VARIABLE DECLARATIONS:

[const] ***type identifier*** **[=** ***initial value*****];**

Ex: char children;
int nPages, cashFare;
const double pi = 3.14159265;

NAMING CONVENTIONS:

starts with a letter or an underscore (_) and contains any combination of letters, digits and underscores (_)
contains less than 32 characters (some compilers allow more, others do not) and is not be a C reserved word

CASTING:

conversions from one type to another

Cast Expression	**Meaning**
(long double)operand	long double version of operand
(double) operand	double version of operand
(float) operand	float version of operand
(long long) operand	long long version of operand
(long) operand	long version of operand
(int) operand	int version of operand
(short) operand	short version of operand
(char) operand	char version of operand

OPERATIONS:

+, -, *, /, % Arithmetic ops. /truncates on integers, % is remainder.
++i --i Add or subtract 1 from i, assign result to i, return new val
i++ i-- Remember i, inc or decrement i, return remembered value
&& || Logical ops. Right side of && and || unless necessary
& | ^ ~ Bit logical ops: and, or, xor, complement.
>> < <= >= Comparison operators (useful only on primitive types)
?: If-like expression: (x%2==0)?"even":"odd"
, computing value is last: a, = b,c,d; exec's b,c,d then a=d

INPUT AND OUTPUT:

scanf(*format*, *address*); // input
printf(*format*, *expression*); // output

format:

Specifier	**Input Text is a**	**Destination Type**
%c	character	char
%d	decimal	int, short
%f	floating-point	float
%lf	floating-point	double

Address: contains the address of the destination variable. We use the prefix & to refer to the 'address of' of a variable.

Expression: A placeholder for the source variable.

ARRAYS:

Definition: data_type <array_name> [array_size] ;
Ex: int grades [10];

Elements: identifier[index]
Ex: grades[1]

Initialization:

data_type <array_name> [array_size] = {value1, .. valuen};
Ex: int grades[5] = {23, 34, 45, 56, 67};

CHARACTER STRINGS:

A string is a char array with a special terminator element(*null* or escape sequence '\0')

Allocating memory:

char name[31]; // 30 chars plus 1 char for the terminator

Initializing memory:

const char name[31] = {'M','y',' ','n','a','m','e',' ','i','s',' ','A','r','n','o','l','d','\0'};
or
const char name[31] = "My name is Arnold";

String Functions:˘

strlen(s) return length of string; number of characters before ASCII 0
strcpy(d,s) copy string s to d and return d;
strncpy(d,s,n) copy at most n characters of s to d and terminate;
stpcpy(d,s) like strcpy, but returns pointer to ASCII 0 terminator in d
strcmp(s,t) compare strings s and t and return first difference;
strncmp(s,t,n) stop after at most n characters;
memcpy(d,s,n) copy exactly n bytes from s to d;

STRUCTURES:

```
struct Product {
    int sku;
    double price;
};
```

Allocating memory: Define an object of a structure to allocate memory for that object. Our definition takes the form
struct Tag identifier;
Tag - name of the structure; **identifier** - name of the object.
struct Student harry; // allocates memory for harry
Initialization: Initialize an object of a structure add a braces-enclosed, comma-separated list of values.
struct Student harry = { 975, {75.6f, 82.3f, 68.9f}};
Member access:
Use the dot or arrow operator to access a member of an object of a structure
Object.member // member access using dot operator
Object -> member // member access using arrow operator
The program that uses the Product structure is listed below.

```
#include <stdio.h>
struct A
{
    int x;
    double r;
}; // struct declaration
void foo(struct A* c);      // function prototype
struct A goo(struct A d);  // function prototype
int main(void)
{
    struct A a = {4, 6.67}, b;  // declare a struct of type A
    foo(&a);  // pass by address
    printf("00%d.%.3lf.111\n", a.x, a.r);
    b = goo(a);  // pass by value
    printf("00%d.%.3lf.112\n", a.x, a.r);
    printf("%d.%.3lf.113\n", b.x, b.r);
}
void foo(struct A* c)
{
    int i;
    i = c->x;
    c->x = c->r;
    c->r = c->x % i + 202.134;
}
struct A goo(struct A d)
{
    struct A e;
    d.x = d.r - 62;
    e = d;
    return e;
}
```

FUNCTIONS: A function is a pointer to some code, parameterized by formal parameters, that may be executed by providing actual parameters. Functions must be declared before they are used, but code may be provided later. A sqrt function for positive n might be declared as:

```
double sqrt(double n) {
        double guess;
        for (guess = n/2.0; abs(n-guess*guess)>0.001; guess = (n/guess+guess)/2);
        return guess;
}
```

This function has type double (s*sqrt)(double).
printf("%g\n", sqrt(7.0)); //calls sqrt; actuals are always passed by value

Functions parameters are always passed by value. Functions must return a value.
The return value need not be used. Function names with parameters returns the function pointer. Thus, an alias for sqrt may be declared:
double (*root)(double) = sqrt;
printf("%g\n", root(7.0));
Procedures or valueless functions return 'void'.
There must always be a main function that returns an int.
int main(int argc, char **argv) OR int main(int argc, char *argv[])
Program arguments may be accessed as strings through main's array argv with argc elements. First is the program name. Function declarations are never nested.

POINTERS:

Pointer: A variable that holds an address is called a pointer. To store the variable's address, define a pointer of the variable's type and assign the variable's address to that pointer.
type *identifier;
type * is the type of the pointer. identifier is the name of the pointer.
The * operator stands for 'data at address' or simply 'data at' and is called the dereferencing or indirection operator.
This operator applied to a pointer's identifier evaluates to the value in the address that that pointer holds.

Pointer types:

Type	Pointer Type	Type	Pointer Type
char	char *	float	float *
double	double *	long double	long double *
Product	Product *	short	short *
int	int *	long	long *
long long	long long *		

NULL Address:
Each pointer type has a special value called its null value.
The constant NULL is an implementation defined constant that contains this null value (typically, 0).
This constant is defined in the <stdio.h> and <stddef.h> header files.
It is good style to initialize the value of a pointer to NULL before the address is known.
Ex: **int *p = NULL;**
Addresses may be computed with the ampersand (&) operator.
An array without an index or a struct without field computes its address:
int a[10], b[20]; // two arrays
int *p = a; // p points to first int of array a
p = b; // p now points to the first int of array b
An array or pointer with an index n in square brackets returns the nth value:
int a[10]; // an array
int *p;
int i = a[0]; // i is the first element of a
i = *a; // pointer dereference
p = a; // same as p = &a[0]
p++; // same as p = p+1; same as p=&a[1]; same as p = a+1
Bounds are not checked; your responsibility not to run off. Don't assume.
An arrow (-> no spaces!) dereferences a pointer to a field:
struct { int n; double root; } s[1]; //s is pointer to struct or array of 1
s->root = sqrt)s->n = 7); //s->root same as (*s).root or s[0].root
printf("%g\n", s->root);

PROTOTYPES:

A function prototype identifies a function type.
type identifier(type [parameter], ..., type [parameter]);
A prototype ends with a semicolon and may exclude the parameter identifiers. The identifier, the return type, and the parameter types are sufficient to validate a function call.
For example, the prototype for our power() function
int power(int, int);
We insert prototypes near the head of our source file and before any function calls.

```
/* Raise an integer to the power of a integer
 * power.c
 */
#include <stdio.h>
int sum(int no1, int no2); //prototype of function sum
int main(void)
{
      int num1=2, num2=4, sum=0;
      sum = sum(num1, num2);
      printf("%d^%d = %d\n", num1, num2, sum);
      return 0;
}
int sum(int num1, int num2)
{
      return num1+num2;
}
```

PASSING ARRAYS:

To grant a function access to an array, we pass the array's address as an argument in the function call. The call takes the form
function_identifier(array_identifier, ...)
Ex:

```
int grade[] = {10,9,10,8,7,9,8,10};
      display(grade);
```

Parameters:
A function header that receives an array's address takes the form
type function_identifier(type array_identifier[], ...)
or
type function_identifier(type *array_identifier, ...)
The empty brackets following identifier in the first alternative tell the compiler that the parameter holds the address of a one-dimensional array. For example

```
void display(int g[], int n)
{
     for(i = 0; i < n; i++)
          printf("%d ", g[i]);
}
```

PASSING STRUCTURES:

Structures can be passed to a function in two ways:
1. pass by value or pass by address
void display(const struct Student s); // pass by value
void set(struct Student* st); // pass by address
void display(const struct Student st); // pass by address
Arrow Notation:
The syntax (*s).no is awkward to read. Arrow notation provides cleaner alternative. It takes the form:
address->member
The arrow operator takes a pointer to an object on its left and a member identifier on its right.

```
for (i = 0; i < st->no_grades_filled; i++)
          printf("%.1f\n", st->grade[i]);
```

I/O (#include <stdio.h>):
fopen(name, "r") opens file name for read, returns FILE *f;
fclose(f) closes file f
getchar() read 1 char from stdin
putchar(c) write 1 char, c, to stdout
fgetc(f) same as getchar(), but reads from file f
fputc(c,f) same as putchar(c), but onto file f
fgets(s,n, f) read string of n-1 chars to a s from f or til eof or \n
fputs(s,f) writes string s to f: e.g. fputs("Hello world\n", stdout);
fprintf(f,p,...) print to file f
fscanf(f,p,...) read from file f
feof(f) return true iff at end of file f