline may have an interpretation on some devices, others not.



same function, some variations:

- 1) infinite loop: 1 is non-zero which is true
- 2) get out of loop: break. exist from closest-enclosing loop construct.
note style of typing
- 3) put a character on specified file. in this case, file is "stdout", so this is equivalent to putchar(c)
- 4) stdout is declared in stdio.h, declared as FILE *stdout

1

2

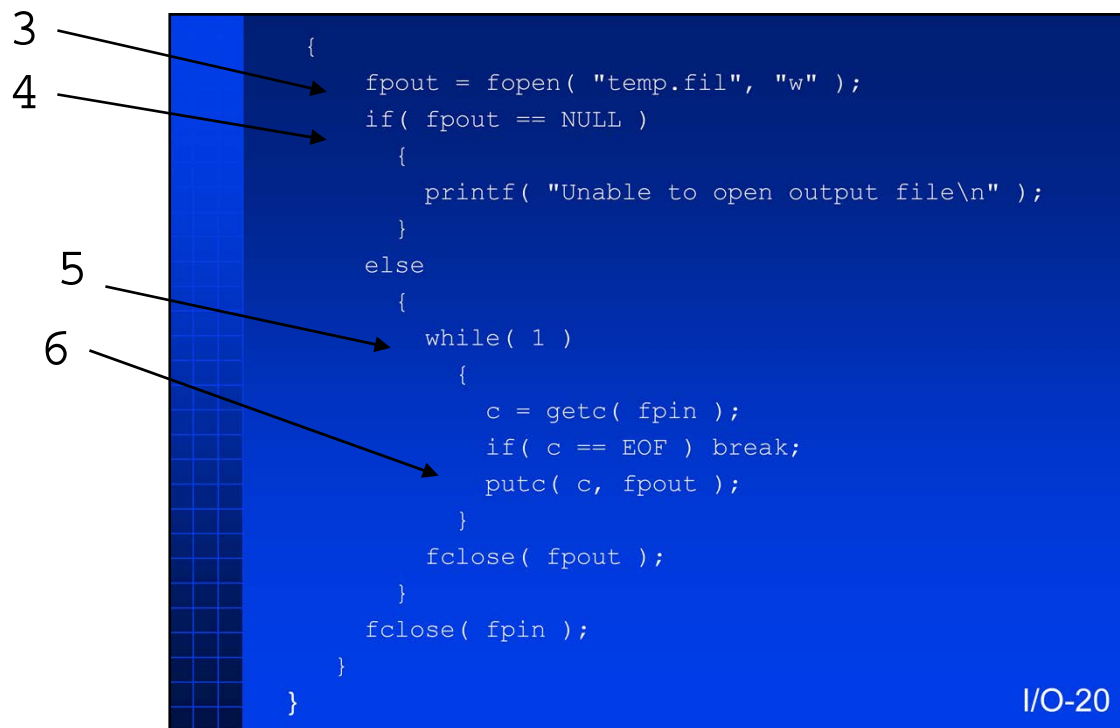
```
/* io-putcf.c : example putc to file program */  
  
#include <stdio.h>  
  
main()  
{  
    FILE *fpin, *fpout;  
    int c;  
  
    fpin = fopen( "student.fil", "r" );  
    if( fpin == NULL )  
    {  
        printf( "Unable to open input file\n" );  
    }  
    else
```

I/O-19

now, write to a different place than stdout: make a copy of the disk file

1) need two file variables: input file and output file

2) open input file and make sure it opened OK



- 3) open output file: name is "temp.fil", mode is "w" for write
- 4) as opening for read, make sure file opens OK
- 5) loop has same structure: copy characters, ignore line structure
- 6) reference our fpout instead of stdout

also: make sure not to close a file that didn't open, so have to pay attention to nesting etc.

1

```
/* io-gets.c : example fgets program */

#include <stdio.h>

#define MAXLINE 100

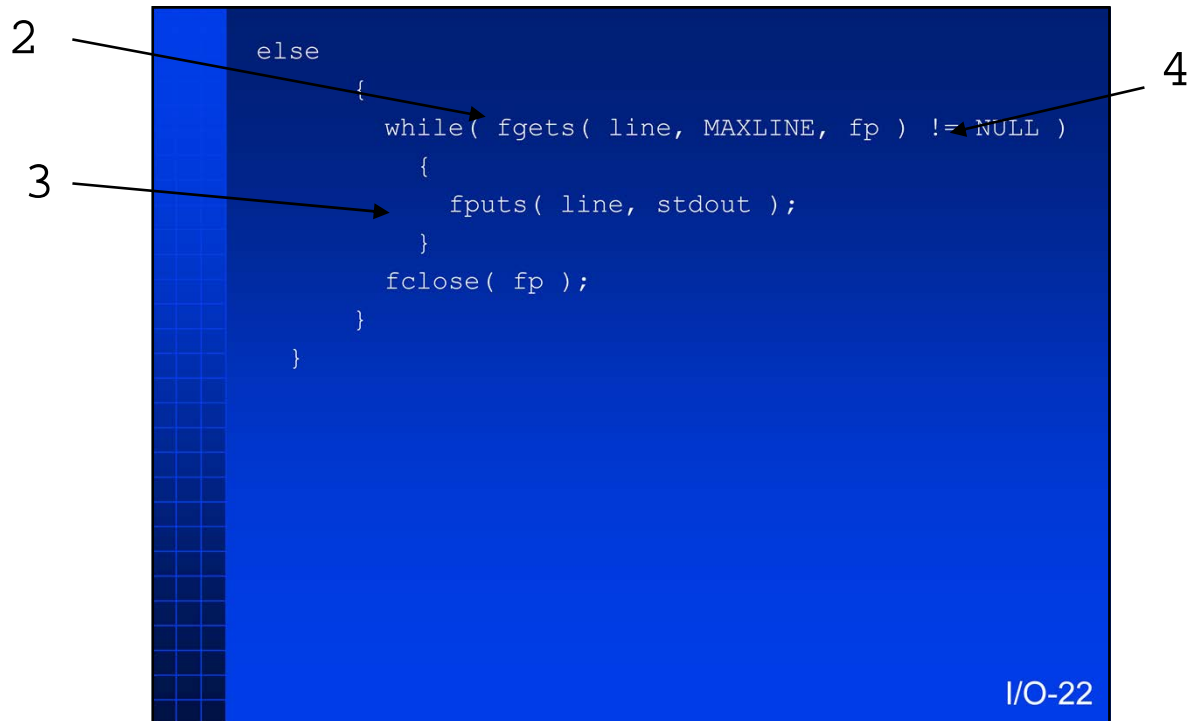
main()
{
    FILE *fp;
    int c;
    char line[ MAXLINE ];

    fp = fopen( "student.fil", "r" );
    if( fp == NULL )
    {
        printf( "Unable to open file\n" );
    }
}
```

I/O-21

sometimes want to maintain record structure. program to read file one line at a time and display on stdout

1) declare a string variable to be used as a line/record buffer. use a symbolic constant for the length, since we'll need this later



2) fgets: gets a string up to and including \n (if any) from a file. destination is "line", source is fp. read no more than MAXLINE characters (use of symbolic constant in both places guarantees no buffer overrun)

3) put a string on given file (stdout here). note that string already contains a \n, so no additional stuff

4) loop termination: fgets returns null if eof or error, so keep going as long as fgets doesn't return null

1

```
/* io-fscanf.c : example fscanf program */

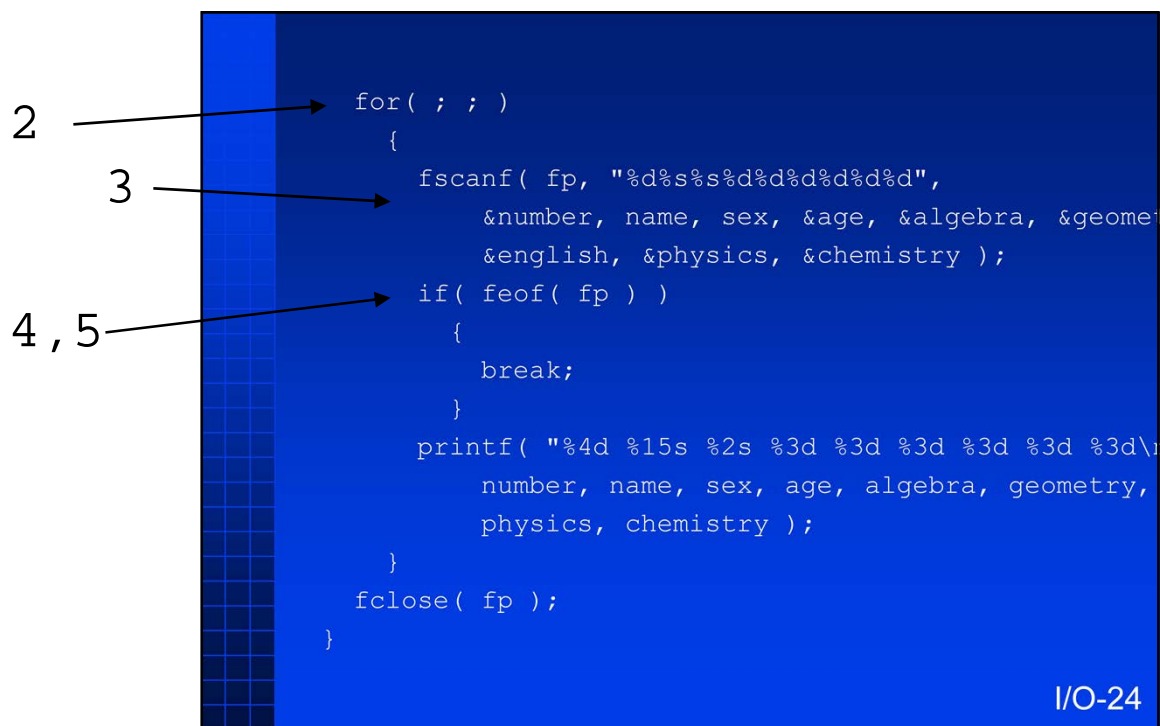
#include <stdio.h>

main()
{
    FILE *fp;
    int number;
    char name[ 15 ], sex[ 2 ];
    int age;
    int algebra, geometry, english, physics, chemistry

    fp = fopen( "student.fil", "r" );
```

I/O-23

can use scanf to get at individual fields
construct control string to match record layout
1) declare bunch of variables to receive fields



2) infinite loop with for statement instead of while

3) fscanf instead of scanf: first parameter is file variable, rest are the same. note mixture of & and not &

- note also that data file is carefully constructed to ensure that each field is blank-delimited.

4) unlike getchar, gets, have no indication from scanf about EOF. have to test explicitly: feof returns true if no more characters in the file.

5) alternate style for if -- break

1

```
/* io-sprin.c : example dynamic control string */

#include <stdio.h>

main()
{
    unsigned int total;
    unsigned int decimals;
    float value;
    char control[ 25 ];

    printf( "Total width?\n" );
    scanf( "%d", &total );

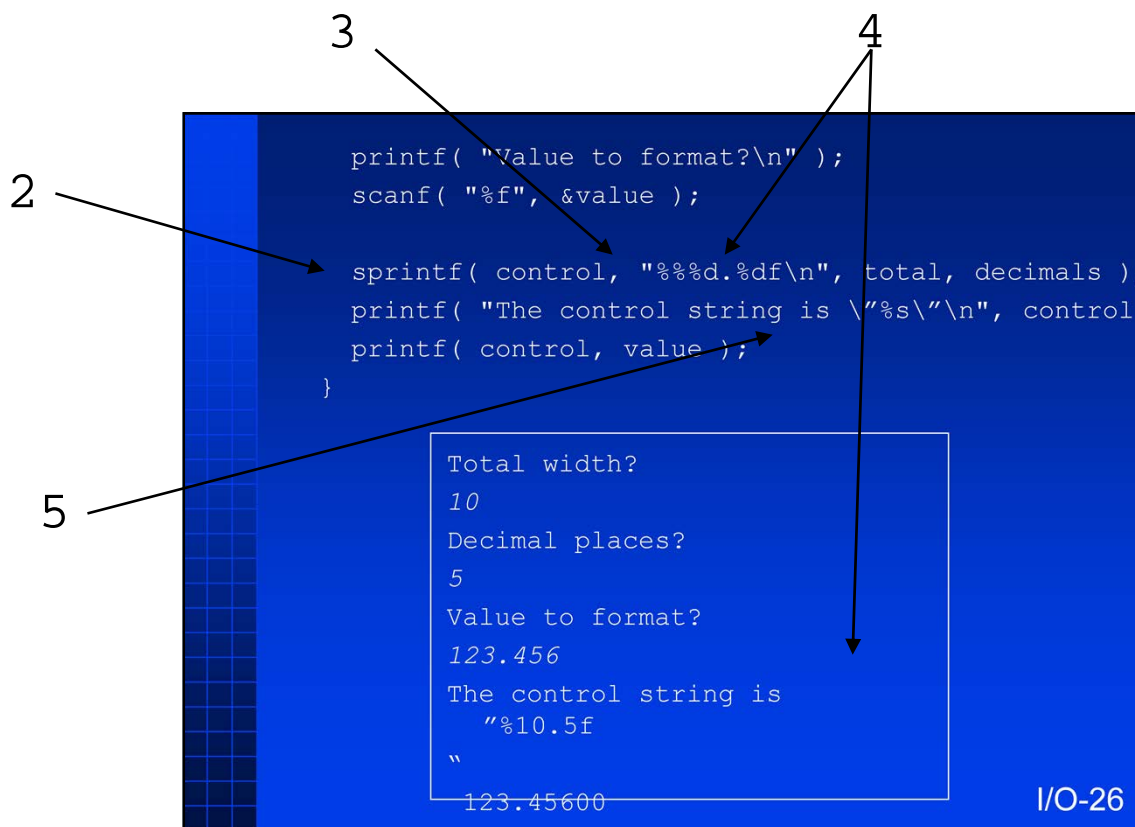
    printf( "Decimal places?\n" );
    scanf( "%d", &decimals );
}
```

I/O-25

Remember that control strings are interpreted at run-time. They can be created and modified at run-time.

Example creates a control string from input provided by the user, then displays a number according to that control string.

1) string variable that will contain the control string, 25 is arbitrary.



- 2) `sprintf`: like `printf`, except that target is not a file, target is a string variable
- 3) want to produce a control string with a %, so need %% [go over control string char by char]
- 4) note `\n` in created control string; note how displayed
- 5) use `\"` to display a “

Summary of printf/scanf

```
printf( control, arg1, arg2, . . . );  
scanf( control, arg1, arg2, . . . );
```

```
fprintf( fp, control, arg1, arg2, . . . );  
fscanf( fp, control, arg1, arg2, . . . );
```

```
sprintf( string, control, arg1, arg2, . . . );  
sscanf( string, control, arg1, arg2, . . . );
```

I/O-27

printf/scanf to standard files, arbitrary files, string variables.

Control Structures

control-1

Control structures, control execution sequence of statements in program.

```
{  
    <statement-1>;  
    <statement-2>;  
    <statement-3>;  
    .  
    .  
    .  
}
```

- a sequential set of statements enclosed in brace-brackets, '{' and '}', form a *compound statement* or *statement block*.

control-2

semi-colons after statements, never after braces

"if" Structure

```
/* cn-if1.c : example if structure */

#include <stdio.h>

#define PASS 65

main()
{
    int mark;

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

control-3

if statement -- choose between two alternative actions called object statements


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

- based on expression, execute one of “then part” or “else part” (object statement)
- the *else* clause is optional

control-4

object statements: then part and else part

objects are single statements, use braces to form compound statement

“nested if” Structure

```
/* cn-if2.c : example nested if structure */

#include <stdio.h>

#define PASS      65
#define HONOURS  85

main()
{
    int mark;

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

control-5

increase complexity, decisions within decisions represented by nested if statements

```

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

```

- *else* is associated with the closest previous *else-less if*

control-6

if-else associations cannot “cross” braces -- if-else must associate within same compound statement. (nesting level)

Eg: get rid of first else; 2nd else doesn't associate with 2nd if because wrong level

Illustration following

```
/* cn-if3.c : example "else-less" if structure */

#include <stdio.h>

#define PASS    65
#define HONOURS 85

main()
{
    int mark;

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

control-7

Braces are necessary to resolve ambiguities.

classic problem in this style of language called “dangling else”

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

printf( "\n" );
}
```

control-8

this illustrates situation just mentioned -- which if does else go with?
use of braces makes it clear: don't cross boundaries

"nested if" Ambiguity

1

```
if( <expression-1> )  
    if( <expression-2> )  
        <statement-1>;  
else  
    <statement-2>;
```

2

```
if( <expression-1> )  
    if( <expression-2> )  
        <statement-1>;  
else  
    <statement-2>;
```

3

```
if( <expression-1> )  
{  
    if( <expression-2> )  
        <statement-1>;  
}  
else  
    <statement-2>;
```

control-9

- 1) correct structure (no braces) for nesting: else goes with closest if
- 2) misleading indentation: still no braces present, so association is same as before
- 3) must have braces to associate else with outer if

“if-else-if” Structure

```
/* cn-elseif.c : example if ... else if ... structure */

#include <stdio.h>

#define PASS      65
#define HONOURS  85

main()
{
    int mark;

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

control-10

Choose one action from a set of actions

```

if( mark >= HONOURS )
{
    printf( "Passed with Honours\n" );
}
else if( mark >= PASS )
{
    printf( "Passed\n" );
}
else if( mark >= 0 )
{
    printf( "Failed\n" );
}
else
{
    printf( "Incomplete\n" );
}
}

```

control-11

- C has no explicit “elseif”, so use cascading if-else-if; presentation is very stylized, but functionally equivalent.

True form:

```

if( blah )
{
    asdfasdf
}
else
{
    if( blah )
    {
        asdfasdf
    }
    else
    {
        if( blah )
        {
            asdfasdf
        }
    }
}
}

```

eliminate brace preceding if:

```

if( blah )
{
    asdfasdf
}
else
    if( blah )
    {
        asdfasdf
    }
    else
        if( blah )
        {
            asdfasdf
        }
}

```

rearrange indendation and spacing

```

if( blah )
{
    asdfasdf
}
else if( blah )
{
    asdfasdf
}
else if( blah )
{
    asdfasdf
}
}

```

- always uses braces to avoid problems

“switch” Structure

```
/* cn-case.c : example case structure */  
  
#include <stdio.h>  
  
main()  
{  
    char grade_letter;  
  
    printf( "Enter grade : " );  
    scanf( "%c", &grade_letter );
```

control-12

if-else-if is very common, explicit statement for handling situation called “switch”, analogous to “case” statement in other languages

```
switch( grade_letter )
{
    case 'A' :
        printf( "Passed with Honours\n" );
        break;
    case 'B' :
    case 'C' :
        printf( "Passed\n" );
        break;
    case 'D' :
        printf( "Failed\n" );
        break;
    default :
        printf( "Incomplete\n" );
}
```

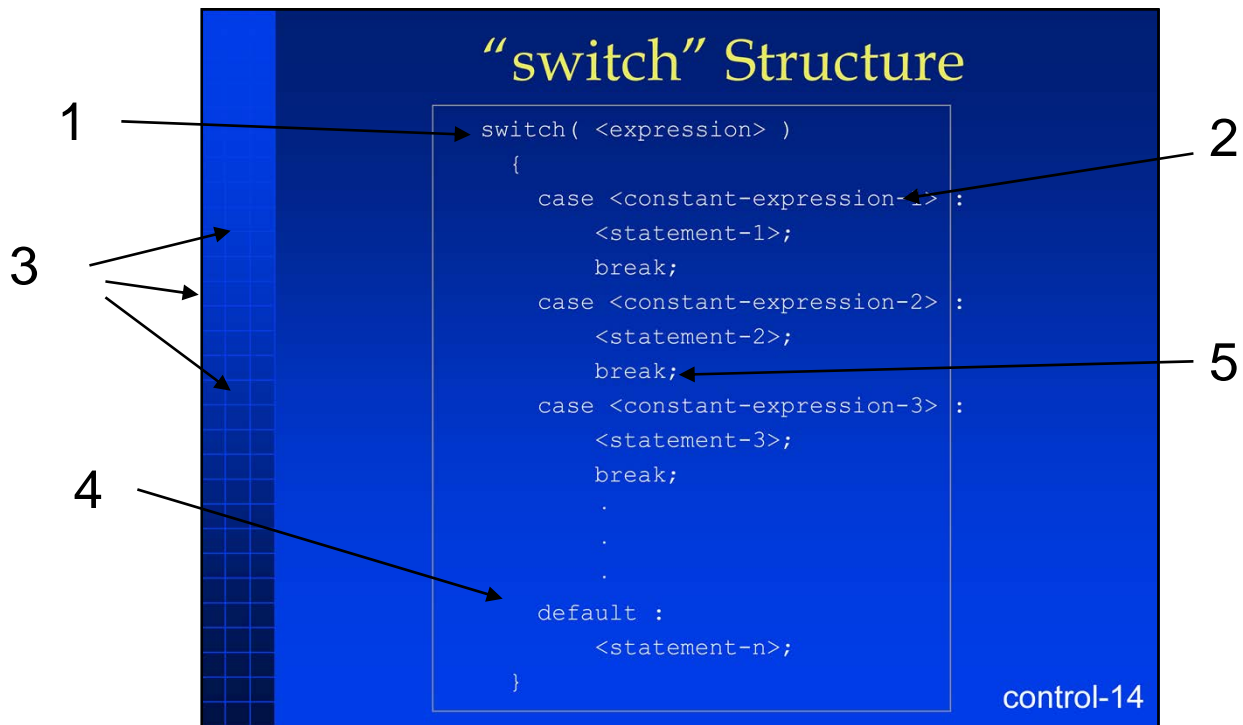
control-13

an example of a switch statement

mechanics: evaluate expression, find matching case label.

execute statements sequentially, if break found, goto end of switch

if none found goto default (aka otherwise)



more formally:

- 1) expression evaluates to one of the case labels
- 2) case labels must evaluate to constants compile-time. only one permitted (but note fall-through technique)
- 3) switch object statement is a statement block (sequence), control transfers to indicated label (like a goto); execution proceeds from that point forward. braces not required within cases, since not object statements
- 4) if no match, go to default case. if no default, do nothing
- 5) break transfers control to end of switch object. if no break, "fall through" to next case. Comes from "computed goto" and "label in statement sequence" idea. Allows multiple labels per action

“switch” Structure

- case labels must be integer or character constant expression
- cases are labels in a statement block
- the *default* case is executed if there exists no matching case label
- the *default* case is optional
- *break* explicitly exits the *switch* structure
- execution “fall through” to next case

control-15

summary, as noted

1

“switch” vs “if-else-if” Structure

```
switch( <expression> )
{
    case <constant-expression-1> :
        <statement-1>;
        break;
    case <constant-expression-2> :
        <statement-2>;
        break;
    .
    .
    .
    default :
        <statement-n>;
}
```

control-16

difference between switch and if-else-if, mostly style and taste. switch can sometimes express “choose one of” idea (dense cases)

some practical differences:

1) expression evaluated once

- some compilers may be able to generate branch table or other special optimization.
- duplicate cases easy to identify
- but: missing break can be pesky

3

"switch" and "if-else-if" Structure

```
if( <expression> == <constant-expression-1> )  
    <statement-1>;  
else if( <expression> == <constant-expression-2> )  
    <statement-2>;  
    .  
    .  
    .  
else  
    <statement-n>;
```

control-17

3) expression evaluate lots (every time up until match) -- could be expensive, if optimizer cannot hoist common code.

- necessary if non-constant tests
- good for sparse cases
- not so good if same action in multiple places: requires "or" expressions that can become messy

“while” Structure

```
/* cn-while.c : example while structure */
#include <stdio.h>

#define FIRST 10
#define LAST 20
#define STEP 2

main()
{
    int i;

    i = FIRST;
    while( i <= LAST )
    {
        printf( "%3d %3d\n", i, i * i );
        i = i + STEP;
    }
}
```

10	100
12	144
14	196
16	256
18	324
20	400

control-18

looping construct to repeat statement

“while” Structure

```
while( <expression> )  
    <statement>;
```

- while <expression> is true (non-zero), execute <statement>

control-19

expression evaluates to false (zero) or true (non-zero)

statement can be a compound statement

if expression initially false, never executes statement

“do-while” Structure

```
/* cn-dowhl.c : example do ... while structure */  
#include <stdio.h>
```

```
#define FIRST 10  
#define LAST 20  
#define STEP 2
```

```
main()  
{  
    int i;  
  
    i = FIRST;  
    do  
    {  
        printf( "%3d %3d\n", i, i * i );  
        i = i + STEP;  
    }  
    while( i <= LAST );  
}
```

10	100
12	144
14	196
16	256
18	324
20	400

control-20

variation of while: do-while

upside-down while, iteration test at end of loop

loop object always executes at least once

“do-while” Structure

```
do  
    <statement>;  
while( <expression> )
```

- similar to *while*, except *<statement>* is always executed as least once

control-21

not much difference between the two, useful if computation of expression depends on execution of statement

“for” Structure

```
/* cn-for.c : example for structure
*/
#include <stdio.h>
#define FIRST 10
#define LAST 20
#define STEP 2

main()
{
    int i;

    for( i = FIRST; i <= LAST; i = i
+ STEP )
    {
        printf( "%3d %3d\n", i, i * i
);
    }
}
```

10	100
12	144
14	196
16	256
18	324
20	400

control-22

ordinary for statement as already seen -- nothing new

“for” Structure

```
for( <expression-1>; <expression-2>; <expression-3>
{
    <statement>;
}
```

- 1 execute <expression-1>
 - 2 execute <expression-2>
 - 3 if true (non-zero), execute <statement>
followed by <expression-3>
- repeat steps 2 & 3; finished when
<expression-2> is false

control-23

three parts: initialization, loop control, incrementor

three steps: initialize, test termination, do statement and increment

not dependent on integral steps, no associated variables (eg read a file)

might never execute statement or expr3

“for” vs “while” Structures

```
for( <expression-1>; <expression-2>; <expression-3>
{
    <statement>;
}
```

```
<expression-1>;
while( <expression-2> )
{
    <statement>;
    <expression-3>;
}
```

control-24

not much difference

for stmt guarantees that incrementor will be done after the stmt; syntax make incrementor very explicit

while relies on user to implement incrementor and put in proper place (nb continue statement)

“break” Statement

```
/* cn-break.c : example break statement */
```

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

```
#define FIRST 10
```

```
#define LAST 20
```

```
#define STEP 2
```

```
main()
```

```
{
```

```
    int i;
```

```
    i = FIRST;
```

```
10 100
```

```
12 144
```

```
14 196
```

```
16 256
```

```
18 324
```

```
20 400
```

control-25

```
while( 1 )
{
    printf( "%3d %3d\n", i, i * i );
    if( i >= LAST )
    {
        break;
    }
    i = i + STEP;
}
```

- *break* causes explicit exit from enclosing *for*, *while*, *do-while*, or *switch* statement
- also:

```
if( i >= LAST ) break;
```

control-26

use break to get out of loops & switch
not if statement
one level at a time

“continue” Statement

```
/* cn-cntnu.c : example continue statement */

#include <stdio.h>

#define FIRST -3
#define LAST  3
#define STEP  1

main()
{
    int i;

    printf( "Squares of i\n\n" );
```

control-27

continue: somewhat of a novelty?


```

for( i = FIRST; i <= LAST; i = i + STEP )
{
    if( i == 0 )
    {
        continue;
    }
    printf( "%3d %3d\n", i, i * i );
}

```

-3	9
-2	4
-1	1
1	1
2	4
3	9

- *continue* causes explicit initiation of next iteration of enclosing *for*, *while*, or *do-while* statement

control-28

“go around again” for all looping structures (closest enclosing, no way to be explicit)

in for stmts, proceeds directly to for incrementor expression, then test
in while stmts, goes directly to top and tests: if incrementor not placed carefully, problem

ok for use with for, generally dangerous for others

Program Structure

program-1

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" );
    }
}

```

program-2

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

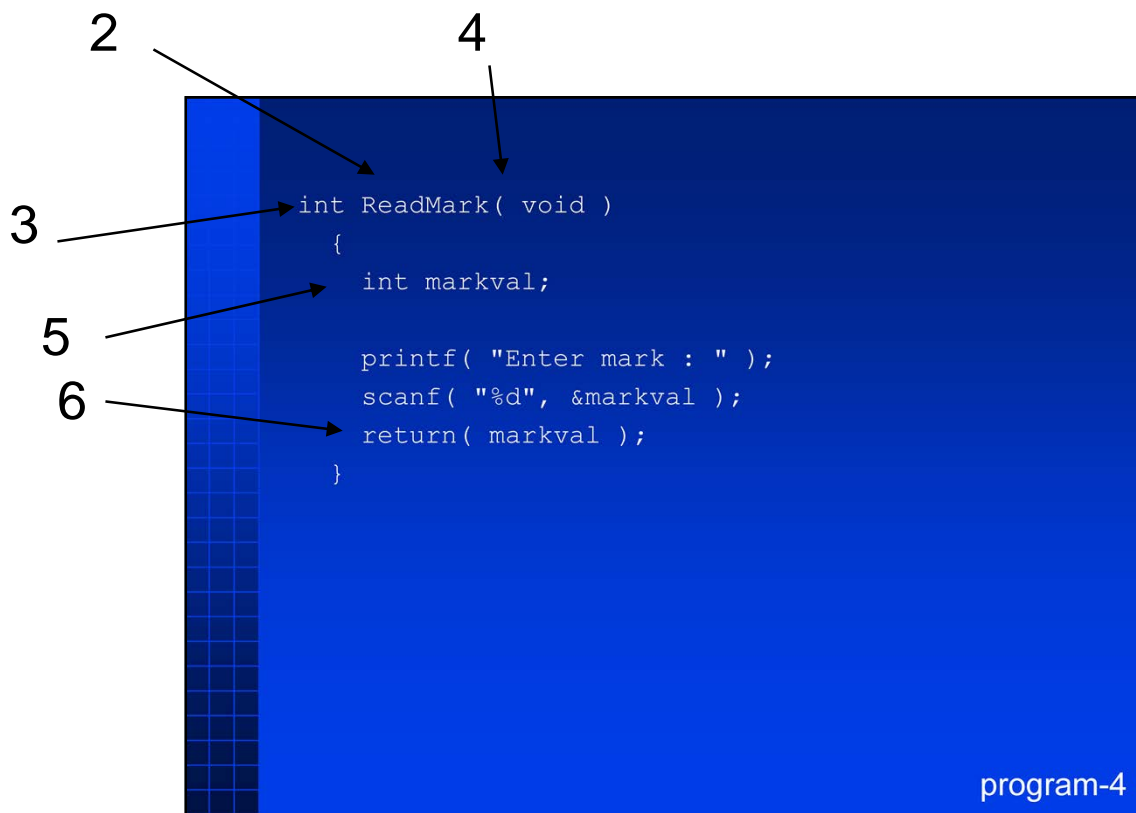
main() {
    int mark;

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

program-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



- 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. missing return might generate a warning, not an error

```

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

program-5

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 refer to mark in readmark, 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

program-6

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

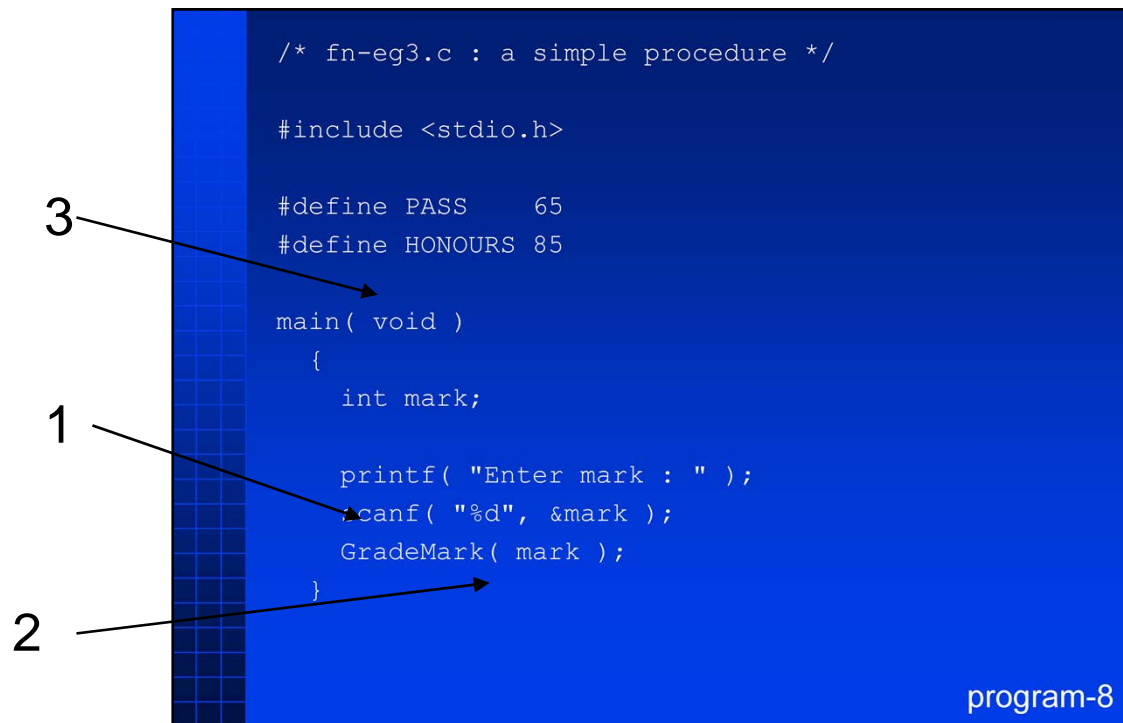
```
<function-type> <function-name>( void )  
{  
    .  
    .  
    return( <expression> );  
    .  
    .  
}
```

- *<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

program-7

general 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" (following Pascal nomenclature)



different kind of function: 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

1

2

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

program-9

1) procedure definition: return-type is void

2) 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.

No return statement -- nothing to return -- returns at end-of-procedure

```
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

program-10

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 (eg strings, arrays) are passed by reference
- *void* for parameters means no parameters
- *void* for <fn-type> means no value will be returned

program-11

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 variable

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

two kinds of function:

- returns a value
- has a side-effect

either type accepts args, C library fns almost always return a value

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();
```

```
}
```

program-12

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" );
    }
}
```

program-13

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

nothing but a big side-effect

Source management

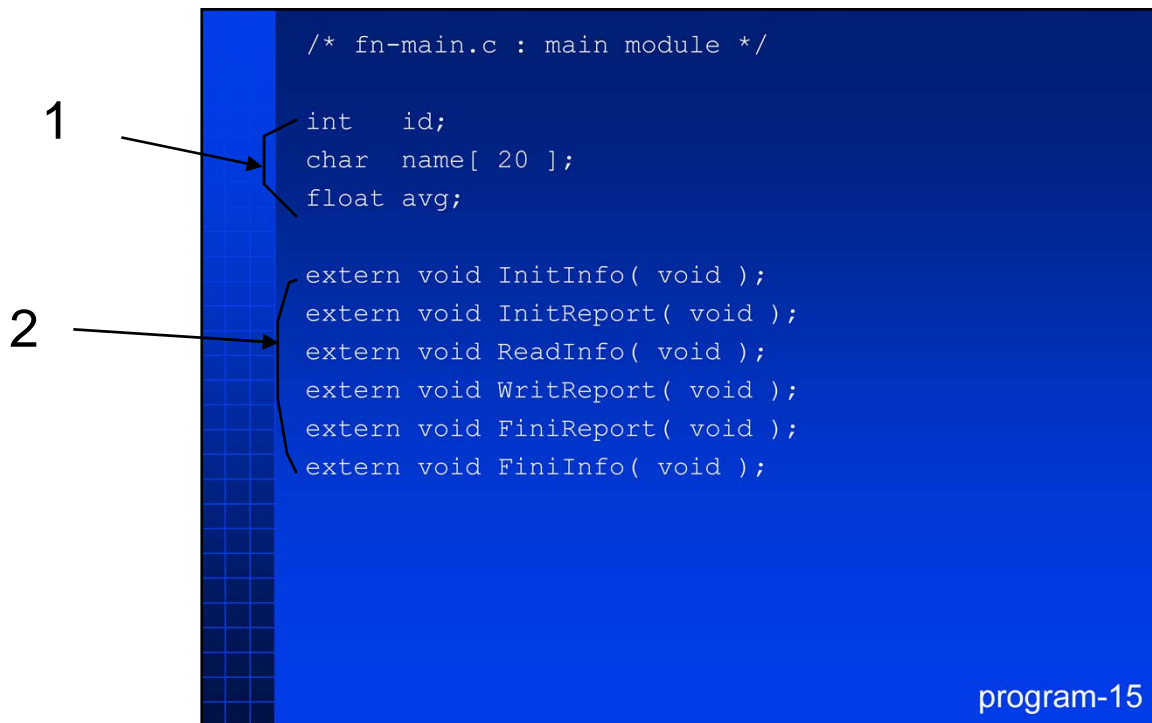
- Programs composed of functions
- Related functions gathered into modules (files, compilation units)
- Need to manage inter-module visibility of functions, variables

program-14

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.

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



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

Program which processes marks, three modules: main, info (reads) and report (write) [flip forward & back]

Big part of the issue of compilation-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();
}
```

program-16

definition of main (same source file as previous slide)

very stylized organization: initialize stuff, do stuff, finish stuff (object oriented, modular)

uses lots of functions: 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.

Function prototypes

- Declarations of functions defined elsewhere
- “Elsewhere” can mean:
 - another module
 - somewhere else in the same module
- External declaration vs forward declaration
- Use private “.h” file for declarations

program-17

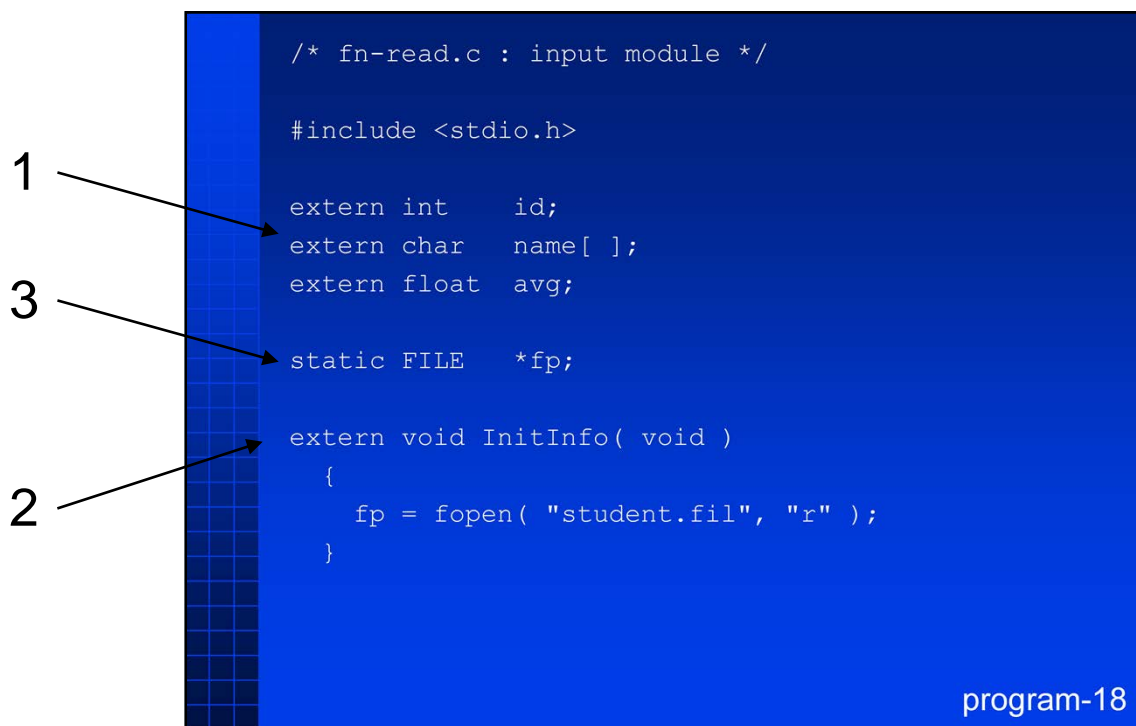
Formally, declarations are called “function prototypes”

Terminology and syntax clash: “extern” makes us think of external definition (defined in another module), but that’s not correct. Extern means globally visible, it’s defined if the definition is given.

- duplication definitions not caught until after compilation

Forward declaration idea: things must be defined before used. may have ordered source-code such that need to have a forward declaration before definition. Perfectly acceptable, even advocated.

put all declarations into h file and include everywhere. provides some degree of inter-module typechecking: one of the files will be the definition -- if that definition and the declaration don’t agree, error



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. maintains value between invocations

1

```
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 = ttl / 5.0;
    }
}

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

program-19

remaining function definitions.

(note use of file scope variable and program scope variables.

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

1) %* directives: scan the thing, then throw it away.

1

```
/* 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 )
{
}
```

program-20

last module; provides definition of Report/write 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

program-21

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;

static int    count = 0;
static float  total = 0.0;

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

program-22

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. Static vars, visible only within this module.

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). Could also be initialized in InitReport.

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

extern void FiniReport( void )
{
    printf( "\n      Average      %5.1f\n", total / c
}

```

program-23

references to variables.

modular development: enhance functionality, replace empty function.

main program stays the same

(program computes average of averages. in WritReport, which is called once per student, aggregate student's average and count. in finireport, compute average of aggregate

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

program-24

lovely new output

2

```

/* fn-writ3.c : output module (version 3) */
#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() );
}

```

program-25

1

Final revision: change original program to output letter grades instead of numbers. Need a function that converts 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. Would be required for mutual recursion, intra-module calling of global function.

```
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 );
}

extern void FiniReport( void )
{ }
```

program-26

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

program-27

lovely new output

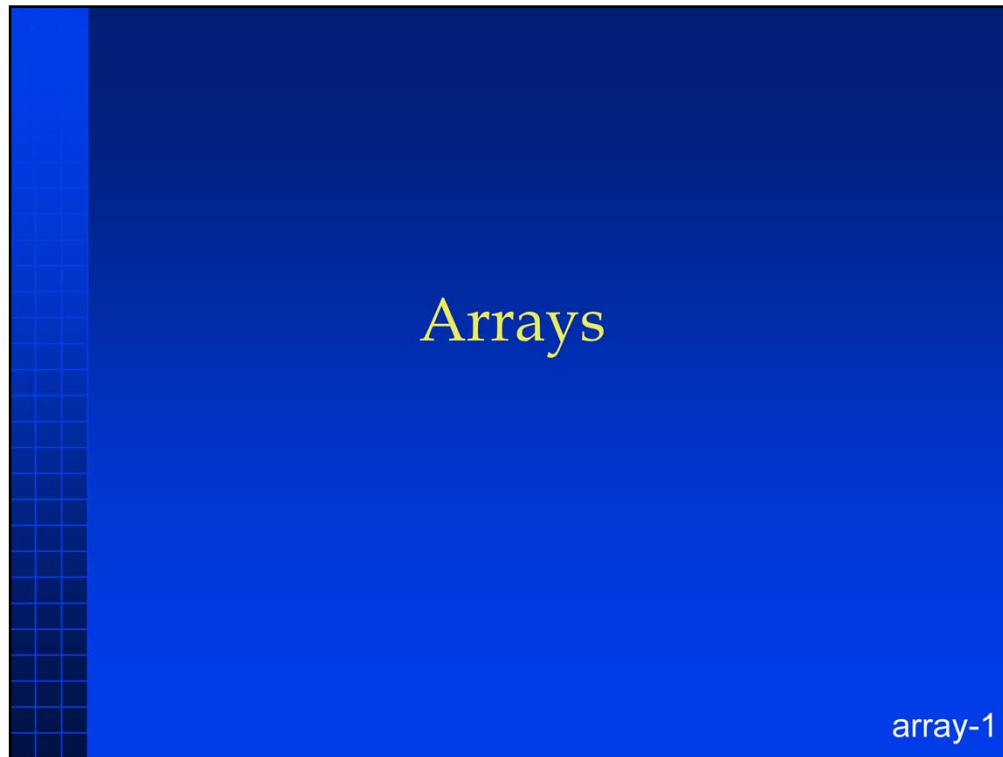
Summary

- Programs are collections of functions
- Functions are grouped into modules (files, compilation units)
- Functions can return values (or not)
- Functions can accept parameters (or not)
- An entity has one definition, possibly many declarations
- Function prototypes for external and forward declarations; also inter-module checking
- Three levels of scope: global, file, local

program-28

function prototyping especially useful with .h files

scoping applies equally to functions and variables



Arrays:

groups or collections of variables of the same type (homogeneous set), individual variables called elements. Elements are numbered: 0, 1, etc.

aka: vector, matrices, tensors

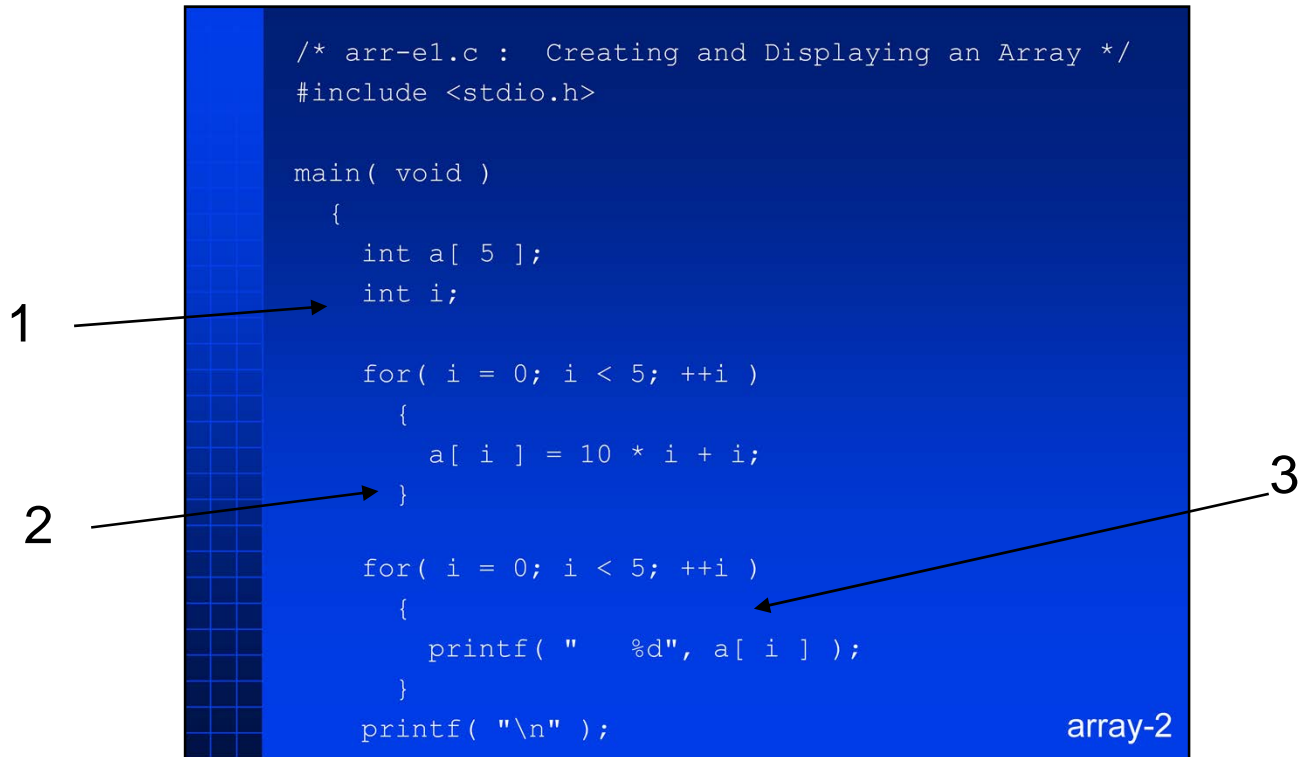
Have to be able to manipulate elements and entire arrays. Operation called subscripting which selects elements from entire arrays. Subscripting involves integer value that says which element selected

arrays are direct-access structures: can select elements in any order, no restrictions on access

In C, fixed # elements; not all elements have to be used (eg strings).

Numbering system for elements starts at 0, no choice.

No checking on bounds! run of the end without notification



- 1) Array definition: 5 elements numbered 0 through 4 inclusive
- 2) subscripting operation: select ith element from 0. In this eg, individual elements are a[0], ..a[4]
- 3) array elements can be used like ordinary variables. subscripting chooses an element that is equivalent to a simple value.

manipulate the array (access all the elements), use a for statement, use int i to select elements.

```
for( i = 4; i >= 0; --i )
{
    printf( "    %d", a[ i ] );
}
printf( "\n" );
}
```

0	11	22	33	44
44	33	22	11	0

Memory representation of "a"

0	11	22	33	44
---	----	----	----	----

array-3

print the array backwards

program output

array in memory: no null chars, no indication of # elements


```

#define SIZE ...
.
.
int table[ SIZE ];
unsigned int i;
.
.
for( i = 0; i < SIZE; i++ )
{
    table[ i ] ...
}

```

- `table[0]` is the first element
- `table[i]` is the $i+1^{\text{st}}$ element
- `table[SIZE-1]` is the last element
- remember the NULL character!

array-4

Key things in array definitions:

- basetype (type of individual elements)
- # elements

multiple dimensions: add more []. abstraction is array of array, row-major ordering.

string definitions: remember to add 1 to size to hold nullchar.

Eg: want 10 “visible” characters, need 11 characters of storage:

`str[11] == str[0] .. str[9]` are 10 visible characters

`str[10]` for the nullchar

1) `table[SIZE]` is past the end of the array

1

```
/* arr-elb.c : Reading in and Printing an Array */  
  
#include <stdio.h>  
  
main( void )  
{  
    int a[ 5 ];  
    int i;  
  
    printf( "Enter 5 int's : " );  
    for( i = 0; i < 5; ++i )  
    {  
        scanf( "%d", &a[ i ] );  
    }  
}
```

array-5

1) array elements just like simple scalar variables, so need & as before in order to address elements.

```
for( i = 0; i < 5; ++i )
{
    printf( "    %d", a[ i ] );
}
printf( "\n" );

for( i = 4; i >= 0; --i )
{
    printf( "    %d", a[ i ] );
}
printf( "\n" );
}
```

1 3 5 7 9

1 3 5 7 9
9 7 5 3 1

array-6

continued

Passing arrays as parameters

/* Arrays as parameters to a function */

main(void)

{

int numbers[10];

...

zero(numbers, 10);

...

}

void zero(int x[], int size)

{

for(--size; size >= 0; --size)

x[size] = 0;

}

- Only a reference to the array, not the whole value, is passed to "zero".

array-7

Individual elements are OK, want to be able to deal with whole collection, too.

Eg want to write a function that sets all elements to zero.

1) definition of array, 10 elements numbered 0..9

2) invoke function, pass array and number of elements. have to do this because there is no implicit record of # elements.

3) definition of function. note empty [] in parameter list to indicate incoming array.

Array passing sends only a reference/pointer to the array. the expression "value" of an entire array is its address/reference.

Any changes to "x" in function are really changes to numbers in main.

Think of scanf of strings -- no &, redundant.

Note structure of zero. clever pre-decrements to yield elements 9..0

String parameters

```
main( void )
{
    char  message[ 100 ];
    ...
    fill( message, 100, '*' );
    ...
}

void fill( char c[], int size, char fillchar )
{
    c[ --size ] = '\\0';
    for( --size; size >= 0; --size )
        c[ size ] = fillchar;
}
```

array-8

since strings are arrays, can pass them, too. same rule applies

note use of size parameter, says how big the array is. we put the nullchar in the last position, which has index size-1. target will contain size-1 visible characters, indexed 0..size-2.

String parameters

- String literals are arrays, too
- What about:

```
fill( "hello world", 5, '?' );
```

- Not detected! For example:

```
fill( "hello world", 5, '?' );  
printf( "%d <%s>", strlen( "hello world" ),  
        "hello world" );
```

produces:

```
4 <????>
```

array-9

as seen, string literals (enclosed in double-quotes) are literals that are constructed the same way as string variables. string literals are “array constants” too bad can’t generalize to arrays of other types.

can certainly pass string literals to functions, but what if the function tries to modify the incoming parameter (its a reference, after all)

unfortunately, not detected. compiler changes first string literal, but, because identical string literals are merged, all other uses of identical literal are also modified. (fortran has same problem “1=2”).

can declare parameters as const. If you want to write a function that can accept literals, declare as constant. Wouldn’t help above, since function is intended to modify parameter.

any attempt to modify const parameter will be an error -- unfortunately this doesn’t work in watcom c.

previous versions of notes: 6 pages of sample programs follow

Multi-dimensioned Arrays

```
/* arr-e5.c : Matrix manipulation */

#include <stdio.h>

#define HEIGHT 5
#define WIDTH 2

main( void )
{
    int m[ HEIGHT ] [ WIDTH ];
    int h, w;
```

array-10

simple matrix, row-column (height==row; width == column); has HEIGHTxWIDTH individual elements

```

for( h = 0; h < HEIGHT; ++h )
    for( w = 0; w < WIDTH; ++w )
    {
        m[ h ] [ w ] = 10 * h +
w;
    }

```

```

for( h = 0; h < HEIGHT; ++h )
{
    for( w = 0; w < WIDTH; ++w )
    {
        printf( "    %d", m[ h ]
[ w ] );
    }
    printf( "\n" );
}

```

0	1
10	11
20	21
30	31
40	41

array-11

fill the array: select row, then traverse columns: coding view is array of 5 elements, each element is an array itself

nested for statements common usage. outer loop selects row, inner loop traverses columns.

display the matrix, \n after each row

1

2

```
/* arr-e5b.c : Reading arrays of strings */
#include <stdio.h>

main( void )
{
    char names [ 5 ] [ 20 ];
    int i;

    for( i = 0; i <= 4; i++ )
    {
        printf( "Enter name : " );
        scanf( "%s", names[ i ] );
    }
    for( i = 4; i >= 0; i-- )
    {
        printf( "%s\n", names[ i ] );
    }
}
```

array-12

1) an array of strings, really a matrix of characters, either 5, 20-char strings or 5x20 matrix of characters; 5x20=100 characters total

use the array or array concept to advantage:

2) missing subscript, names[i] select an entire array, which is a string still don't need &, since expression yields an array.

Multi-dimensioned Arrays

```
char names [ 5 ] [ 20 ]
```

- “names[i]” refers to a row (i.e., 20-character array)
- “names[i] [j]” refers to a single character

array-13

as stated

Multi-dimensioned Arrays

```
#define DIM1 ...  
#define DIM2 ...  
.  
.  
type matrix[ DIM1 ] [ DIM2 ] ;
```

- `matrix` is DIM1 x DIM2 elements

array-14

elements is product of each dimension size

good coding practice to use symbolic constants as shown here, write code to use constants in for-loops etc.

previous versions: 3 page example program followed

Array initialization

/* arr-e7.c : array initialization */

#include <stdio.h>

int points[5] =

{
 13, 0x50, -967, 42, 8888
};

unsigned int sequence[] =

{
 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
};

array-15

many arrays are tables of information, commonly want to be able to initialize them

C supports “structured” constants for initialization.

1) # elements given explicitly, followed by initial values for elements. if elements missing, some default value (zero) [this generalizes to all variables, in fact]

2) implicit # elements, # initial values determines # elements in definition.

3

```
#define INTSIZE sizeof( int )

main( void )
{
    printf( "points: %d elements\n",
           sizeof( points ) / INTSIZE );
    printf( "sequence: %d elements\n",
           sizeof( sequence ) / INTSIZE );
}
```

points: 5 elements
sequence: 10 elements

array-16

code that computes # elements

3) sizeof compile-time function that given #bytes storage for a type or a variable

String initialization

```
/* arr-e7a.c : Char vs. String Initialization */
#include <stdio.h>
#include <string.h>

char str1[ ] = { 't','h','i','s','\0' };
char str2[ ] = { "this" };

main( void )
{
    printf( "%s\n",
            ((strcmp(str1, str2)==0) ? "same" : "diff")
    )
}
```

same

array-17

strings are arrays, they can be initialized, too

string initialization: two styles

1) as array of characters: have to place `\0` ourselves

2) using string literal, compiler inserts `\0`

choice depends on how you want to think about things

3) ? operator



proof

```
/* arr-e8.c : Convert month number to string name */  
  
#include <stdio.h>  
  
static char Months[ 12 ] [ 10 ] =  
{  
    "January",  
    "February",  
    "March",  
    "April",  
    "May",  
    "June",  
    "July",  
    "August",  
    "September",  
    "October",  
    "November",  
    "December"  
};
```

array-19

initializing an array of string == initializing a matrix of characters.

use representation that makes sense: 12 rows of 10-char strings

cannot omit second dimension in definition here: must know big strings are
(could omit first, compiler can count # strings), due to matrix idea,
rectangular block of characters, compiler will pad out to 10-char each.


```

main( void )    {
    int day, month, year;

    scanf( "%d %d %d", &day, &month, &year );
    printf( "%s %d, %d.\n",
            Months[ month-1 ], day, year );
}

```

J	a	n	u	a	r	y	•	
F	e	b	r	u	a	r	y	•
M	a	r	c	h	•			
.								
.								
.								
D	e	c	e	m	b	e	r	•

array-20

storage representation of array
bad program, doesn't bounds-check

Structured initializers

```
/* arr-e9.c : Initialize a 3 X 4 matrix. */  
  
const int I[ 3 ] [ 4 ] =  
    {  
        { 1, 0, 0, 2 },  
        { 0, 1, 0, 2 },  
        { 0, 0, 1, 0 }  
    };
```

array-21

matrix initialization: can use braces to construct an initializer that follows the structure of the array.

here, 3 rows of 4 integers each

generally, can construct initializers for any number of dimensions, use {} to show dimensions

previous versions: 4-page example program.

Pointers, etc.

pointer-1

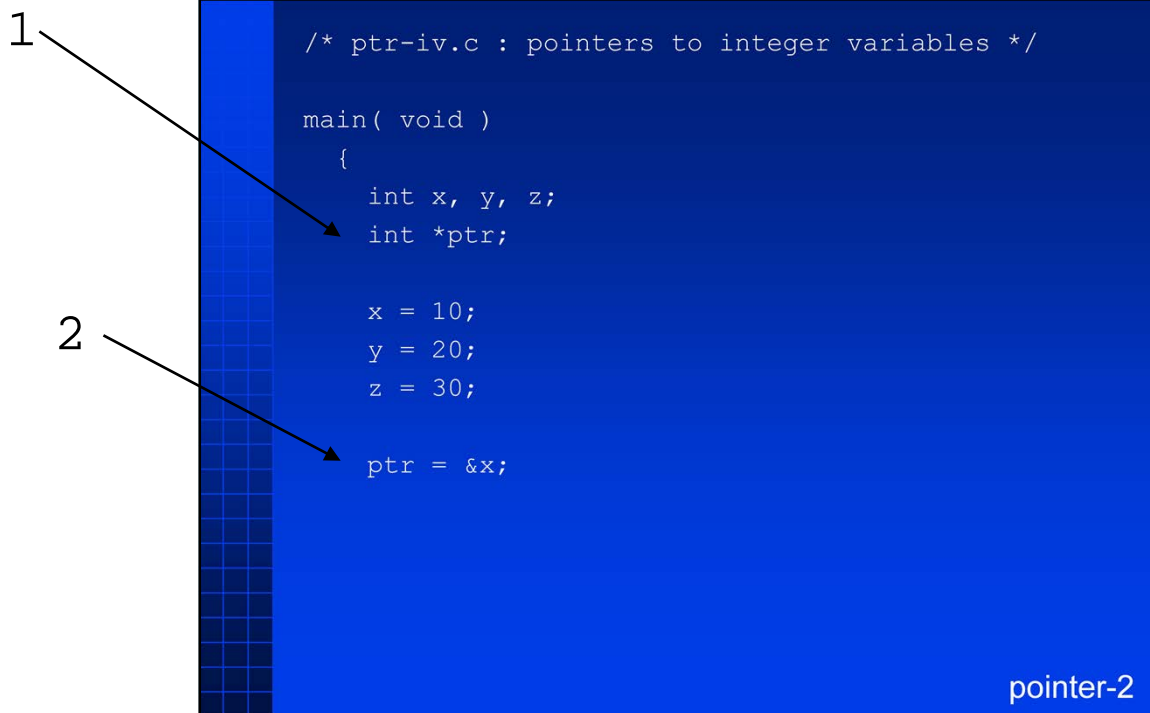
significant part of C's expressive power comes from ability to manipulate pointers. aka references, addresses.

consider a variable. sometimes we're not interested in the contents of the variable, we want the variable itself (box and contents analogy)

useful for efficiency, eg arrays, don't want to transmit entire set of values

also useful for accessing underlying hardware: computer memory is big array of integers or bytes/characters

later, will see equivalence of pointers and arrays



- 1) ptr is a variable that contains a reference to an integer. ptr is not an int, it is a pointer to an int.
 - 2) ptr is assigned a reference to the variable x. ptr contains the address of x.
- unary & is the “address of” operator

		/*	x	y	z	ptr	*ptr	*/
		/*	10	20	30	&X	10	*/
1	x = 40;	/*	40	20	30	&X	40	*/
	ptr = 50;	/	50	20	30	&X	50	*/
	ptr = &y;	/*	50	20	30	&Y	20	*/
2	*ptr = 60;	/*	50	60	30	&Y	60	*/
	z = *ptr;	/*	50	60	60	&Y	60	*/
	ptr = x;	/	50	50	60	&Y	50	*/
	}							

pointer-3

explain slide:

comments show contents of variables. ignore last column for a second starts out as...

assign 40 to x

1) assign 50 to wherever ptr points; assign 50 to the variable whose address is contained in ptr. opposite of & (taking reference) -- call dereferencing. aka indirection.

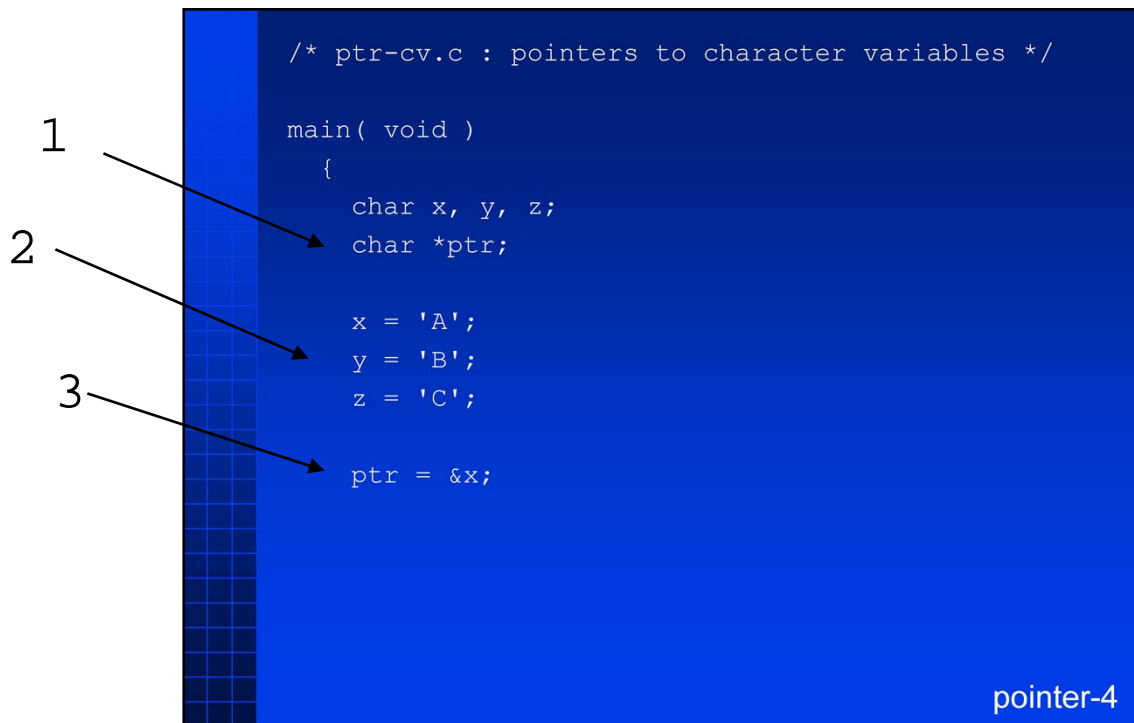
in this case, dereference as a target of assignment,

last column shows current dereference of ptr. go back and review.

[explain each line]

...

2) dereference as an expression



- 1) ptr is a pointer to a single character
- 2) single-char values
- 3) ptr is assigned the address of x. note that this is the same expression as assigning an integer address. & yields a pointer in any case, the definition specify precisely what it points to

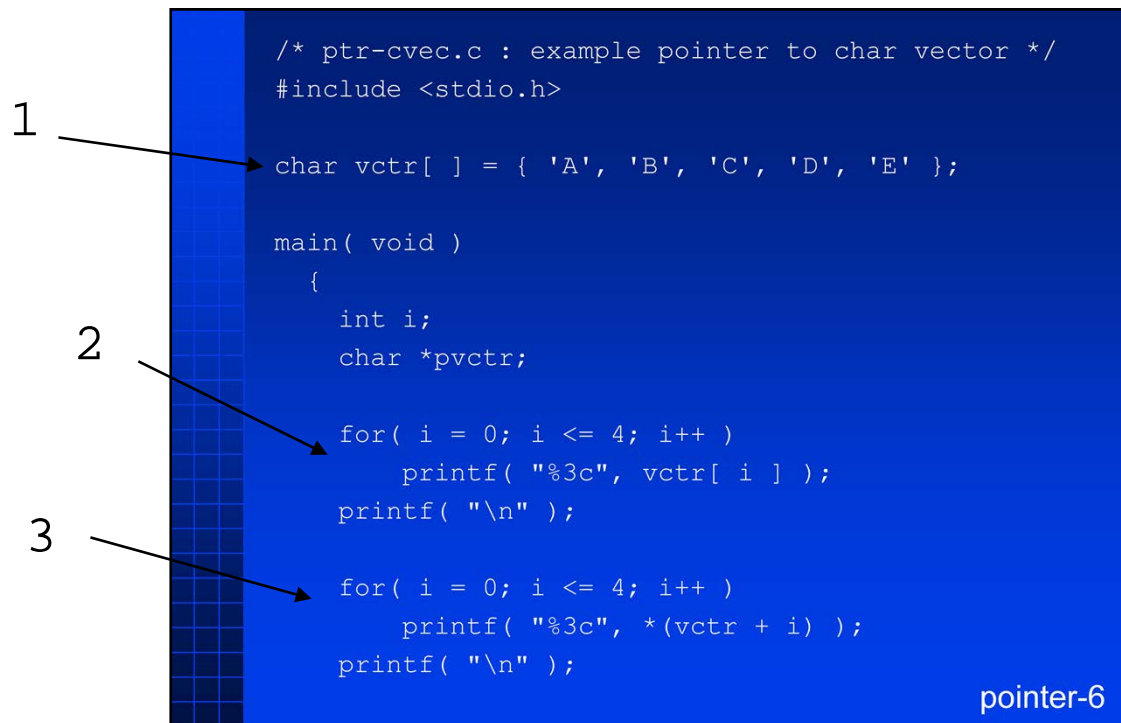
1

```
/* x y z ptr *ptr */
/* A B C &X A */
x = 'D';
/* D B C &X D */
*ptr = 'E';
/* E B C &X E */
ptr = &y;
/* E B C &Y B */
*ptr = 'F';
/* E F C &Y F */
z = *ptr;
/* E F F &Y F */
*ptr = x;
/* E E F &Y E */
}
```

pointer-5

same structure as before, comment columns show values

1) dereferencing a pointer, same as before. since its a ptr to char, we assign a char.



have seen that array-names are already a reference, don't need & to get address of array. what would be an appropriate thing to store the address of an array? eg foo = vector, how to declare foo? its a reference, so suggests a pointer variable; but -- pointer to what? pointer to same thing as the basetype of the array.

Eg: arrayname is pointer to first element, ignore the rest [chalkboard]

So arrayname is reference is pointer to basetype. what is subscripting? start at first element and move along. can do this with [], can also do with pointer arithmetic [chalkboard: +1 to char*]

So subscripting is equivalent to pointer arithmetic.

These concepts are fundamental to C: can use pointer variables and array-names interchangeably

can subscript pointers, can do pointer dereferencing to array-names.

1) array of characters, not a "string" (no nullchar). all strings are array of char, but not all array of char are string.

2) simple array subscripting, [] yields a single character

3) ptr arith on array-name (compute address and dereference)


```

4  pvctr = &vctr[ 0 ];
5  for( i = 0; i <= 4; i++ )
   printf( "%3c", *(pvctr + i) );
   printf( "\n" );

6  for( i = 0; i <= 4; i++ )
   printf( "%3c", pvctr[ i ] );
   printf( "\n" );

7  for( pvctr = vctr; pvctr <= &vctr[ 4 ]; pvctr++ )
   printf( "%3c", *pvctr );
   printf( "\n" );
   }

```

pointer-7

4) pvctr gets address of first element of vctr

can also use `pvctr = vctr` since array name is already reference

5) pure pointer arithmetic. note same as array-name case

6) subscripting a pointer. the `[]` are really operators that are defined to work on things containing an address -- any address will do. in this case, operation is pretty much identical to preceding

7) cursoring: changing the pointer variable itself (moves the pointer along the array)

8) loop termination: address comparison `&vctr[4]` is the address of the last (5th) element. loop continues as long as not advances past end of array

1

```
/* ptr-ivec.c : example pointer to int vector */
#include <stdio.h>
```

```
int vctr[ ] = { 10, 20, 30, 40, 50 };
```

```
main( void )
```

```
{
```

```
    int i;
```

```
    int *pvctr;
```

```
    for( i = 0; i <= 4; i++ )
```

```
        printf( "%4d", vctr[ i ] );
```

```
    printf( "\n" );
```

```
    for( i = 0; i <= 4; i++ )
```

```
        printf( "%4d", *(vctr + i) );
```

```
    printf( "\n" );
```

pointer-8

2

3

Now, similar to previous, but array of integers instead of array of characters.

1) pointer to integer

2) simple array subscripting

3) pointer arithmetic on array-name (compute address and dereference)

--> same code as before (vctr + i): how can this work? adding one would point at the middle of an integer

--> answer: C adjust for this size of the thing being dereference of for the size of the thing to which a pointer points. So, if vctr points at an 4-byte integer, (vctr+i) becomes (vctr + i*sizeof(int))

so each increment of i adds 4 instead of 1

```

pvctr = &vctr[ 0 ];
for( i = 0; i <= 4; i++ )
    printf( "%4d", *(pvctr + i) );
printf( "\n" );

for( i = 0; i <= 4; i++ )
    printf( "%4d", pvctr[ i ] );
printf( "\n" );

for( pvctr = vctr; pvctr <= &vctr[ 4 ]; pvctr++ )
    printf( "%4d", *pvctr );
printf( "\n" );
}

```

10	20	30	40	50
10	20	30	40	50
10	20	30	40	50
10	20	30	40	50
10	20	30	40	50

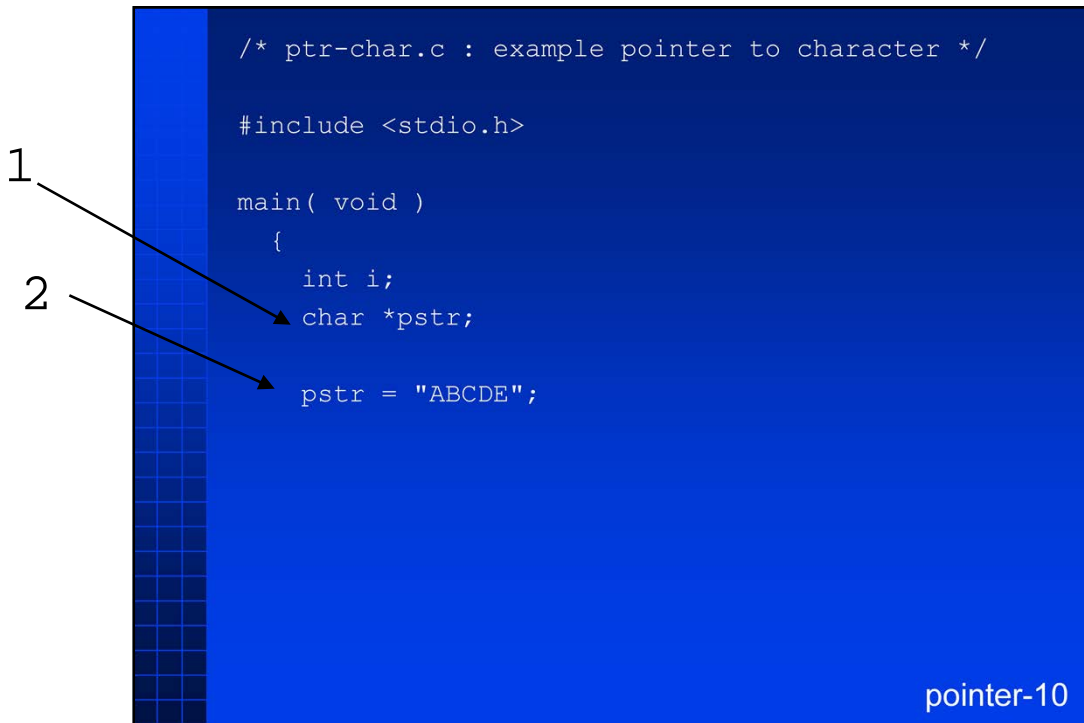
pointer-9

4) pure pointer arithmetic. `+i` adds `sizeof(*pvctr)`

5) subscripting a pointer. full definition of subscripting is

`x[i] == *(x + i*sizeof(*x))`

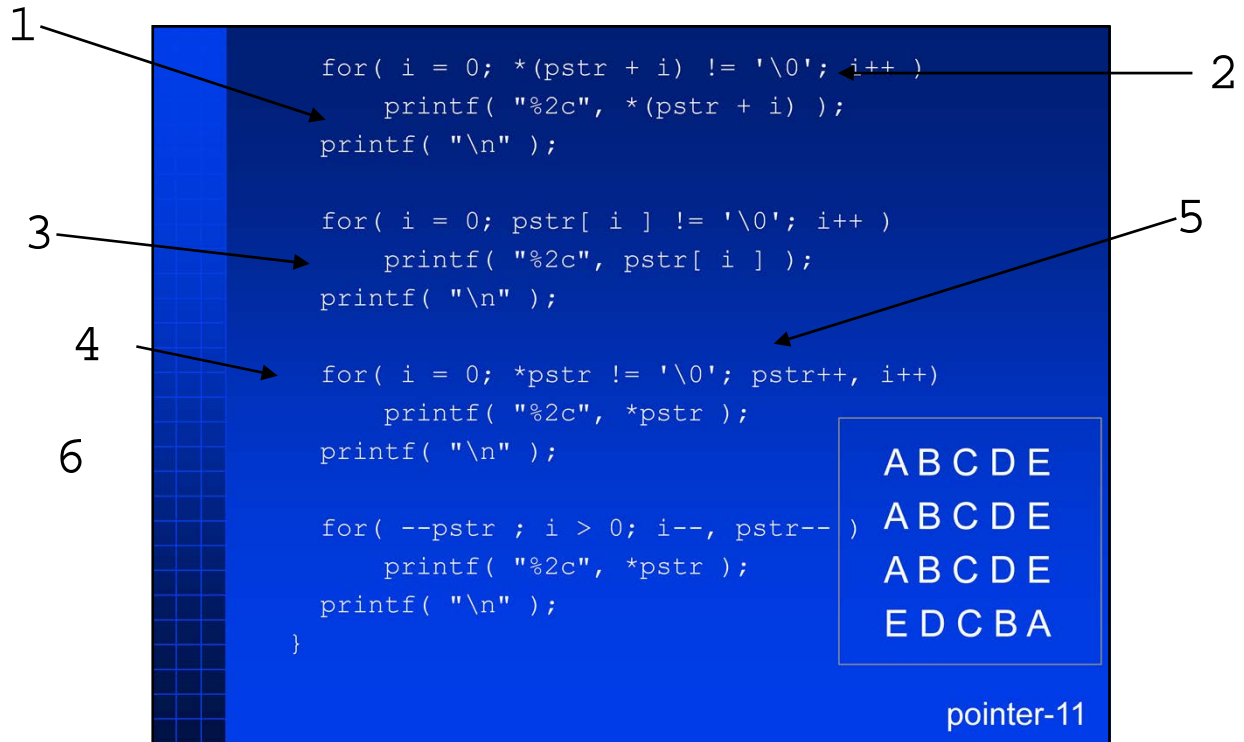
6) cursoring, moving the pointer. `++` here means “plus the size of the thing to which I point”



Now, use character pointer to manipulate traditional null-terminated strings. look at ways to get at individual characters within strings.

1) pointer to character

2) assign the address of the string literal to the pointer var. a literal string is represented as an array of char (as discussed), so assigning the string is really just assigning the address. the string literal automatically contains a null (C adds it).



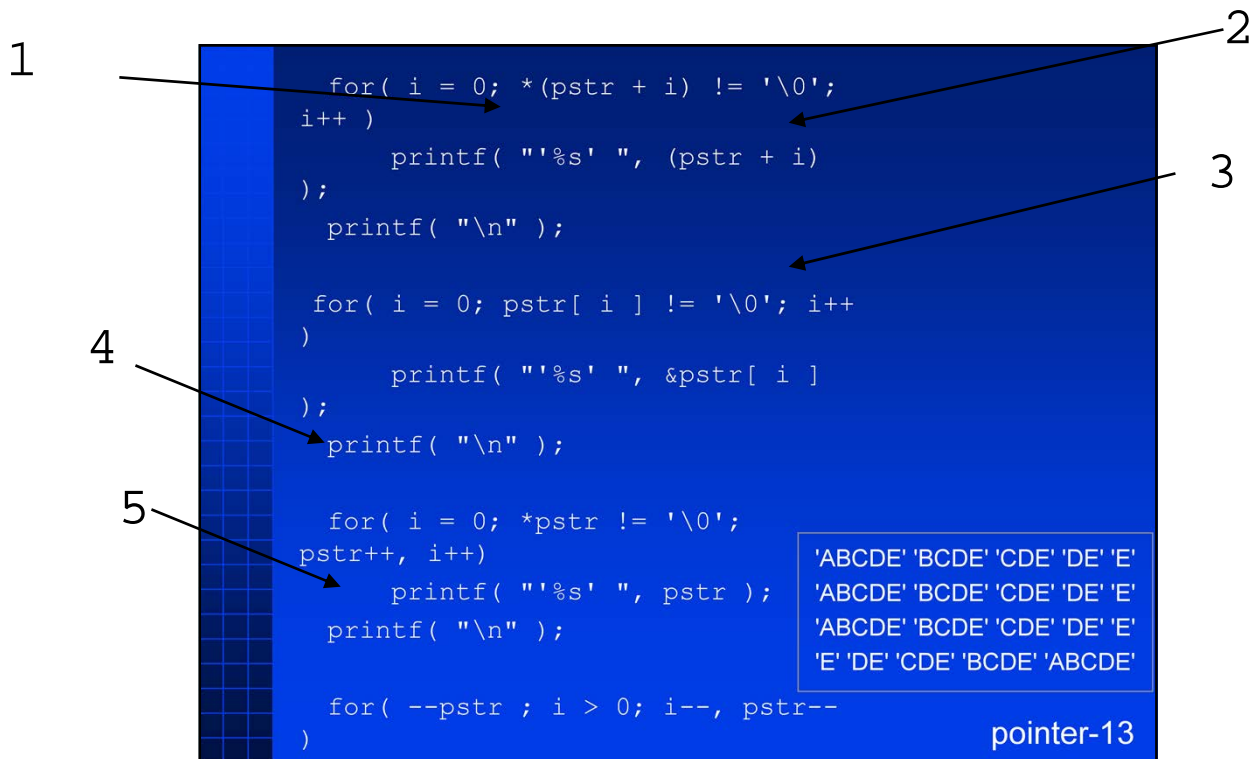
- 1) pure pointer arithmetic. compute and dereference to a single character
- 2) termination condition: while not at a nullchar
- 3) subscripting a pointer variable
- 4) cursor (moving the pointer) forwards. termination same (not null), but simpler expression, since pstr itself is moving
- 5) Note comma op. want to advance pointer and counter together. counter doesn't actually do anything for us in this loop (used in next one)
- 6) cursor backwards. note clever predecrement to move from nullchar (where previous loop ended). use i here for termination test. Note comma operator

```
/* ptr-str.c : example pointer to string */  
  
#include <stdio.h>  
  
main( void )  
{  
    int i;  
    char *pstr;  
  
    pstr = "ABCDE";
```

pointer-12

similar ideas. instead of dealing with individual chars, manipulate strings.

printf control strings will have %s instead of %c



loop structure: display substrings of varying lengths. terminate at nullchar

1) print strings instead of a char

2) no dereference: %s wants the address of a string, (pstr+i) is an address expression that yields the address of a null-terminated string.

3) we want the address of a string, pstr[i] is a char, toss in the & to get the address of the character, suitable as string address

4) move the pointer along the string. no dereference.

5) move the pointer backwards. still no dereference.

```

/* ptr-strf.c : example pointer, string, char */
#include <stdio.h>
#include <string.h>

main( void )
{
    char *pstr1, *pstr2;

    pstr1 = "ABCDE";
    pstr2 = strchr( pstr1, 'C' );

    printf( "pstr1: %s\n", pstr1 );
    if( pstr2 != NULL ) {
        printf( "pstr2: %s\n", pstr2 );
        printf( "*pstr2: %c\n", *pstr2 );
    }
}

```

```

pstr1:ABCDE
pstr2: CDE
*pstr2: C

```

pointer-14

another example of relationship between pointers, characters and string

1) strchr is one of many functions that manipulate string. see library reference. many are standard, many more are not.

Strchr looks for occurrence of single character in a string. returns pointer to string, or null if not found

2) display string, no dereference, string address for %s

3) display character, dereference, single char for %c

String literals, variables and pointers

```
{
    char str[100];
    char *pstr1, *pstr2;

    strcpy( str, "ABCDE" );
    pstr1 = str;
    pstr2 = "ABCDE";
}
```

- `str` is a variable, storage is read/write
- `"ABCDE"` is a literal, might be read-only
- `*pstr1` can be modified, `*pstr2` should not

pointer-15

preceding examples used pointer to string literal, not typical use.

in fact, trying to change a string literal might cause an error -- analogous situation to array/string parameters discussed earlier.

more typical usage is to define storage (eg string variable), and then define a pointer to it

Note: can't change the value of a string variable name (eg can't do `"str = x"`)

Pointers and parameters

- Function parameters: value or reference
- Array and strings by reference
- Scalars (int, float) by value
- Use references to pass pointers to scalars; allows modification

pointer-16

As noted, parameters to functions are passed either as actual values, or as references to values.

Arrays (incl strings) are passed by reference, scalars by value.

This means that function can modify caller's data (if array), but cannot modify caller's scalar data.

Sometimes want opposite of either. As noted not much protection for arrays -- cannot stop dereferencing a pointer. For other case, where want to modify caller's scalar, pass a pointer to the scalar instead

Passing scalars by reference

```
main( void )
{
    int i, j;
    int *jptr;

    jptr = &j;
    initialize( &i, jptr );
}

void initialize( int *iref, int
               *jref)
{
    *iref = 42;
    *jref = -42;
}
```

pointer-17

can use either style: get reference explicitly and store, or use &

for functions that only modify a single value, use a return. for functions that need to modify more than one thing, use references -- common use is return value and error condition

Pointer vs array parameters

- Arrays passed by reference

- Previously, used:

```
char msg[50]
...
func( msg );
...
void func( char m[] )
```

- Equivalence of arrays and pointers allows

```
void func( char *m )
```

pointer-18

Previously, used array notation to declare an array parameter to a function.

But, arrays and ptrs are equivalent, so could use ptr notation just as well

Choice is personal preference, technically equivalent. Depends what the function does (eg just pass to along to other, to element-by-element traversal etc).

Summary

- Pointers are addresses of variables; use indirect reference to contents
- Array subscripting is equivalent to pointer operations
- Pointers must point to something:

```
char *msg  
is useless by itself
```

pointer-19

if a variable is a container, then pointers refer to the containers, not the contents. to get at the contents, first have to get to the container.

strong equivalence in C between array processing and pointer processing

- pointer variables are useless by themselves, they must point to something

declaring a pointer does not allocate storage to which the pointer points, you must do this, either by assigning the address of a variable, or with dynamic memory (next time)

Enumerated, Structure and Union Types

types-1

On the general topic of data structuring.

have seen builtin-types int, char float, pointers

composite type array, collection of elements of same type.

other structuring facilities, strongly influenced by Pascal and related.

underlying principle is to model data according to application. could be business-rules, hardware abstractions, whatever.

Data Types in C

- scalar
 - ordinal
 - integer
 - char
 - enumerated (enum)
 - float
 - pointer
- composite
 - array (includes string)
 - struct
 - union

types-2

basic categories

we've seen most of the scalar, enum in a minute

for composite, will look at struct and union

struct: like records, dsect, structs in other langs

unions: several structs overlaid; assembler ORG, common block, redefines, variant

Enumerated types

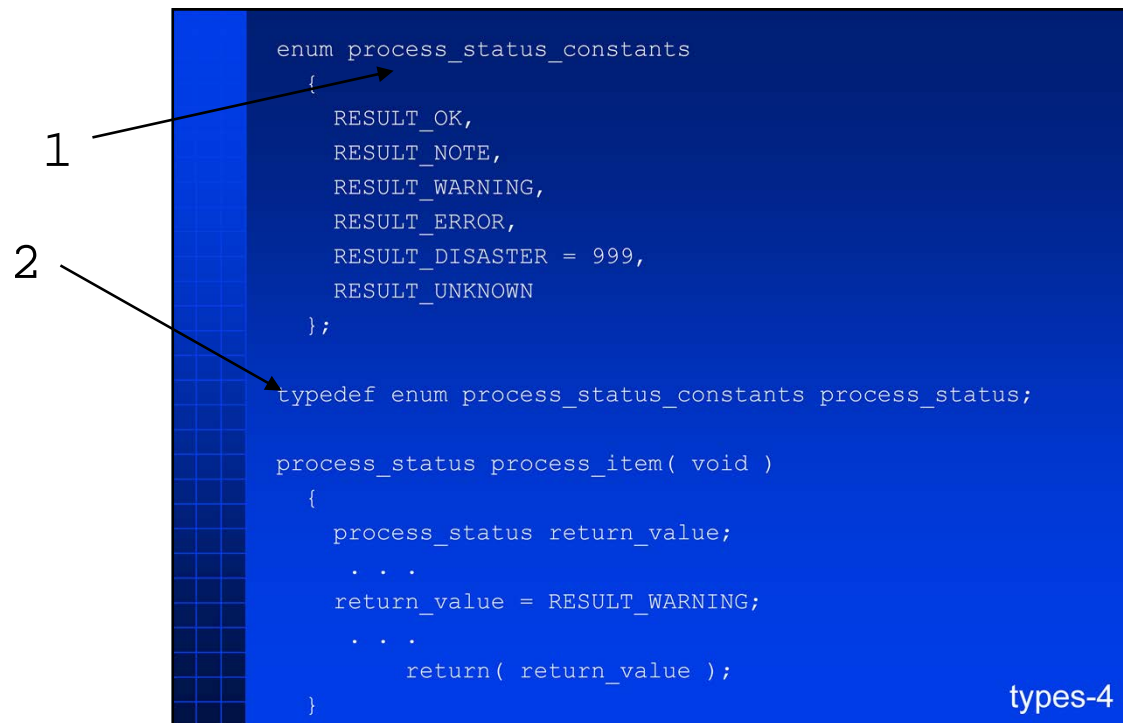
- symbolic constants (integer)
- compiler assigns values, or can be user-defined

types-3

automatic way of generating a sequence of integer constants.

eg error codes, want to reference with symbolic names, don't care what the actual numbers are

very important for defining format interfaces eg Windows has thousands



eg.

1) define a set of constants “enum process_status_constants”, given symbolic names, compiler generates integer values

if we want, can specify value

behave same as #define or const int,

2) also define a new typename that can be used in same circumstances as a built-in: var, function declarations and definitions.

Handy, too bad compiler doesn't enforce them (unlike Pascal): they're all integers. Some compilers may issue warnings, may be some other source-code management tools that can detect (eg lint)

Alternate syntax

```
typedef enum process_status_constants
{
    RESULT_OK,
    RESULT_NOTE,
    RESULT_WARNING,
    RESULT_ERROR,
    RESULT_DISASTER = 999,
    RESULT_UNKNOWN
} process_status;
```

types-5

preceding syntax requires two distinct definitions. can combine into one:

- 1) the declaration of the enumeration
- 2) the declaration of the new type

Can also omit the enum name, so have

```
typedef enum
{
    etc
} process_status;
```

Typedef

- Define alias / synonym for a type
- Examples:

```
typedef int Integer;  
typedef char * StringPtr;  
typedef unsigned long CardinalNumber;  
  
Integer i;  
StringPtr message;  
CardinalNumber size;
```

types-6

in general typedef just declares an alias for another type. eg want to rename int to integer, just say: `typedef int integer`

doesn't work for arrays:

```
typedef char[50] string;
```

won't work -- typedef is for type names, but an array specification isn't a name, its "repetition" of its basetype.

Struct

- group of data items
- usually different types
- each “piece” of the struct is called a *field*
- the dot “.” is the *field selection* operator

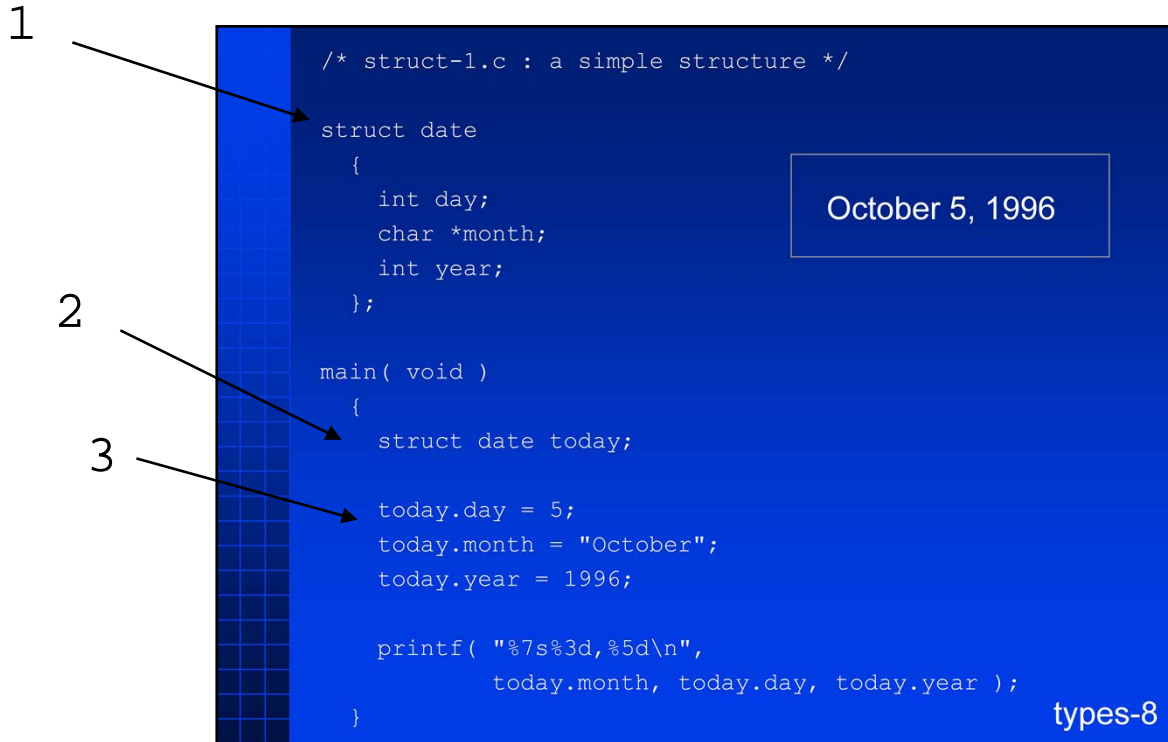
types-7

Struct is another example of a composite type.

arrays are homogeneous (all the same type); structs are heterogeneous (can be different types).

just a way of grouping things together and giving a common name

declares only the shape: storage definition occurs when struct used to define a variable or function.

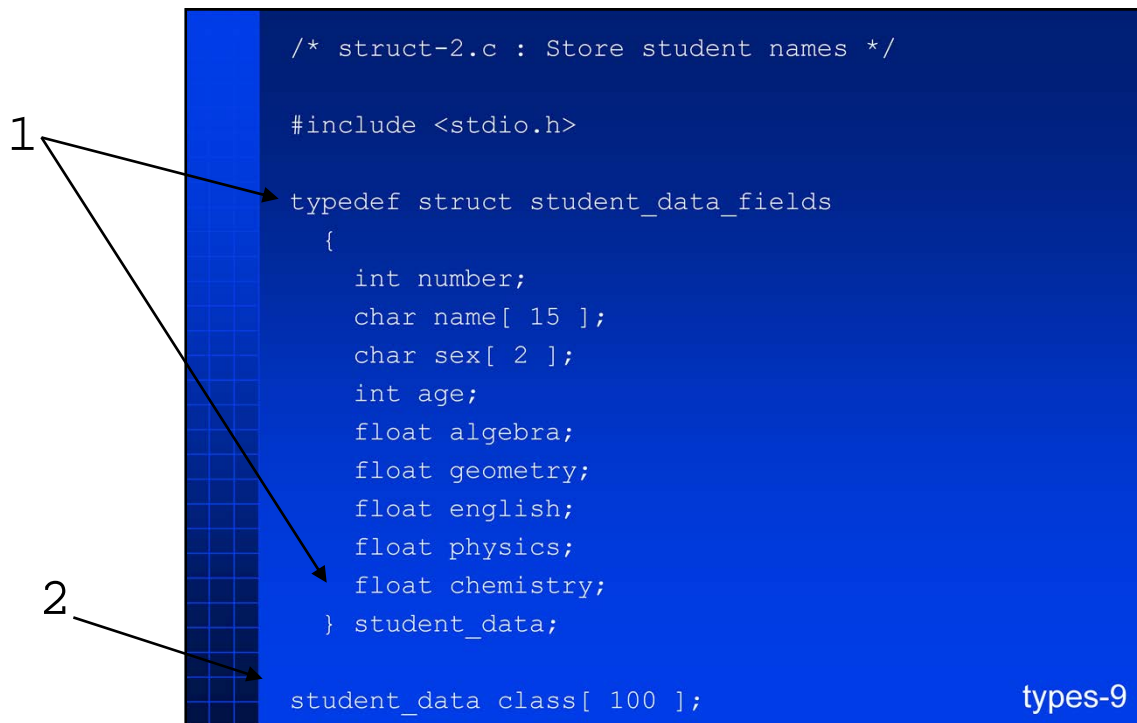


1) declaration of structure: three fields

Note style of syntax, no typedef in this case.

2) definition of variable; since no typedef, use struct-name directly.

3) references to the fields. use the . operator (field selection)



```
/* struct-2.c : Store student names */

#include <stdio.h>

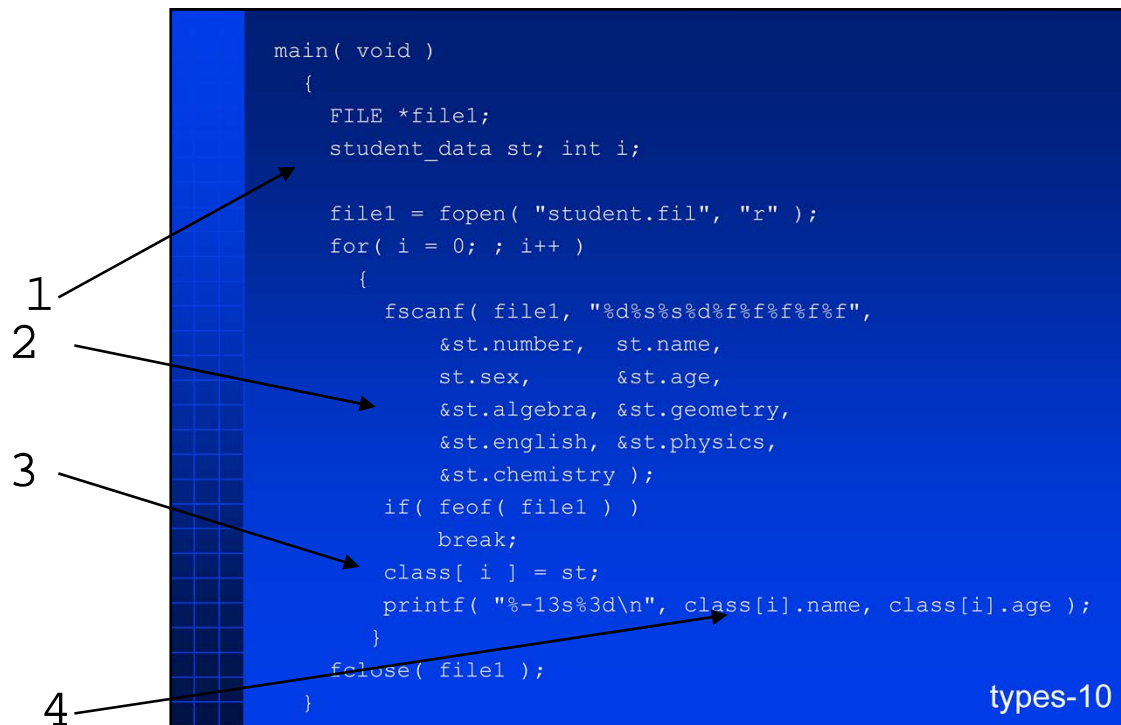
typedef struct student_data_fields
{
    int number;
    char name[ 15 ];
    char sex[ 2 ];
    int age;
    float algebra;
    float geometry;
    float english;
    float physics;
    float chemistry;
} student_data;

student_data class[ 100 ];
```

types-9

Another example

- 1) use a type definition, similar ideas to enum. define the fields (struct student_data_fields), then define a type-name (student_data)
- 2) Use the type-name to define variable class as array of structs.



- 1) definition of local variable (single struct)
- 2) field references. use & as required. dot operator is higher priority than &.
- 3) assign a struct in a single operation
- 4) array subscript, then dot. choose the element from the array. it's a struct, so choose the field from the struct.

More structs

- Can use pointers to structs:

```
student_data * me;  
...  
me = &st;  
(*me).age = 39;
```

- With typedef:

```
typedef student_data * Sdp;  
Sdp me;
```

- Structs are passed by value (like scalars)
- Can be inefficient for large structs, pass by reference instead

types-11

structs are assignable, and are passed by value

use of () in (*me).age, is necessary, depends on priority of . wrt *
who can remember, use parentheses (in fact . is higher, so the
parentheses are not optional)

also : pointers to structs

```
student_data *me;  
typedef student_data * sdp;  
sdp me;
```



```
typedef struct student_data_fields
{ /* etc */
} student_data;

typedef student_data * Sdp;

main( void )    {
    student_data st;
    ...
    initialize( &st );
}

void initialize( Sdp s_ptr )
{
    strcpy( (*s_ptr).name, "" );
    (*s_ptr).age = 0;
    ...
}
```

types-12

Union

- overlapping group of data items
- syntax is the same as a struct declaration
- size of the union is at least as large as the largest member
- dot “.” is the *member selection* operator

types-13

as stated

unions: several structs overlaid; assembler ORG, common block, redefines, variant

like struct, declares only the shape

```
/* union-1.c : example of unions */
```

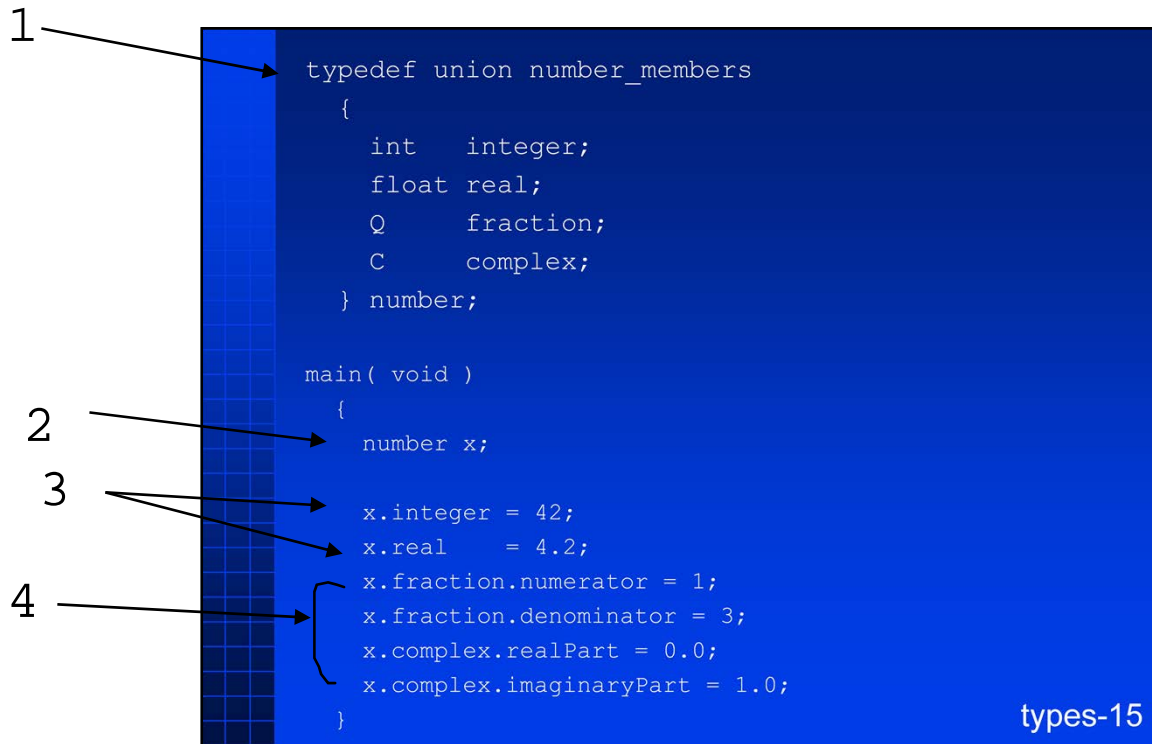
```
#include <stdio.h>
```

```
typedef struct Q_fields  
{  
    int numerator;  
    int denominator;  
} Q;
```

```
typedef struct C_fields  
{  
    float realPart;  
    float imaginaryPart;  
} C;
```

types-14

define a couple of structs, to be used in next



1) declare a union, similar syntax to structs (can also be separated into two declarations, the union and the typedef)

four members:

an int called integer

a float called real

a Q called fraction -- ie a struct Q_fields

a C called complex -- ie a struct C_fields

2) define a variable

3) select the member of the union, and assign

4) select the member, then select the field



Need to fill in some details, go over some realities

C Preprocessor

- Already seen `#define` and `#include`
- Also:
 - `#if -- #else -- #endif`
 - `#ifdef`
 - `#ifndef`
 - `#line, #error`
 - `#pragma`
- Standard symbols: `__LINE__`, `__FILE__`,
`__DATE__`, `__TIME__`

details-2

We've seen simple uses of preprocessor.

almost a compiler in a compiler: variables, functions, conditional evaluation,

control compilation in various ways

`#include`:

copy contents of file in place of `#include` line

can be nested, can get ugly-complicated

#define

- Simple textual replacement
`#define SYMBOL a sequence of text`
- Macros with argument substitution:
`#define pct(x, y) (x/y)*100`
`#define percent(x,y) ((y==0) ?`
`(x/y)*100 : 0)`
`...`
`printf("Score: %d\n", percent(23,34)`
`)`
- Also:
`#undef SYMBOL`

details-3

- have seen simple replacement: replace symbol with replacement text; "in place", traditional to use upper-case for constants

- can also parameterize to create macros, much like function calls. if symbol followed by (with no space, macro definition

Consists of macro prototype and substitution text; identifiers in prototype also occur in sub-text.

- compiler replaces macro-name with sub-text and replaces macro args with actual values.

- macro definitions can be nested (eg could use pct in percent)

- macros vs functions? hard to say, remember macros expand. use macros to avoid overhead of function calls (especially if lots of args); macro args are typeless;

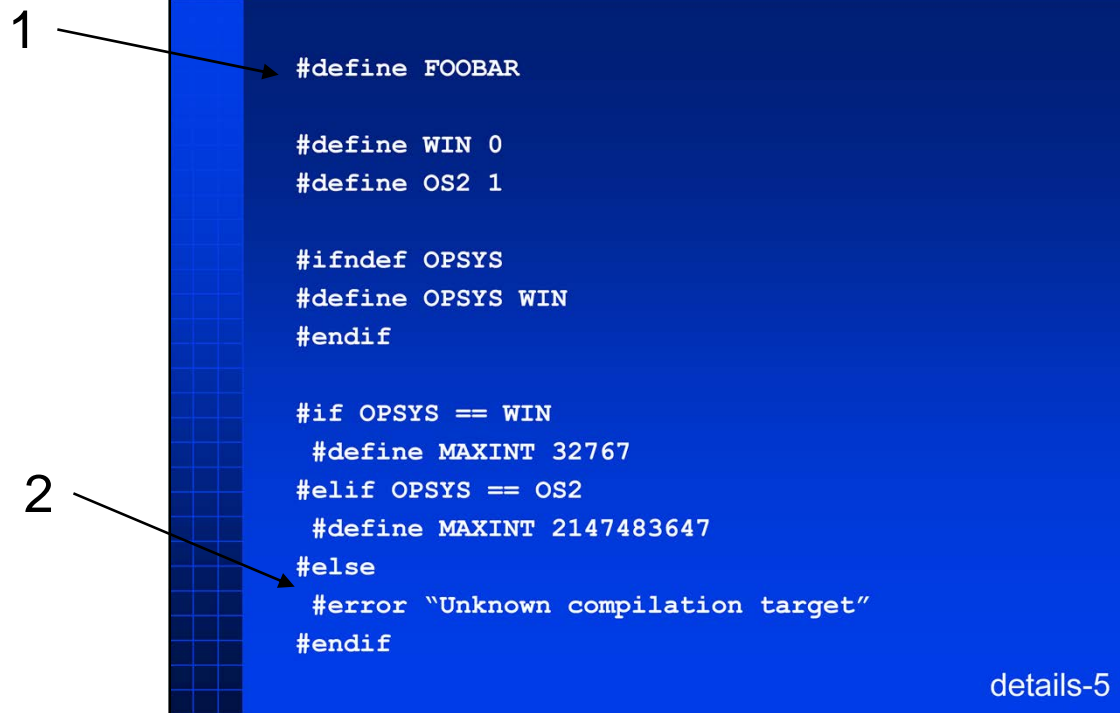
- undef: get rid of a symbol. OK if wasn't there

#if, etc.

- “Conditional compilation” -- controls which statements are compiled
- Useful for multi-target, multi-host programs
- `#if` -- `#else` -- `#endif` as usual, `#elif` for “else if”
- `#ifdef` and `#ifndef` test existence of preprocessor symbols, also `#if defined (symbol)`
- Examples:

details-4

- preprocessor decides which code is to be compiled
- mostly used is “big” programs that have multiple target systems or multiple hosts



typical sequence of stuff

- 1) defined with no value, useful with ifdef/ifndef only.
- 2) #error: generate an error message. useful for complex macro & preprocessor stuff

Other preprocessor items

- Predefined preprocessor symbols:

```
printf( "This is line %d of %s compiled at %s  
on %s\n",  
        __LINE__,  
        __FILE__,  
        __TIME__,  
        __DATE__ );
```

- `#line` resets line-number and filename:
 `#line 999 "foobar.c"`
- `#pragma` controls compiler

details-6

symbols useful for generating timestamps, etc in object code

`#line` directive for c code that is generated from elsewhere -- want to give a reference back to original source (eg application generators)

pragmas: tell compiler to do something. like options, but while program is compiling. Eg control code generation, error-message levels

[switch to development environment]

[look at `stdio.h`; ...`\watcom\h`

(reference `wc users-guide` search for `pragma`)

Module Organization

- Functions contained in modules (compilation units)
- Use extern declarations as function prototypes
- Provides inter-module type-checking

details-7

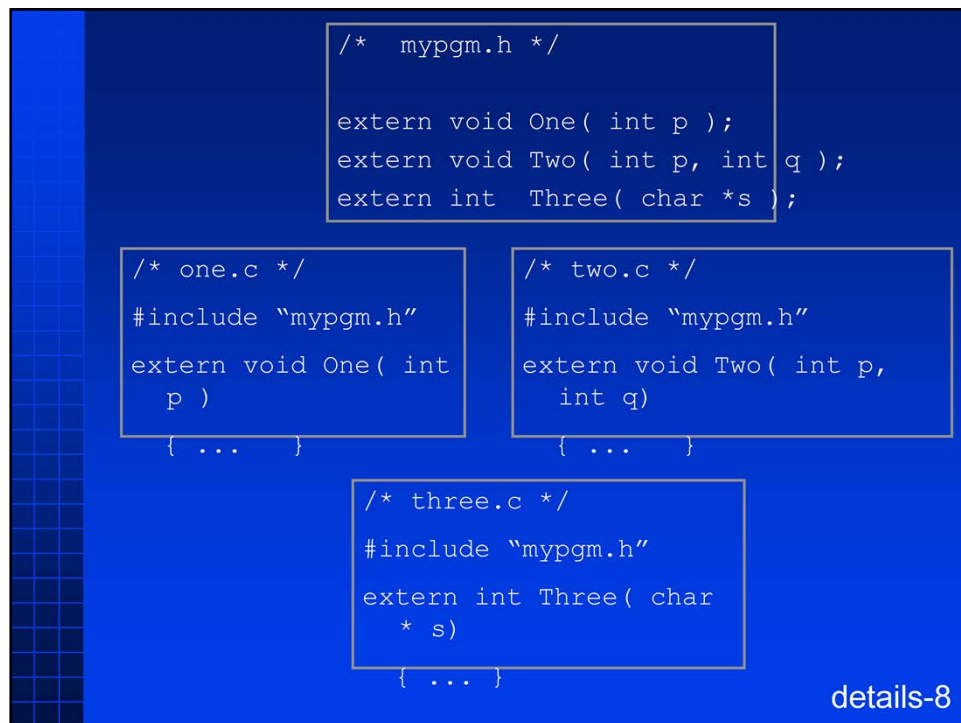
have seen how programs are arranged into functions

need a declaration in order to use a function that is defined elsewhere

use extern declaration to make function known. strictly, this is called a function prototype.

common way to organize: gather together all external functions into a private header file, then include that header file in every module.

provides cross-check of function and argument types.



include same header-file everywhere

same technique for data, externs in the header file, create separate C file that has nothing but data definitions

same include-file can contain data-structure, constants, other project-wide entities.

In fact, standard files are nothing more that this, a bit formalized

Arguments to `main()`

- System-dependent interpretation
- Usually “commandline” arguments
- Definition:

```
main( int argc, char *argv[] )
```
- `Argc` is number of arguments
- `Argv` is array of pointers to string, one string per argument

details-9

very unix -- whitespace discarded

some systems may provide access as a single simple string

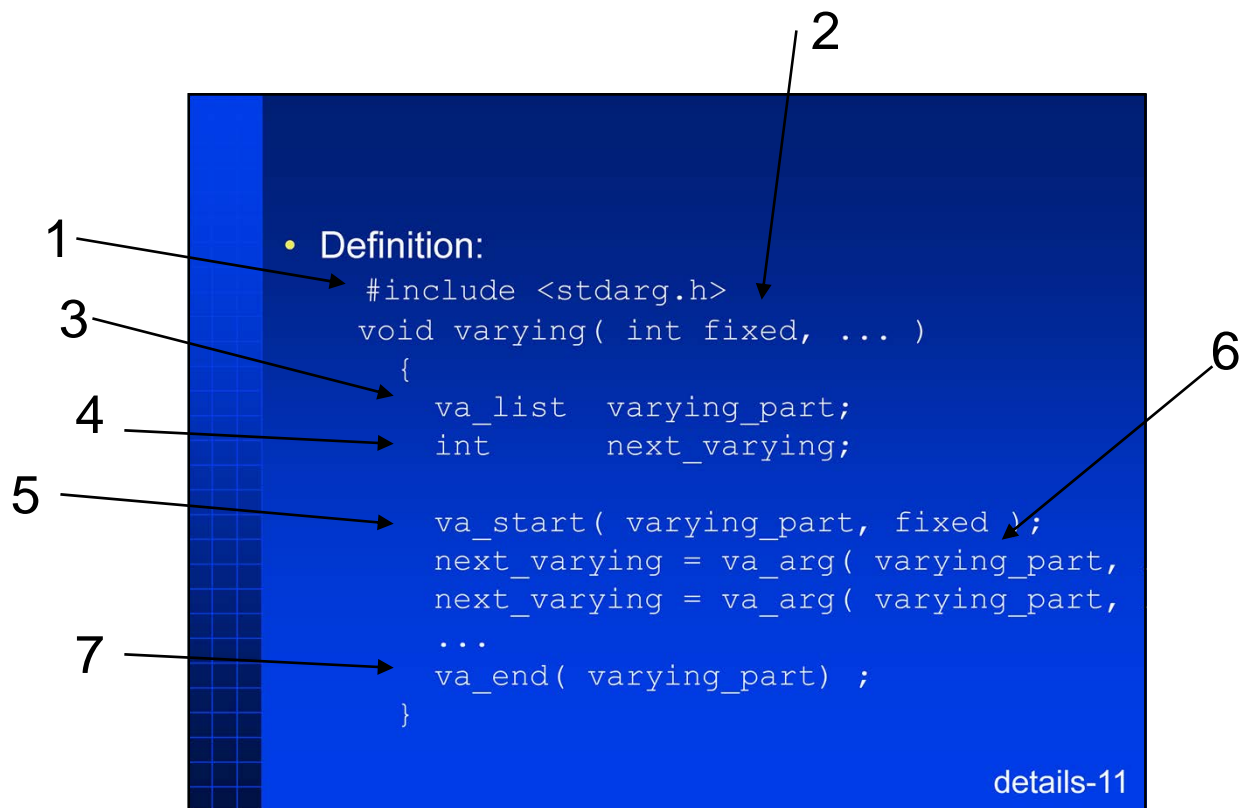
Varying argument-lists

- Functions can receive varying number arguments
- Printf, etc.
- Can also be user-defined
- Declaration:

```
void varying( int fixed, ... );
```

details-10

printf etc have variable # args, can be implemented in C
special syntax and features designed explicitly for this purpose (nice rule
about languages, must be able to be implemented in themselves)
... is part of syntax



- 1) required header file (stuff is implemented as macros)
- 2) ... is part of syntax
- 3) va_list keeps track of things
- 4) we have to know what types to expect
- 5) va_start initializes things, need to know name of last non-varying arg (must be at least one)
- 6) va_arg gets an arg of indicated type
- 7) va_end end varying processing, deallocates any storage

Function pointers

- Functions names are addresses (like arrays)
- Can invoke a function indirectly, given its address
- Syntax:

```
int (*funcptr)( int p );
```

```
typedef int (*function_ptr)(  
int p );  
function_ptr funcptr;  
function_ptr jump_vector[10];
```

details-12

A function name is its address (like arrays)

Assigning a function-name to something yields its address.

how to declare/define a variable that will contain a function address?

1)

funcptr is a pointer (because of the *)

that takes an integer parameter (because of the arglist)

and returns an int (return type)

2) using a typedef to define a function-pointer type; define a variable and even an array

(*jump_vector[5])(12)

Each different kind of function (return types args) requires its own kind of function-pointer declaration

this is a really ugly bit of syntax, since you have to read it from the middle outwards

Example

```
/* funcptr.c */
#include <stdio.h>
void foo( int p );
void bar( int q );

void (*funcptr)( int
p);

void main()
{
    funcptr = foo;
    (*funcptr)( 12
);
    funcptr = bar;
    (*funcptr)( 12
```

```
void foo( int p )
{
    printf( "foo: %d\n", p
);
}

void bar( int p )
{
    printf( "bar: %d\n", p
);
}
```

details-13

Typecasting

- Explicit type conversion
- Not generally needed
- Example:

```
float f; int i;  
...  
f = (float) i;
```
- Does not affect `i`
- Often used with pointers

details-14

In most cases, conversions are automatic.. Can be useful as a documentation feature to remind that conversions will occur.

If function prototypes are omitted, function parameter types will be unknown, so automatic conversion not possible. can use typecasting, although providing a prototype is a better solution.

for scalars, conversion changes representation; eg 2 byte int converted to 4byte float

Typecasting pointers

- Malloc returns `void *` (“pointer to nothing”); generic pointer type
- Strictly, `void *` can be assigned to any pointer, use typecasting for documentation:

```
typedef mystruct * msp;  
msp p;  
...  
p = (msp) malloc( sizeof( mystruct  
    ) );
```

- Typecast also used to change pointer dereference type

details-15

eg. if p is a pointer to some struct, but want to look at storage with a different shape, use a cast

- **Example:**

```
typedef struct a_fields
{
    int f;
} a;

typedef struct b_fields
{
    char f_h;
    char f_l;
} b;

a * ap;
int high;

ap = malloc( sizeof( a ) );
*ap = 0x1234;
high = (*(b *) (ap)).f_h;    /* or ((b *) (ap))->f_h;
*/
```

details-16

allocate a structure containing an int, use another structure to overlay the int and get at sub-components.

expression: `*(b *) (ap).f_h`

sp is a ptr; cast it to a ptr to b; dereference it; select field f_h

the `()` around ap are not needed in this case

Dynamic Variables

dynamic-1

Memory Management is library based not language based

Problem

- storage requirements are not known at compile-time
- fixed-size variables are not suitable
- use *dynamic memory* techniques to create (at execution time) exactly the storage needed
- data-structuring methodologies: linked-lists, trees; dynamic arrays, strings

dynamic-2

- example: storing varying amounts of data provided interactively
- arrays either waste space, or occasionally too small
- pre-defined arrays are not suitable in general
- many data-structuring techniques, add-on products

Storage classes

extern, static

created at beginning of program execution

local

created at function activation; destroyed
at function exit

dynamic

created by program control

dynamic-3

extern -- global scope

static -- module/file scope

local -- function scope

dynamic -- known only if provided with address (not exactly scope in the traditional sense)

Dynamic variables

- created by `malloc()`
- referred to using pointers
- deallocated by `free()`

dynamic-4

ANSI standard functions:

`malloc` - get a piece of storage from operating environment

`free` - give storage back

get piece of storage, then use that storage as a variable.

`Malloc` returns a pointer to the storage, so all references to dynamically-allocated storage are indirect (via a pointer)

Example: dynamic array

```
/* dyn-1.c:  allocate an array dynamically */

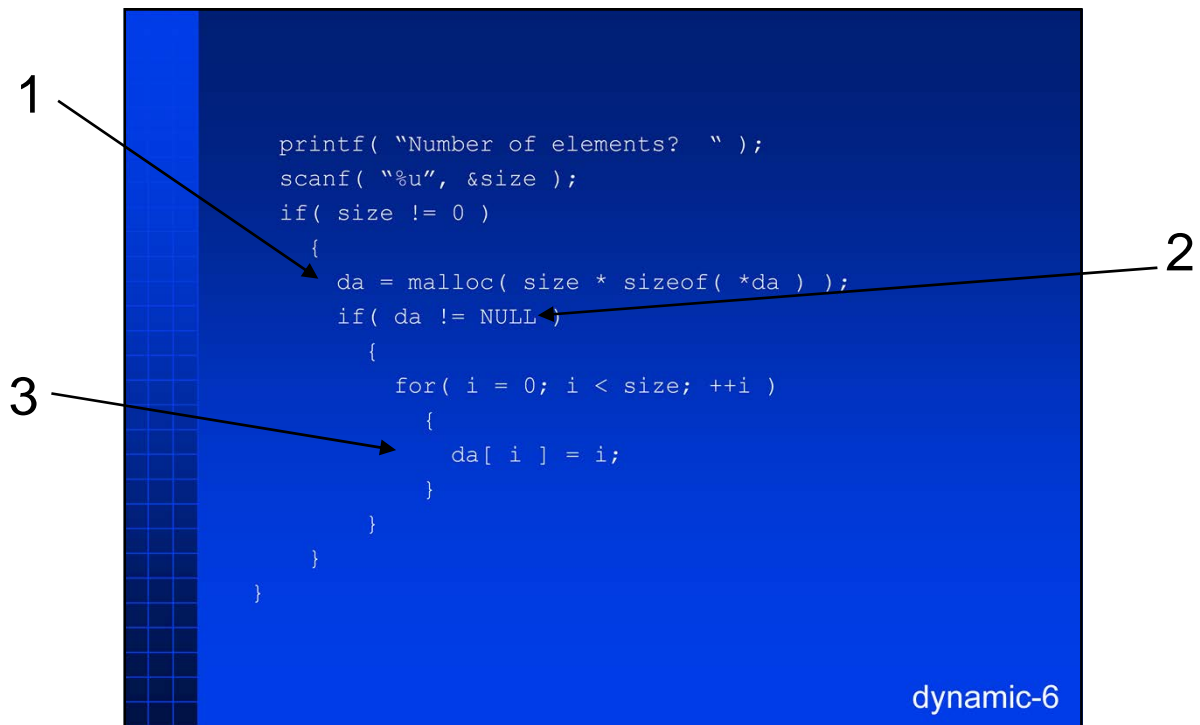
#include <stdio.h>
#include <stdlib.h>

main( void )
{
    int *da;
    unsigned int size;
    unsigned int i;
```

dynamic-5

Example, want an array of integers, but don't know how big until program is running.

some definitions first; remember that array name is same as pointer to element of the basetype of the array.



1) get storage, return pointer and store. want "size" elements, each of which is sizeof(*da) (ie the size of the entity to which da points ie int). could also use "int" here, but prefer *da (more descriptive)

total storage is size*element

note, not sizeof(da) which is the size of the pointer.

2) malloc returns null if not available

3) da is pointer to int, but can also be used as array-name

NULL: sort of zero, actually "a pointer value toat doesn't point at anything"

1

Example: dynamic string

```
/* dyn-2.c : simple dynamic variables */  
  
#include <stdio.h>  
#include <stdlib.h>  
  
typedef struct dstring_fields  
{  
    char      *storage;  
    unsigned int  size;  
} dstring;
```

dynamic-7

- 1) stdlib.h -- standard file containing memory-manipulation functions
- 2) define struct and pointer types

1

```
main( void )
{
    dstring s;
    char *p;
    unsigned int size;

    printf( "String size?\n" );
    scanf( "%u\n", &size );
    if( size != 0 )
    {
        p = malloc( size );
        if( p != NULL )
        {
            s.storage = p;
            s.size = size;
            fgets( s.storage, size, stdin );
        }
    }
}
```

2

dynamic-8

1) get storage, return pointer and store

2) malloc returns null if not available

NULL: sort of zero, actually "a pointer value that doesn't point at anything"

[chalkboard walkthrough]

note use of intermediate variables p and size not necessary, could use fields in s directly.

Example: list/stack

```
/* dyn-3.c : allocating dynamic variables */
```

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

```
typedef struct list_element_fields * l_e_ptr;
```

```
typedef struct list_element_fields
{
    int      data;
    l_e_ptr link;
} list_element;
```

```
l_e_ptr ListHead;
```

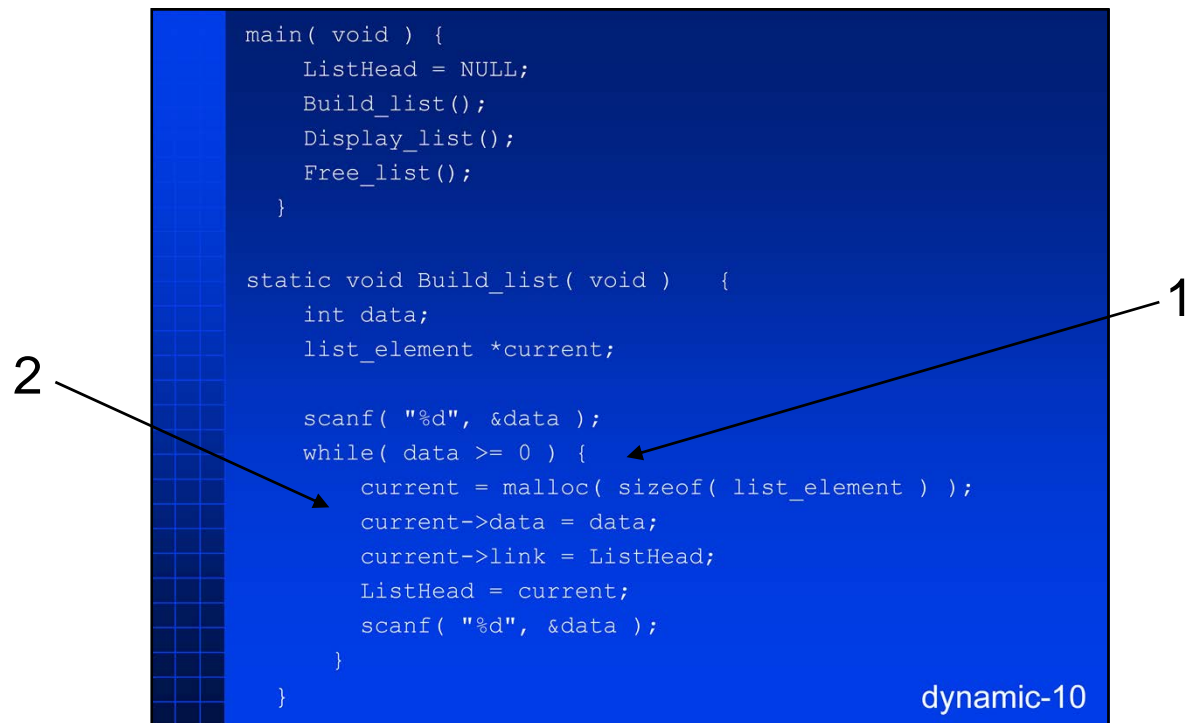
ListHead



dynamic-9

Example, build and display a singly-linked list

1) unresolved forward pointer declaration



1) sizeof == compile-time; "get storage big enough to hold a list-element")

- Note: program doesn't check return of malloc!!

2) notation : dereference, then field select == (*current).data

builds the list backwards: [do blackboard walkthrough of construction]

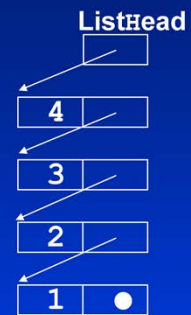
note definition of current doesn't use ptr type, uses *

```

static void Display_list( void )
{
    l_e_ptr current;

    current = ListHead;
    while( current != NULL )
    {
        printf( "%d\n", current->data );
        current = current->link;
    }
}

```



dynamic-11

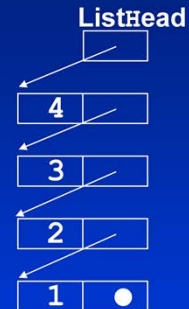
displays the list. typical list-traversal

```

static void Free_list( void )
{
    l_e_ptr current, next;

    current = ListHead;
    while( current != NULL )
    {
        next = current->link;
        free( current );
        current = next;
    }
}

```

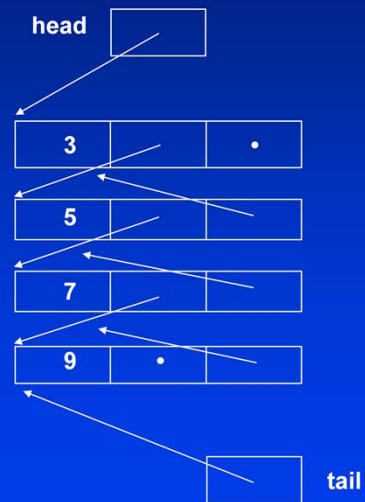


dynamic-12

free the list. typical list-traversal. once storage is freed, cannot (should not) be referenced. therefore, need two pointers, must get next value before freeing.

programs should always free what they allocate. some environments may provide a “free all” that undoes everything since the program started. Others may provide a checkpoint facility that lets pgm free everything back to checkpoint

Example: doubly-linked search list



dynamic-13

```
/* dyn-4.c : dynamic list insertion and removal */

#include <stdio.h>
#include <stdlib.h>

typedef struct list_element
{
    int data;
    struct list_element *next;
    struct list_element *previous;
} list_element;

list_element *head, *tail;
static void init_list( void );
static void insertit( int data );
static list_element *search( int data );
static void removeit( list_element *current );
```

dynamic-14

```
main( void )
{
    int data;
    list_element *current;

    init_list();
    for( ; ; )
    {
        scanf( "%d", &data );
        if( data < 0 ) break;
        insertit( data );
    }
}
```

```
for( ; ; )
{
    scanf( "%d", &data );
    if( data < 0 ) break;
    current = search( data );
    if( current != NULL )
        removeit( current );
    else
        printf( "not found\n" );
}

static void init_list( void )
{
    head = NULL;
}
```

dynamic-16

```
static void insertit( int data )
{
    list_element *current;

    current = malloc( sizeof(
list_element ) );
    if( head == NULL )
    {
        head = current;
        current->previous = NULL;
    }
    else
    {
        tail->next = current;
        current->previous = tail;
    }
    current->next = NULL;
    tail = current;
    current->data = data;
```

dynamic-17

```
static list_element *search( int data )
{
    list_element *current;

    current = head;
    while( current != NULL )
    {
        if( current->data == data )
            break;
        else
            current = current->next;
    }
    return( current );
}
```

dynamic-18

```
static void removeit( list_element *current )
{
    if( head == tail )
    {
        head = NULL;
    }
    else if( head == current )
    {
        head = current->next;
        current->next->previous = NULL;
    }
    else if( tail == current )
    {
        tail = current->previous;
        current->previous->next = NULL;
    }
    else
```

dynamic-19

```
{  
    current->previous->next = current->next;  
    current->next->previous = current->previous;  
}  
free( current );  
}
```


Dynamic variables—summary

- use when storage requirements are determined at execution time
- explicit allocation and deallocation via standard library
- use “classical” data structuring methods

dynamic-21

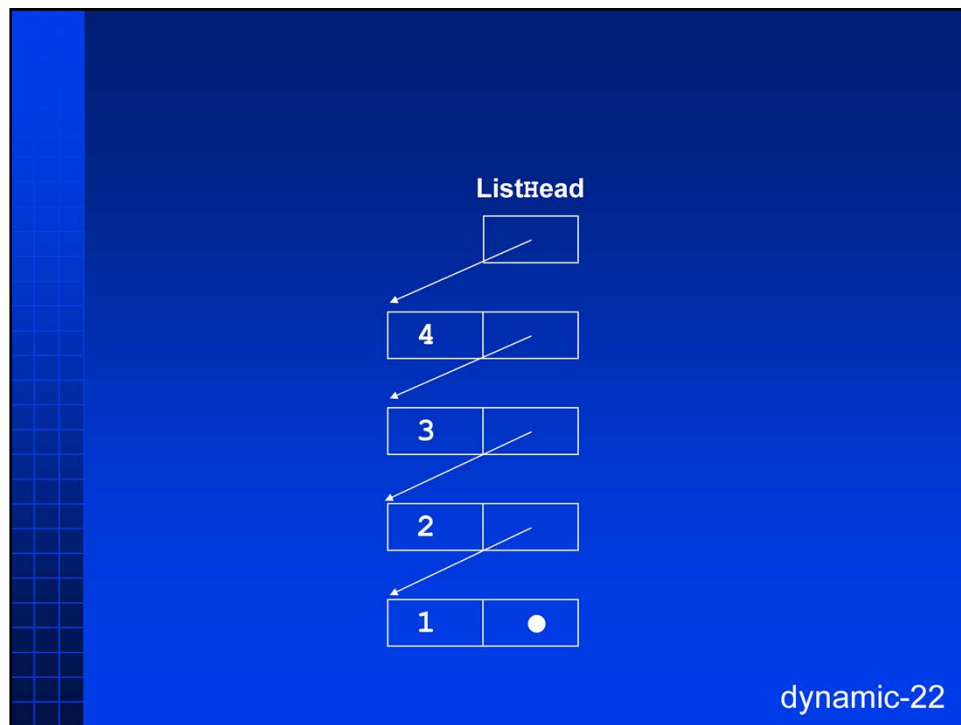
1) library not language

2) standard defines names and basic semantics; implementations are free to implement as suits [sys call or not, efficiency, garbage collect etc.]

3) almost always layered for real programs

4) there exists exception handling libraries for extraordinary situations

Code reusability becomes important for data-manipulation etc. Need well-designed libraries



input 1 2 3 4 -1

C Standard Library

library-1

We have seen some of the available library facilities.
Have a more detailed look.

Overview

- Not part of the language *per se*
- Implications for operating environment
- Groups of related functions
- Standard definitions, plus vendor-specific and common system-specific

library-2

Library is distinct from language, but still standardized. We'll take a look at the ANSI standard.

Much of the C std library is influenced by UNIX, notably I/O, program/process control etc.

Every implementation provides more than the std lib. In some cases the additional libraries are standardized by other organizations, eg posix libraries, database libraries like ODBC & embedded SQL bindings, gui standards like windows API, X, os2 PM etc.

These days, any significant software technology is rendered in C and C++.

The std defines library in terms of related function groups, like io, string management, error handling etc. We'll see these in a minute.

Pros and Cons

- Pro: saves writing code yourself
- Pro: improves portability (maybe)
- Pro: improves readability of code
- Con: long learning curve
- Con: efficiency is unknown
- Con: run-time system requirements (program packaging)

library-3

Why do we like standard libraries?

Why don't we like standard libraries?

last pt: for embedded systems (instrumentation), want lean and mean, can't stand the overhead implied by an operating system, want to run C programs on "bare machine" Such programs typically can't use the standard library, because of the interconnectivity.

learning curve. one of the reasons why we must delay this discussion for so long: library functions use full range of language features, especially pointers

Organization

- Groups of related items
- One header file (".h") per group
- Functions, variables, constants, macros, types

<assert.h> <ctype.h>

 <errno.h>

<float.h> <limits.h>

 <locale.h>

<math.h> <setjmp.h>

 <signal.h>

<stdarg.h> <stddef.h>

library-4

As noted, basic organization is groups of related items.

Each group is identified by a header file, which contains declarations, definitions and references for any or all of functions, variables, constants, datatypes, preprocessor macros etc (we'll see datatype and macros later)

here are the standard headers. some have just constants & datatypes, others have dozens of functions. about 150 functions total.

take a quick look at what's in each one. No details, not parameters: use vendor reference.

to use: include the appropriate header. defined in such a way that any interrelation is handled internally, users don't need to be aware. can include more than once (but why would you?)

Non-standard standards

- All implementations have non-standard libraries
- May define categories (e.g. Watcom C):

ANSI	POSIX 1003.1	BIOS
DOS	Intel	OS/2
PC graphics	Windows	Watcom

library-5

As noted, lots of non-standard stuff. as soon as a program used a non-standard function, portability is at a risk. If using an industry-standard function library, might be OK.

Eg: Watcom don't mention the following function categories. ANSI is what we're about to discuss. Some of these other categories are trivial, others are more complex than ANSI.

Intel -- hardware instructions and related

Watcom -- their own set of useful functions

[help library; contents; search "ANSI"]

Character handling: <ctype.h>

- manipulate/classify single characters:

```
isalnum isalpha iscntrl isdigit  
isgraph islower isprint ispunct  
isspace isupper isxdigit  
tolower toupper
```

library-6

individual character functions

is* take a char, return true/false

Mathematics: <math.h>

- Trig, exponential, integer:

acos	asin	atan	atan2	cos	sin
tan	cosh	sinh	tanh	exp	frexp
ldexp	log	log10	modf		pow
	sqrt				
ceil	fabs	floor	fmod		

library-7

all the math, nothing but the math. take and return doubles

I/O: <stdio.h>

- File I/O:

clearerr	fclose	feof	ferror
fflush	fgetc	fgetpos	fgets
fopen	fprintf	fputc	fputs
fread	freopen	fscanf	fseek
fsetpos	ftell	fwrite	getc
getchar	gets	perror	printf
putc	putchar	puts	remove
rename	rewind	scanf	setbuf
setvbuf	sprintf	sscanf	
tmpfile			
tmpnam	ungetc	vfprint	
fprintf			
vfprintf			

library-8

we've seen a good number of these

also various types (eg FILE) and constants that define defaults for buffer sizes etc

a number of these "functions" are actually macros, as we'll see next week..

these functions generally all return something. some we've already discussed. print functions return # characters processed. scan returns # items scanned.

btw scanf formats can contain ordinary characters: these are expected to match input stream

Extensions: fcloseall, fdopen, vscaf, vsscanf

General purpose utilities: <stdlib.h>

- String conversion:
atof atoi atol strtod strtol
strtoul
- Random numbers:
rand srand
- Memory management:
calloc free malloc realloc
- Program control:
abort atexit exit getenv system

library-9

The miscellaneous category: lots of subcategories
converting strings to numbers of various sizes

random numbers

memory mgmt: allocating and freeing dynamic memory. More next time.

program control: how to give up gracefully. UNix influence here.
Atexit: register a function that will be called at program exit time.
system: invoke command shell, if possible

<stdlib.h> continued

- Searching & sorting:
bsearch qsort
- Integer arithmetic:
abs div labs ldiv
- Multibyte characters:
mblen mbtowc wctomb mbstowcs
wcstombs

library-10

as noted.

I can honestly say I've never used the multibyte functions, so don't ask.

String handling: <string.h>

- String (null terminated) and memory (length):

memcpy	memmove	strcpy	strncpy
strcat	strncat	memcmp	strcmp
strcoll	strncmp	strxfrm	memchr
strchr	strcspn	strpbrk	strrchr
strcpn	strstr	strtok	memset
strerror	strlen		

library-11

String: this one is important.

deals with strings, as we have discussed them: null char at the end.

also deals with “memory” vector of bytes with no null char. user maintains length.

memmove vs copy: overlapping or not (copy not, move OK)

some string-scanning and searching functions like chr, cspn, pbrk (pattern break), tok (build tokens)

Time and date: <time.h>

- Time-of-day (“wall-clock”) and computational:

```
clock    difftime    mktime    time  
asctime  ctime      gmtime    localtime  
strftime
```

library-12

support for processor time, time-of-date, formatting times, time arithmetic

Others:

- Error diagnosis: `<errno.h>`
- Implementation limits: `<limits.h>` `<float.h>`
- Program diagnostics: `<assert.h>`
`assert`
- Localization: `<locale.h>`
`setlocale`

library-13

Other standard headers, minor ones:

`errno`: variable containing error number of most recent error, also constants

`limits`: `maxint`, `minint`, fp characteristics

`locale`: part of posix influence: used for controlling multi-byte behaviour and some of the time functions (eg timezone difference)

continued

- Non-local jumps (goto): `<setjmp.h>`
`setjmp longjmp`
- Signal handling: `<signal.h>`
`signal raise`
- Variable argument lists: `<stdarg.h>`
`va_start va_arg va_end`
- Common definitions: `<stddef.h>`

library-14

`setjmp`: C actually has a `goto` stmt that we conveniently omitted mentioning. normally it is function scope: can only `goto` within a function. `setjmp/longjmp` are a mechanism for taking the “big flying leap”

`signal`: asynchronous error handling, also includes stuff like arithmetic exceptions

`vararg`: c functions can have avariable # arguments. this stuff is implemented with a combination of library functions and language syntax. full details next week.

`stddef`: gathers together common types

Summary

summary-1

The C programming language

- general purpose language
- widespread use in industry and education
- developed at Bell Labs
- ANSI standard X3.159.1989
- reference: *The C Programming Language, Second Edition*, Kernighan and Ritchie, Prentice-Hall, 1988
- implementation and system dependencies require vendor documentation

summary-2

Data types

- integer
- float
- pointer
- array
- struct
- union
- enum
- void

summary-3

as seen

integers are king, most operators defined to work conveniently with ints

Storage classes and qualifiers

- extern (known to entire program)
- static (known to compilation unit where defined)
- register, auto (local – known in block where defined)
- dynamic (known where explicitly made available)
- const, volatile (read only, modified in unknown ways)

summary-4

1) auto -- default class for local variables, never need the keyword so omitted, register means allocate to a machine register if possible

2) Volatile: modified in a way the compiler cannot know (e.g., interrupt vector)

implications for optimizers flow control, storage alloc, etc.

```
void some_function( int p )
{
    auto char *str;  /* as usual */
    register int i; /* in a machine register
    */
    const double PI = 3.14159;
    volatile long int IOPSW;

}
```

summary-5

Control structures

- if
- while
- for
- do
- switch
- break
- continue

summary-6

also goto

Program structure facilities

- functions
- separate compilation
- preprocessor and macro language
- block structure
- scope facilities
- “make”

summary-7

functions are important

Standard C library

- **commonly used facilities**
 - string and character manipulation (with international support)
 - floating-point functions (math, conversions)
- **system facilities**
 - input/output
 - memory allocation
 - date and time, resource usage
 - exception handling
- **numerous add-on libraries**
 - access to system-dependent facilities
 - data structures (AVL trees, B-trees, ...)

summary-8

reference TOC of C library help

Primary benefits of C

- development: easier and more reliable than assembler
- efficiency: close to assembler
- convenience: high-level-language benefits
- portability: available on most systems

summary-9

easier to program

small run-time environment

nice for ASM-class apps, system programs

vendor independence on most platforms

Downside: hard to learn, harder to get good at, easy to write bad, buggy programs