

C EXAM NOTES

Mark E. Donaldson

PRECEDENCE CHART

Descrip	Oper	Asso	Preced
Function expr	()	Left	Highest
Array expr	[]		
struct indirec	->		
struct memb	.		
Incr/decr	++ --	Right	
One's comp	~		
Unary not	!		
Address	&		
Dereference	*		
Cast	(type)		
Unary plus	+		
Unary minus	-		
Size in bytes	sizeof		
Multiplication	*	Left	
Division	/		
Modulus	%		
Addition	+	Left	
Subtraction	-		
Shift left	<<	Left	
Shift right	>>		
Less than	<	Left	
Less than or =	<=		
Greater than	>		
Greater than or	>=		
Equal	==	Left	
Not equal	!=		
Bitwise AND	&	Left	
Bitwise XOR	^	Left	
Bitwise OR		Left	
Logical AND	&&	left	
Logical OR		left	
Conditional	? :	Right	
Assignment	= %= += -= *= /= >>= <<= &= ^= =	Right	
Comma	,	Left	Lowest

VARIABLES – CONSTANTS

char constants are enclosed in single quotes, for example: `c = 'L';` internally this assigns the integer value 76 to `c` (ASCII decimal for L). There is a significant difference between a digit as a character ('3') and a digit as an integer (3). For ASCII the system, the code:

```
char c, d;
c = '3';
d = 3;
```

assigns `c` the value decimal 51 (the ASCII value of the character 3) and `d` the value decimal 3. **String constants** are delimited by double quotation marks. For example: `x = "This is a string constant"`

Conditional Expressions

Basic syntax is: `expr 1 ? expr 2 : expr 3;`
 Example: `x = flag ? y : y * y;` means if `flag` is true, then `x = y`. If `flag` is not true, then `x = y * y`. Example: `x = (l > j) ? j : l;` means if `l` is greater than `j`, then `x = j`; else if `l` is not greater than `j`, then `x = l`.

FLOW CONTROL – STATEMENTS

The do while Statement

```
do {
    action
} while (expression);
The if Statement
if (expression1) {
    action 1
}
else if (expression 2) {
    action 2
}
else {
    action 3
}
The for Statement
for (expr 1; expr 2; expr 3) {
    action
}
The switch Statement
switch (expression) {
case constant 1:
    statements 1;
case constant 2:
    statements 2;
    default:
    statements;
}
```

FUNCTIONS

Function Terminology

Any C function can be **invoked** or **called** by another. The **invoking** function may **pass** information to the **invoked** function, and the invoked function may **return** information to its invoker. By passing and returning information, *functions communicate with each other*. An invoking function passes information by passing **arguments**. In C, any expression can be an argument. In the invoking function, arguments are evaluated, and the values are passed to the invoked function. An invoked function has **parameters** that catch the information passed to it.

Function Definition - Every function has a **header** and a **body**. A function is **defined** by giving its header and body. A functions header consists of: 1) The data type returned, or the keyword **void** if a function does not return a value. Specifying the data type is optional, but if omitted, it defaults to `int`. 2) The functions name. 3) In parentheses, a list of parameters and their data types and names, separated by commas, or the keyword **void** if the function has no parameters. There is no limit to the number of parameters a function can have, except that the *number of parameters must be the same as the number of arguments in any invocation of the function*. **Definition example:**
`char grade(int exam1, int exam2, float ave)`

Function Declaration - A functions **declaration** is *terminated with a semicolon*. A **function prototype** is the form of writing function definitions and declarations, in which the data types are included within parentheses. **Declaration example:**
`char grade(int mid, int fin, float wt);`

Function Invocation - When a function is invoked, the arguments that are sent to it from the invoking function need not have the same names as its parameters. Data types of the invoking function are not listed in the

invocation. If the invoked function is to return a value, the invoking function will typically need to be an assignment argument. A function is invoked with zero or more arguments. If a function does not return a value, the function is invoked just by naming it. **Invocation examples:**
`letter_grade = grade(mid_term, final, wt);`
`print_stars(value);`
`echo_line();`

ARRAYS

An array groups distinct variables of the same type under a single name. A definition of an array reserves one or more cells in memory and associates a name with the cell or cells that the programmer can use to access the cells. An array can be initialized in the definition. The initial values are enclosed in braces and separated by commas. If, in a definition, fewer initial values are given than there are cells, the cells beginning with the first are initialized with the given values. After the initial values supplied are exhausted, each of the remaining cells are initialized to 0. It is an error to supply more initial values than there are cells. If we define and initialize arrays, we can omit the integers that specify the number of cells. Examples:
`int cars[12];`
`int id[4] = { 45, 2, 800, 81 };`
`int age[] = { 6, 0, 1, 7, 3 };`

Arrays and Pointers

An array's name is a **pointer constant**. Its value is set to the address of the first cell of the array when the array is defined and it cannot be changed thereafter. To access a specific element in an array, we can use the array's name together with an index. The array's name provides the address of its first cell, and the index (beginning from 0) provides the offset from this first cell. Examples: `age[0] = 6;` `age[4] = 3;` `age = &age[0].` A **pointer variable** also holds the address of some cell, but unlike a pointer constant, a pointer variable can have its value set through an assignment operator.

Examples:

```
int* ptr;
ptr = age; /* points to first cell */
ptr = &age[0]; /* point to first cell */
ptr = age[2]; /* point to third cell */
```

A **char array** uses one cell for each character in the string and a final cell to hold the null character '\0', which marks the end of the string. Three main ways to initialize:

```
char stoooge1[4];
char stoooge2[6];
char stoooge3[6] = "Larry";
stoooge1[0]='M';stoooge1[1]='0';stoooge1[2]='e';
stoooge1[3] = '\0';
scanf( "%s", stoooge2 ); /* type in Curly */
```

An array can be passed as an argument from one function to another. Two parameters are required: 1) An array parameter `a` to catch the array passed; 2) a parameter index `n` to catch the index of the last item in the array to be summed. Example:

```
int sum( int[a], int n ) /* function definition */
int sum[12];
x = sum( b, m );
```

String Handling Functions

strcat and **strncat** are short for string concatenation: **strcat**(string1, string2);
strncmp(string1, string2, 5); The functions **strncmp** and **strncmp** are short for string compare. The function **strcmp** expects two arguments, compares them, and returns: 0 if the two are equal; a negative integer if the first string is **lexicographically** less than the second; a positive integer if the first string is **lexicographically** greater than the second:
if (**strcmp**(string1, string2) < 0)
strncpy(string1, string2, 5); The functions **strcpy** and **strncpy** are short for string copy. Each copies all or part of the second argument into the first argument:
strcpy(string1, string2);
strncpy(string1, string2, 5); The function **strlen** is short for string length:
printf("Strings is %d\n", strlen(string));

Function to Compute string's Length

```
int length( char string[] );
{
    int count;
    for ( count = 0; string[count] != '\0'; ++count )
        ;
    return count;
}
```

Multidimensional Arrays

Arrays of more than one dimension are **multidimensional arrays**. An arrays definition shows how many dimensions it has. An array with one pair of brackets [] is one dimensional, and an array with two pairs of brackets [] [] is two dimensional. The total number of cells allocated is determined by multiplying the number within the brackets: float tolerance[10] [50] has 500 cells. The following example allocates storage for 100 rows of four columns each:

```
/* rows-jobs*/; columns-job attributes */
int job_table[100][4];
```

In functions, a *multidimensional array can be passed like a single dimensional array by giving its name as an argument*. However, in the **declaration**, we must specify the number of cells in all dimensions beyond the first. Examples:

```
void print_table( int jobs[], [4] )
print_table( job_table );
```

POINTERS

A variable that holds an address is called a **pointer variable**, or a **pointer**. Example:

```
int l = 10;
int *ptr;
ptr = &l; or int* ptr = &l;
```

We can access the contents of the cell whose address is stored in ptr by writing *ptr; *ptr = 10; Using the *ptr is called **dereferencing** the pointer. A pointer can point to an array as well:
float on_time_rate[100];
float* ptr;
ptr = on_time_rate;

Pointer Arithmetic

Using **array syntax**, an array is accessed using the array's name along with an index to access each element. With **pointer syntax**, cells in an array are accessed using a pointer. Example:
char letters[5];
char* ptr;
int count;

```
/* initialize ptr to address of letter */
ptr = letters;
/* read five characters into letters */
for (count = 0; count < 5; ++count){
    *ptr = getchar();
    ++ptr;
}
```

Array of Pointers

In general, a function that receives an array can calculate the address of any cell in the array because the parameter declaration of the array includes the address of the first cell and the cell size can be deduced from the array type. Example:
/* declare function with pointer array args */
void print_names(char* [p], int n);
char* cs[4]; /* define array of pointers */
print_names(cs, 4); /* pass pointer array */
void print_names(char* [p], int n)

malloc() and **calloc()** are used to allocate storage at **run time** instead of **compile time**:
ptr = **malloc**(sizeof(int)); The value returned by **malloc** is copied into a variable defined to point to the type of storage allocated. **free()** is used to release run-time storage. It expects one argument, a pointer to storage allocated by **malloc** or **calloc**:
free(ptr);

INPUT/OUTPUT

Files

FILE* fp;
FILE *fin, *fout;
fin = fopen("infile.dat, "r");
FILE* is a data type like **char** or **int**. The functions **fopen** and **fclose** are used to open and close files. **fopen** expects two arguments: the name of the file given a char string, and the mode (binary add b):

Mode	If File Exists	Not Exist
"r"	open reading	Error
"w"	open new	Creates new
"a"	open append	Creates new
"r+"	open read/write	Error
"w+"	open new r/w	Creates new
"a+"	open appen r/w	Creates new
"wb"	open binary w	Creates new
"r+b"	open binary r/w	Error

Characters

```
while ( ( c = getchar ( ) ) != EOF )
    putchar ( c ); or
*ptr = getchar;
```

Reading Characters	Examples
	FILE* fp;
	char c;
fgetc	c = fgetc(fp);
getc	c = getc(fp);
getchar	c = getchar();
Writing Characters	Examples
	FILE* fp;
	char c;
fputc	fputc(c, fp);
putc	putc(c, fp);
putchar	putchar(c);

Strings

```
char ans[2]; gets( ans ); puts ( ans);
```

Function	Src/dest	Newline	Returns
fgets	Any file	Reads	s& /NUL
gets	stdin	!read	s& /NUL
fputs	Any file	!append	!-/EOF
puts	stdout	appends	!-/EOF

Formatted Input/Output

```
scanf( "%C", &charvariable); & for %d%f too.
scanf( "%S", stringvariable);
```

Function	Input Source
scanf(fmat_str, ptr...)	stdin
fscanf(file_ptr, fmat_str,ptr)	A file
sscanf(array, fmat_str, ptr)	Array of char
Function	Output Dest
printf(fmat_str, arg1....)	Stdout
fprintf(file_ptr, fmat_str,arg)	A file
sprintf(array, fmat_str, arg)	Array of char

Example	Meaning
printf("%d", var)	Decimal output
printf("%O", var)	Octal output
printf("%X", var)	Hexadecimal output
printf("%c", var)	Character output
printf("%ld", var)	Long decimal output
printf("%f", var)	Float output
printf("%u", var)	Unsigned output
printf("%7c", var)	Field width 7
printf("%-7c", var)	Left justify width 7
printf("%12c", var)	Field width 12
printf("%7.3f", var)	3 decimal places
printf("%-7.3f", var)	Left justify

To convert lower case to upper, and upper case to lower:

```
toupper( string );
tolower( string );
```

STRUCTURES

A **structure** aggregates variables of different data types in order to represent an object such as an element. A structure must be **declared, defined, and initialized**.

Declaration: The declaration creates the data type and tells the system how much storage needs to be allocated whenever an instance of the structure is defined. A **structure declaration** begins with the keyword **struct** and is followed by a **tag**, or user supplied name:

```
struct element{
    char name[10];
    char symbol[5];
    float aWgt;
    float mass;
}; e1, e2, e3, e4, es[30];
```

Initialization:

To access a structure member, write the structure variable name, a period (**member operator**), and the member name:

```
e1.mass = 3.0; or for an array member,
es[6].mass += 2.0; Or a structure variable can be initialized at definition time with values for members enclosed in braces:
```

```
e1 = { "hydrogen", "H", 3.0, 2.2 }; Or with stringcopy: strcpy( e1.name, "hydrogen" );
```

To initialize an array:

```
es[3] =
{ { "hydrogen", "H", 3.0, 2.2 },
{ "oxygen", "O", 4.0, 2.4 } };
Assignment operator can be used to initialize entire structure variable or variable members of same data type:
struct element {
} e1, e2
e2.mass = e1.mass; or e2 = e1;
```

Pointers to Structures:

```
typedef struct element {
} element e1, e2, e3, e4, es[3];
element ptr;
```

ptr = &e1
ptr.mass = 2.2; or (*ptr).mass = 2.2; or
ptr -> mass = 2.2, where -> is **pointer operator**. Here, e1.name, (*ptr).name, ptr -> name, *(ptr -> name), *ptr -> name are all equiv. Same if add + 1 (second byte). A **self-referential structure** is a structure that includes among its members a pointer to itself: **struct e1* ptr;**