Data Structures: Organization of data stored within a program.

Algorithms: Procedures for solving problems. (all C programs are essentially a collection of algorithms)

* Choosing the right data structure or algorithm will affect how efficient you are as a programmer and how efficient your program is during execution.

Efficiency: Data structures help to organize data in a way that makes algorithms more efficient.

Efficient algorithms help to solve common problems.

Abstraction: Data structures provide different ways of abstracting, managing, and storing data. Allows programs to choose how to understand and a represent data.

Abstraction helps to solve large problems by breaking the problem down into smaller problems for which efficient algorithms can be developed.

Reusability: Data structures can be effeciently reused as they are typically independent of the overall problem being solved.

Algorithms are often reusable because many common problems appear across programs.

Software Engineering

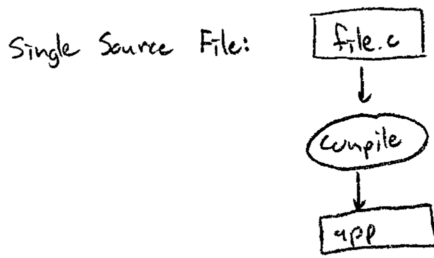
Modularity: Defining modules that provide an interface for users to access that module, where the implementation details can be hidden. (i.e. black box)

Readability: Document programs so that other developers can follow its logic simply by reading the code and comments.

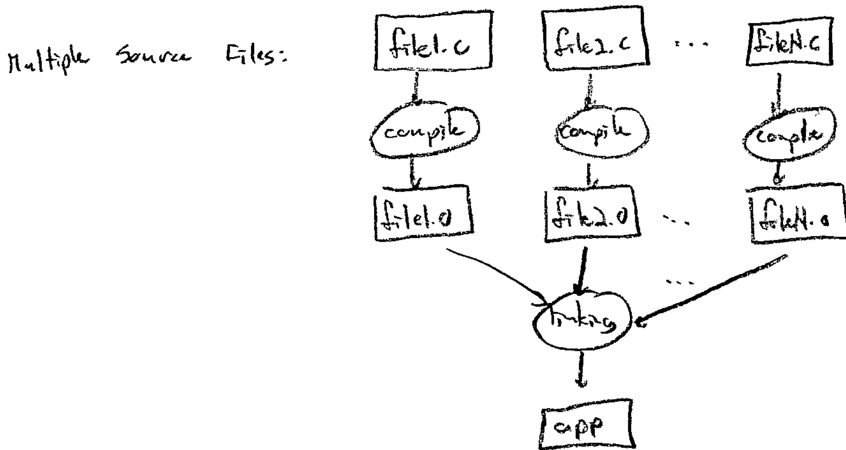
Simplicity: Intelligent solutions are often the simplest (but they may be hard to find)

Consistency: Consistency in style, conventions, software design, etc. help to improve readability and simplicity.

C Compilation and Linking



Single file can be directly compiled to application executable
(Note: linking still happens for C standard libraries)



- Related functions typically organized into C files for modularity, reusability, and project management.

- Separate compilation and linking steps are needed to build application

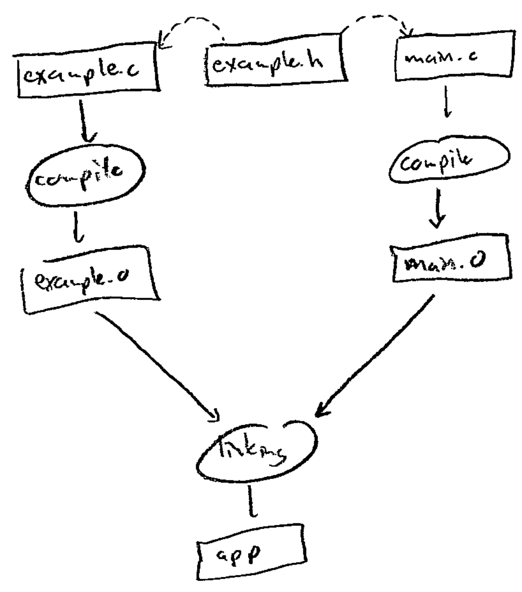
- Compiling: Each source file is compiled separately. During compilation the include header files describe what external functions and variables exist. The compiler will generate an <u>object file</u> that has placeholders for external functions/variables

- Linking: Linker combines objects files and library code (e.g. standard libraries) into final application. Linker resolves external references determined during compilation.

- Header Files: In order for a function defined in one source file to be used by another file, its existence can be defined in a header file

  + Typically defines function prototypes and variables that will <u>potentially</u> be used in other files.

```
example.h:   int funcA(int input);

example.c:   #include "example.h"
             int
             funcA(int input) {

                 return input + 5;
             }

main.c:      #include "example.h"

             int main() {
                 int a = 10;
                 int b;
                 b = funcA(a);

                 return 0;
             }
```

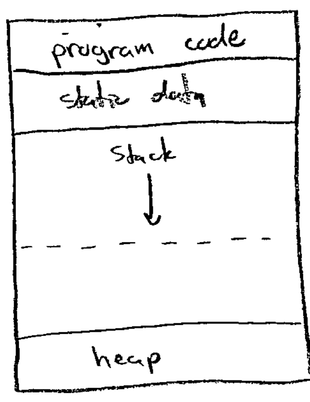


compilation: object file has a placeholder for calling funcA, but implementation of funcA is not known at compilation phase

Linking: linker finds reference to funcA in main.o and finds implementation of funcA in example.o.

Note:
When compiling large programs, two stage compile-link build process allows for faster build times. Only those object files whose source files have changed, need to be recompiled.

C Program Memory Organization



Program Code: Compiled instructions for executing program

Static Data: Data that persists throughout program execution (i.e. global variables)

Stack: Temporary memory used for function calls
Note: Data does **NOT** persist after returning from function

Heap: Dynamically allocated memory (e.g. memory allocated by calls to malloc())

* All data (and code) within a C program is stored within memory at a specific memory location (i.e. **memory address**)

C Function Calls

- The program stack is used to keep track of information during function calls.
- This information is referred to as the activation record. (or stack frame)

Activation Record: include storage for input parameters, return value, temporary storage, saved state information (i.e. information used to manage stack and return to location function was called from)

```
#include <stdio.h>

int main() {

    int val = 10;
    int mod;


    mod = FuncE(val);
(3) printf("%i %i\n", val, mod);

    return 0;
}
```

```
int FuncE(int input) {
(1)
    if (input <=10) input += 5;

(2) return input-1;
}
```

Stack at (1):

val mod	10 ??	main's activation record
input return value	10 ??	FuncE's activation record
temp. storage return location	(3)	

Stack at (2):

val mod	10 ??
input return value	15 14
temp. storage return location	(3)

Stack at (3):

val	10
mod	14

Pointers: A C pointer stores the address at which data is located.
↳ e.g. although there are some exceptions, a pointer is simply a memory address

Need methods to: ① operate on pointers themselves
② operate on the data pointed to by the pointer

* what if pointer doesn't point to anything?

```
int a;
int *iptr;
int **iptr_ptr;
```

iptr = &a; ⇒ & returns the address at which the variable a is located, stored in pointer

*iptr = 10; ⇒ * access the item pointed to by the pointer (in this case, 10 is written to the memory location pointed to by the pointer. Thus a is now 10)

iptr_ptr = &iptr; ⇒ pointers are also variables stored in memory, so they also have an address

Array Accesses using Pointers (Pointer Arithmetic):

Pointers are commonly used to access arrays. C supports pointer arithmetic to make doing so easy (but it can be confusing).

- a pointer can be directly assigned to an array. Same as assigning pointer to address of the first element
- incrementing (or decrementing) a pointer will increment (or decrement) the memory location stored to access the next (or previous) location based on the type of data being accessed.

Example:

```
int   test_array[100];
int   sum;
int   *iptr;
     ...
for(iptr=test_array; iptr < &test_array[100]; iptr++) {
      sum +=  *iptr;
}
```

Note: there is an error in the above example:

The following are equivalent:

① `iptr = test_array;`

② `iptr = &test_array[0];`

① `test_array[0+5] = 42;`

② `iptr = test_array;`
`*(iptr+5) = 42;`

Dynamic Memory Allocation:

- we often do not know how much memory we will need before executing our program.
- Can dynamically allocate memory as needed using malloc, realloc, calloc, and free

```
void * malloc (size_t size);
void * calloc (size_t size);
void * realloc (void* ptr, size_t size);
void free (void *ptr);
```

- malloc() allocates a block of memory of size bytes in the program heap and returns a pointer to the first memory location if successful.
  Returns **NULL** if memory cannot be allocated
  - ↳ malloc doesn't know what you are allocating (int, char array, etc.) so it returns a void *. Must be cast to proper pointer type
  - ↳ always check the return value.

Typical method of dynamic memory allocation:

```
int *iptr;

iptr = malloc (sizeof (int));
if (iptr == NULL) {
    printf ("could not allocate memory\n");
    exit(-1);
}
```

Pointers to pointers (common in function parameters):

```
int g (int **iptr) {
    *iptr = (int *) malloc (sizeof (int) * 5);
    if ( *iptr == NULL) return -1;
    return 5;
}
```

```
int *new_array;
int num-elements;

num-elements = g(&new-array);
if (num-elements >= 0) {
    int i=0;
    int aptr = new-array;
    while (i < num_elements) {
        *aptr = 0;
        aptr ++;
        i++;
    }
}
```