Multidimensional Arrays:

Know the aspects of how these work:

- Declaring a multidimensional array
- Assigning a multidimensional array values
    - How and where values are stored

See full explanation on *pages 218 - 221*.

Arrays of Pointers:

See full explanation on *pages 228 - 231*.

Chapter 9 - Strings, Characters, and Bytes ✓

String:

- **String:** a sequence of zero or more characters followed by a NULL byte, which is a sbyte whose bits are all zero
    - Therefore, it is not possible for a string to *contain* a NULL as one of its characters
- **Header:** `string.h`
- The Standard reserves all function names that begin with `str` for future expansion of the library
    - Prototype for the library **strlen**: `size_t strlen(char const *string);`
- Most commonly used string functions are ***unrestricted***: they determine the length of their string arguments solely by looking for the terminating NULL byte *(pg. 244 - 245)*

Copying Strings:

- Prototype for **strcpy**: `char *strcpy(char *dst, char const *src);` *(pg. 246)*

Consider this example:

```
char  message[] = "Original message";
...
if( ... )
      strcpy( mesaage, "Different" );
```

If the condition is true and the copy is performed, the array will contain the following:

The characters after the first NUL byte are never accessed by the string functions and are, for all practical purposes, lost.

'D'	'i'	'f'	'f'	'e'	'r'	'e'	'n'	't'	0	'e'	's'	's'	'a'	'g'	'e'	0

It is up to the programmer to make sure that the destination array is large enough to hold the string. If the string is longer than the array, the excess characters will be copied anyway and will overwrite whatever values happen to be after the array in memory, `strcpy` is unable to avoid this problem because it cannot determine the size of the destination array. For example:

```
char  message[] = "Original message";
...
strcpy( message, "A different message" );
```

The second string is too long to fit in the array, so the `strcpy` will run off the end of the array and overwrite whatever variables happen to follow it in memory. You can avoid a lot of debugging by making sure that the destination argument is large enough before calling `strcpy`.

Concatenating Strings:

- Prototype for **strcat**: `char *strcat(char *dst, char const *src);`

The following example shows a common use of this function.

```
strcpy( message, "Hello " );
strcat( message, customer_name );
strcat( message, ", how are you?" );
```

Each of the arguments to `strcat` are appended to the string already in `message`. The result is a string such as this one:

```
Hello Jim, how are you?
```

String Comparison:

- Prototype for **strcmp**: `int *strcmp(char const *s1, char const *s2);`
    - Returns a value less than zero if `s1` is less than `s2`
    - Returns a value greater than zero if `s1` is greater than `s2`
    - Returns zero if the two strings are equal

Finding a Character:

- The easiest way to locate a character is with **strchr** and **strrchr**
    - Prototype for **strchr**: `char *strchr(char const *str, int ch);`
        - Searches from the beginning of the string
    - Prototype for **strrchr**: `char *strrchr(char const *str, int char);`
        - Searches from the end of the string

```
The strchr() function returns a pointer to the first occurrence of the character c in the string s.

The strrchr() function returns a pointer to the last occurrence of the character c in the string s.

The  strchrnul() function is like strchr() except that if c is not found in s, then it returns a pointer to
the null byte at the end of s, rather than NULL.
```

```
char  string[20] = "Hello there, honey.";
char  *ans;

ans = strchr( string, 'h' );
```

    - `ans` will get the value `string + 7` because the first `'h'` appears in this position.

Finding Any of Several Characters:

- Prototype for **strpbrk**: `int *strpbrk(char const *s1, char const *s2);`
    - This function returns a pointer to the first character in str that matches ***any*** of the characters in group
    - Returns NULL if none matched

```
char  string[20] = "Hello there, honey.";
char  *ans;

ans = strpbrk( string, "aeiou" );
```

    - `ans` will get the value `string + 1` because this position is the first that contains any of the characters in the second argument.

Finding a Substring:

- Prototype for **strstr:** `char *strstr(char const *s1, char const *s2);`
    - This function finds the first place in `s1` where the entire string `s2` begins and returns a pointer to this location.
    - If `s2` doesn't' appear in its entirety anywhere in `s1`, then NULL is returned

Finding Tokens:

- Prototype for **strtok:** `char *strtok(char *str, char const *sep);`

The `sep` argument is a string that defines the set of characters that are used as separators. The first argument specifies a string that is assumed to contain zero or more tokens separated from one another by one or more characters from the `sep` string. `strtok` finds and NUL-terminates the next token in `str`, and returns a pointer to the token.

While it is doing its work, `strtok` modifies the string that it is processing. If the string must not be changed, copy it and use `strtok` on the copy.

If the first argument to `strtok` is not NULL, the function finds the first token in the string, `strtok` also saves its position in the string. If the first argument to `strtok` is NULL, the function uses the saved position to find the *next* token from the same string as before. `strtok` returns a NULL pointer when there aren't any more tokens in the string. Typically, a pointer to a string is passed on the first call to `strtok`. The function is then called repeatedly with NULL first argument until it returns NULL.

Error Messages:

- Prototype for **strerror:** `char *strerror(int error_number);`
    - Errors that occur are reported by setting an external integer variable called `errno` to an error code. The `strerror` function takes one of these error codes as an argument and returns a pointer to a message describing the error

Character Operations and Classification:

Function	Returns True if its Argument is
`iscntrl`	any control character
`isspace`	a whitespace character: space `' '`, form feed `'\f'`, newline `'\n'`, carriage return lab `'\t'`, or vertical tab `'\v'`.
`isdigit`	a decimal digit `0` through `9`.
`isxdigit`	a hexadecimal digit, which includes the decimal digits and the letters `a` through `f` and `A` through `F`.
`islower`	a lowercase letter `a` through `z`.
`isupper`	an uppercase letter `A` through `Z`.
`isalpha`	an alphabetic character `a` through `z` or `A` through `Z`.
`isalnum`	an alphabetic or a numeric character `a` through `z`, `A` through `Z`, or `0` through `9`.
`ispunct`	punctuation: any character with a graphic (printable symbol) associated with it that is not alphanumeric.
`isgraph`	any character with a graphic associated with it.
`isprint`	any printing character, which includes the graphic characters and the space character.

Memory Operations:

- **`memcpy`:** copies *length* bytes, beginning at `src`, into the memory beginning at `dst`
    - `void *memcpy( void *dst, void const *src, size_t length );`
- **`memmove`:** behaves exactly like memcpy except that its source and destination operands may overlap
    - `void *memmove( void *dst, void const *src, size_t length );`
- **`memcmp`:** compares the `length` bytes of memory beginning at `a` to the bytes beginning at `b`; the values are compared byte by byte as *unsigned characters*, and the function returns the same type of value as `strcmp`— a negative value if `a < b`, a positive value if `a > b`, and zero if they are equal
    - `void *memcmp( void const *a, void const *b, size_t length );`
- **`memchr`:** searches the `length` bytes beginning at `a` for the first occurrence of the character `ch` and returns a pointer to the location
    - `void *memchr( void const *a, int ch, size_t length );`
- **`memset`:** sets each of the length bytes begging at `a` to the character value `ch`
    - `void *memset( void *a, int ch, size_t length );`

Chapter 10 - Structures and Unions ✓

Structure Basics:

- **Aggregate data type:** one that can hold more than one individual piece of data at a time
- **Structure:** a collection values, called *members*, but the members of a structure may be of different types
- **Structure variable:** is a scalar, so you can perform the same kinds of operations with it that you can with other scalars

Structure Declarations:

- **`struct tag {member-list} variable-list;`**
  - *Ex.* This declaration creates a single variable named `x`, which contains 3 members: an integer, a character, and a float.

```
struct      {
            int         a;
            char        b;
            float c;
} x;
```

  - *Ex.* This declaration creates an array `y` of twenty structures and z, which is a pointer to the structure of this type.

```
struct      {
            int         a;
            char        b;
            float c;
} y[20], *z;
```

- ***Tag* field:** allows a name to be given to the member list so that it can be referenced in subsequent declarations
  - The tag allows many declarations to use the same member list, thus creating structures of the same type

```
struct      SIMPLE      {
            int         a;
            char        b;
            float c;
};

struct      SIMPLE      x;
struct      SIMPLE      y[20],
```

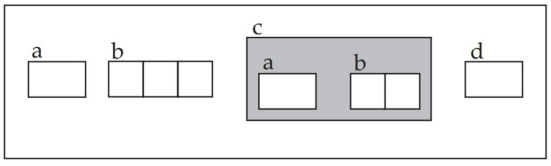
Member Access:

- Direct member access: (see page 272 - 273)
- Indirect member access: (see page 273)

Structures, Pointers, and Members:

```
typedef     struct      {
            int         a;
            short b[2];
} Ex2;
typedef     struct      EX {
            int         a;
            char        b[3];
            Ex2         c;
            struct      EX    *d;
} Ex;
```

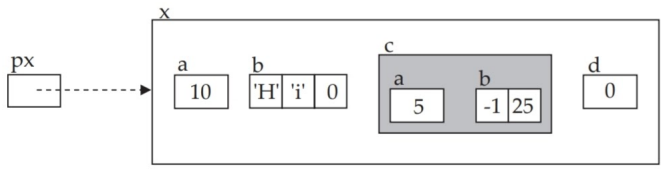
Structures of type `Ex` will be pictured like this:



The first examples will use these declarations:

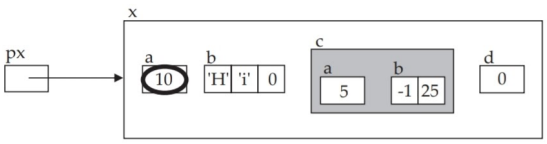
```
Ex    x = { 10, "Hi", {5, { -1, 25 } }, 0 };
Ex    *px = &x;
```

which produce the following variables:



Pointers:

Now let's look at the arrow operator. The R-value of the expression `px->a` is



The -> operator applies indirection to `px` (indicated by the solid arrow) in order to get the structure, and then selects the a member. The expression `px->a` is used when you have a pointer to a structure but do not know its name. If you knew the name of this structure, you could use the equivalent expression `x.a` instead.

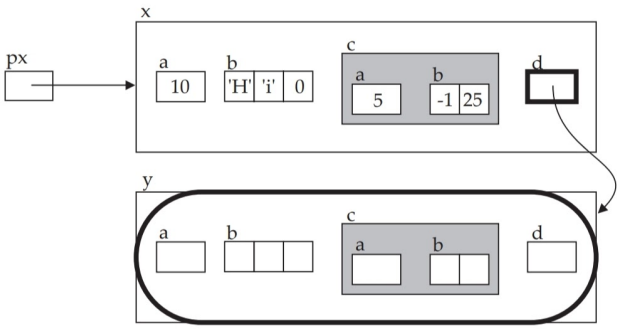
Let's pause here and compare the expressions `*px` and `px->a` to each other. In both cases, the address in `px` is used to fold the structure. But the first member in the structure is `a`, so the address of `a` is the same as the address of the structure. It would seem, then, that `px` points to the structure *and* to the first member of the structure: after all, they both have the same address. This analysis is only half correct, though. Although both addresses have the same value, they have different types. The variable `px` was declared as a pointer to a structure, so the result of the expression `*px` is the whole structure, not its first member.

Accessing Pointer Member:

Let's create another structure and set `x.d` to point to it

```
Ex     y;
x.d = &y;
```

Now we can evaluate `*px->d`.



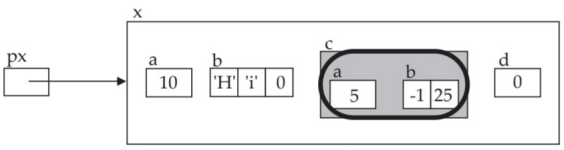
Bit Fields:

- Programs intended to be portable should avoid bit fields because bit fields may work differently on various systems
- They should typically be declared as *unsigned* or *signed*
- Ex. of bit field declaration:

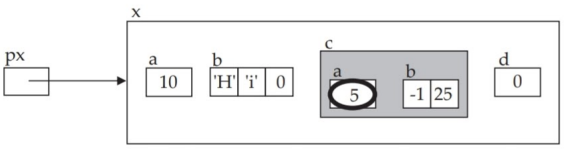
```
struct      CHAR  {
            unsigned ch       : 7;
            unsigned font     : 6;
            unsigned size     : 19;
};
struct      CHAR  ch1;
```

Accessing Nested Structure:

To access the member c, which is a structure, use the expression `px->c`. Its R-value is the entire structure.



The dot operator can be added to this expression to access specific members of `c`. For example, the expression `px->c.a` has the following R-value:



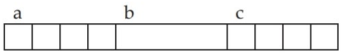
This expression contains both the dot and arrow operators. The arrow is used because `px` is not a structure, it *points to* a structure. Then the dot operator is used because `px->c` does not point to a structure, it *is* a structure.

Storage Allocation:

To illustrate, consider this structure:

```
struct      ALIGN {
            char        a;
            int         b;
            char        c;
};
```

On a machine whose integers occupy four bytes and must begin at a byte whose address is evenly divisible by four, this structure would appear like this in memory:



Structures as Function Arguments:

```
typedef     struct      {
            char        product[PRODUCT_SIZE];
            int         quantity;
            float       unit_price;
            float       total_amount;
} Transaction;
```

When a transaction occurs, there are many steps involved, one of which is printing the receipt. Let's look at some different ways to perform this task.

```
void
print_receipt( Transaction trans )
{
      printf( "%s\n", trans.product );
      printf( "%d @ %.2f total %.2f\n", trans.quantity,
            trans.unit_price, trans.total_amount );
}
```

Read pages 288 - 291.

Unions:

- Declared like a structure but doesn't work like one
    - All members of a union refer to ***the same location(s) in memory***
    - Read *pages 291 - 294*.

Chapter 11 - Dynamic Memory Allocation ✓

Malloc and Free:

- Use the library `stdlib.h`
- Prototypes:
    - `void *malloc( size_t size );`
    - `void free( void *pointer );`
- **`malloc:` DON'T CAST MALLOC**
    - The argument to `malloc` is the number of bytes (characters) of memory that are needed. If the desired amount of memory is available, `malloc` returns a pointer to the beginning of the allocated block.
    - If the pool of memory is empty or does not contain a big enough block, `malloc` calls the operating system to obtain more memory and begins allocating pieces from this new chunk.
        - It the operating system is unable to give more memory, then a NULL pointer is returned
- **`free:`**
    - The argument to `free` must either be NULL or a value that was previously returned from `malloc`, `calloc`, or `realloc`.
    - Passing a NULL argument to `free` has no effect

Calloc and Realloc:

- Prototypes:
    - `void *calloc( size_t num_elementa, size_t element_size );`
    - `void *realloc( void *ptr, size_t new_size );`
- **`calloc:`**
    - Allocates memory; similar to `malloc`, but initializes the memory to zero before returning a pointer to it
    - Takes the number of elements desired and the amber of bytes in each element, then computers the total number of bytes needed
- **`realloc:`**
    - Used to change the size of a previously allocated block of memory
    - You can make a block larger or smaller
    - If the block cannot be resized, `realloc` will allocate a different block of the right size and copy the contents of the old block to the new one
        - Thus, you must not use the old pointer to the block after a call to `realloc`. Use a new pointer that is returned instead

Keywords and Notes:

- **Memory leak:** allocating memory by not freeing it later
- Read *pages 306 - 317*.
    - This contains more code examples of malloc, free, calloc, and realloc.
- Read Lecture 9 tldr notes.
    - This contains code examples and dynamic memory errors.

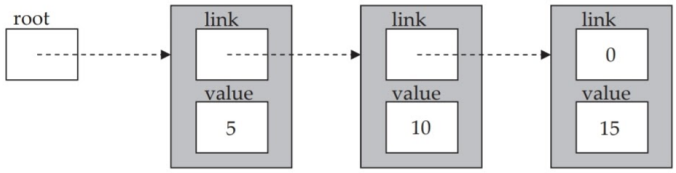
Chapter 12 - Using Structures and Pointers ✓

Linked List:

- A collection independent structures (often called *nodes*) that contain data
- Access the nodes in a list by following pointers

Singly Linked List:

- Each node contains a pointer to the next node in the list
- The pointer field of the last node in the list contains NULL to indicate that there are no more nodes in the list

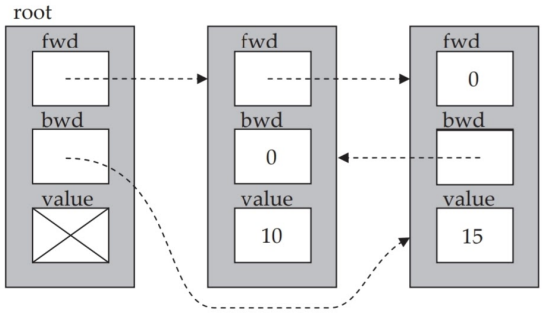


The nodes in this example are structures created with the following declaration.

```
typedef     struct      NODE  {
            struct      NODE  *link;
            int               value;
} Node;
```

Doubly Linked List:

- Each node has 2 pointers: 1 to the next node in the list and one to the previous node
- The back pointer lets us traverse doubly linked lists in either direction



Here is the declaration for the node type.

```
typedef     struct      NODE  {
            struct      NODE  *fwd;
            struct      NODE  *bwd;
            int               value;
} Node;
```

Notes:

- Read the chapter to see code examples of these 2 list types

Chapter 13 - Advanced Pointers Topics ✓

Pointers to Pointers:

Given this information, what is the effect of each of these statements?

```
int   i;
int   *pi;
int   **ppi;
```

```
❶ printf( "%d\n", ppi );
❷ printf( "%d\n", &ppi );
❸ *ppi = 5;
```

❶ If `ppi` is an automatic variable, it is uninitialized and a random value is printed. If it is a static variable, zero is printed.
❷ This statement prints, as a decimal integer, the address where `ppi` is stored. This value is not very useful.
❸ The result of this statement is unpredictable. Indirection should not be performed on `ppi` because it is not yet initialized.

- Function that returns a pointer to an integer: `int *f();`
- **`f`** is a pointer to a function that returns an integer: `int (*f)();`
- **`f`** is a pointer to a function that returns a pointer to an integer: `int *(*f)();`
- **`f`** is an array of pointers to integers: `int *f[];`
- **`f`** is an array of pointers to functions that return integers: `int (*f[])();`
- **`f`** is an array of pointers to functions that return pointers to integers: `int *(*f[])();`

Callback Functions:

- Simple function that locates a value in a singly linked list; its arguments are a pointer to the first node in the list and the value to locate; only works with linked lists whose values integers

```
Node*
search_list( Node *node, int const value )
{
       while( node != NULL ){
             if( node->value == value )
                   break;
             node = node->link;
       }
       return node;
}
```

- **Callback function:** the user passes a pointer to a function to some other routine, which then "calls back" to the user's function
- **Typeless linked list search:**

```
/*
** Function to search a linked list for a specific value.  Arguments
** are a pointer to the first node in the list, a pointer to the
** value we're looking for, and a pointer to a function that compares
** values of the type stored on the list.
*/
#include <stdio.h>
#include "node.h"

Node *
search_list( Node *node, void const *value,
    int (*compare)( void const *, void const * ) )
{
      while( node != NULL ){
            if( compare( &node->value, value ) == 0 )
                  break;
            node = node->link;
      }
      return node;
}
```

- **Comparison function for searching a list of integers:**

```
int
compare_ints( void const *a, void const *b )
{
      if( *(int *)a == *(int *)b )
            return 0;
      else
            return 1;
}
```

The function would be used like this:

```
desired_node = search_list( root, &desired_value,
      compare_ints );
```

If you wish to search a list of strings, this code will do the job:

```
#include <string.h>
...
desired_node = search_list( root, "desired_value",
      strcmp );
```

Jump Tables:

- **Jump table:** an array of pointers to functions
- **Example with switch statement:**

```
switch( oper ){
case ADD:
      result = add( op1, op2 );
      break;

case SUB:
      result = sub( op1, op2 );
      break;

case MUL:
      result = mul( op1, op2 );
      break;

case DIV:
      result = div( op1, op2 );
      break;

...
```

In order to use a `switch`, the codes that represent the operators must be integers. If they are consecutive integers starting with zero, we can use a *jump table* to accomplish the same thing. A jump table is just an array of pointers to functions.

There are two steps in creating a jump table. First, an array of pointers to functions is declared and initialized. The only trick is to make sure that the prototypes for the functions appear before the array declaration.

```
double      add( double, double );
double      sub( double, double );
double      mul( double, double );
double      div( double, double );
...
double      (*oper_func[])( double, double ) = {
            add, sub, mul, div, ...
};
```

The proper order for the functions' names in the initializer list is determined by the integer codes used to represent each operator in the program. This example assumes that `ADD` is zero, `SUB` is one, `MUL` is two, and so forth.

The second step is to replace the entire `switch` statement with this one!

```
result = oper_func[ oper ]( op1, op2 );
```

`oper` selects the correct pointer from the array, and the function call operator executes it.

Command Line Arguments:

```
/*
** A program to print its command line arguments.
*/
#include <stdio.h>
#include <stdlib.h>

int
main( int argc, char **argv )
{
      /*
      ** Print arguments until a NULL pointer is reached (argc is
      ** not used).  The program name is skipped.
      */
      while( *++argv != NULL )
            printf( "%s\n", *argv );
      return EXIT_SUCCESS;
}
```

- Read *pages 362 - 369* for more information.

String Literals:

- This expression computes the sum of the pointer value plus 1: **`"xyz"+1`**
  - The result is a pointer to the second character in the literal: **`y`**
- When indirection is applied to a pointer, the result is the thing to which it points: **`*"xyz"`**
  - The type of a string literal is "pointer to character", so the result of the indirection is the character to which it points: **`x`**. Not that the result is *not* the entire strings just the first character
- **`"xyz"[2]`**: the value of this expression is the character **`z`**
- Read *pages 369 - 372* for more information.

Chapter 14 - The Preprocessor ✓

Notes:

- Should be a review. Read *pages 384 - 409* for more information.

Chapter 15 - Input/ Output Functions ✓

Error Reporting:

- **`perror:`** function that reports errors in a simple, uniform way
  - **Prototype:** `void perror( char const *message );`
- **`errno:`** defined in `errno.h`; indicates exactly why the operation failed

Terminating Execution:

- **`exit:`** function used to terminate the execution of a program; found in stdlib.h
  - **Prototype:** `void exit( int status );`

Streams:

- **Fully buffered:** "reading" and "writing" copies data out of and into an area in memory called the *buffer*
- **Flushed:** physically written to the device or file only when it becomes full
  - *Ex.* `fflush(stdout);`
- **Binary streams:** written to the file or device exactly as the program wrote them and are delivered to the program exactly as they were read from the file or device

The standard library functions make it very convenient to perform I/O to and from files in C programs. Here is a general overview of file I/O.

1. The program declares a pointer variable of type `FILE *` for each file that must be simultaneously active. This variable will point to the `FILE` structure used by the stream while it is active.
2. The stream is *opened* by calling the **`fopen`** function. To open a stream, you must specify which file or device is to be accessed and how it is to be accessed (for example, reading, writing, or both). `fopen` and the operating system verify that the file or device exists (and, on some operating systems, that you have permission to access it in the manner you specify) and initializes the `FILE` structure.
3. The file is then read and/or written as desired.
4. Finally, the stream is *closed* with the **`fclose`** function. Closing a stream prevents the associated file from being accessed again, guarantees that any data stored in the stream buffer is correctly written to the file, and releases the `FILE` structure so that it can be used again with another file.

Type of Data	Function or Family Name		Description
	Input	Output	
Character	*getchar*	*putchar*	Read (write) a single character
Line	*gets*	*puts*	Unformatted input (output) of a line
	scanf	*printf*	Formatted input (output)
Binary	`fread`	`fwrite`	Read (write) binary data

Table 15.1 Functions to perform character, line, and binary I/O

Family Name	Purpose	To Any Stream	Only `stdin` and `stdout`	String in Memory
getchar	Character input	`fgetc, getc`	`getchar`	❶
putchar	Character output	`fputc, putc`	`putchar`	❶
gets	Line input	`fgets`	`gets`	❷
puts	Line output	`fputs`	`puts`	❷
scanf	Formatted input	`fscanf`	`scanf`	`sscanf`
printf	Formatted output	`fprintf`	`printf`	`sprintf`

❶ Use a subscript or indirection on a pointer to get/put single characters to/from memory,
❷ Use `strcpy` to get/put lines to/from memory.

Opening Streams:

- **`fopen:`** function opens a specific file and associates a stream with the file
  - **Prototype:** `FILE *fopen( char const *name, char const *mode );`
- Modes used most frequently:

	Read	Write	Append
Text	"r"	"w"	"a"
Binary	"rb"	"wb"	"ab"

- Example of **`fopen`**:

```
FILE  *input;

input = fopen( "data3", "r" );
if( input == NULL ){
      perror( "data3" );
      exit( EXIT_FAILURE );
}
```

Closing Streams:

- **`fclose:`** closes streams with this function **--------->**
  - **Prototype:** `int fclose( FILE *f );`

Character I/O:

- ***getchar* family of functions prototypes:**
  - `int fgetc ( FILE *stream );`
  - `int getc ( FILE *stream );`
  - `int getchar ( void );`
- ***putchar* family of functions prototypes:**
  - `int fputc ( int character, FILE *stream );`
  - `int putc ( int character, FILE *stream );`
  - `int putchar ( int character );`
- **Macros:** `getc`, `putc`, `getchar`, and `putchar` are `#define` macros

Undoing Character I/O:

- **`ungetc` prototype:** `int ungetc( int character, FILE *stream );`

```
#include <stdlib.h>
#include <stdio.h>

int
main( int ac, char **av )
{
      int   exit_status = EXIT_SUCCESS;
      FILE  *input;

      /*
      ** While there are more names ...
      */
      while( *++av != NULL ){
            /*
            ** Try opening the file.
            */
            input = fopen( *av, "r" );
            if( input == NULL ){
                  perror( *av );
                  exit_status = EXIT_FAILURE;
                  continue;
            }

            /*
            ** Process the file here ...
            */

            /*
            ** Close the file (don't expect any errors here).
            */
            if( fclose( input ) != 0 ){
                  perror( "fclose" );
                  exit( EXIT_FAILURE );
            }
      }

      return exit_status;
}
```

```
/*
** Convert a series of digits from the standard input to an integer.
*/

#include <stdio.h>
#include <ctype.h>

int
read_int()
{
      int   value;
      int   ch;

      value = 0;

      /*
      ** Convert digits from the standard input; stop when we get a
      ** character that is not a digit.
      */
      while( ( ch = getchar() ) != EOF && isdigit( ch ) ){
            value *= 10;
            value += ch - '0';
      }

      /*
      ** Push back the nondigit so we don't lose it.
      */
      ungetc( ch, stdin );
      return value;
}
```

Unformatted and Formatted Line I/O:

- **Unformatted I/O:** simply reads or writes

```
char *fgets( char *buffer, int buffer_size, FILE *stream );
char *gets( char *buffer );

int fputs( char const *buffer, FILE *stream );
int puts( char const *buffer );
```

- **Formatted I/O:** performs conversions between internal and external representations of numeric and other variables

Scanf Family:

- `int fscanf( FILE *stream, char const *format, … );`
  - **`fscanf`** is the stream given as an argument
- `int scanf( char const *format, … );`
  - **`scanf`** reads from the standard input
- `int sscanf( char const *string, char const *format, … );`
  - **`sscanf`** takes input characters from the character spring given as the 1st argument
- **`scanf` format codes:**

Code	Argument	Meaning
`c`	`char *`	A single character is read and stored. Leading whitespace is *not* skipped. If a width is given, that number of characters are read and stored; *no NUL byte is appended;* the argument must point to a character array that is large enough.
`i` `d`	`int *`	An optionally signed integer is converted. `d` interprets the input as decimal; `i` determines the base of the value by its first characters as is done with integer literal constants.

`u`
`o`
`x`
	`unsigned *`	An optionally signed integer is converted, but is stored as unsigned. The value is interpreted as decimal with `u`, octal with `o`, and hexadecimal with `x`. The code `X` is a synonym for `x`.

`e`
`f`
`g`
	`float *`	A floating-point value is expected. It must look like a floating-point literal constant except that a decimal point is not required. `E` and `G` are synonyms for `e` and `g`.

`s`	`char *`	A sequence of nonwhitespace characters is read. The argument must point to a character array that is large enough. Input stops when whitespace is found; the string is then NUL-terminated.
`[xxx]`	`char *`	A sequence of characters from the given set is read. The argument must point to a character array that is large enough, input stops when the first character that is not in the set is encountered. The string is then NUL-terminated. The code `%[abc]` specifies the set including `a`, `b`, and `c`. Beginning the list with `^` complements the set, so `%[^abc]` means all characters *except* `a`, `b`, and `c`. A right bracket may be included in the list only if it is first it is implementation dependent whether a dash (for example, `%[a-z]`) specifies a range of characters.
`p`	`void *`	The input is expected to be a sequence of characters such as those produced by the `%p` format code of `printf` (see below). The conversion is performed in an implementation-dependent manner, but the result will compare equal to the value that produced the characters when printed as described above.
`n`	`int *`	The number of characters read from the input so far by this call to `scanf` is returned. `%n` conversion are not counted in the value returned by `scanf`. No input is consumed.
`%`	`(none)`	This code matches a single `%` in the input, which is discarded.

	Result when used with qualifier		
Format code	**`h`**	**`l`**	**`L`**
`d, I, n`	`short`	`long`	
`o, u, x`	`unsigned short`	`unsigned long`	
`e, f, g`		`double`	`long double`

```
/*
** Line-oriented input processing with sscanf
*/
#include <stdio.h>
#define     BUFFER_SIZE 100   /* Longest line we'll handle */

void
function( FILE *input )
{
      int   a, b, c, d, e;
      char  buffer[ BUFFER_SIZE ];

      while( fgets( buffer, BUFFER_SIZE, input ) != NULL ){
            if( sscanf( buffer, "%d %d %d %d %d",
                &a, &b, &c, &d, &e ) != 4 ){
                  fprintf( stderr, "Bad input skipped: %s",
                      buffer );
                  continue;
            }

            /*
            ** Process this set of input.
            */
      }
}
```

Program 15.4 Processing line-oriented input with `sscanf`

The printf Family:

- `int fprintf( FILE *stream, char const *format, … );`
  - **`fprintf:`** any output stream can be used
- `int printf( char const *format, … );`
  - **`printf:`** formats the values in its argument list according to the format codes and other characters in the `format` argument
- `int sprintf( char const *buffer, char const *format, … );`
  - **`sprintf:`** writes its results as a NULL-terminated string in the specified `buffer` rather than to a stream

Code	Argument	Meaning
c	int	The argument is truncated to `unsigned char` and printed as a character.
d i	int	The argument is printed as a decimal integer. If a precision is given and the value has fewer digits, zeros are added at the front.
u o x,X	unsigned int	The argument is printed as an unsigned value in decimal (`u`), octal (`o`), or hexadecimal (`x` or `X`). `x` and `X` are identical except that `abcdef` are used for `x` conversions, and `ABCDEF` are used with `X`.
e E	double	The argument is printed in exponent form; for example, `6.023000e23` for the `e` code, and `6.023000E23` for the `E` code. The number of digits behind the decimal point is determined by the precision field; the default is six digits.
f	double	The argument is printed in conventional notation. The precision determines the number of digits behind the decimal point; the default is six.
g G	double	The argument is printed in either `%f` or `%e` (or `%E`, if `G` is given) notation, depending on its value. The `%f` form is used if the exponent is greater than or equal to –4 but less than the precision. Otherwise the exponent form is used.
s	char *	A string is printed.
p	void *	The value of the pointer is converted to an implementation-dependent sequence of printable characters. This code is used primarily in conjunction with the `%p` code in `scanf`.
n	int *	This code is unique in that it does not produce any output. Instead, the number of characters of output produced so far is stored in the corresponding argument.
%	(none)	A single `%` is produced in the output.

Flag	Meaning
-	Left justify the value in its field. The default is to right justify.
0	When right justifying numeric values, the default is to use spaces to fill unused columns to the left of the value. This flag causes zeros to be used instead, and it applies to the `d`, `i`, `u`, `o`, `x`, `X`, `e`, `E`, `f`, `g`, and `G` codes. With the `d`, `i`, `u`, `o`, `x`, and `X` codes, the zero flag is ignored if a precision is given. The zero flag has no effect if the minus flag is also given.
+	When used with a code that formats a signed value, this forces a plus sign to appear when the value is not negative. If the value is negative, a minus sign is shown as usual. By default, plus signs are not shown.
space	Useful only for codes that convert signed values, this flag causes a space to be added to the beginning of the result when the value is not negative. Note that this flag and + are mutually exclusive; if both are given the space flag is ignored.
#	Selects an alternate form of conversion for some codes. These are described in Table 15.8.

Modifier	Used With …	Means the Arguments is …
h	`d, I, u, o, x, X`	a (possibly unsigned) `short` integer
h	`n`	a pointer to a `short` integer
l	`d, I, u, o, x, X`	a (possibly unsigned) `long` integer
l	`n`	a pointer to a `long` integer
L	`e, E, f, g, G`	a `long double`

Used With …	The # Flag …
`o`	guarantees that the value produced begins with a zero.
`x, X`	prefixes a nonzero value with `0x` (`0X` for the `%X` code).
`e, E, f`	ensures the result always contains a decimal point, even if no digits follow it.
`g, G`	does the same as for the `e`, `E`, and `f` codes above; in addition, trailing zeros are not removed from the fraction.

Format Code	Number Converted			
	1	-12	12345	123456789
%d	1	-12	12345	123456789
%6d	¤¤¤¤¤1	¤¤¤-12	¤12345	123456789
%.4d	0001	-0012	12345	123456789
%6.4d	¤¤0001	¤-0012	¤12345	123456789
%-4d	1¤¤¤	-12¤	12345	123456789
%04d	0001	-012	12345	123456789
%+d	+1	-12	+12345	+123456789

Format Code	Number Converted			
	1	.01	.00012345	12345.6789
%f	1.000000	0.010000	0.000123	12345.678900
%10.2f	¤¤¤¤¤¤1.00	¤¤¤¤¤¤0.01	¤¤¤¤¤¤0.00	¤¤12345.68
%e	1.000000e+00	1.000000e-02	1.234500e-04	1.234568e+04
%.4e	1.0000e+00	1.0000e-02	1.2345e-04	1.2346e+04
%g	1	0.01	0.00012345	12345.7

Binary I/O:

- **`fread:`** used to read binary data
  - **Prototype:** `size_t fread( void *buffer, size_t size, size_t count, FILE *stream );`
- **`fwrite:`** used to write binary data
  - **Prototype:** `size_t fwrite( void *buffer, size_t size, size_t count, FILE *stream );`

Flushing and Seeking Functions:

- **`fflush:`** forces the buffer for oan output stream to be physically written even if it is not yet full
  - **Prototype:** `int fflush( FILE *stream );`
- **`ftell:`** returns the current position in the stream, which is the offset from the beginning of the file at which the next read or write would begin
  - **Prototype:** `long ftell( FILE *stream );`
- **`fseek:`** allows you to seek on a stream; this operations changes the position at which the next read or write will occur
  - **Prototype:** `int fseek( FILE *stream, long offset, int from );`

If `from` is...	Then you will seek to...
SEEK_SET	`offset` bytes from the beginning of the stream; `offset` must be non-negative.
SEEK_CUR	`offset` bytes from the current location in the stream; `offset` may be positive or negative.
SEEK_END	`offset` bytes from the end of the file; `offset` may be positive or negative, positive values seek beyond the end of the file.

Changing the Buffering:

- **`setbuf:`** installs an alternate array to be used for buffering the stream; the array must be **`BUFSIZ`** (which is defined in `stdio.h`) characters long
  - **Prototype:** `void setbuf( FILE *stream, char *buf );`
- **`int setvbuf:`** indicates what type of buffering is desired
    - **`_IOFBF`** indicates a fully buffer stream
    - **`_IONBF`** indicates an unbuffered stream
    - **`_IOLBF`** indicates a line buffer stream
  - **Prototype:** `int setvbuf( FILE *stream, char *buf, int mode, size_t size );`

Temporary Files:

- **`tmpfile:`** use a file to hold data temporarily; automatically removed when the file is closed or the program terminates
  - **Prototype:** `FILE *tmpfile( void );`
- **`tmpnam:`** temporary file that can be opened with a different mode, or created by 1 program and read by another
  - **Prototype:** `char *tmpnam( char *name );`

File Manipulation:

- **`remove`:** deletes specified file; returns zero if it succeeds and non zero if it fails
    - **Prototype:** `int remove( char const *filename );`
- **`rename`:** changes specified file name; returns zero if it succeeds and non zero if it fails
        - If a file already exists with the new name, the behavior implementation is dependent
        - If this function fails, the file will still be accessible with its original name
    - **Prototype:** `int rename( char const *oldname, char const *newname );`

Chapter 16 - The Standard Library ✓

Arithmetic <stdlib.h>

- **Absolute value:** `int abs( int value );`
- **Absolute value for long integer values:** `long int labs( long int value );`
- **Division:** `div_t div( int numerator, int denominator );`
- **Division for long values:** `ldiv_t ldiv( long int numer, long int denom);`

Random Numbers <stdlib.h>

- **Random number:** `int rand( void );`
- **Random number seed generator:** `void srand( unsigned int seed );`

String Conversion <stdlib.h>

- **`atoi`:** converts characters to integer values
    - **Prototype:** `int atoi( char const *string);`
- **`atol`:** converts characters to long integer values
    - **Prototype:** `long int atol( char const *string);`
- **`strtol`:** converts the argument string to a long in the same manner as `atol`
        - Returns LONG_MIN if the value is too large and negative; returns LONG_MAX if the value is too large and positive
    - **Prototype:** `long int strtol( char const *string, char **unused , int base);`
- **`stroul`:** behaves in the same manner but produces an unsigned long instead
        - Returns **`ULONG_MAX`** if the value is too large
    - **Prototype:** `unsigned long int strtoul( char const *string, char **unused , int base);`

Trigonometry <math.h>

```
double sin( double angle );
double cos( double angle );
double tan( double angle );
double asin( double angle );
double acos( double angle );
double atan( double angle );
double atan2( double angle );
```

Hyperbolic <math.h>

```
double sinh( double angle );
double cosh( double angle );
double tanh( double angle );
```

Logarithm and Exponent <math.h>

```
double exp( double x);
double log( double x);
double log10( double x);
```

Floating-Point Representation <math.h>

```
double frexp( double value, int *exponent);
double ldexp( double value, int exponent);
double modf( double value, int *ipart);
```

Power <math.h>

```
double pow( double x, double y);
double sqrt( double x);
```

Floor, Ceiling, Absolute Value, and Remainder <math.h>

```
double floor( double x );
double ceil( double x );
double fabs( double x );
double fmod( double x, double y );
```

String Conversion <stdlib.h>

```
double atof( char const *string );
double strtod( char const *string, char **unused );
```

Processor Time <time.h>

- Returns amount of time since program began executing: `clock_t clock( void );`
- Returns time of day: `time_t time( time_t *returned_value );`

Date and Time Conversions <time.h>

Read *pages 463 - 466* for more information:

- `char *ctime( time_t const *time_value );`
- `double difftime( time_t time1, time_t time2 );`
- `asctime( localtime( time_value ) );`
- `struct tm *gmtime( time_ const *time_value );`
- `struct tm *localtime( time_t const *time_value );`
- `char *asctime( struct tm const *tm_ptr );`
- `size_t strftime( char *string, size_t maxsize, char const *format, struct tm const *tm_ptr );`
- `time_t mktime( struct tm const *tm_ptr );`

Code	is Replaced By
%%	a %
%a	the day of week, using the locale's abbreviated weekday names
%A	the day of week, using the locale's full weekday names
%b	the month, using the locale's abbreviated month names
%B	the month, using the locale's full month names
%c	the date and time, using %x %X
%d	the day of month (01-31)
%H	the hour, in 24-hour clock format (00-23)
%I	the hour, in 12-hour clock format (01-12)
%j	the day number of the year (001-366)
%m	the month number (01-12)
%M	the minute (00-59)
%p	the locale's equivalent of AM or PM, whichever is appropriate
%S	the seconds (00-61)
%U	the week number of the year (00-53), starting with Sunday
%w	the day of the week; Sunday is day 0
%W	the week number of the year (00-53), starting with Monday
%x	the date, using the locale's date format
%X	the time, using the locale's time format
%y	the year within the century (00-99)
%Y	the year, including century (for example, 1984)
%Z	the time zone abbreviation

Type & name	Range	Meaning
int tm_set;	0 - 61	Seconds after the minute†
int tm_min;	0 - 59	Minutes after the hour
int tm_hour;	0 - 23	Hours after midnight
int tm_mday;	0 - 31	Day of the month
int tm_mon;	0 - 11	Months after January
int tm_year;	0 - ??	Years after 1900
int tm_wday;	0 - 6	Days after Sunday
int tm_yday;	0 - 365	Days after January 1
int tm_isdst;		Daylight Savings Time flag

Nonlocal Jumps <setjmp.h>

```
int setjmp( jmp_buf state );
void longjmp( jump_buf state, int value );
```

Signal Names <signal.h>

Read *pages 470 - 473* for more information:

Signal	Meaning
SIGABRT	The program has requested abnormal termination.
SIGFPE	An arithmetic error occurred.
SIGILL	An illegal instruction was detected.
SIGSEGV	An invalid access to memory was detected.
SIGINT	An interactive attention signal was received.
SIGTERM	A request to terminate the program was received.

Printing Variable Argument Lists <stdarg.h>

```
int vprintf( char const *format, va_list arg );
int vfprintf( FILE *stream, char const *format, va_list arg );
int vsprintf( char *buffer, char const *format, va_list arg );
```

Terminating Execution <stdlib.h>

```
void abort( void );
void atexit( void (func)( void ) );
void exit( int status );
```

The Environment <stdlib.h>

```
char *getenv( char const *name );
```

Executing System Commands <stdlib.h>

```
void system( char const *command );
```

Sorting and Searching <stdlib.h>

```
void qsort( void *base, size_t n_elements, size_t el_size, int
(*compare) (void const *, void const *) );
```

Locales:

```
char *setlocale( int category, char const *locale );
```

Numeric and Monetary Formatting <locale.h>

```
struct lconv *localeconv( void );
```

Value	Changes
LC_ALL	The entire locale.
LC_COLLATE	The collating sequence, which affects the behavior of the strcoll and strxfrm functions (see below).
LC_CTYPE	The character type classifications used by the functions defined in ctype.h.
LC_MONETARY	The characters to be used when formatting monetary values.
LC_NUMERIC	The characters to be used when formatting nonmonetary values. Also changes the decimal point character used by the formatted input/output functions and string conversion functions.
LC_TIME	The behavior of the strftime function.

Table 16.5 setlocale categories

Field and Type	Meaning
char *decimal_point:	The character to use as a decimal point. This value will never be an empty string.
char *thousands_sep	The character used to separate groups of digits that appear to the left of the decimal point.
char *grouping	Specifies how many digits are in each digit group to the left of the decimal point.

Monetary Formatting:

Field and Type	Meaning
char *currency_symbol	The local currency symbol.
char *mon_decimal_point	The decimal point character.
char *mon_thousands_sep	The character used to separate digit groups that appear to the left of the decimal point.
char *mon_grouping	Specifies the number of digits in each group appearing to the left of the decimal point
char *positive_sign	The string used to indicate a nonnegative amount.
char *negative_sign	The string used to indicate a negative amount.
char frac_digits	The number of digits appearing to the right of the decimal point.
char p_cs_precedes	1 if the currency_symbol precedes a nonnegative value; 0 If it follows.
char n_cs_precedes	1 if the currency_symbol precedes a negative value; 0 if it follows.
char p_sep_by_space	1 if the currency_symbol is separated by a space from a nonnegative value; else 0.
char n_sep_by_space	1 if the currency_symbol is separated by a space from a negative value; else 0.
char p_sign_posn	Indicates where the positive_sign appears in a nonnegative value. The following values are allowed: 0 Parentheses surround the currency symbol and value. 1 The sign precedes the currency symbol and value. 2 The sign follows the currency symbol and value. 3 The sign Immediately precedes the currency symbol. 4 The sign immediately follows the currency symbol.
char n_sign_posn	Indicates where the negative_sign appears in a negative value. The values used for p_sign_posn are also used here.

Table 16.7 Parameters for formatting local monetary values

Strings and Locale <string.h>

```
int strcoll( char const *s1, char const *s2 );
size_t strxfrm( char *s1, char const *s2, size_t size );
```

Chapter 17 - The Standard Library

Pdf pg 494

***Starred topics are not completed.** Always read the textbook for more information.

Debugger Notes - tldr

Commands:

Command	Description	Examples
`-g`	Compiler	`gcc -g program.c -o program.x`
`gdb`	Starts debugging mode	`gdb program.x`
`start`	Jumps to main function	
`next` or `n`	Tells you the line that is about to run	
`quit`	Type to exit	`quit`
`step` or `s`	Goes into next function call	
`print` or `p`	Prints the variable value	
`display`	Always show the variable value on every step	`display_variable1`
`info locals`	Shows all the current local variables	
`info args`	Shows function argument values	

Base Conversion Tips:

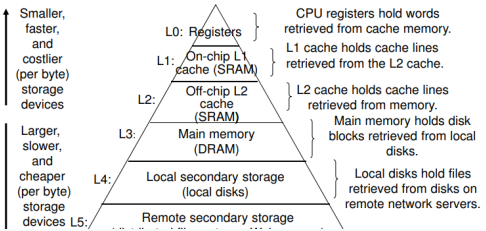
- Text

Lecture Summaries - tldr

Lecture 1: *Chp. 1* ✓

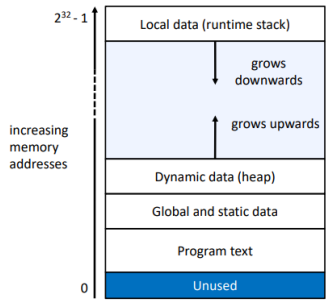
- **Bit:** single binary digit (1 or 0)
- **Byte:** 8 bits; basically an aggregate of bits
- **Word:** multiple bytes
- **Machine words/ language:** the only language a computer understands is in binary
- **Assembly language:** human-readable form of machine language
- ***Compiler:** converts high-level languages to machine language; every CPU has its own form of machine language
- ***Assembler:** converts high-level programs to machine language so it can be run

The memory hierarchy



- **Hardware organization:**
  - **Machine registers:** storage locations that are internal to the CPU; faster access than memory is
  - **Main memory:** long, linear list of locations, each with individual memory location addresses
  - ***Mass storage (hard drives, SSDs):**
  - ***Remote storage:**
- **Operating system:** manages computer resources; protects the computer from misuse; provides an abstraction and mechanisms so that programs can safely manipulate hardware
- **Processes:** running program (1 or more threads of control), along with all the data associated with it *(an address space)*
  - **Context switching:** operating system uses this to give the appearance of multiple processes executing at once on a single processor

UNIX virtual address space (simplified)



- **C vs. Java:** no classes or objects, notion of current object, inheritance (or interfaces), overloading, overriding, reference/ object dichotomy, garbage collection, exceptions, or library collections (no data structions in the library like the Java Collections Framework)
  - C is typically fully compiled (to machine code), not to bytecode
  - It facilitates direct manipulation of memory
  - Only uses `/*  */` for comments in **C90**
  - Methods don't exist in C. They are functions.
  - All variable declarations in **C90** need to come before statements
- **The C preprocessor:**
  - **Source files:** contain code and commonly some related definitions
  - **Header files:** contain only definitions
  - ***Preprocessing*** occurs before compilation in C
- Also explains `printf()` and `scanf()`

Preprocessor directives
- **`#include`**
- Two forms:
  `#include <file1.h>`
  vs.
  `#include "file2.h"`

Lecture 2: *Chp 1, 2, 3* ✓

- **`feof(stdin)`**: reading until the end of input
  - **`feof(stdin)`** is true if the previous input operation could not read because the last data in the input had already been read. Otherwise it is NOT true.
  - *See examples on Grace:* **~/216public/examples/lecture02**

```
scanf("%d", &n);  /* read a number */
while (!feof(stdin)) {
  count++;
  scanf("%d", &n);  /* read a number */
}
```

- **`EOF`**: another way to read until the end of input

- The `scanf()` function returns a special value `EOF` when trying to read after the last data element has already been read (EOF = end of file)

```
while (scanf("%d", &n) != EOF)  /* read a number and check for eof
*/
   count++;

printf("%d integers were read.\n", count);
```

- **Compilation stages in C:**
  - **Preprocessor:** See "*The C preprocessor*" in lecture 1
  - **Translation (actuall compilation):**
    - Produces object code (object file)
  - **Linking:** connects everything (source/ header files, object code, etc.) before executing the program
    - **gcc**'s **-c** option suppresses automatic linking
      - *Ex.* **gcc file1.c file2.c - program.x**
        - This compiles file1 and file2, then links them to create a file executable called program
      - *Ex.* **gcc -c file1.c**
        gcc -c file2.c
        gcc file1.o file2.o -o program.x
        - This compiles file1 and file 2, and it stops before linking. It will create **file1.o** and **file2.o**. Then the last piece of code will link those object code files to create a file executable called program
- **Basic types:**
  - **Scalar Types:** integer types, floating-point types, and pointer types
    - Integer types include characters
    - Integer types include *signed and unsigned types*
      - **Unsigned**: cannot store a negative value
      - **Signed:** can have both positive and negative values
  - **Aggregate types:** arrays and structures
  - **Integer type sizes:** char,short, int, long

Integer type	**Minimum size**	**Size on Grace**
`char`	1 bytes	1 bytes
`short`	2 bytes	2 bytes
`int`	2 bytes	4 bytes
`long`	4 bytes	8 bytes

  - **Floating type sizes:** float, double, long double

Floating type	**Minimum size**	**Size on Grace**
`float`	4 bytes	4 bytes
`double`	8 bytes	8 bytes