String/Character Functions

In C, a string is a ***null-terminated array of characters***. Declarations for the string functions are in **`strlng.h`** and declarations for character functions are in **`ctype.h`**. C has *no bounds-checking* on array operations, programmer prevent an array overflow.

Header Function	***Description*** *#include <ctype.h>* ***must be included before use***	***Example***
#include <ctype.h> *int isalnum(int ch);*	**`isalnum()`** function returns ***nonzero*** if its argument is letter/digit. Otherwise **`0`** is returned.	**char** ch; **if**(***isalnum***(ch)) **printf**("alphanumeric");
int isalpha(int ch);	**`isalpha()`** returns nonzero if **`ch`** is a letter of the ***alphabet***; otherwise **`0`** is returned.	**char** ch; **if** (***isalpha***(ch)) **printf**("letter")
int iscntrl(int ch);	**`iscntrl()`** function returns nonzero if **`ch`** is between **`0`** and **`0x1F`** or is equal to **`0x7F (DEL);`** otherwise **`0`** is returned.	**char** ch; **if**(***iscntrl***(ch)) **printf**(" In control ");
int isdigit(int ch);	**`isdigit()`** function returns nonzero if **`ch`** is a ***digit 0*** through **`9`**; otherwise **`0`** is returned.	**char** ch; **if**(***isdigit***(ch)) **printf**("is a digit");
int isgraph(int ch);	**`isgraph()`** returns nonzero if **`ch`** is any ***printable***, character other than a space (**`0x21`** through **`0x7E);`** otherwise **`0`** is returned.	**char** ch; **if**(***isgraph***(ch)) **printf** ("printing char");
int islower(int ch);	**`islower()`** returns nonzero if **`ch`** is a ***lowercase*** letter (a through z); otherwise **`0`** is returned.	**char** ch; **if**(***islower***(ch)) **printf**("lowercase");
int isupper(int ch);	**`isupper()`** returns nonzero if **`ch`** is a ***uppercase*** letter (a through z); otherwise **`0`** is returned.	**char** ch; **if**(***isupper***(ch)) **printf**("uppercase");
int isprint(int ch);	**`isprint()`** returns nonzero if **`ch`** is a ***printable*** character (**`0x20`** through **`0x7E`**), including a space, otherwise **`0`** is returned.	**char** ch; **if**(***isprint***(ch)) **printf**("printable");
int ispunct(int ch);	**`ispunct()`** returns nonzero if **`ch`** is a ***punctuation*** (neither alphanumeric nor a space); otherwise **`0`** is returned.	**char** ch; **if**(***ispunct***(ch)) **printf**("punctuation ");
int isspace(int ch);	**`isspace()`** returns nonzero if **`ch`** is either a *space*, *tab*, *vertical tab*, *form feed*, *carnage return*, or *newline character*, otherwise **`0`** is returned.	**char** ch; **if**(***isspace***(ch)) **printf**("White-Space ");
int isxdigit(int ch);	**`isxdigit()`** returns nonzero if **`ch`** is a ***hexadecimal*** digit (**`A–F`** or **`a–f`** or **`0–9`**); otherwise **`0`** is returned.	**char** ch; **if**(***isxdigit***(ch)) **printf**("hexadecimal");
int tolower(int ch);	**`tolower()`** returns the ***lowercase*** equivalent of **`ch`** if **`ch`** is a letter; otherwise **`ch`**, is returned unchanged.	**putchar**(***tolower***('Q'));
int toupper(int ch);	**`toupper()`** returns the ***uppercase*** equivalent of **`ch`** if **`ch`** is a letter; otherwise **`ch`**, is returned unchanged.	**putchar**(***toupper***('t'));

	#include <string.h> must be included before use below funcs.	
#include <string.h> *char *strcat(char *strl, const char *str2);*	**`strcat()`** ***concatenates*** a copy of **`str2`** to **`str1`** (ensure that **`str1`** is large enough to hold both its original contents those of **`str2`**) and terminates **`str1`** with a ***null***. The null terminator originally ending **`str1`** is overwritten by the first character of **`str2`**. **`str2`** is untouched by the operation. The **`strcat()`** function returns **`str1`**.	***gets(s1); gets(s2) ;*** ***strcat(s2. s1);***
#include <string.h> *char *strcpy(char *str1, const char *str2);*	**`strcpy()`** is used to ***copy the contents*** of **`str2`** into **`str1`**: **`str2`** must be a pointer to a null-terminated string. **`strcpy()`** returns a pointer to **`str1`**. If **`str1`** and **`str2`** overlap, the behavior of **`strcpy()`** is undefined.	**char** str[80]; **strcpy**(str, "hello");
#include <string.h> *char *strchr(const char *str, int ch);*	**`strchr()`** returns a pointer to the first ***occurrence of the low-order byte*** of **`ch`** in the string pointed to by **`str`**. If no match is found, a ***null*** pointer is returned.	**char** *p; p = **strchr**("test", ' ') **printf(p)**;
*int strcmp(const char *str1, cost char *str2);*	**`strcmp()`** function lexicographically compares two null-terminated strings and returns an integer based on the outcome, as: ***Result*** / ***Meaning*** less than 0 — str1 < str2 0 — str1 = str2 greater than 0 — str1 > str2	**if(strcmp**(s, "pass ")) { **printf** (" Invalid PW"); **return** 0;}
*size_t strlen(const char *str);*	**`strlen()`** rerums the length of the null-terminated string pointed to by **`str`**. The null is not counted. The **`size_type`** is defined in **`string.h`**.	**strcpy**(s, "hello"); **printf**("%d", **strlen**(s));
*char *strtok(char *str1, const char *str2);* ***Example: The | summer | soldier***	**`strtok()`** returns a pointer to the next token in the string pointed to by **`str1`**. The characters making up the string pointed to by **`str2`** are the ***delimiters*** that separate each token. A ***null*** pointer is returned when there are no more tokens.	**char** *p; p = **strtok**("The summer soldier", ",") **printf**(p); **do**{ p = **strtok**('\0' , ",") **if**(p) **printf**("|%s", p); } **while**(p)
*char *strstr(const char *str1, const char *str2);* ***Example: " is a test "***	**`strstr()`** returns a pointer to the first occurrence of the string pointed to by **`str2`** in the string pointed to by **`str1`** (except **`str2`**'s null terminator). It returns a null pointer for no match.	**char** *p; p = **strstr**("this is a test", "is"); **printf**(p);

Dynamic allocation

Two primary ways a C program can store information in the main memory of the computer. The first uses ***global*** and ***local*** variables—including arrays and structures. The second way information can be stored is with C's ***dynamic allocation system***. In this method, storage for information is allocated from the free memory area (called the ***heap***) as it is needed. ***Dynamic allocation*** system is in **`stdlib.h`**, here the type **`size_t`** is defined. This type is used extensively by the allocation functions and is essentially the equivalent of **`unsigned`**.

Header Function	***Description*** *#include <stdlib.h>* ***must be included before use***	***Example***
#include <stdlib.h> *void *calloc(size_t num, size_t size);*	**`calloc()`** returns a pointer to the allocated memory. Allocated memory is equal to ***num *size***. i.e, **`calloc()`** allocates sufficient memory for an array of ***num*** objects of size **`size`** and returns a pointer to the first byte of the allocated region. A null pointer is returned for not enough memory.	p = ***calloc***(100, ***sizeof***(***float***));
*void free(void *ptr)*	**`free()`** de-allocates the memory pointed to by **`ptr`**. It is called only with a pointer that was previously allocated using **`malloc()`** or **`calloc()`** etc. Invalid pointer destroy the memory management mechanism and cause a system crash.	**for**(i=0; i<100; i++) **free**(str[i]);
*void *malloc(size_t size);*	**`malloc()`** returns a pointer to the first byte of a region of memory of size **`size`** that has been allocated from the ***heap***. (Remember, the heap is a region of free memory managed by C's dynamic allocation subsystem.) A ***null pointer*** is returned if there is insufficient memory in the ***heap***. Always verify that the return value is not a null pointer before attempting to use it. ***null pointer*** will usually result in a system crash.	**if**((p = **malloc**(**sizeof**(**struct** addr)))==**NULL**) { **printf**("Allocation error - aborting.\n"); **exit**(0); }
*void *realloc(void *ptrt size_t size);*	**`realloc()`** changes the size of the allocated memory pointed to by **`ptr`** to that specified by **`size`**. **`size`** may greater or less than the original. A pointer to the memory block is returned since it may be necessary for **`realloc()`** to move the block to increase its size. Contents of the old block are copied into the new block—no information is lost. A ***null pointer*** is returned if there is not enough free memory in the heap. Verify the success of **`real1oc().`**	**char** *p; p = **malloc**(17); **if**(!p) { **printf**("Alloc error"); **exit**(1); } **strcpy**(p, "this is 16 chars'); p = **realloc**(p,18);

Mathematics functions

ANSI C defines several mathematics functions that take double arguments and return double values. Function categories:

[1] ***Trigonometric*** **[2]** ***Hyperbolic*** **[3]** ***Exponential and logarithmic*** **[4]** ***Miscellaneous***

All the math functions require that the header ***math.h*** be included in any program that uses them. This header defines a macro called ***huge_val*** for overflowing double causing ***range error***. A ***domain error*** occurs if the input value is not in the domain. All angles are specified in ***radians***.

Header Function	***Description*** *#include <math.h>* ***must be included before use***	***Example***
#include <math.h> *double sin(double arg);*	***sin()*** returns the ***sine*** of ***arg***. The ***arg*** must be in ***radians.***	**printf**("%f ", **sin**(x))
double cos(double arg);	***cos()*** returns the ***cosine*** of ***arg***. The ***arg*** must be in ***radians.***	**printf**("%f ", **cos**(x));
double tan(double arg);	***tan()*** returns the ***tangent*** of ***arg***. The ***arg*** must be in ***radians.***	**printf**("%f ", **tan**(x));
double asin(double arg);	***asin()*** returns the ***arc sine*** of ***arg***. The ***range*** -1 through 1; otherwise a ***domain error*** occur.	**printf**("%f ", **asin**(x));
double acos(double arg);	***acos()*** returns the ***arc cosine*** of ***arg***. The ***range*** -1 through 1; otherwise a ***domain error*** occur.	**printf**("%f ", **acos**(x));
double atan(double arg);	***atan()*** returns the ***arc tangent*** of ***arg***.	**printf**("%f ", **atan**(x));
double atan2(double y, double x);	***atan2()*** returns the ***arc tangent*** of ***y/x.*** Signs of args are used to determine quadrant.	*printf*("%f ", *atan2*(y, x));
double sinh(double arg);	***sinh()*** returns the ***hyperbolic sine*** of ***arg***.	**printf**("%f ", **sinh**(x));
Double cosh(double arg);	***cosh()*** returns the ***hyperbolic cosine*** of ***arg***.	**printf**("%f ", **cosh**(x));
double tanh(double arg);	***tanh()*** returns the ***hyperbolic tangent*** of ***arg***.	**printf**("%f ", **tanh**(x));
double ceil (double num);	***ceil()*** returns ***smallest integer*** (represented as a ***double***) that is not less than ***num***.	***printf***("%f", ***ceil***(9.9)); out: 10.0
double floor(double num);	***floor()*** returns the ***largest integer*** (represented as a ***double***) not greater than ***num***.	***printf***("%f", ***floor***(9.9)); out: 9.0
double pow(double base, double exp);	***pow()*** returns $base^{exp}$. A domain error may occur if $base = 0$ and $exp \leq 0$. A domain error will occur if $base < 0$ and ***exp*** is not an ***integer***. An overflow produces a range error.	***printf***("%f ", ***pow***(x, y));
double sqrt(double num);	***sqrt()*** returns $\sqrt{num}$. If $num < 0$ a domain error will occur.	**printf**("%f ", **sqrt**(4.0));
double exp(double arg);	***exp()*** returns the ***natural logarithm e*** raised to the ***arg*** power e^{arg}.	**printf**("%f ", **exp**(1.0));
double log (double num);	***log()*** returns the $\ln(num)$. A domain error for $num < 0$ and a range error for $num = 0$	**printf**("%f ", **log**(8.0));
double log 10(double num);	***log10()*** returns the $\log_{10}(num)$. domain error for $num < 0$ and range error for $num = 0$	**printf**("%f ", **log10**(8.0));
double fabs(double num);	***fabs()*** function the ***absolute*** value of ***num***.	**printf**("%f ", **fabs**(-1.0));

Time And Date Functions

The time and date functions require the header ***time.h*** for their prototypes. This header file also defines four ***types*** and two ***macros***. The ***type time_t*** is able to represent the **system time and date** as a ***long integer***. This is called the **calendar time**. The ***structure type tm*** holds date and time broken down into its elements. The ***tm structure*** is defined as shown here:

- The value of ***tm_isdst*** will be ***+ve*** if Daylight Saving is in effect, ***0*** if it is not in effect, and ***-ve*** if there is no information available. When the date and time are represented in this way, they are referred to as ***broken-down time***.
- The type ***clock_t*** is defined the same as ***time_t***. The header file also defines ***size_t***.
- The macros defined are ***NULL*** and ***CLOCKS_PER_SEC***.

```
struct tm {     int tm_sec;     /* seconds, 0-61 */
                int tm_min;     /* minutes, 0-59 */
                int tm_hour;    /* hours, 0-23 */
                int tm_mday;    /* day of the month, 1-31*/
                int tm_mon;     /* months since Jan, 0-11 */
                int tm_year;    /* years from 1900 */
                int tm_wday;    /* days since Sunday, 0-6 */
                int tm_yday;    /* days since Jan 1, 0-365 */
                int tm_isdst;   /* Daylight Saving indicator */
            };
```

Header Function	***Description*** *#include <time.h>* ***must be included before use***	***Example***
#include <time.h> *char *asctime(const struct tm *ptr);*	***asctime()*** returns a pointer to a string that contains the time and date stored in the structure pointed to by ***ptr*** after it has been converted into the following form: ***day month date hours:minutes:seconds year\n\0*** (Eg: ***Wed Jun 19 12:05:34 1999***) ***struct pointer*** passed to ***asctime()*** is generally obtained from either ***localtime()*** or ***gmtime()***. The buffer used by ***asctime()*** to hold the formatted output string is a statically allocated character array and is overwritten each time the function is called. To save the contents of the string, copy it elsewhere.	`struct tm *ptr;` `time_t lt;` `lt = time(NULL);` `ptr = localtime(&lt);` `printf(asctime(ptr));`
clock_t clock(void);	***clock()*** returns the number of system clock cycles that have occurred since the program began execution. To compute the number of seconds, divide this value by the ***CLOCKS_PER_SEC*** macro.	#include <stdio.h> #include <time.h> **int main**(**void**){**int** i; **for**(i=0; i<10000; i++) **printf**("%u", clock()); **return** 0 ;}
*char *ctime(const time_t *time);*	***ctime()*** returns a pointer to a string of the form ***day month date hours:minutes:seconds year\n\0*** given a pointer to the calendar time. The calendar time is generally obtained through a call to ***time()***. ***ctime()*** is equivalent to: ***asctime(localtime(time))*** The buffer used by ***crime()*** to hold the formatted output string is a statically allocated character array and is overwritten each time the function is called. To save the contents of the string, you need to copy it elsewhere.	`time_t lt;` `lt = time(NULL);` `printf( ctime(&lt) );`
double difftime(time_t time2, time_t time1);	***difftime()*** returns the difference, in seconds, between ***time1*** and ***time2***. i.e, ***time2 - time1***. The given program times the number of seconds that it takes for the empty for loop to go from 0 to 500000.	**int main**(**void**) { **time_t** start, end; **long unsigned int** t; start = time(**NULL**); **for**(t=0; t<500000L; t++); end = **time**(**NULL**); **printf**("Loop required %f seconds.\n", **difftime**(end, start)); **return** 0; }
*time_t time(time_t *systime);*	***time()*** returns the system's current ***calendar time***. If the system has no time-keeping mechanism, then ***-1*** is returned. ***time()*** can be called either with a null pointer or with a pointer to a variable of type ***time_t***. The argument will be assigned the calendar time due to using type ***time_t***.	**struct tm** *ptr; **time_t** lt; lt = **time**(**NULL**); ptr = **localtime**(&lt); **printf**(**asctime**(ptr))
*struct tm *localtime(const time_t *time);*	***localtime()*** returns a pointer to the broken-down form of tm structure. The time is represented in local time which is obtained through a call to the ***time()***. This structure statically allocated and is overwritten each call time the function is called. So copy it elsewhere.	**struct tm** *local; **time_t** t; t = **time**(**NULL**); local = **localtime**(&t); **printf**("Local'time: %s", **asctime**(local));
*strut tm *gmtime(const time_t *time);*	***gmtime()*** returns a pointer to the ***broken-down form*** of tm structure. The time is represented in Coordinated Universal Time (i.e., ***Greenwich Mean Time***). The time value is generally obtained through a call to ***time()***. This structure statically allocated and is overwritten each call time the function is called. So copy it elsewhere.	**struct tm** *gmt; **time_t** t; t = **time**(**NULL**); gmt = **gmtime**(&t); **printf**("GMT time: %s", **asctime**(gmt));

Miscellaneous Functions

Header Function	Description #include <stdlib.h> must be included before use	Example
#include <stdlib.h> *void abort(void);*	**abort()** causes immediate termination of a program. Whether it closes any open files is defined by the implementation, but generally it won't.	**for**(;;) **if**(**getche**()=='A') **abort**();
void exit(int status);	**exit()** causes immediate ***normal termination*** of a program. The value of status is passed to the calling process, (usually the operating system, if the environment supports). By convention, if the value of status is **0**, normal program termination is assumed. A nonzero value may be used to indicate an error. You may also use the predefined macros **EXIT_SUCCESS** and **EXIT_FAILURE** as arguments to **exit()**.	**if**(ch=='Q') **exit**(0);
int abs(int num);	**abs()** returns the absolute value of the integer ***num***.	gets(num); ***return abs(atoi***(num));
long labs(long num);	**labs()** returns the absolute value of the ***long int num***.	gets(num); ***return labs(atol***(num));
*double atof(const char *str);*	**atof()** converts the string pointed to by **str** into a **double** value. **str** must contain a valid **float** number. Otherwise **0** is returned. The number may be terminated by any character that cannot be part of a valid floating-point number. This includes whitespace characters, punctuation (other than periods), and characters other than 'E' or 'e'. Thus, **atof(100.00HELLO)** returns **100.00**.	**printf**("%f", **atof**(num));
*int atoi(const char *str);*	**atoi()** converts the string pointed to by **str** into an **int** value. **str** must contain a valid integer number. Otherwise **0** is returned . The number may be terminated by any character that cannot be part of a integer number. This includes whitespace characters, punctuation, and other characters. Thus, **atoi(123.23)** returns **123** and **0.23** ignored.	**printf**("%d", **atoi**(num));
*long atol(const char *str);*	**atol()** converts the string pointed to by **str** into an **long int** value. **str** must contain a valid long integer number. Otherwise **0** is returned . The number may be terminated by any character that cannot be part of an integer number. This includes whitespace characters, punctuation, and other characters. Thus, **atol(123.23)** returns **123** and **0.23** ignored.	**printf**("%ld", **atol**(num));
int rand(void);	**rand()** generates a sequence of ***pseudo-random*** numbers. Each time it is called, an integer between **0** and **RAND_MAX** is returned. **RAND_MAX** is defined in **STDLIB.H**. The ANSI standard stipulates that the macro **RAND_MAX** will have a value of at least **32,767**.	**printf**("%d", **rand**());
void srand(unsigned seed);	**srand()** function is used to set a starting point for the sequence generated by **rand()**, which returns pseudo random numbers. Generally **srand()** is used to allow multiple program runs to use different sequences of ***pseudo-random*** numbers. Eg: randomly initialize the **rand()** using **srand()**	**int** i, utm; **long** ltime; ltime = **time**(**NULL**); utm = (**unsigned int**) **ltime**/2; **srand**(utime); **for**(i=0; i<10; i + +) **printf**("%d ", **rand**());
*void qsort(void *base, size_t num, size_t size,* *int(*compare)(const void*, const void*));* ***Function pointed to*** by **compare** is used to *compare two elements in the array.* It must return the values: ***-ve : arg1 < arg2 0 : arg1= arg2 +ve : arg1 > arg2*** The form of **compare** must be ***int*** *function_name(**const void** *arg1, **const void** *argl2)*	**qsort()** function sorts the array pointed to by **base** using a **Quicksort** (developed by C.A.R. Hoare). The **Quicksort** is generally considered the best general-purpose sorting algorithm. Upon termination, the array will be sorted. The number of elements in the array is specified by **num** and the **size (in bytes)** of each element is described by **size**. (The **size_t** type is defined in **STDLIB.H** and is equivalent of **unsigned**.) . The array is sorted in **ascending order**, with the **lowest address** containing the **lowest element**.	**int** comp(**const void** *i, **const void** *j) int num[5]= {8, 7, 6, 2, 0}; **int main**(**void**){ **int** i; **qsort**(num, 5, **sizeof**(**int**), comp); **printf**("Sorted array: "); **for**(i=0; i<10; i++) **printf**("%d ", num[i]); **return** 0;} /* compare the integers */ **int** comp(**const void** *i, **const void** *j){ **return** *(**int** *)i – *(**int** *)j; }
*void *bsearch(const void *key, const void *base,* *size_t num, size_t size,* *int(*compare)(const void *, const void *));* ***Function pointed to by compare*** is used to compare two elements in the array with the key. It must return values: ***-ve : arg1 < arg2 0 : arg1= arg2 +ve : arg1 > arg2*** The form of **compare** must be ***int*** *function_name(**const void** *arg1, **const void** *argl2)*	**bsearch()** performs a ***binary search*** on the sorted array pointed to by **base** and returns a pointer to the first member that matches the key pointed to by **key**. The number of elements in the array is specified by **num** and the **size (in bytes)** of each element is described by **size**. (**size_t** type e defined in **STDLIB.H** and equivalent of **unsigned**). The array must be sorted in **ascending order**, with the lowest address containing the lowest element. If the array does not contain the key, then a null pointer is returned.	**char** *alph = "abcdefghijklmnopqrstuvwxyz"; **int** comp(**const void** *ch, **const void** *s); **int main**(**void**){**char** ch, *p; **do** { **printf**("Enter a character: "); **scanf**("%c%*c",&ch); ch = **tolower**(ch); p = **bsearch**(&ch, alph, 26, 1, comp) **if**(p) **printf**("is in alphabet.\n"); **else printf**("is not in alphabet.\n") } **while**(p); **return** 0;} **int** comp(**const void** *ch, **const void** *s){ **return** *(**char** *)ch – *(**char** *)s; }
	Description **#include <setjmp.h>** **must be included before use**	
void longjmp(jmp_buf envbuf, *int val);*	**longjmp()** causes program execution to resume at the point of the last call to **setjmp()**. These two functions are the way ANSI C provides for a ***jump between functions***. Notice that the header **SETJMP.H** is required. The **longjmp()** function operates by resetting the stack as described in **envbuf**, which must have been set by a **prior** call to **setjmp()**. This causes ***program execution to resume*** at the statement following the **setjmp()** invocation—the computer is 'tricked' into thinking that ***it never left the function that called*** **setjmp()**. (As a somewhat graphic explanation, the **longjmp()** function '***warps***' across ***time*** and (memory) ***space*** to a previous point in your program, without having to perform the ***normal function-return*** process.) The buffer **envbuf** is of type **jmp_buf**, which is defined in the header **SETJMP.H**. The buffer must have been set through a call to **setjmp()** prior to calling **longjmp()**. The value of **val** becomes the return value of **setjmp()** and may be interrogated to determine where the ***long jump*** came from. The only value not allowed is **0**. It is important to understand that the **longjmp()** function must be called before the function that called **setjmp()** returns. If not, the result is technically undefined. In actuality, a crash will almost certainly occur. By far the most common use of **longjmp()** is to return from ***a deeply nested set of routines*** when a ***catastrophic error*** occurs.	#include <setjmp.h> #include <stdio.h> **void** f2(**void**); **jmp_buf** ebuf; **int main**(void){ **char** first=1; **int** i; **printf**("1") ; i = **setjmp**(ebuf) ; **if**(first) {first = !first; f2(); **printf**("Not printed"); } **printf**("%d",i); **return** 0;} **void** f2(**void**){ **printf**("2"); **longjmp**(ebuf, 3); }
#include <setjmp.h> *int setjmp(jmp_buf envbuf);*	**setjmp()** saves the contents of the system stack in the buffer **envbuf** for later use by **longjmp()**. **setjmp()** returns **0** upon invocation. However, **longjmp()** passes an argument to **setjmp()** when it executes, and it is this value (always nonzero) that will appear to be the value of **setjmp()** after a call to **longjmp()**.	

C Keyword Summary

There are 32 keywords in C. All keywords are in lowercase. Following table list the keywords alphabetically. However the summery of those keywords are in GroupWise

auto	*const*	*double*	*float*	*int*	*short*	*struct*	*unsigned*
break	*continue*	*else*	*for*	*long*	*signed*	*switch*	*void*
case	*default*	*enum*	*goto*	*regiser*	*sizeof*	*typedef*	*volatile*
char	*do*	*extern*	*if*	*return*	*static*	*union*	*while*

Keyword	Control : Keyword Summery and general form	Example
if	The general form of the **if** statement is **if**(condition){ statement block 1 } **else** { statement block 2 } If single statements are used, the braces are not needed. The **else** is optional. The condition may be any expression. If that expression evaluates to any value other than **0**, then statement **block 1** will be executed; otherwise, if it exists, statement **block 2** will be executed.	ch = **getche**(); **if** (ch=='q'){**printf**("Prog. Terminated"); **exit**(0); } **else** proceed() ;
else	See the **if** section.	
for	The **for** loop allows automatic initialization and incrementation of a counter variable. The general form is: ***for(initialization; condition; increment) { statement block }*** The braces are not necessary for only one statement. Although the **for** allows a number of variations, generally the ***initialization*** is used to set a counter variable to its starting value. The ***condition*** is generally a ***relational statement*** that checks the counter variable against a termination value, and the ***increment*** increments (or decrements) the counter value. The loop repeats until the ***condition*** becomes ***false***.	The following code will print ***hello*** 10 times. **for**(t=0; t<10; t++) **printf**("Hello \n");
do	The **do** loop is one of three loop constructs in available in C. The general form: ***do{ statement block } while(condition);*** The braces are not necessary for only one statement. The **do** loop repeats as long as the ***condition*** is ***true***. The **do** loop is the only loop in C that will always have ***at least one iteration*** because the condition rs tested at the ***bottom*** of the ***loop***.	**do** {ch=**getche**();} **while**(ch!='q');
while	The **while** loop has the general form: ***while(condition){ statement block }*** The braces are not necessary for only one statement. The loop will repeat as long as the condition is true. The **while** tests its condition at the ***top*** of the ***loop***. Therefore, if the condition is false to begin with, the loop will not execute at all. The condition may be any expression. reads characters until ***end-of-file***	t = 0; **while** (!**feof**(fp)) {s[t] = **getc**(fp); t++;}
switch	The **switch** statement is C's ***multi-path branch*** statement. It is used to route execution in one of several ways. The general form : **switch**(value) { **case** constant_1:statement sequence; **break**; **case** constant_2: statement sequence; **break**; **case** constant_3: statement sequence: **break**; . . . . . . **default** : statement sequence; **break**;} Each statement-sequence may be one or many statements long. The **default** portion is optional. The expression controlling the **switch** and all case constants must be of **integral** or **character** types. The switch works by checking the value of ***int-expression*** against the ***constants***. As soon as a match is found, that set of statements is executed. If the **break** statement is ***omitted***, execution will continue into the ***next case***. cases are similar to labels. Execution will continue until a **break** statement is found or the **switch** ends.	ch = **getche**(); **switch**(ch) { **case** 'e': enter(); **break**; **case** 'l': list(); **break**; **case** 's': sort'(); **break**; **case** 'q': exit(O); **break**; **default**: **printf**("Unknon cmd\n"); **printf**("Try Again \n"); }
case	**case** is covered in conjunction with **switch**.	
default	**default** is used in the **switch** statement to signal a default block of code to be executed If no matches are found in the **switch**.	
continue	**continue** is used to bypass portions of code in a loop and forces the conditional expression to be evaluated. The call to **process()** will not occur until **ch** contains the character **s** and char entry won't stop until **s** is entered.	**while**(ch=**getche**()) { **if**(ch != 's') **continue**; process(ch); }
break	**break** is used to exit from a **do**, **for**, or **while** loop, bypassing the normal loop condition. It is also used to exit from a **switch** statement (in a **switch**, **break** effectively keeps program execution from "***falling through***" to the next case).	**while**(x<100){ x = get_new_x( ); */ * key hit on keyboard */* **if** (**kbhit**()) **break** ; process(x) ; }
goto	The **goto** causes program execution to jump to the label specified in **goto**. The general form: **goto** label; . . . . . label: All **labels** must end in a colon and must not conflict with keywords or function names. Furthermore, a **goto** can branch only within the ***current function***, and not from one function to another.	The following example will prim the message "right" but not the "wrong": **goto** lab_1; **printf**("wrong"); lab_1: **printf**("right");

Data type specifier : Keyword Summery and general form		Example
int	**int** is the type specifier used to declare integer variables. Eg: to declare **count** as an integer	**int** count;
char	**char** is a data type used to declare character variables. In C, a character is one byte long.	**char** ch;
float	**float** is a data type specifier used to declare floating-point variables. To declare **f** to be of type float:	**f loat** f;
double	**double** is a data type specifier used to declare double-precision floating-point variables. To declare **d** to be of type double	**double** d;

Data type Modifier : Keyword Summery and general form			Example
short	**short** is a data type modifier used ·to declare small integers. Eg: to declare **sh** to be a short integer		***short*** int sh;
long	**long** is a data type modifier used to declare long integer and long double variables. Eg: to declare **count** as a long int		**long int** count;
signed	The **signed** type modifier is most commonly used to specify a signed char data type.		**signed char** ch;
unsigned	The **unsigned** type modifier tells the compiler to create a variable that holds only **unsigned** (i.e., positive) values. Eg: to declare **big** to be an unsigned integer you would write		**unsigned int** big;
const	The **const** modifier tells the compiler that the ***contents of a variable cannot be changed***. It is also used to prevent a function from modifying the object pointed to by one of its arguments.	***Access Modifiers***	***const*** **int** i=10;
volatile	The **volatile** modifier tells the compiler that a variable may have its contents altered in ways not explicitly defined by the program. Variables that are changed by the hardware, such as real-time clocks, interrupts, or other inputs are examples.		***volatile*** **unsigned** u;
typedef	The **typedef** statement allows you to create a new name for an existing data type. The general form **typedef type-specifier** new-name;	***statement***	'**balance** ' in place of '**float** ': **typedef float** balance;

Structure : Keyword Summery and general form		Example
struct	The **struct** statement is used to create aggregate data types, called ***structures***, that are made up of one or more members. The general form: **struct** *struct-name* {**type** *member'*; **type** *member2* ; . . . . **type** *memberN* ; } *variable-list*; The individual members are referenced using the dot or arrow operators.	**struct** *catalog* { **char** name [40] ; /* author name */ **char** title[40] ; /* title */ **char** pub[40] ; /* publisher */ **unsigned** date; /* copyrit date */ **unsigned char** ed; /* edition */ }card;
union	The **union** keyword creates an aggregate type in which two or more variables share the same memory location. The form of the declaration and the way a member is accessed are the same as for **struct**. The general form is **union** union-name {**type** *member1* ; **type** *member2* ; . . . **type** *member N* ; } *variable-list*;	**union** item {**int** m; **float** x; **char** c;} code;
enum	The **enum** type specifier is used to create ***enumeration*** types. An enumeration is simply a ***list of named integer constants***. For example, the code declares an enumeration called **color** that consists of three constants: **red, green, and yellow**.	**enum** color {red, green, yellow}; **enum color** c; **int main**(**void**){ c = red; **if** (c==red) **printf**("is red\n"); **return** 0; }

Memory mangmnt : Keyword Summery and general form		***Example***
auto	***auto*** is used to create ***temporary variables*** that are created upon entry into a block and destroyed upon exit. The use of auto is optional since ***local variables*** are auto by default. In the example, the variable t is created only if the user strikes an ***a***. Outside the ***if*** block, ***t*** is completely unknown; and any reference to it would generate a compile-time syntax error.	**if**(**getche**()=='a'){ **auto int** t; **for**(t=0; t<'a'; t++)**printf**("%d", t); **break**;}
static	The ***static*** keyword is a data type modifier that causes the compiler to create permanent storage for the local variable that it precedes. This enables the specified variable to maintain its value between function calls. ***static*** can also be used on ***global*** variables to limit their scope to the file in which they are declared.	to declare ***last_time*** as a ***static integer***: **static int** last_time;
extern	The ***extern*** data type modifier tells the compiler that a variable is defined elsewhere in the program. This is often used in conjunction with separately compiled files that share the same ***global*** data and are linked together. In essence, it notifies the compiler of a variable without redefining it.	As an example, if first were declared in another file as an ***integer***, the following declaration would be used in subsequent files: **extern int** first;
register	The ***register*** modifier requests that a variable be stored in the way that allows the fastest possible access. In the case of characters or integers, this usually means a ***register of the cpu***.	declare ***i*** to be a ***register*** integer: **register** int i;
Miscellanious : Keyword Summery and general form		***Example***
void	The ***void*** type specifier is primarily used to declare ***void functions*** (functions that do not return values). It is also used to create ***void pointers*** (pointers to void) that are ***generic pointers*** capable of pointing to any type of object and to specify an empty parameter list.	**void** func_1( );
sizeof	The ***sizeof*** keyword is a ***compile-time operator*** that returns the length of the variable or type it precedes. If it precedes a type, the type must be enclosed in parentheses. The ***sizeof*** statement's principal use is in helping to generate portable code when that code depends on the size of the C built-in data types.	***operator*** **printf**("%d", **sizeof**(**short int**)); will print 2 for most C implementations.
return	The ***return*** statement forces a return from a function and can be used to transfer a value back to the calling routine. Keep in mind that as soon as a ***return*** is encountered, ***the function will return***, skipping any other code in the function.	***statement*** **int** mul(**int** a, **int** b){ **return** a*b;}