The standard C library

Header files and commonly used functions

Systems Programming

ANSI libraries and header files (1/2)

<assert.h>	Consistency checking
<ctype.h>	Case conversion, classification / handling of characters
<errno.h>	Error codes, error checking, error reporting
<float.h>	Floating point parameters
<limits.h>	Width of types, limits for files, general limits, selecting conversions, reserved names
<locale.h>	Basic support for internationalization
<math.h>	Mathematics functions (e.g., rounding, normalization, absolute value, floating point classes)
<setjmp.h>	Non-local exits and jumps
<signal.h>	System signal processing

ANSI libraries and header files (2/2)

<stdarg.h>	“Variadic” functions
<stddef.h>	Definitions of some standard data types
<stdio.h>	Basic, stream-based input and output
<stdlib.h>	Several assorted functions, including ones for memory allocation, array manipulation and sorting, system handling and command execution, etc.
<string.h>	Basic string manipulation
<time.h>	Working with, and formatting, time

Before we start… (1/2)

- Although we will not cover that, you should be aware that the standard C library contains functions for handling multi-byte character sets (e.g., Unicode), and “strings” comprising such characters
- These functions should be preferred over their “simple” equivalents that we will cover here when writing programs for an international user base
- In most cases, these functions follow the same naming conventions as their non-multi-byte counterparts, but are declared in different header files, and may require that you compile / link with additional library modules
- We will cover also some functions that are marked GNU extension and are only available in the GNU version of the standard C library
  - To use them, you must include this line in your source files, before other #include statements: `#define _GNU_SOURCE`

Before we start… (2/2)

- What we have **already looked at**
  - Stream-based input / output
  - Formatted, character-based input / output

- What we **will not cover**
  - “Variadic” functions with an undefined number of parameters
  - Specific mathematical functions
  - Non-local “jumps”
  - Floating point arithmetic
  - Internationalization

Character class tests (<ctype.h>) (1/4)

`int` **`islower`** `(int c)`

- Returns true if *c* is a lower-case letter. The letter need not be from the Latin alphabet, any alphabet representable is valid.

`int` **`isupper`** `(int c)`

- Returns true if *c* is an upper-case letter. The letter need not be from the Latin alphabet, any alphabet representable is valid.

`int` **`isalpha`** `(int c)`

- Returns true if *c* is an alphabetic character (a letter). If islower or isupper is true of a character, then isalpha is also true.
- In some locales, there may be additional characters for which isalpha is true – letters which are neither upper case nor lower case. But in the standard "C" locale, there are no such additional characters.

Character class tests (<ctype.h>) (2/4)

`int` **`isdigit`** `(int c)`

» Returns true if *c* is a decimal digit (0 through 9)

`int` **`isalnum`** `(int c)`

» Returns true if *c* is an alphanumeric character (a letter or number); in other words, if either **`isalpha`** or **`isdigit`** is true of a character, then **`isalnum`** is also true

`int` **`isxdigit`** `(int c)`

» Returns true if *c* is a hexadecimal digit. Hexadecimal digits include the normal decimal digits 0 through 9 and the letters A through F and a through f

`int` **`ispunct`** `(int c)`

» Returns true if *c* is a punctuation character. This means any printing character that is not alphanumeric or a space character

Character class tests (<ctype.h>) (3/4)

`int` **`isspace`** `(int c)`

» Returns true if *c* is a *whitespace* character. In the standard "C" locale, **`isspace`** returns true for only the standard whitespace characters:

`' '`	space	`'\r'`	carriage return
`'\f'`	formfeed	`'\t'`	horizontal tab
`'\n'`	newline	`'\v'`	vertical tab

`int` **`isblank`** `(int c)`

» Returns true if *c* is a blank character; that is, a space or a tab

» **This function is a GNU extension**

`int` **`isgraph`** `(int c)`

» Returns true if *c* is a graphic character; that is, a character that has a glyph associated with it. The whitespace characters are not considered graphic

Character class tests (<ctype.h>) (4/4)

`int` **`isprint`** `(int c)`

- Returns true if *c* is a printing character. Printing characters include all the graphic characters, plus the space “ “ character

`int` **`iscntrl`** `(int c)`

- Returns true if *c* is a control character (that is, a character that is not a printing character)

`int` **`isascii`** `(int c)`

- Returns true if *c* is a 7-bit unsigned char value that fits into the US/UK ASCII character set
  - **This function is a BSD extension and is also an SVID extension.**

Error checking (<errno.h>)

- Most library functions return a special value to indicate that they have failed. The special value is typically **`-1`**, a **`null` pointer**, or a **constant** such as **`EOF`** that is defined for that purpose.

- But this return value tells you only that an error has occurred. To find out what kind of error it was, you need to look at the error code stored in the variable **`errno`**.
  - This “variable” is declared in the header file **`<errno.h>`**.

- All error codes that may be returned by individual library functions have a corresponding “symbolic” name, defined in **`<errno.h>`**.

- Although you can change the value of **`errno`**, it is recommended that you only do so if you *really* know what you are doing (except setting it to 0!).

» Setting **`errno`** to zero immediately before the function call is **necessary** to get the correct return value / error code for some library functions such as **`sqrt`**, **`atan`**, etc.

Error reporting functions (1/2)

`char * strerror (int errnum)`

- The strerror function maps the error code specified by the *errnum* argument to a descriptive error message string. The return value is a pointer to this string. The value *errnum* normally comes from the variable errno.
- You should never modify the string returned by strerror (static shared buffer)
- If you make subsequent calls to strerror, the string might be overwritten.
- The function strerror is declared in <string.h>.

`char * strerror_r (int errnum, char *buf, size_t n)`

- The strerror_r function works like strerror but instead of returning the error message in a statically allocated buffer shared by all threads in the process, it returns a private copy for the thread. This might be either some permanent global data or a message string in the user supplied buffer starting at *buf* with the length of *n* bytes. At most *n* characters are written (including the NUL byte) so it is up to the user to select the buffer large enough.

Error reporting functions (2/2)

(cont.)

- This function should be used in multi-threaded programs since there is no way to guarantee the string returned by strerror really belongs to the last call of the current thread.
- This function strerror_r is a **GNU extension** and it is declared in <string.h>.

```
void perror (const char *message)
```

- This function prints an error message to the stream stderr. If you call perror with a *message* that is either a null pointer or an empty string, perror just prints the error message corresponding to errno, adding a trailing newline.
- If you supply a non-null *message* argument, then perror prefixes its output with this string. It adds a colon and a space character to separate the *message* from the error string corresponding to errno.
- The function perror is declared in <stdio.h>.

String functions (<string.h>) (1/11)

`size_t` **`strlen`** `(const char *s)`

- The `strlen` function returns the length of the NUL-terminated string *s* in bytes
  - In other words, it returns the offset of the terminating NUL character within the array

`size_t` **`strnlen`** `(const char *s, size_t maxlen)`

- The `strnlen` function returns the length of the string *s* in bytes if this length is smaller than *maxlen* bytes. Otherwise it returns *maxlen*. Therefore this function is equivalent to

  ```
  (strlen (s) < n ? strlen (s) : maxlen)
  ```

  but it is more efficient and works even if the string *s* is not NUL-terminated

String functions (<string.h>) (2/11)

`char * strcpy (char *to, const char *from)`

» This copies characters from the string *from* (up to and including the terminating NUL character) into the string *to*. This function has undefined results if the strings overlap. The return value is the value of *to*.

`char * strncpy (char *to, const char *from, size_t size)`

» This function is similar to `strcpy` but always copies exactly *size* characters into *to*. If the length of *from* is more than *size*, then `strncpy` copies just the first *size* characters. Note that in this case there is **no NUL terminator written** into *to!*

» If the length of *from* is less than *size*, then `strncpy` copies all of *from*, followed by enough NUL characters to add up to *size* characters in all. This behaviour is rarely useful, but it is specified by the ISO C standard.

» The behaviour of `strncpy` is undefined if the strings overlap.

String functions (<string.h>) (3/11)

`char * strdup (const char *s)`

- This function copies the NUL-terminated string *s* into a newly allocated string
- The string is allocated using `malloc`
- If `malloc` cannot allocate space for the new string, `strdup` returns a NULL pointer, otherwise it returns a pointer to the new string

`char * strndup (const char *s, size_t size)`

- This function is similar to `strdup` but always copies at most *size* characters into the newly allocated string
- If the length of *s* is more than *size*, then `strndup` copies just the first *size* characters and adds a closing NUL terminator, otherwise all characters are copied and the string is terminated
- This function is different from `strncpy` in that it **always** NUL-terminates the destination string!
- **`strndup`** is a **GNU extension**

String functions (<string.h>) (4/11)

`char * strcat (char *to, const char *from)`

- The strcat function is similar to strcpy, except that the characters from *from* are concatenated or appended to the end of *to*, instead of overwriting it
- That is, the first character from *from* overwrites the NUL character marking the end of *to*
- This function has undefined results if the strings overlap

`char * strncat (char *to, const char *from, size_t size)`

- This function is like strcat except that not more than *size* characters from *from* are appended to the end of *to*
- A single NUL character is also **always** appended to *to*, so the total allocated size of *to* must be at least *size* + 1 bytes longer than its initial length
- The behaviour of `strncat` is undefined if the strings overlap

String functions (<string.h>) (5/11)

`int` **`strcmp`** `(const char *s1, const char *s2)`

- The `strcmp` function compares the string *s1* against *s2*, returning a value that has the same sign as the difference between the first differing pair of characters (interpreted as unsigned char objects, then promoted to int)
- If the two strings are equal, `strcmp` returns 0.
- A consequence of the ordering used by `strcmp` is that if *s1* is an initial substring of *s2*, then *s1* is considered to be "less than" *s2*
  - The NUL (=0) of s1 is less than whatever character is in s2
- `strcmp` does not take sorting conventions of the language the strings are written in into account. To get that one has to use `strcoll`.

`int` **`strncmp`** `(const char *s1, const char *s2, size_t size)`

- This function is the similar to `strcmp`, except that no more than *size* characters are compared
- In other words, if the two strings are the same in their first *size* characters, the return value is zero

String functions (<string.h>) (6/11)

`int` **`strcasecmp`** `(const char *s1, const char *s2)`

- This function is like `strcmp`, except that differences in case are ignored
- How uppercase and lowercase characters are related is determined by the currently selected locale
  - In the standard "C" locale the characters Ä and ä do not match but in a locale which regards these characters as parts of the alphabet they do

`int` **`strcoll`** `(const char *s1, const char *s2)`

- The `strcoll` function is similar to `strcmp` but uses the collating sequence of the current locale for collation (the LC_COLLATE locale)

`char *` **`strchr`** `(const char *string, int c)`

- The `strchr` function finds the first occurrence of the character *c* (converted to a char) in the NUL-terminated string beginning at *string*
- The return value is a pointer to the located character, or a NULL pointer if no match was found

String functions (<string.h>) (7/11)

`char * **strrchr** (const char *string, int c)`

» The function `strrchr` is like `strchr`, except that it searches backwards from the end of the string *string* (instead of forwards from the front)

`char * **strstr** (const char *haystack, const char *needle)`

» This is like `strchr`, except that it searches *haystack* for a substring *needle* rather than just a single character

» It returns a pointer into the string *haystack* that is the first character of the substring, or a NULL pointer if no match was found

» If *needle* is an empty string, the function returns *haystack*

`char * **strcasestr** (const char *haystack, const char *needle)`

» This is like `strstr`, except that it ignores case in searching for the substring

» Like `strcasecmp`, it is locale-dependent how uppercase and lowercase characters are related

String functions (<string.h>) (8/11)

`char * strtok (char *newstring, const char *delimiters)`

- A string can be split into tokens by making a series of calls to the function `strtok`. **The string to be split up is passed as the *newstring* argument on the first call only**. The `strtok` function uses this to set up some internal state information.
- Subsequent calls to get additional tokens from the same string are indicated by passing a NULL pointer as the *newstring* argument
- Calling `strtok` with another non-NULL *newstring* argument reinitializes the state information. It is guaranteed that no other library function ever calls `strtok` behind your back (which would mess up this internal state information).
- The *delimiters* argument is a string that specifies a set of delimiters that may surround the token being extracted. All the initial characters that are members of this set are discarded. The first character that is *not* a member of this set of delimiters marks the beginning of the next token. The end of the token is found by looking for the next character that is a member of the delimiter set. This character in the original string *newstring* is **overwritten by a NUL character**, and the pointer to the beginning of the token in *newstring* is returned.

String functions (<string.h>) (9/11)

(cont.)

- On the next call to `strtok`, the searching begins at the next character beyond the one that marked the end of the previous token. Note that the set of delimiters *delimiters* do not have to be the same on every call in a series of calls to `strtok`.
- If the end of the string *newstring* is reached, or if the remainder of string consists only of delimiter characters, `strtok` returns a NULL pointer.
- Note that "character" is here used in the sense of byte. In a string using a multibyte character encoding (abstract) character consisting of more than one byte are not treated as an entity. Each byte is treated separately. The function is not locale-dependent.

- **Warning:** Since `strtok` alters the string it is parsing, you should always copy the string to a temporary buffer before parsing it with `strtok`. If you allow `strtok` to modify a string that came from another part of your program, you are asking for trouble; that string might be used for other purposes after is has beed modified, and it would not have the expected value!

String functions (<string.h>) (10/11)

`void * memcpy (void *to, const void *from, size_t size)`

» The `memcpy` function copies *size* bytes from the object beginning at *from* into the object beginning at *to*. The behaviour of this function is undefined if the two arrays *to* and *from* overlap; use `memmove` instead if overlapping is possible. The value returned by `memcpy` is the value of *to*.

`void * memmove (void *to, const void *from, size_t size)`

» `memmove` copies the *size* bytes at *from* into the *size* bytes at *to*, **even if those two blocks of space overlap**. In the case of overlap, `memmove` is careful to copy the original values of the bytes in the block at *from*, including those bytes which also belong to the block at *to*. The value returned by `memmove` is the value of *to*.

`void * memset (void *block, int c, size_t size)`

» This function copies the value of *c* (converted to an unsigned char, i.e. a single byte) into each of the first *size* bytes of the object beginning at *block*. It returns the value of *block*.

String functions (<string.h>) (11/11)

`int` **`memcmp`** `(const void *a1, const void *a2, size_t size)`

- The function `memcmp` compares *size* bytes of memory beginning at *a1* against *size* bytes of memory beginning at *a2*
- The value returned has the same sign as the difference between the first differing pair of bytes (interpreted as unsigned char, then promoted to int)

`void *` **`memchr`** `(const void *block, int c, size_t size)`

- This function finds the first occurrence of the byte *c* (converted to an unsigned char) in the initial *size* bytes of the object beginning at *block*
- The return value is a pointer to the located byte, or a NULL pointer if no match was found

`void *` **`memrchr`** `(const void *block, int c, size_t size)`

- The function `memrchr` is like `memchr`, except that it searches backwards from the end of the block defined by *block* and *size* (instead of forwards from the front)

Time and date functions (<time.h>) (1/7)

`time_t`

- This is the datatype used to represent simple time. Sometimes, it also represents an elapsed time. When interpreted as a calendar time value, it represents the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time.
  - This calendar time is sometimes referred to as the *Unix time* or *Unix epoch*
- Note that a simple time has no concept of local time zone. Calendar Time *T* is the same instant in time regardless of where on the globe the computer is.
- In the GNU C library, time_t is equivalent to long int. In other systems, time_t might be either an integer or floating-point type.

`time_t` **`time`** `(`*`time_t *result`*`)`

- The time function returns the current calendar time as a value of type time_t. If the argument *result* is not a NULL pointer, the calendar time value is also stored in **result*
- If the current time is not available, the value **`(time_t)(-1)`** is returned

Time and date functions (<time.h>) (2/7)

`struct tm`

» This is the data type used to represent a broken-down time. The structure contains at least the following members, which can appear in any order.

`int tm_sec` … number of full seconds since the top of the minute (normally in the range 0 through 59, but the actual upper limit is 60, to allow for leap seconds if leap second support is available)

`int tm_min` … number of full minutes since the top of the hour (in the range 0 through 59)

`int tm_hour` … number of full hours past midnight (in the range 0 through 23)

`int tm_mday` … ordinal day of the month (in the range 1 through 31). Watch out for this one! As the only ordinal number in the structure, it is inconsistent with the rest of the structure (which are 0-based)!

`int tm_mon` … number of full calendar months since the beginning of the year (in the range 0 through 11). Watch out for this one! People usually use ordinal numbers for month-of-year (where January = 1).

`int tm_year` … number of full calendar years since 1900

Time and date functions (<time.h>) (3/7)

(cont.)

`int tm_wday` … number of full days since Sunday (in the range 0 through 6)
`int tm_yday` … number of full days since the beginning of the year (range 0 – 365)

`int tm_isdst` … flag that indicates whether Daylight Saving Time is (or was, or will be) in effect at the time described. The value is positive if Daylight Saving Time is in effect, zero if it is not, and negative if the information is not available.

```
struct tm * localtime (const time_t *time)
```

- The `localtime` function converts the simple time pointed to by *time* to broken-down time representation, expressed relative to the user's specified time zone
- The return value is a pointer to a static broken-down time structure, which might be overwritten by subsequent calls to `ctime`, `gmtime`, or `localtime`
  - But no other library function overwrites the contents of this object
- The return value is the NULL pointer if *time* cannot be represented as a broken-down time; typically this is because the year cannot fit into an int

Time and date functions (<time.h>) (4/7)

`time_t` **`mktime`** `(`*`struct tm *brokentime`*`)`

- The `mktime` function is used to convert a broken-down time structure to a simple time representation. It also "normalizes" the contents of the broken-down time structure, by filling in the day of week and day of year based on the other date and time components. The `mktime` function ignores the specified contents of the tm_wday and tm_yday members of the broken-down time structure. It uses the values of the other components to determine the calendar time; it's permissible for these components to have unnormalized values outside their normal ranges. The last thing that `mktime` does is adjust the components of the *brokentime* structure (including the tm_wday and tm_yday).
- If the specified broken-down time cannot be represented as a simple time, `mktime` returns a value of **`(time_t)(-1)`** and does not modify the contents of *brokentime*.

Time and date functions (<time.h>) (5/7)

`char * asctime (const struct tm *brokentime)`

- The `asctime` function converts the broken-down time value that *brokentime* points to into a string in a standard format: "`Tue May 21 13:46:22 1991\n`" Abbreviations for the days of week are: Sun, Mon, Tue, Wed, Thu, Fri, and Sat.
- The abbreviations for the months are: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec
- These strings are fixed and not localized to the locale in use
- The return value points to a statically allocated string, which might be overwritten by subsequent calls to `asctime` or `ctime`
  - But no other library function overwrites the contents of this string

`char * asctime_r (const struct tm *brokentime, char *buffer)`

- This function is similar to `asctime` but instead of placing the result in a static buffer it writes the string in the buffer pointed to by the parameter *buffer*
- This buffer should have room for at least 26 bytes, including the terminating null
- If no error occurred the function returns a pointer to the string the result was written into, i.e., it returns *buffer; o*therwise it returns NULL

Time and date functions (<time.h>) (6/7)

```
char * ctime (const time_t *time)
```

» The `ctime` function is similar to `asctime`, except that you specify the calendar time argument as a time_t simple time value rather than in broken-down local time format. It is equivalent to `asctime (localtime (time))`

```
char * ctime_r (const time_t *time, char *buffer)
```

» This function is similar to `ctime`, but places the result in the string pointed to by *buffer*. If no error occurred the function returns a pointer to the string the result was written into, i.e., it returns *buffer; o*therwise it returns NULL.

```
size_t strftime (char *s, size_t size,
                 const char *template,
                 const struct tm *brokentime)
```

» This function is similar to the `sprintf` function, but the conversion specifications that can appear in the format template *template* are specialized for printing components of the date and time *brokentime* according to the locale currently specified for time conversion

Time and date functions (<time.h>) (7/7)

```
char * strptime (const char *s, const char *fmt,
                 struct tm *tp)
```

- The `strptime` function parses the input string *s* according to the format string *fmt* and stores its results in the structure *tp*. The input string could be generated by a `strftime` call or obtained any other way. It does not need to be in a human-recognizable format; e.g. a date passed as "`02:1999:9`" is acceptable, even though it is ambiguous without context. As long as the format string *fmt* matches the input string the function will succeed.
- The user has to make sure, though, that the input can be parsed in an unambiguous way. The string "`1999112`" can be parsed using the format "`%Y%m%d`" as 1999-1-12, 1999-11-2, or even 19991-1-2. It is necessary to add appropriate separators to get reliable results.
- The format string consists of the same components as the format string of the `strftime` function

Standard utility functions (<stdlib.h>) (1/8)

- **Memory (de-)allocation**

```
void * malloc (size_t size)
```

  - This function returns a pointer to a newly allocated block *size* bytes long, or a NULL pointer if the block could not be allocated

```
void * calloc (size_t count, size_t eltsize)
```

  - This function allocates a block long enough to contain a vector of *count* elements, each of size *eltsize*
  - Its contents are cleared to “all bits zero” before calloc returns

```
void free (void *ptr)
```

  - This function deallocates the block of memory pointed at by *ptr*

Standard utility functions (<stdlib.h>) (2/8)

`void * realloc (void *ptr, size_t newsize)`

- The `realloc` function changes the size of the block whose address is *ptr* to be *newsize*. Since the space after the end of the block may be in use, `realloc` may find it necessary to copy the block to a new address where more free space is available. The return value of `realloc` is the new address of the block. If the block needs to be moved, `realloc` copies the old contents.
- If you pass a NULL pointer for *ptr*, `realloc` behaves just like `malloc(`*newsize*`)`. This can be convenient, but beware that older implementations (before ISO C) may not support this behaviour, and will probably crash when `realloc` is passed a NULL pointer.
- Like `malloc, realloc` may return a NULL pointer if no memory space is available to make the block bigger. When this happens, the original block is untouched; it has not been modified or relocated.
- This function can also be used to shrink the reserved space
- Passing zero for `newsize` is the equivalent of a `free`
- **Note**: when `realloc` moves the block of data in memory, other pointers still point to the old addresses and need to be adjusted manually!
- **Note**: `ptr` must be the beginning of the block (i.e. what was returned my `malloc`) and cannot be a pointer “somewhere in there!

Standard utility functions (<stdlib.h>) (3/8)

- **Array functions**

```
void * bsearch (const void *key, const void *array,
                size_t count, size_t size,
                comparison_fn_t compare)
```

  - The `bsearch` function searches the sorted array *array* for an object that is equivalent to *key*. The array contains *count* elements, each of which is of size *size* bytes. The *compare* function is used to perform the comparison. This function is called with two pointer arguments and should return an integer less than, equal to, or greater than zero corresponding to whether its first argument is considered less than, equal to, or greater than its second argument. The elements of the *array* must already be sorted in ascending order according to this comparison function.
  - The return value is a pointer to the matching array element, or a NULL pointer if no match is found. If the array contains more than one element that matches, the one that is returned is unspecified.
  - This function derives its name from the fact that it is implemented using the binary search algorithm

Standard utility functions (<stdlib.h>) (4/8)

```
void qsort (void *array, size_t count, size_t size,
            comparison_fn_t compare)
```

- The `qsort` function sorts the array *array*. The array contains *count* elements, each of which is of size *size*. The *compare* function is used to perform the comparison on the array elements. This function is called with two pointer arguments and should return an integer less than, equal to, or greater than zero corresponding to whether its first argument is considered less than, equal to, or greater than its second argument.
- **Warning:** If two objects compare as equal, their order after sorting is unpredictable. That is to say, the sorting is not stable. This can make a difference when the comparison considers only parts of the elements. Two elements with the same sort key may differ in other respects.

Standard utility functions (<stdlib.h>) (5/8)

- In the previous slide, we made use of the **comparison_fn_t** function pointer type. This type is a GNU extension and is defined in <stdlib.h> as follows:

```
int comparison_fn_t (const void *, const void *);
```

- If your functions comply with the prototype above, then you don't need to worry about whether **comparison_fn_t** has been defined in the libc implementation you are using.

- To search through unsorted arrays, you can also use the **lsearch(…)** functions, defined in <search.h>.

Standard utility functions (<stdlib.h>) (6/8)

- **Random numbers**

`int` **`RAND_MAX`**

» The value of this macro is an integer constant representing the largest value the rand function can return. In the GNU library, it is 2147483647, which is the largest signed integer representable in 32 bits. In other libraries, it may be as low as 32767.

`int` **`rand`** `(`*`void`*`)`

» The rand function returns the next pseudo-random number in the series. The value ranges from 0 to RAND_MAX.

`void` **`srand`** `(`*`unsigned int seed`*`)`

» This function establishes *seed* as the seed for a new series of pseudo-random numbers. If you call `rand` before a seed has been established with `srand`, it uses the value 1 as a default seed. To produce a different pseudo-random series each time your program is run, do `srand(time(0))`.

Standard utility functions (<stdlib.h>) (7/8)

- **Miscellaneous functions**

`int` **`system`** `(const char *command)`

» This function executes *command* as a shell command. In the GNU C library, it always uses the default shell sh to run the command. In particular, it searches the directories in PATH to find programs to execute. The return value is -1 if it wasn't possible to create the shell process, and otherwise is the status of the shell process. If the *command* argument is a NULL pointer, a return value of zero indicates that no command processor is available.

`void` **`exit`** `(int status)`

» The exit function tells the system that the program is done, which causes it to terminate the process. *status* is the program's exit status, which becomes part of the process' termination status.

» This function does not return

Standard utility functions (<stdlib.h>) (8/8)

```
int atexit (void (*function) (void))
```

- The `atexit` function registers the function *function* to be called at normal program termination
- The *function* is called with no arguments
- The return value from `atexit` is zero on success and nonzero if the function cannot be registered