JTSK-320112

Programming in C II

C-Lab II

Lecture 3 & 4

Dr. Kinga Lipskoch

Spring 2019

Planned Syllabus

- The C Preprocessor
- Bit Operations
- Pointers and Arrays (Dynamically Allocated Multi-Dimensional Arrays)
- **Pointers and Structures (Linked Lists)**
- **Compiling, Linking and the `make` Utility**
- **Pointers and Functions (Function Pointers)**
- Stacks and Queues
- Modifiers and Other Keywords
- Binary I/O (File Handling)

Structures

- A structure (i.e., `struct`) is a collection of variables
  - Variables in a structure can be of different types
- The programmer can define its own structures
- Once defined, a structure is like a basic data type, you can define
  - Arrays of structures,
  - Pointers to structures,
  - ...

Example: Points in the Plane

- A point is an object with two coordinates (= two properties)
  - Each one is a `double` value
- Problem: Given two points, find the point lying in the middle of the connecting segment
  - It would be useful to have a point data type
  - C does not provide such a type, but it can be defined

Defining the point struct

- The keyword struct can be used to define a structure

```
struct point {
  double x;
  double y;
};
```

- A point is an object with two doubles, called x and y of type double

Defining point Variables

- To declare a point (i.e., a variable of data type `point`), the usual syntax is used: type followed by variable name
  `struct point a, b;`
- `a` and `b` are two variables of type `struct point`

Accessing the Components of a `struct`

To access (read / write) the components (i.e., fields) of a structure, the selection operator . is used

```c
struct point a;
a.x = 34.5;
a.y = 0.45;
a.x = a.y;
```

struct Initialization

- Like in the case of arrays, a structure can be initialized by providing a list of initializers
  ```
  struct point a = { 3.0, 4.0 };
  ```
- Initializations can use explicit field names to improve readability and code robustness (e.g., if `struct` definitions are modified)
  ```
  struct point a = { .x = 3.0, .y = 4.0 };
  ```
- As for arrays, it would be an error to provide more initializers than members available
- Initializers' types must match the types of the fields

struct Assignment

- The assignment operator (=) can be used also with structures

```
struct point a, b;
a.x = a.y = 0.2345;
b = a;
```

- The copying is performed field by field (keep this in mind when your structures have pointers)
- Warning: the relational operators (including equality test) are not defined for structures

Structures and Functions

A function can have parameters of type structure and can return results of type structure

```
struct point middle(struct point a,
    struct point b) {
  struct point retp;
  retp.x = (a.x + b.x ) / 2;
  retp.y = (a.y + b.y ) / 2;
  return retp;
}
```

Arrays of Structures

- It is possible to define arrays of structures
- The selection operator must then be applied to the elements in the array (as every element is a structure)

```
struct point list[4];
list[0].x = 3.0;
list[0].y = 7.3;
```

Pointers to Structures

- Structures reside in memory, thus it is possible to get their address
- Everything valid for the basic data types still holds for pointers to structures

```
struct point p;
struct point *pointpointer;
pointpointer = &p;
```

The Arrow Operator

- A structure can be modified by using a pointer to it and the dereference operator
  `(*pointpointer).x = 45;`
  - Parenthesis are needed to adjust the precedence of the operators `*` and `.`
- The arrow operator achieves the same goal giving the same result
  `pointpointer->x = 45;`

Dynamic Structures

- Pointers to structures can be used to allocate dynamically sized arrays of structures

```c
struct point *ptr;
int number;
scanf("%d\n", &number);
ptr = (struct point *)malloc(sizeof(
  struct point) * number);
```

- You can access the array as we have already seen

```c
ptr[0] = { 0.9, 9.87 };
ptr[1].x = 7.45;
ptr[1].y = 57.3;
```

Pointers and Structures: Self-referential Structures

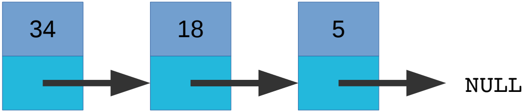
- Is it possible for a structure A to have a field of type A? No
- Is it possible for a structure A to have a field which is a pointer to A? Yes
  - This is called self reference
  - You will encounter many data structures organized by mean of self references
- Trees, Lists, ...

An Example: Lists

- A list is a data structure in which objects are arranged in a linear order
- The order in a list is determined by a pointer to the next element
  - While a vector has indices
- Advantages: lists can grow and shrink
- Disadvantages: access is not efficient

Linked Lists

- It is a standard way to represent lists
- A list of integers: every element holds an `int` plus a pointer to the next one
  - Recursive definition
- The last element's pointer points to `NULL`



Linked Lists in C

- Every element (node) holds two different information
  - The value (integer, float, double, char, array, ...)
  - Pointer to the next element
- This "calls" for a structure

```
struct list {
  int info;
  struct list *next;  /* self reference */
};
```

Building the Linked List

```
struct list a, b, c;
struct list *my_list;
my_list = &a;
a.info = 34;
a.next = &b;
b.info = 18;
b.next = &c;
c.info = 5;
c.next = NULL;    /* defined in stdlib.h */
```

- `NULL` is a constant indicating that the pointer is not holding a valid address
- In self-referential structures it is used to indicate the end of the data structure

Printing the Elements of a Linked List

```c
void print_list(struct list* my_list) {
  struct list *p;
  for(p = my_list; p; p = p->next) {
    printf("%d\n", p->info);
  }
}
/* Using a while loop
void print_list(struct list* my_list) {
  while (my_list != NULL) {
    printf("%d\n", my_list->info);
    my_list = my_list->next;
  }
}*/
```

To print all the elements of a list, `print_list` should be called with the address of the first element in the list

Dynamic Growing and Shrinking

- Elements added and deleted to lists are usually allocated dynamically using the `malloc` and `free` functions
  - The example we have seen before is not the usual case (we assumed the list has content)
- Initially the list is set to empty (i.e., it is just a `NULL` pointer)

```
struct list *my_list = NULL;
```

Inserting an Element in a Linked List (1)

```c
/* Inserts a new int at the beginning of the list
   my_list list where element should be inserted
   value integer to be inserted
   Returns the updated list
*/

struct list* push_front(struct list *my_list, int value) {
  struct list *newel;
  newel = (struct list *) malloc(sizeof(struct list));
  if (newel == NULL) {
    printf("Error allocating memory\n");
    return my_list;
  }
  newel->info = value;
  newel->next = my_list;
  return newel;
}
```

Inserting an Element in a Linked List (2)

```
/* Like the previous one, but inserts at the end */

struct list* push_back(struct list* my_list, int value) {
  struct list *cursor, *newel;
  cursor = my_list;
  newel = (struct list *) malloc(sizeof(struct list));
  if (newel == NULL) {
    printf("Error allocating memory\n");
    return my_list;
  }
  newel->info = value;
  newel->next = NULL;
  if (my_list == NULL)
    return newel;
  while (cursor->next != NULL)
    cursor = cursor->next;
  cursor->next = newel;
  return my_list;
}
```

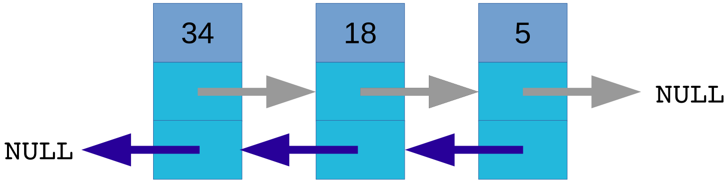
Freeing a Linked List

```c
/*
  Disposes a previously allocated list
*/

void dispose_list(struct list* my_list) {
  struct list *nextelem;
  while (my_list != NULL) {
    nextelem = my_list->next;
    free(my_list);
    my_list = nextelem;
  }
}
```

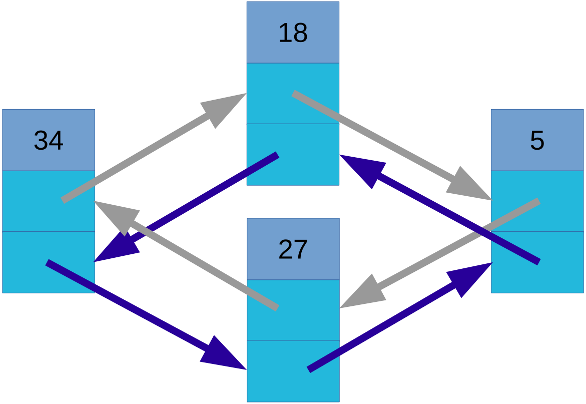
Using a Linked Lists

```c
/*
  Here go the definitions we have seen before
*/

int main() {
  struct list* my_list = NULL;

  my_list = push_front(my_list, 34);
  my_list = push_front(my_list, 18);
  my_list = push_back(my_list, 56);
  print_list(my_list);
  dispose_list(my_list);
}
```

Doubly Linked Lists



Circular Doubly Linked Lists



Declarations and Definitions

- Declaration: Introduces an object. After declaration the object can be used
  - Example: functions prototypes
- Definition: Specifies the structure of an object
  - Example: function definition
- Declarations can appear many times, definitions just once

Dealing with Larger Programs

- Functions are a first step to break big programs in small logical units
  - Breakup of specific tasks into functions
- A further step consists in breaking the source into many modules (files)
  - Smaller files are easy to handle
  - Objects sharing a context can be put together in one module and easily reused
- C allows to put together separately compiled files to have one executable

Libraries

- Libraries are collections of compiled definitions
- You include header files to get the declarations of objects in libraries
- At linking time libraries are searched for unresolved declarations
- Some libraries are included by `gcc` even if you do not specifically ask for them

Example: Linking math Functions

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

int main() {
  double n;
  double  sn;

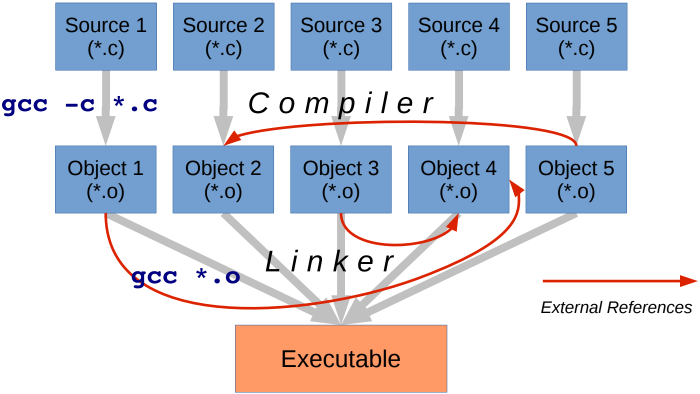
  scanf("%lf", &n); /* double needs %lf */
  sn = sqrt(n);
  /* conversion from double to float ok */
  printf("Square root of %f is %f\n", n, sn);
  return 0;
}
```

```
gcc -lm -o compute compute.c
```

Building from Multiple Sources

- C compilers can compile multiple sources files into one executable
- For every declaration there must be one definition in one of the compiled files
  - Indeed also libraries play a role
  - This control is performed by the linker
- `gcc -o name file1.c file2.c file3.c`

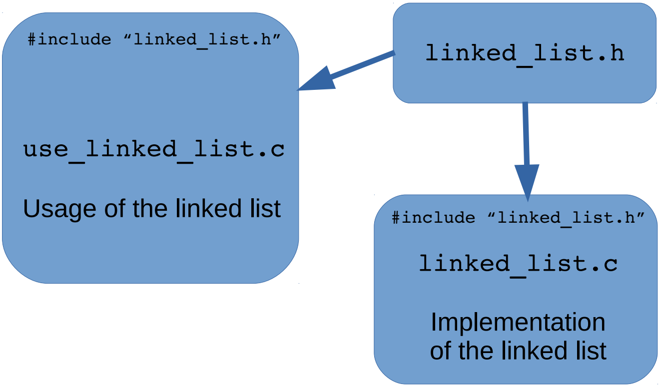
Linking


Source 1 (*.c)
Source 2 (*.c)
Source 3 (*.c)
Source 4 (*.c)
Source 5 (*.c)
gcc -c *.c
Compiler
Object 1 (*.o)
Object 2 (*.o)
Object 3 (*.o)
Object 4 (*.o)
Object 5 (*.o)
gcc *.o
Linker
External References
Executable

Linked List Header File

```
/*****************************************************
 *                                                   *
 * A simply linked list is linked from node structures *
 * whose size can grow as needed. Adding more elements *
 * to the list will just cause it to grow and removing *
 * elements will cause it to shrink.
 *---------------------------------------------------*
 * struct ll_node                                    *
 *      used to hold the information for a node of a *
 *      simply linked list                           *
 *---------------------------------------------------*
 * Function declaration (routines)                   *
 *                                                   *
 *      push_front -- add an element in the beginning *
 *      push_back -- add an element in the end       *
 *      dispose_list -- remove all the elements      *
 *      ...                                          *
 *****************************************************/
```

Definition Import via `#include`



Compile Linked List from Multiple Sources

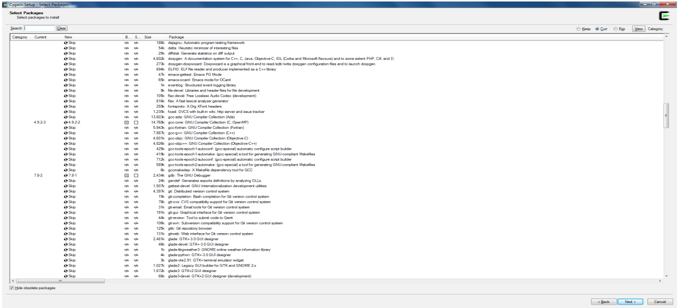
- Create a project with your IDE, add all files including the header file and then compile and execute
- or
- Compile: `gcc -Wall -o use_linked_list linked_list.c use_linked_list.c`
- Execute: `./use_linked_list`

Cygwin

- Cygwin is a Unix-like environment and command-line interface for Microsoft Windows
- Cygwin provides native integration of Windows-based applications, data, and other system resources with applications, software tools, and data of the Unix-like environment
- Thus it is possible to launch Windows applications from the Cygwin environment, as well as to use Cygwin tools and applications within the Windows operating context

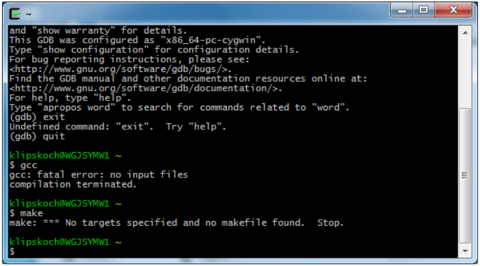
Install Cygwin on Windows (1)

- Go to `https://cygwin.com/install.html`, download `setup-x86_64.exe` and install it
- During installation add `gdb`, `gcc-core` and `make` listed under `Devel`



Install Cygwin on Windows (2)

- Once installed under `C:/cygwin64` you will have a Unix-like environment
- You can use it to compile and debug your code using `gcc` and `gdb`



`make` (1)

- `make` is special utility to help programmer compiling and linking programs
- Programmers had to type in compile commands for every change in program
- With more modules more files need to be compiled
  - Possibility to write script, which handles sequence of compile commands
- Inefficient

`make` (2)

- Compiling takes time
- For only small change in one module, not necessary to recompile other modules
- `make`: compilations depends upon whether file has been updated since last compilation
- Also possible to specify dependencies
- Also possible to specify commands to compile (e.g., depending of suffix of source)

Makefile (1)

- A makefile has the name "Makefile"
- Makefile contains following sections:
  - Comments
  - Macros
  - Explicit rules
  - Default rules

Makefile (2)

- Comments
  - Any line that starts with a # is a comment
- Macro format
  - `name = data`
  - Ex: `OBJ=linked_list.o use_linked_list.o`
  - Can be referred to as `$(OBJ)` from now on

Makefile (3)

Explicit rules

- ```
  target:source1 [source2] [source3]
          command1
          [command2]
          [command3]
  ```
- `target` is the name of file to create
- File is created from `source1` (and `source2`, ...)
- ```
  use_linked_list: use_linked_list.o linked_list.o
                   gcc -o use_linked_list
                       use_linked_list.o linked_list.o
  ```

Makefile (4)

Explicit rules

- ```
  target:
          command
  ```
  Commands are unconditionally executed each time make is run
- Commands may be omitted, built-in rules are used then to determine what to do
  `use_linked_list.o: linked_list.h use_linked_list.c`
- Create `use_linked_list.o` from `linked_list.h` and `use_linked_list.c` using standard suffix rule for getting to `use_linked_list.o` from `linked_list.c`
- `$(CC) $(CFLAGS) -c file.c`
  ```

Example Makefile (1)

- Header file with `struct` definition and function prototypes
  - `header_file.h`
- Implementation file with usage of the `struct` and function definitions
  - `implementation.c`
- Main function where implemented behaviour can be used
  - `main.c`
- Makefile with different targets for different purposes
  - `Makefile.txt`

Run Makefile

- `make`
  Default makefile called `Makefile` and default target all
- `make TargetName`
  Default makefile called `Makefile` and target `TargetName`
- `make -f MyMakeFile.txt`
  Makefile called `MyMakeFile.txt` and default target all
- `make -f MyMakeFile.txt TargetName`
  Makefile called `MyMakeFile.txt` and default target `TargetName`

Example Makefile (2)

```
CC = gcc
CFLAGS = -Wall

OBJ = linked_list.o use_linked_list.o

all: use_linked_list

use_linked_list: $(OBJ)
			$(CC) $(CFLAGS) -o use_linked_list $(OBJ)

use_linked_list.o: linked_list.h use_linked_list.c

linked_list.o: linked_list.h linked_list.c

clean:
	rm -f use_linked_list *.o
```

Function Pointers

- A pointer may not just point to a variable, but may also point to a function
- In the program it is assumed that the function does what it has to do and you use it in your program as if it was there
- The decision which function will actually be called is determined at run-time

Function Pointer Syntax

- `void (*foo)(int);`
  - `foo` is a pointer to a function taking one argument, an integer, and that returns `void`
- `void *(*foo)(int *);`
  - `foo` is a pointer to a function that returns a `void *` and takes an `int *` as parameter
- `int (*foo_array[2])(int);`
  - `foo_array` is an array of two pointer functions having an `int` as parameter and returning an `int`
- Easier and equivalent:

```
typedef int (*foo_ptr_t)(int);
foo_ptr_t foo_ptr_array[2];
```

Function Pointers: Simple Examples

```c
void (*func) (void);   /* define pointer to function */
void a(void) { printf("func a\n"); }
void b(void) { printf("func b\n"); }

int main() {
  func = &a;   // calling func() is the same as calling a()
  func = a;   // calling func() is the same as calling a()
  func();
}
```

One may have an array of function pointers:

```c
int func1(void);
int func2(void);
int func3(void);
int (*func_arr[3])(void)
                          = {func1, func2, func3};
```

Another Function Pointer Example

```
#include <stdio.h>
void output(void) {
  printf("%s\n", "Please enter a number:");
}
int sum(int a, int b) {
  return (a + b);
}
int main() {
  int x, y;
  void (*fptr1)(void);
  int (*fptr2)(int, int);
  fptr1 = output;
  fptr2 = sum;
  fptr1();      // cannot see whether function or pointer
  scanf("%d", &x);
  (fptr1)();    // some prefer this to show it is pointer
  (*fptr1)();   // complete syntax, same as above
  scanf("%d", &y);
  printf("The sum is %d.\n", fptr2(x, y));
}
```

Alternatives for Usage

```
int (*fct) (int, int);
/* define pointer to a fct */
int plus(int a, int b) {return a+b;}
int minus(int a, int b) {return a-b;}
int a=3; int b=4;
fct = &plus;
/* calling fct() same as calling plus() */
printf("fct(a,b):%d\n", fct(a,b));  /* 7 */
```

or

```
printf("fct(a,b):%d\n", (*fct)(a,b));  /* 7 */
fct = &minus;
/* calling fct() same as calling minus() */
printf("fct(a,b):%d\n", fct(a,b));   /* -1 */
```

Printing a List with Function Pointers

```
void foreach_list_simple(struct list *my_list,
    void (*func)(int num)) {
  struct list *p;
  for (p = my_list; p != NULL; p = p->next) {
    func(p->info);
  }
}
void printnum(int num) {
  printf("%d ", num);
}
int main() {
  ...
  foreach_list_simple(my_list, printnum);
  return 0;
}
```

Summing Up a List with Function Pointers

```
void foreach_list(struct list *my_list,
    void (*func)(int num, void *state),
      void *state) {
  struct list *p;
  for (p = my_list; p != NULL; p = p->next) {
    func(p->info, state);
  }
}
void sumup(int num, void *state) {
  int *p = (int *) state;
  *p += num;
}
int main() {
  ...
  int sum = 0;
  foreach_list(my_list, sumup, &sum);
  printf("sum=%d\n", sum); return 0;
}
```

Sorting with Function Pointers

- An array of lines (strings) can be sorted according to multiple criteria:
  - Lexicographic comparison of two lines (strings) is done by `strcmp()`
  - Function `numcmp()` compares two lines on the basis of numeric value and returns the same kind of condition indication as `strcmp` does
- These functions are declared ahead of the main and a pointer to the appropriate one is passed to the function `qsort` (implementing quick sort)

Function `strcmp()`

- `strcmp()` compares the two strings `s1` and `s2`
- It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`

```c
#include <stdio.h>
#include <string.h>
int main() {
  char s1[30], s2[30];
  scanf("%29s", s1);
  scanf("%29s", s2);
  // avoid buffer overflow on the strings
  if (!strcmp(s1, s2)) {
    printf("Both strings are equal!\n");
  }
  return 0;
}
```

Function numcmp()

- Function strcmp() compares two strings and returns <0, 0, >0
- Here you see function numcmp(), which compares two strings on a leading numeric value, computed by calling atof

```c
#include <stdlib.h>
/* numcmp: compare s1 and s2
    numerically */
int numcmp(char *s1, char *s2 ){
  double v1, v2;
  v1 = atof(s1);
  v2 = atof(s2);
  if (v1 < v2)
    return -1;
  else if (v1 > v2)
    return 1;
  else
    return 0;
}
```

Further Refinement of the Sorting Problem

- You want to write a sorting function
- The sorting algorithm is the same, but the comparison function may be different (i.e., you want ordering by different keys, different data types, increasing/decreasing sequence)
- Can we have a pointer to a comparison function as parameter for the sort function and write the sort function only once, always calling it with different comparison functions?

Function Pointer as Function Argument

```
int my_sort(int *array, int n,
    int (*my_cmp) (int ,int)) {
  ...
  if ( my_cmp(array[i],array[i+1]) == 1)  {
    ...
  }
  ...
}
```

Usage of Function Pointers as Function Arguments

```c
int fct1(int a, int b) {
  ...
}
int *array, n;
/* pass your function as argument */
my_sort(array, n, &fct1);
```

Using the qsort() from stdlib.h

This version of the qsort is declared in stdlib.h:

```
void qsort(void *base,
           size_t nmemb,
           size_t size,
           int(*compare)(const void *,
                         const void *));
```

User Supplied Comparison Function

```c
int my_compare(const void *va, const void *vb) {
  const int* a = (const int*) va;
  const int* b = (const int*) vb;
  if (*a < *b) return -1;
  else if (*a > *b) return 1;
  else return 0;
}
```

Calling qsort()

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define NUM_ELEMENTS 50
int my_compare(const void *va, const void *vb) {
  const int* a = (const int*) va;
  const int* b = (const int*) vb;
  if (*a < *b) return -1;
  else if (*a > *b) return 1;
  else return 0;
}
int main() {
  srand(time(NULL)); // initialize random number generator
  int arr[NUM_ELEMENTS];
  int i;
  /* fill array with random numbers */
  for (i = 0; i < NUM_ELEMENTS; i++)
    arr[i] = rand() % 1000;
  qsort(arr, NUM_ELEMENTS, sizeof(arr[0]), my_compare);
  for (i = 0; i < NUM_ELEMENTS; i++)
    printf("%d\n", arr[i]);
  return 0;
}
```

Why useful?

- Can use `qsort()` or other functions with your own data types (`struct`), just need to write the comparison function, but no need to duplicate the sorting function itself
- Change comparison function to reverse the order
- Change comparison function to sort by different key (member of your `struct`), e.g., sort by first name, last name, age, ...