3. Arrays of structures

Unions: struct where members share space to accomodate memory

Enum: allows set of int constants to be replaced by symbols (default to 0)

```
enum months { JAN = 1, ... DEC };
main() {
    enum months month;
    char *monthName[] = { "", "January", ... };
    for (month = JAN; month <= DEC; month++) printf("... %s", monthName[month]);
```

* Strings (%s) - arrays ending in NULL (char → int by char - 'a')

* Declaration & Initialization

- `char word[20] = "hello";` / `word[1] = 'h'` ... `word[5] = 'o'`
- `char str1[] = { "Hi" };`
- `char *str1 = "By gosh";`
- `char *str1; str3 = "Hello";`

NOT
`char *str1; str1 = { "Hi" };`

* <string.h> functions

1. `strcpy(first, second)` → [first ← second]
2. `strlen()`: returns string length (w/o including NULL)
3. `strcat(first, second)` → [first second]
4. `strcmp()`: compares 2 strings lexicographically (closer to a = ↓ number)

strcmp() returns	
a == b	strcmp(a,b) == 0
a != b	strcmp(a,b) != 0
a < b	strcmp(a,b) < 0
a > b	strcmp(a,b) > 0
etc.	etc.

- Arrays of strings: 2-dimensional → `char strings[# of strings][max length];`

* Advanced File I/O

- Streams: sequences of 0s and 1s · Lines: 254 chars, including end-of-line
  - 0 Input
  - 1 Output
  - 2 Error
- Adv. printf(): `"%[flag][field width][.precision]specifier"`

Flags	
-	left
+	sign
0	000
#	0 if octal

Specifiers	
d	int
i	signed decimal int
u	unsigned decimal int
f	float
lf	double
c	char
s	string
o	unsigned octal int
p	pointer

Field width: · Min # of chars printed · Right-justify

Precision	
int	# of digits 0 if >
float	# of digits after .
string	# of bytes to be printed

- Adv. scanf(): `scanf("format-control-string", other arguments);`
  - defines input format
  - stops after whitespace
  - & used, except for strings

- Escape sequences for printf()

\n — newline	\' = '
\t — tab	\\ = \
\" — "	\0 = NULL
%% — %	

- String functions:
  1. puts()
  2. putchar() > printf w/ new line
  3. gets(): reads char from standard input
  4. getchar(): inputs next char from standard input → int

- time(): returns time

+ Basic I/O

1. Declare ptr of type FILE → `FILE *fin, fout;` (fin: file in, fout: file out)
2. Create new file or open existing → `fopen("file_name", "action");`
   - fopen returns
     - address related to file
     - NULL — `if((fin = fopen("file.dat", "r")) == NULL) printf("File could not be opened.");`

Actions	
"r"	read
"w"	write
"a"	append
"r+"	open update
"w+"	write update
"a+"	append update

3. Read from file → `fscanf(fin, "%d", &x);` ↕ file pointer
   OR write to a file → `fprintf(fout, "%d", x);`
4. Close the file → `fclose(fin);`

- Sequential data files
  - Simple
  - Can spill over
  - Fields of diff. size
  - ⇒ Randomly-accessed file
    - Fields of same size
    - Easy update
    - Search
    - prevents corruption
    - allows other programs to use file
- feof: 0 — not end of file; 1 — end of file

`while(fscanf(fin, "%d", &x) != EOF)` for scanning files

* Pointers (top): return (...) variable values

* Declaration int * ptr;

* Initialize ALWAYS, to 0 or NULL

ptr = &x

- Operators

&	returns address	int * ptr = &y; return (ptr) = Address of y
*	returns value indirectly	int y; int * ptr = *y; return (*ptr) = Value of y

```
void cube_it(int *);
void main () {
  int n = 5;          // n -> 5
  cube_it (&n);       // n -> 25
void cube_it (int * nptr)
  * nptr = (*nptr) * (*nptr) * (*nptr);
```

- Const & pointer-passing — 4 cases

Data / Pointer

	Non-constant	Constant
Constant	const int * a;	int x = 5; const int * const ptr = &x;
Non-constant	int a; int * ptr = &a;	int x; int * const ptr = &x;

(access ← → security)

- Pointer Math: value +/- is # of memory elements (4 bits for int in 32-bit)

int * yptr = 3000;
yptr += 2; → yptr = 3000 + 2 (4 bits)

- Double Pointers

int ** dbl_ptr;
var = *(*dbl_ptr); → var = value pointed at by * dbl_ptr

- Pointers to functions

double (*foo) (double);

double square (double x)
return x*x;
foo = & square;

* Simple Data Structures

+ Arrays: same data types in sequential/contiguous memory

* Declaration: int array [SIZE];

* Initialization 1. N of automatic
2. Size specification — more members than initialized → others = 0; less members than initialized → compilation error

- Strings: 8-bit char arrays with 1 space for NULL
- VLA: May not work
- Loops: usually counter-controlled

+	−
Pointer math, Loops/calls	Static memory for while loop

- Pointers & Arrays

a[0] ⟷ *a
&a[0] ⟷ a

- n-dimensional

int c[2][2] = {{1,2},{3,4}};
int array [rows] [columns];

	c[0]	c[1]
c[0]	1	2
c[1]	3	4

- Passing to functions

```
void f1 (int [], int);
void f2 (int);
void main () {
  int a [5] = {..., ...};
  f1 (a, 5);  // pass a by ref.
  f2 (a[3]);  // pass a[3] by value
```

- Arrays of pointers: char * suit[4] = {...}

+ Structures: related but dissimilar data [Global]

* Creating Instance Variables from template

In Body	W/ struct	W/ Typedef
struct tag { <members>; } instance1, instance2[10], *inst_ptr; (array of 10 structures; pointer to data type "tag")	struct tag { <members>; }; struct tag instance1	typedef struct tag { <members>; } Var_name instance1;
Tag: Optional if only instantiating only in body	Necessary	Necessary

* Accessing Members

Individually

① Dot (member): directly access statically-allocated members [instance1.member1]

② Arrow (pointer): works on pointers [instance1ptr -> member1 ⟷ (*instance1).member1]

At once

struct tag instance1 = {<instance1.member1>, ...};
- Keep track of data types
- If less, others = 0