Leibniz Universität Hannover

Programmieren 1

Program Structure, List of Pointers

Human-Computer Interaction Group

Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

Lectures

#	Date	Topic	HÜ→	HÜ←
1	14.10.	Organization, computers, programming, algorithms, PostFix introduction (execution model, IDE, basic operators, booleans, naming)	1	20.10. 23:59
2	21.10.	PostFix (primitive types, functions, parameters, local variables, tests), recipe for atomic data	2	27.10. 23:59
3	28.10.	PostFix (operators, array operations, string operations), recipes for enumerations, intervals, and itemizations	3	3.11. 23:59
4	4.11.	Recipes for compound and variant data, iteration and recursion, PostFix (loops, association arrays, data definitions)	4	10.11. 23:59
5	11.11.	C introduction (if, variables, functions, loops), Programming I C library	5	17.11. 23:59
6	18.11.	Data types, infix expressions, C language (enum, switch)	6	24.11. 23:59
7	25.11.	Compound and variant data, C language (formatted output, struct, union)	7	1.12. 23:59
8	2.12.	C language (arrays, pointers) arrays: fixed-size collections, linear and binary search	8	8.12. 23:59
9	9.12.	Dynamic memory (malloc, free), recursion (recursive data, recursive algorithms)	9	15.12. 23:59
10	16.12.	Linked lists, binary trees, search trees	10	22.12. 23:59
11	13.1.	C language (program structure, scope, lifetime, linkage), function pointers, pointer lists	11	12.1. 23:59
12	20.1.	List and tree operations (filter, map, reduce), objects, object lists	12	19.1. 23:59
13	27.1.	Dynamic data structures (stacks, queues, maps, sets), iterators, documentation tools	(13)	

Review

- Linked lists
  - Basic list operations
  - Basic list implementation
  - Ordered insertion
- Binary trees
  - Self-referential, hierarchical data structure
  - Either (1) empty or (2) a node with a value, a left binary tree, and a right binary tree
- Search trees
  - Ordered elements allow for efficient search
- Balanced search trees (optional topic)
  - In each search step exclude about half of the elements

Lists and trees involve:
- dynamic memory allocation
- recursion / recursive types

Preview

- Formatted output into a buffer (snprintf)
- Formatted input into a buffer (scanf)
- Program structure, multiple files
- Lifetime, scope, linkage
- Function pointers
- A general pointer list

SNPRINTF AND SCANF

Formatted Output to a String: snprintf

- Formatted printing into a buffer instead of standard output
- Example

```
#include <stdio.h> // contains snprintf function header
int main(void) {
        char *vehicle = "bike";
        int wheel_count = 2;
        double top_speed = 25.9; // km/h
        char buffer[100]; // will store the formatted string
        snprintf(buffer, 100, // buffer and buffer length, max. string length: 99 characters
                        "My %s has %d wheels and a top speed of %.1f km/h.",
                        vehicle, wheel_count, top_speed);
        printf("%s\n", buffer);
        return 0;
}
```

Formatted Input: scanf

- Read characters from standard input and interpret them according to the format string
  - int scanf(char* format, arg_1, arg_2, ...)
- Example

```
#include <stdio.h>
int main(void) {
    int day = 0;
    int month = 0;
    int year = 0;
    int matches = scanf("%d.%d.%d", &day, &month, &year);
    printf("%d: %d %d %d\n", matches, day, month, year);
    return 0;
}
```

expects an integer followed by a dot, followed by an integer, ...

stores the parts in day, month, year

Input/Output:

```
$ ./testscanf
14.1.2022
3: 14 1 2022
$ ./testscanf
14.1
2: 14 1 0
```

PROGRAM STRUCTURE

Structure of C Programs

- A program may consist of multiple .c-files (modules)
- A .c-file may contain multiple functions
- A function may contain multiple (nested) blocks

- Implementation files (.c) and header files (.h)
- .c-file (e.g., basic.c)
  - Implementation of a module
  - Each .c-file is compiled independently (a translation unit)
- .h-file (e.g., basic.h)
  - Interface of a module
  - Declarations (e.g. function headers and type definitions)

Structure of C Programs

- Organize source code into multiple files
  - Shorter source files easier to edit and understand
  - Put related functions and types in the same file

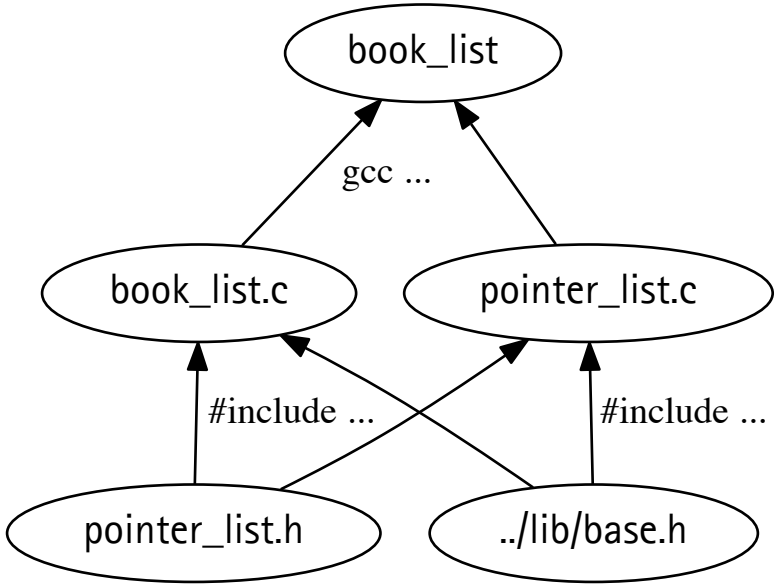
- Mechanisms to modularize programs
  - Organize source code into coherent modules
    - Example: List functions in list.c, book functions in book.c
  - Separate the interface from the implementation
    - Example: Public declarations in list.h and book.h
  - Control what may be accessed from other modules and what is private to a module

Compilation Process

- Compiler treats each .c-file independently
  - Preprocessor textually replaces #includes and #defines — c' ← c h*
  - Compiler compiles each preprocessed .c-file separately into an object file (.o-file) — o ← c'
  - Linker combines object files and static libraries (.a-files) into executable file — exe ← o+ a*
    - Intermediary .o-files then get deleted

- Example: gcc -Wall -o myprog.exe file1.c file2.c
  - Preprocessor includes header files
  - Compiler generates file1.o and file2.o (object files)
  - Linker combines them to myprog.exe (executable file)

Makefile for Multiple Source Files and Header Files




```
# define some variables
CFLAGS = -std=c99 -Wall -Werror -Wpointer-arith -Wfatal-errors
DEBUG = -g

# disable default suffixes
.SUFFIXES:

# book_list(.exe) depends on book_list.c, pointer_list.{c, h}
SOURCES = book_list.c pointer_list.c
HEADERS = pointer_list.h ../lib/base.h
book_list: $(SOURCES) $(HEADERS)
    gcc $(CFLAGS) $(DEBUG) $(SOURCES) -L../lib -lprog1 -lm -iquote../lib -o $@

# automatic variables:
# $@: target (book_list)

# gcc options:
# -L: where to look for additional libraries (in ../lib directory)
# -l: library to use (libprog1.a, libm.a)
# -i: where to look for additional include files (in ../lib directory)
# -o: how to name the output file
```

tab

`-L`: where to look for libraries

`-l`: what libraries to link

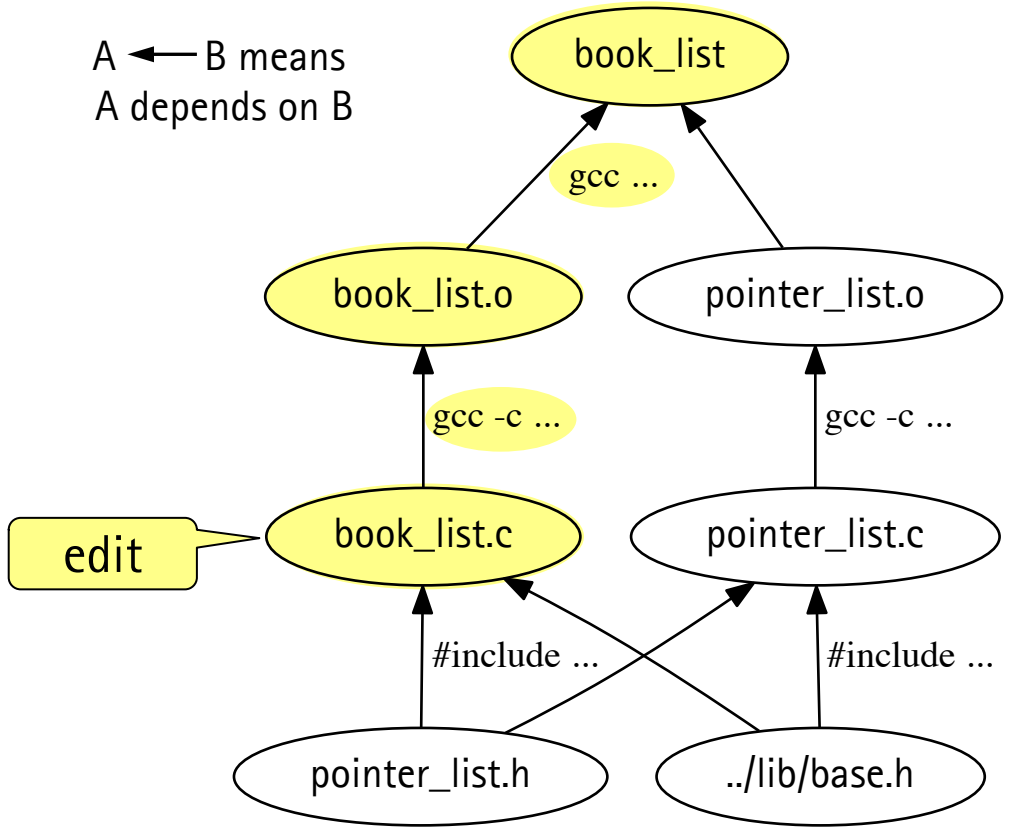
`-i`: where to look for includes

https://www.gnu.org/software/make/manual/html_node/index.html

Dependency Tree

- make book_list

- descend dependency tree to a leaf
- if a dependency is newer than its target or if a dependency does not exist
- then execute the action in the Makefile rule to produce the dependency



https://www.gnu.org/software/make/manual/html_node/index.html

Dependency Tree

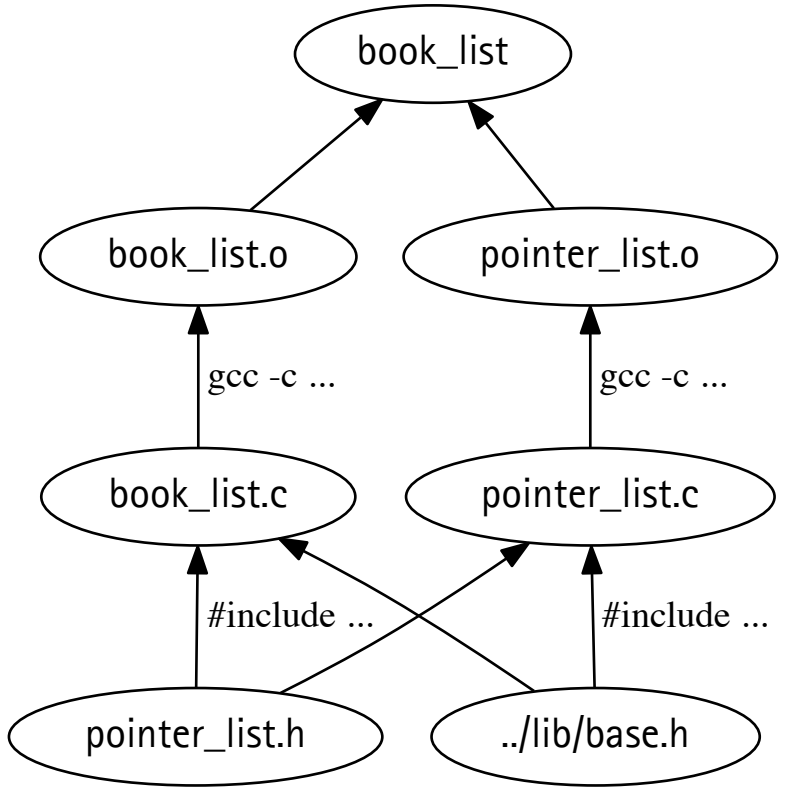
```
# define some variables
CFLAGS = -std=c99 -Wall -Werror -Wpointer-arith -Wfatal-errors
DEBUG = -g

# disable default suffixes
.SUFFIXES:

# book_list(.exe) depends on book_list.o and pointer_list.o
OBJECTS = book_list.o pointer_list.o
book_list: $(OBJECTS)
     gcc $(CFLAGS) $(DEBUG) $(OBJECTS) -L../lib -lprog1 -lm -o $@

# pattern rule: how to generate foo.o from foo.c
%.o: %.c
     gcc -c $(CFLAGS) $(DEBUG) -iquote../lib $<

# dependencies (without actions)
book_list.o: book_list.c pointer_list.h ../lib/base.h
pointer_list.o: pointer_list.c pointer_list.h ../lib/base.h
```



https://www.gnu.org/software/make/manual/html_node/index.html

Inclusion of Header Files

- #include: Preprocessor textually replaces by contents of file
  - Before actual compilation
- How to locate file?
  - `#include "file"` looks in directory of current file, then in standard include directories
  - `#include <file>` looks in standard include directories
- #include does not check if a file has already been included

```
#include "point.h"
#include "point.h"

#include "mysubdir/point.h"
```

Would insert the contents of point.h again

Conditional Inclusion of Header Files

- Ensure that a header file is included only once
- (in point.h) Surround declarations by preprocessor directives #ifndef, #define, #endif

```
#ifndef POINT_H_INCLUDED
#define POINT_H_INCLUDED
// actual contents
// of point.h
#endif
```

If symbol is not defined, then keep the lines between #ifndef and #endif, otherwise don't.

Lifetime, Scope, Linkage

- Modularization only works if there is a way to refer to entities (functions, type definitions, constants, etc.) in other modules
- Lifetime
  - Period of existence of a variable during program execution
- Scope
  - Parts of the source code in which a name may be used
- Linkage
  - Declarations in different scopes may refer to same entity
- Reminder: Declaration vs. definition
  - Declaration: make name and type known
  - Definition: also provide an implementation / reserve storage

“Local” and “Global” Variables and Constants

- Local variables/constants
  - Defined in functions
  - Defined as parameters
  - Lifetime: block execution
  - Scope: to end of function
  - Also known as: automatic variables

```
int f(int a) {
    const int A = 3;
    return A * a;
}
```

a is a “local” variable of f

A is a “local” constant of f

- Global variables/constants
  - Defined outside of functions
  - Lifetime: program execution
  - Scope: to end of file
  - Also known as: external variables

```
#include "base.h"
int g = 0;
const int G = 123;
...
```

g is a “global” variable

G is a “global” constant

Lifetime of Variables

- Lifetime: Period of existence during program execution
- Lifetime: program execution
  - Variable exists as long as the program is executed
  - Initialized once
  - Declaration at file level or in a function with keyword `static`
- Lifetime: block execution
  - Variable exists as long as the block is executed
  - Automatic variable: created/destroyed on each block entry/exit
  - Declared within block (without keyword `static`)

Lifetime of Non-Static Local Variables

- Automatically created / destroyed on entry / exit
- Lifetime: block execution

- Example

```
#include "base.h"

int inc(void) {
        int i = 3;
        i++;
        return i;
}
```

```
int main(void) {
    println(inc());
    println(inc());
    println(inc());
    return 0;
}
```

Output:

```
4
4
4
```

Lifetime of Static Local Variables

- Private state of a function
- Initialized once
- Lifetime: program execution
- Example

```
#include "base.h"

int inc(void) {
        static int i = 3;
        i++;
        return i;
}
```

```
int main(void) {
    println(inc());
    println(inc());
    println(inc());
    return 0;
}
```

Output:

```
4
5
6
```

Scope of Functions and Variables/Constants

- Scope: Parts of the source code in which a name may be used
  - Same name may refer to different things in different scopes
- File scope
  - Declaration at file level
  - Use from point of declaration to end of file
- Block scope
  - Declaration within block or in function parameter list
  - Use from point of declaration to end of block and within nested blocks
- Function header scope
  - Within function header only
- Function scope
  - Labels only (e.g. for goto statement)

Block Scope

- Declarations of variables may occur within a block {...}
- Variables of the same name in inner blocks hide variables in outer blocks (and external variables)
- Example:
  - Block execution lifetime
  - Block scope
    - scope of outer i
    - scope of inner i
  - No linkage

```
#include "base.h"
int main(void) {
    int i = 5;
    if (i > 1) {
        int i = 1; // inner i hides outer i
        println(i); // output: 1
    }
    println(i); // output: 5
    return 0;
}
```

Avoid!

External ("Global") Variables and Functions

- The scope of an external variable or function extends from the point of declaration to the end of the file

- Example:
  - (external variable) x is visible in f
  - x is not visible in main
  - f is not visible in main

test.c

```
int main(void) { ... }

int x = 0;

int f(int d) { ... }
```

Declarations and Definitions

function **declaration**: declares name, parameters, and return type of function

external variable **declaration**: declares name and type of variable

external variable **definition**: declares name and type of function **and** sets aside storage

function **definition**: declares a function **and** provides its implementation

test.c

```
int f(int d);

extern int x;

int main(void) { ... }

int x = 0;

int f(int d) { ... }
```

extern Keyword

- extern keyword indicates that storage is allocated elsewhere
  - Declares a variable, does not set aside storage

- Example:
  - x is visible in f
  - x is visible in main
  - f is visible in main

test.c

```
int f(int d);

extern int x;

int main(void) { ... }

int x = 0;

int f(int d) { ... }
```

Linkage of Functions and Variables/Constants

- Linkage: Declarations in different scopes may refer to same entity
  - Accessible across translation units (.c-files)
  - Controlled via point of declaration and keyword `static`
- Internal linkage
  - Declaration at file level with keyword `static`
  - Access limited to translation unit (.c-file)
- External linkage
  - Declaration at file level without keyword `static`
  - Accessible from other translation units (.c-files)
- No linkage
  - Declaration within a block or a function parameter list
  - Accessible only within that scope

C Standard: "An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called **linkage.**"

Linkage of External Variables and Functions

- Non-static external variables and functions are "public"
  - Accessible in other source files
- Example

```
double f(double x) {...}
```

visible also in other .c-files of the program

- Static external variables and functions are "private"
  - The **static** keyword limits the scope to that source file
  - Advantage: Can use the same name in many files (to refer to different things)
- Example

```
static double f(double x) {...}
static int x = 0;
```

only visible in this .c-file

Non-Static vs. Static External Variables

- Non-static external variables and functions are “public”

file1.c:

```
int x = 1;
int main(void) { return 0; }
```

file2.c:

```
int x = 2;
```

- gcc -Wall -o myprog file1.c file2.c

  Linker error (ld is the linker)

```
duplicate symbol _x in:
    ccn8pJtf.o
    ccVJWDCV.o
ld: 1 duplicate symbol for architecture x86_64
```

Non-Static vs. Static External Variables

- Static external variables and functions are “private”

file1.c:

```
static int x = 1;
int main(void) { return 0; }
```

file2.c:

```
static int x = 2;
```

x in file1.c and x in file2.c are unrelated, refer to different “integer objects”

- gcc -Wall -o myprog file1.c file2.c

- No error (only a warning, because the variables are not used)

```
file1.c:1: warning: 'x' defined but not used
file2.c:1: warning: 'x' defined but not used
```

Summary of Lifetime, Scope, and Linkage

(*) Accessible from other translation unit?

Level	Declaration/Definition	Keyword	Lifetime	Scope	Linkage	Public? *
file	variable definition	`static`	program	rest of file	internal	no
file	variable declaration	`extern`	program	rest of file		a reference
file	variable definition or declaration	none	program	rest of file	external	yes (may be a reference)
file	function declaration or definition	`static`	program	rest of file	internal	no
file	function declaration or definition	`extern` (optional)	program	rest of file	external	yes
block	variable declaration	`extern`	program	rest of block		a reference
block	variable definition	`static`	program	rest of block	no linkage	no
block	variable definition	`auto` (optional)	block	rest of block	no linkage	no

FUNCTION POINTERS

Using Functions as Parameters

- Provide address of machine code to a function
- A powerful way to make a function more useful

- Example: Sorting algorithm
- Same algorithm, but different sorting criteria
  - Sort numbers by decreasing/increasing value
  - Sort addresses by zip code
  - Sort people by age
  - Sort vehicles by top speed

Comparison Function as Parameter to Sorting Algorithm

- Sorting criterion defined as a comparison function
- Sort an array a of n pointers using a comparison function cmp

```
void sort(void* a[], int n, ComparisonFunction cmp);
```

- Comparison function

```
CmpResult cmp(void* x, void* y);    or    int cmp(void* x, void* y);
```

- Comparison result

```
typedef enum {
    LT = -1,    // less than
    EQ = 0,     // equal
    GT = 1      // greater than
} CmpResult;
```

Comparison Function as Parameter to Sorting Algorithm

Standard library sorting function qsort for sorting arrays:

```
#include <stdlib.h>    // qsort
typedef unsigned long int size_t;    // stddef.h

void qsort(void* a, size_t n, size_t size, int (*cmp)(const void*, const void*));
```

element size — element comparison function

cmp is a pointer to a function that takes two pointers and returns an int

Return value { <0 / =0 / >0 } if first argument is { less than / equal to / greater than } second

Comparison Function as Parameter to Sorting Algorithm

```c
#include <stdlib.h> // qsort

typedef struct { // Represents a word and its position in multi-line text.
    char* value; // '\0'-terminated string
    int line; // line number of this word in the text
    int column; // column number of the beginning of this word in the text
} Word;

typedef  struct { // An array of n words.
    int n; // number of words
    Word words[]; // array of words (flexible array member must be last)
} WordArray;
```

Comparison Function as Parameter to Sorting Algorithm

```
WordArray* word_array_create(int word_count) {
    require("not negative", word_count >= 0);
    int n = sizeof(WordArray) + word_count * sizeof(Word);
    WordArray* a = xmalloc(n);
    memset(a, 0, n); // initialize to 0
    a->n = word_count;
    return a;
}

void word_array_sort(WordArray* a) {
    require_not_null(a);
    // sort n words, each of a certain size, use function cmp_words to compare
    qsort(a->words, a->n, sizeof(Word), cmp_words);
}
```

```
typedef  struct {
    int n;
    Word words[];
} WordArray;
```

Comparison Function as Parameter to Sorting Algorithm

```
// Compares two words (for use with qsort). Words are sorted lexicographically
// and by their position of occurrence in the text.
int cmp_words(const void* a,  const void *b) {
    Word* v = (Word*)a;
    Word* w = (Word*)b;
    // compare words
    int c = strcmp(v->value, w->value);
    if (c != 0) return c; // words are different, return comparison result
    // words are equal, check line number
    if (v->line < w->line) return -1;
    if (v->line > w->line) return 1;
    // words and lines numbers are equal, check column number
    if (v->column < w->column) return -1;
    if (v->column > w->column) return 1;
    return 0;
}
```

Pointers to Functions

```c
#include <stdio.h>

char* hello(void) {
    return "hello";
}

int main(void) {
    printf ("%s\n", hello());    // calls function hello
                                 // output: hello
    printf("%p\n", hello);       // address of function
                                 // output: 0x10094bf10
    return 0;
}
```

A function's name is the address of the function's machine code in memory

Declaring Pointers to Functions

```c
#include <stdio.h>

void hello(void) {
    printf("hello\n");
}

int main(void) {
    void (*fp)(void);          // declares variable fp as a pointer to a function
                               // that takes no arguments and returns no result
    fp = hello;                // assigns the address of function hello to fp
    fp();                      // calls function hello
    return 0;
}
```

Complicated Declarations in C

- Operator precedence: ( ) higher than *, ( ) equal to [ ]
- int* f(int i); ← explicit precedence: int(*(f(int i)))
  - f is a function that takes an int and returns a pointer to int
- int (*g)(double d);
  - g is a pointer to function that takes a double and returns an int
- int* (*h)(void);
  - h is a pointer to function that takes no arguments and returns a pointer to an int
- int (*a)[3]; ← explicit precedence: int((*a)[3])
  - a is a pointer to an array of 3 ints
- int* b[3]; ← explicit precedence: int(*(b[3]))
  - b is an array of 3 pointers to int

typedef simplifies Function Pointer Declarations

- typedef creates new type names
  - we have seen typedef with enums and structs

- Example typedef for a function:

```
typedef void (*MyFuncType)(void); // declare new type
                                  // (function, no parameters, no return value)
MyFuncType fp3; // declare variable of this type
fp3 = hello; // assign pointer to function hello
fp3(); // call function hello
```

A POINTER LIST

A Pointer List

- List of pointers to arbitrary elements (can store anything)

```
typedef struct Node {
    void* value; // a pointer to anything, not specified further
    struct Node* next; // self-reference
} Node;
```

- Flexible but no information about the elements
  - Size? How to print? How to compare?
- Some list functions cannot be implemented without knowledge of element details
- Solution: Element-specific functions and function pointers

Printing a List of Integers (from previous lecture)

```
void print_list(Node* list) {
    if (list == NULL) {
        printf("[]");
    } else {
        printf("[%d", list->value);
        for (Node* n = list->next; n != NULL; n = n->next) {
            printf(" %d", n->value);
        }
        printf("]");
    }
}
```

element is an integer, know how to print it

element is an integer, know how to print it

Printing a List of Arbitrary Values

```
void print_list(Node* list, ToStringFunc to_string) {
    require_not_null(to_string);
    if (list == NULL) {
        printf("[]");
    } else {
        String s = to_string(list->value);
        printf("[%s", s);
        free(s);
        for (Node* n = list->next; n != NULL; n = n->next) {
            s = to_string(n->value);
            printf(", %s", s);
            free(s);
        }
        printf("]");
    }
}
```

pointer to function that produces a string representation of an element

call to_string function

release memory

call to_string function

release memory

Printing a List of Arbitrary Values

- `typedef String (*ToStringFunc)(void* element);`
  - ToStringFunc is a function that takes an element (a void*) and returns a string representation of that element
  - The string is dynamically allocated: caller has to release it

- Example: List of books

```
typedef struct {
    String authors;
    String title;
    int year;
} Book;
```

Creating a Book

- Constructor for a book

```
Book* new_book(String authors, String title, int year) {
    Book* b = xcalloc(1, sizeof(Book));
    b->authors = s_copy(authors); // own copy of string (dynamically allocated)
    b->title = s_copy(title); // own copy of string (dynamically allocated)
    b->year = year;
    return b;
}
```

- Copy function for a book

```
void* copy_book(void* x) {
    require_not_null(x);
    Book* b = (Book*) x;
    return new_book(b->authors, b->title, b->year);
}
```

x has to point to a Book!
No type checking!

```
typedef struct {
    String authors;
    String title;
    int year;
} Book;
```

Printing a List of Arbitrary Values

- String format:
  `<authors>: <title>, <year>.`
  `Goethe: Faust, 1808.`

```
typedef struct {
    String authors;
    String title;
    int year;
} Book;
```

```
String book_to_string(void* x) {
    require_not_null(x);
    Book* b = (Book*)x; // type cast void* to Book*
    String year = s_of_int(b->year); // convert int to string
    int n = strlen(b->authors) + strlen(b->title) + strlen(year) + 6;
    String s = xmalloc(n); // allocate space for string representation
    snprintf(s, n, "%s: %s, %s.", b->authors, b->title, year); // print into buffer
    free(year);
    return s;
}
```

Printing a List of Books

- Code

```
Book* b1 = new_book("Alice and Bob", "Cryptography for Dummies", 1987);
Book* b2 = new_book("Doris Doe", "C-Programming in 3 hours", 1999);
Book* b3 = new_book("Emma Erlang", "PostFix-Programming for Experts", 2018);
Node* list = new_node(b1, new_node(b2, new_node(b3, NULL)));
println_list(list, book_to_string); // supply pointer to element-specific function
```

- Output

```
[Alice and Bob: Cryptography for Dummies, 1987.,
 Doris Doe: C-Programming in 3 hours, 1999.,
 Emma Erlang: PostFix-Programming for Experts, 2018.]
```

Freeing a List of Integers (from previous lecture)

```
// Frees all nodes of the list.
void free_list(Node* list) {
    Node* node_next = NULL;
    for (Node* node = list; node != NULL; node = node_next) {
        node_next = node->next;
        free(node);
    }
}
```

only release the node, no need to release the integer value

Freeing a List of Arbitrary Values

```
// Frees the element.
typedef void (*FreeFunc)(void* element);

// Frees all nodes of the list. Uses free_element to free each element.
// If free_element is NULL, does not free the elements.
void free_list(Node* list, FreeFunc free_element) {
    Node* node_next = NULL;
    for (Node* node = list; node != NULL; node = node_next) {
        node_next = node->next;
        if (free_element != NULL) free_element(node->value);
        free(node);
    }
}
```

use free_element function to free the element, don't release if NULL

Freeing a List of Books

- Release a single book

```
void free_book(void* x) {
    Book* b = (Book*) x;
    free(b->title); // title was dynamically allocated
    free(b->authors); // authors was dynamically allocated
    free(b); // release the book structure itself
}
```

- Release a list of books

```
free_list(list, free_book);
```

- Release the list nodes, but not the books

```
free_list(list, NULL);
```

Finding an Element in a List

```c
// Specifies what to look for.
typedef bool (*FilterFunc)(void* element, int i, void* x);

// Finds the first element in list for which pred(element, i, x) is true.
void* find_list(Node* list, FilterFunc pred, void* x) {
    require_not_null(pred);
    int i = 0;
    for (Node* node = list; node != NULL; node = node->next, i++) {
        if (pred(node->value, i, x)) {
            return node->value;
        }
    }
    return NULL;
}
```

Finding a Book by Year

- Look for a book published after 1990

```
bool published_after_1990(void* element, int i, void* x) {
    Book* b = (Book*) element;
    return b->year > 1990;
}
```

a separate function for each relevant year is inconvenient
→ provide year as a parameter

- Finding such a book in a list of books

```
Book* found_book = find_list(list, published_after_1990, NULL);
if (found_book != NULL) {
        String s = book_to_string(found_book);
        printsln(s);
        free(s);
}
```

Finding a Book by Year

- Look for a book published after a given year

```
bool published_after(void* element, int i, void* x) {
    Book* b = (Book*) element;
    int* year = (int*) x;
    return b->year > *year;
}
```

access year parameter

- Finding such a book in a list of books

provide year as a parameter

```
int year = 1990;
Book* found_book = find_list(list, published_after, &year);
if (found_book != NULL) {
        String s = book_to_string(found_book);
        printsln(s);
        free(s);
}
```

Finding a Book by Author

- Look for a book published by a given author

```
bool written_by(void* element, int i, void* x) {
    Book* b = (Book*) element;
    char* authors = (char*) x;
    return strcmp(authors, b->authors) == 0;
}
```

access authors parameter

- Finding such a book in a list of books

```
found_book = find_list(list, written_by, "Emma Erlang");
if (found_book != NULL) {
    String s = book_to_string(found_book);
    printsln(s);
    free(s);
}
```

provide authors as a parameter

```
// Compares s and t. Returns 0 if s and t are equal.
// Otherwise, for the first position at which s and t differ:
// Returns a value <0 if s[i] < t[i].
// Returns a value >0 if s[i] > t[i].
int strcmp(char* s, char* t);
```

Finding a Book by Author

- Look for a book published by a given author

```
bool written_by(void* element, int i, void* x) {
    Book* b = (Book*) element;
    char* authors = (char*) x;
    return strstr(b->authors, authors) != NULL;
}
```

access authors parameter

- Finding such a book in a list of books

```
found_book = find_list(list, written_by, "Erl");
if (found_book != NULL) {
    String s = book_to_string(found_book);
    printsln(s);
    free(s);
}
```

provide part of author name as parameter

```
// Returns a pointer to the first occurrence of t in s.
// Returns NULL if t is not part of s.
char* strstr(char* s, char* t);
```

Testing if an Element is Contained in a List

- Specify the target element using content-based or identity-based equality
- Equality function: Checks whether two elements are equal

```
typedef bool (*EqualFunc)(void* element1, void* element2);
```

- Return true iff element is in list

```
bool contains_list(Node* list, void* element, EqualFunc equal) {
    if (equal == NULL) { // identity-based equality
        ...
    } else { // content-based equality
        ...
    }
    return -1;
}
```

Testing if an Element is Contained in a List

```
// Checks if list contains element. Uses equal for content-based equality.
// Performs identity-based comparison if equal is NULL.
bool contains_list(Node* list, void* element, EqualFunc equal) {
    if (equal == NULL) { // identity-based equality
        for (Node* node = list; node != NULL; node = node->next) {
            if (node->value == element) return true; // compare pointers
        }
    } else { // content-based equality
        for (Node* node = list; node != NULL; node = node->next) {
            if (equal(node->value, element)) return true; // compare content
        }
    }
    return false;
}
```

Finding a Book

- Content-based equality of books

```
bool books_equal(void* x, void* y) {
    Book* a = (Book*) x;
    Book* b = (Book*) y;
    if (a == b) return true; // same address (or both NULL)
    if (a == NULL || b == NULL) return false; // one null, but not the other
    return s_equals(a->authors, b->authors) &&
            s_equals(a->title, b->title) && (a->year == b->year);
}
```

- Finding a book: content-based equality

```
bool is_contained = contains_list(list, my_book, books_equal);
```

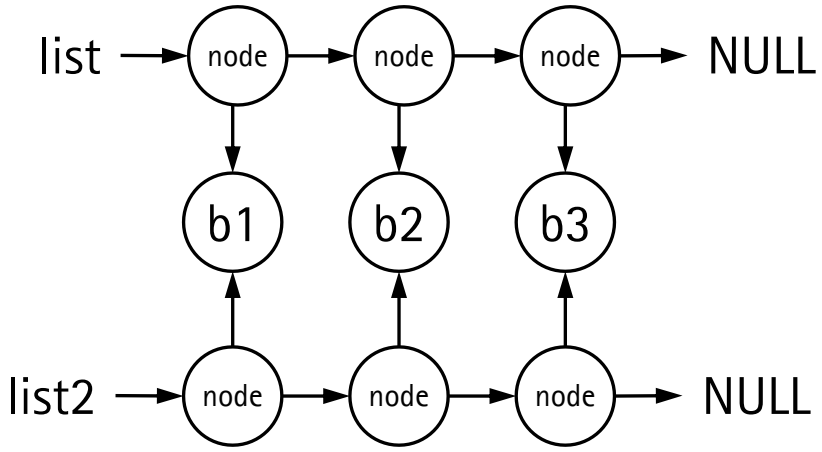
- Finding a book: identity-based equality

```
bool is_contained = contains_list(list, my_book, NULL);
```

Copying a List of Books: Shared Books

- Lists that share books

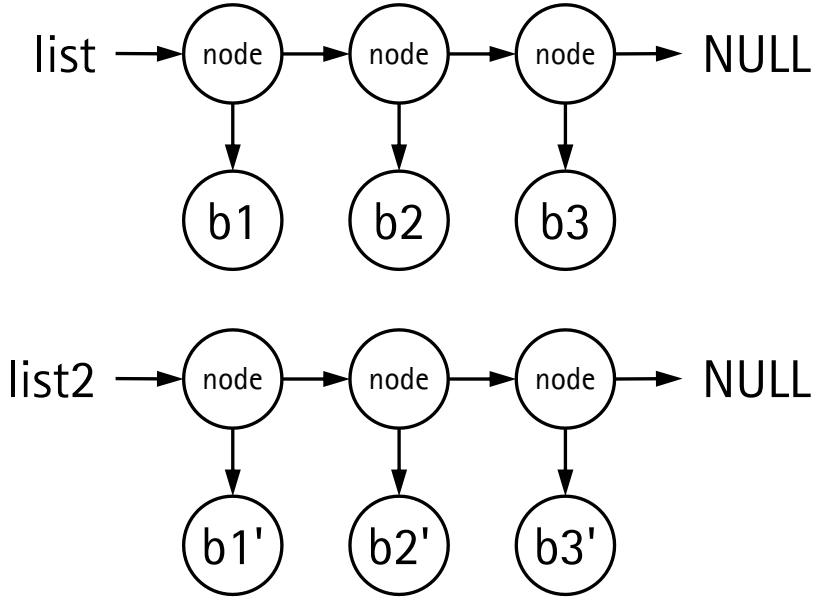
```
Node* list = new_node(b1, new_node(b2, new_node(b3, NULL)));
Node* list2 = copy_list(list, NULL); // do not copy books, lists share books
free_list(list, free_book); // free the list including the books
free_list(list2, NULL); // NULL FreeFunc: shared books already freed above
```



Copying a List of Books: Independent Copies

- Lists with independent copies of books

```
Node* list = new_node(b1, new_node(b2, new_node(b3, NULL)));
Node* list2 = copy_list(list, copy_book);  // use copy_book function
free_list(list, free_book);                // to duplicate the books
free_list(list2, free_book);
```



Copying a List of Arbitrary Elements

- Function copies element

```
typedef void* (*CopyFunc)(void* element);
```

- Copy the list, copy each element if copy function is provided

```
Node* copy_list(Node* list, CopyFunc copy_element) {
    if (list == NULL) {
        return NULL;
    } else {
        if (copy_element == NULL) { // just reassign, do not copy elements
            ...
        } else { // copy elements using copy function
            ...
        }
    }
}
```

Copying a List of Arbitrary Elements

```
if (copy_element == NULL) { // just reassign, do not copy elements
    Node* result = new_node(list->value, NULL);
    for (Node* n = result; list->next != NULL; n = n->next) {
        list = list->next;
        n->next = new_node(list->value, NULL);
    }
    return result;
} else { // copy elements using copy function
    Node* result = new_node(copy_element(list->value), NULL);
    for (Node* n = result; list->next != NULL; n = n->next) {
        list = list->next;
        n->next = new_node(copy_element(list->value), NULL);
    }
    return result;
}
```

Ordered Insertion into a Integer List (from previous lecture)

```
Node* insert_ordered(Node* list, int value) {
    if (list == NULL) { // empty list
        return new_node(value, NULL);
    } else if (value < list->value) { // insert before first
        return new_node(value, list);
    } else { // non-empty list, find insertion position after first node
        for (Node* n = list; n != NULL; n = n->next) {
            if (n->next == NULL) { // end of list?
                n->next = new_node(value, n->next);    break;
            } else if (value < n->next->value) { // found position?
                n->next = new_node(value, n->next);    break;
            }
        }
        return list;
    }
}
```

integers can be compared with built-in operators

how to compare arbitrary elements? → define a comparison function

Ordered Insertion into a List of Arbitrary Elements

- Function that compares two elements

```
typedef int (*CompFunc)(void* x, void*y);
```

- Return value indicates comparison result
  - 0 if x and y are equal
  - < 0 if x is smaller than y
  - > 0 if x is larger than y
- Alternatively could use an enumeration

```
typedef enum {
    LT = -1,    // less than
    EQ = 0,     // equal
    GT = 1      // greater than
} CmpResult;
```

Ordered Insertion into a List of Arbitrary Elements

```
Node* insert_ordered(Node* list, void* value, CompFunc compare) {
    require_not_null(compare);
    if (list == NULL) { // empty list
        return new_node(value, NULL);
    } else if (compare(value, list->value) < 0) { // insert before first
        return new_node(value, list);
    } else { // non-empty list, find insertion position after first node
        for (Node* n = list; n != NULL; n = n->next) {
            if (n->next == NULL) { // end of list?
                n->next = new_node(value, NULL);    break;
            } else if (compare(value, n->next->value) < 0) { // found position?
                n->next = new_node(value, n->next);    break;
            }
        }
        return list;
} }
```

Sorting a List of Books through Ordered Insertion

- Comparison function

```
int compare_year(void* x, void* y) {
    Book* a = (Book*) x;
    Book* b = (Book*) y;
    if (a == b) return 0; // same instance (or both NULL)
    if (a == NULL) return -1;
    if (b == NULL) return 1;
    return a->year - b->year; // just compare by year
}
```

Sorting a List of Books through Ordered Insertion

```
Book* b1 = new_book("Emma Erlang", "PostFix-Programming for Experts", 2018);
Book* b2 = new_book("Doris Doe", "C-Programming in 3 hours", 1999);
Book* b3 = new_book("Alice, Bob", "Cryptography for Dummies", 1987);
Node* list = new_node(b1, new_node(b2, new_node(b3, NULL)));

Node* sorted_list = NULL;
for (Node* n = list; n != NULL; n = n->next) {
    sorted_list = insert_ordered(sorted_list, copy_book(n->value), compare_year);
}
println_list(sorted_list, book_to_string);
```

sorted list gets its own copy of each book

Summary

- Formatted output into a buffer (snprintf)
- Formatted input into a buffer (scanf)
- Program structure, multiple files
- Lifetime
  - Period of existence of a variable during program execution
- Scope
  - Parts of the source code in which a name may be used
- Linkage
  - Referring to entities that are defined in another scope
- Function pointers
- A general pointer list