COMP1511: Multi file (module) C Programs

Session 2, 2018

Single vs Multiple files

- A large C program in a **single file** is **not very convenient** to browse and edit, may need lots of scrolling! and result in unintentional edits (by mistake).
- Importantly, difficult for multiple programmers to work on a single file at the same time! Not suitable for real word problems.
- **Larger programs** are normally divided into smaller subtasks, across **multiple files**.
- Each file implements say a subtask or a partial solution.
- We can compile each file (module) separately or together.
- Importantly, modules improve **reusability** of code. For example, `list.c` and `list.h` can be used by many *client* C programs to create and manipulate lists.

Multiple C files

- We can write C programs that use functions defined in **multiple `.c` files**
- Allows developers to divide a large programming task into smaller subtasks, with say one `.c` file (and corresponding `.h` file) for each subtask.

  For example, `list.c` and `list.h` to handle list operations.

test_list.h

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

#include  "myList.h"

int main(int argc, char *argv[]) {

    struct node *head = create_node(67, NULL);

    head = create_node(33, head);
    head = create_node(77, head);
    head = create_node(25, head);
    head = create_node(10, head);

    print_list(head);
    printf("\n");

    int fails = noFail(head, 50);
    printf("noFail: %d \n", fails);

    struct node *p = findR(head, 60);
    if(p != NULL) {
        printf("p->data is: %d \n", p->data);
    }
    else {
        printf("Value not found! \n");
    }

    return 0;
}
```

Multiple C files

- Allows **separation** between how to ***use*** a function (in .h file), and its ***implementation*** (in .c file).

- `.h` file, also known as a ***header file***, contains function prototypes/signatures offering information on how to use a function. The corresponding `.c` file implements the required function.

Example

myList.h

```c
#ifndef MYLIST_H
#define MYLIST_H

struct node {
    struct node *next;
    int         data;
};


struct node *create_node(int data, struct node *next);
void print_list(struct node *head);
struct node *find_node(struct node *head, int data);
int findMax(struct node *head);
struct node *findR(struct node *head, int val);
int noOdd(struct node *head);

/**
Calculates no of failed students ...
**/
int noFail(struct node* head, int passMark);

#endif
```

myList.c

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

#include  "myList.h"


// Create a new struct node containing the specified data,
// and next fields, return a pointer to the new struct node.
struct node *create_node(int data, struct node *next) {
    struct node *n = malloc(sizeof (struct node));
    if (n == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
    n->data = data;
    n->next = next;
    return n;
}



// print contents of list in Python syntax
void print_list(struct node *head) {
    printf("[");
    for (struct node *n = head; n != NULL; n = n->next) {
        printf("%d", n->data);
        if (n->next != NULL) {
            printf(", ");
        }
    }
    printf("]");
}
```

Example: #include

- To call a function we need to know its *signature (prototype)*, that includes function *name*, *return type* and *input parameters*.
- `.h` file contains function *signature (prototype)*. We call this the API - Application Programmers Interface
- For example, we include **`myList.h`** file in a “client” program **`test_list.c`** before using functions implemented in **`myList.c`**

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

#include  "myList.h"

int main(int argc, char *argv[]) {

    struct node *head = create_node(67, NULL);

    head = create_node(33, head);
    head = create_node(77, head);
    head = create_node(25, head);
    head = create_node(10, head);

    print_list(head);
    printf("\n");

    int fails = noFail(head, 50);
    printf("noFail: %d \n", fails);

    struct node *p = findR(head, 60);
    if(p != NULL) {
        printf("p->data is: %d \n", p->data);
    }
    else {
        printf("Value not found! \n");
    }

    return 0;
}
```

Another Example: multi file (module) C program

main.c

```
#include <stdio.h>
#include "world.h"
#include "graphics.h"

int main(void)
{
   ...
   drawPlayer(p);
   spin(...);
}
```

world.h

```
typedef ... Ob;
typedef ... Pl;

extern addObject(Ob);
extern remObject(Ob);
extern movePlayer(Pl);
```

world.c

```
#include <stdlib.h>

addObject(...)
{ ... }

remObject(...)
{ ... }

movePlayer(...)
{ ... }
```

graphics.h

```
extern drawObject(Ob);
extern drawPlayer(Pl);
extern spin(...);
```

graphics.c

```
#include <stdio.h>
#include "world.h"

drawObject(Ob o);
{ ... }

drawPlayer(Pl p)
{ ... }

spin(...)
{ ... }
```

Avoid Multiple Inclusion

- If multiple modules include the same header file, the variables/functions in it will be included/declared twice.
- We can use C preprocessor to introduce conditional compilation to address this problem.

- In the following example, the first time "**MYLIST_H**" is not defined, so the contents of the **myList.h** are included, and during this we also define "**MYLIST_H**".
- For the following include statements, "**MYLIST_H**" is already defined, so **"#ifndef MYLIST_H"** will be false, and nothing will be included.

myList.h

```
#ifndef MYLIST_H
#define MYLIST_H

struct node {
    struct node *next;
    int          data;
};


struct node *create_node(int data, st
void print_list(struct node *head);
struct node *find_node(struct node *h
int findMax(struct node *head);
struct node *findR(struct node *head,
int noOdd(struct node *head);

/**
Calculates no of failed students ...
**/
int noFail(struct node* head, int pas

#endif
```

Compilers

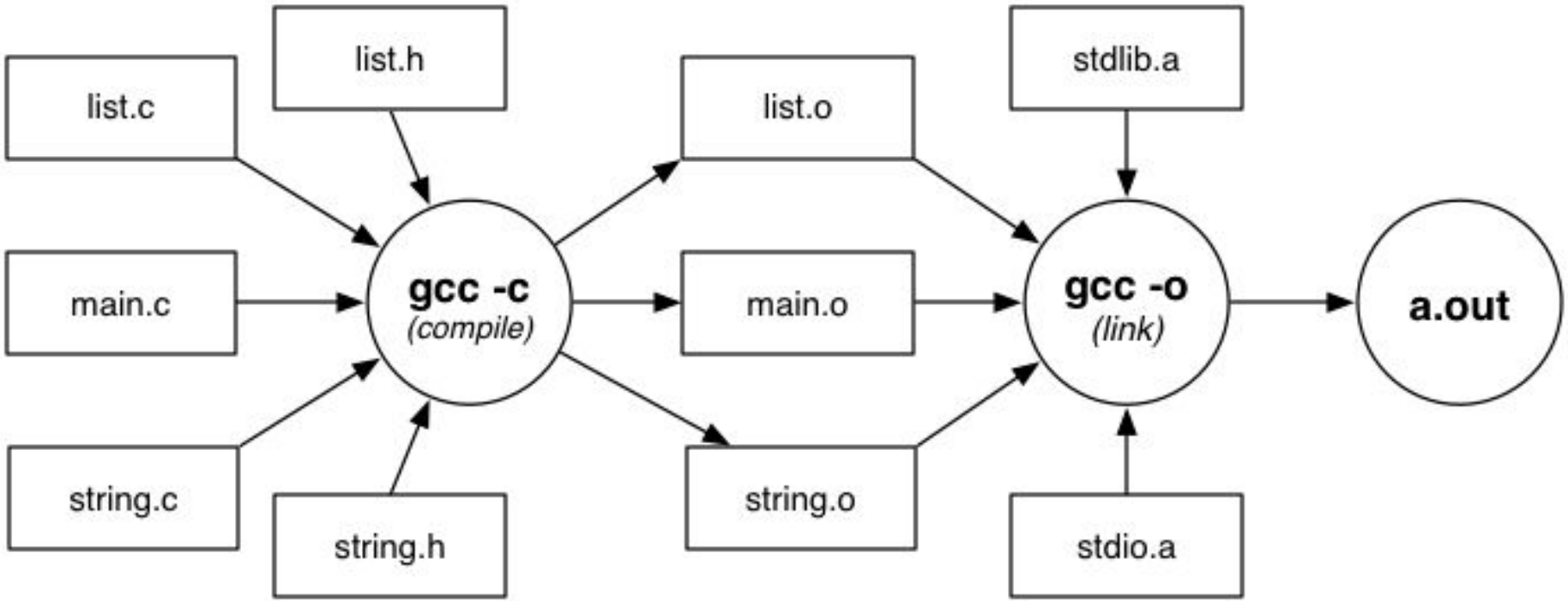
Compilers are programs that

- convert program source code to executable form
- "executable" might be machine code or bytecode

The Gnu C compiler (**gcc**)

- applies source-to-source transformation (**pre-processor**)
- **compiles** *source code* to produce *object files* (***.o files***)
- **links** object files (.o files) and *libraries* to produce ***executables***

Compilers


list.c
list.h
main.c
string.c
string.h
gcc -c
(compile)
list.o
main.o
string.o
stdlib.a
stdio.a
gcc -o
(link)
a.out

gcc/dcc multi-purpose tools

gcc / dcc is a multi-purpose tool

- compiles (**-c**), links, makes executables (**-o**)

```
dcc -c     myList.c
produces myList.o, from myList.c

dcc -c     test_list.c
produces test_list.o, from test_list.c and myList.h

dcc -o     test_list     myList.o     test_list.o
links myList.o, test_list.o and libraries
producing executable program called test_list
```