Leibniz Universität Hannover

Programmieren 1

Dynamic Memory

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

- Standard input and standard output
  - Reading from (writing to) the console or from (to) a file
- Arrays
  - Sequences of elements of the same type, access by index, no bounds checking
- Linear search and binary search
  - Efficient lookup of ordered elements in an array
- Pointers
  - A pointer is a variable that contains a memory address (or NULL)

Review: How to declare a pointer to an int?

A. `int* p;`
B. `int *p;`
C. `int * p;`
D. `int*p;`

Review: How to declare two pointers to int?

A. `int* p1, p2;`

B. `int *p1, *p2;`

C. `int *p1; int *p2;`

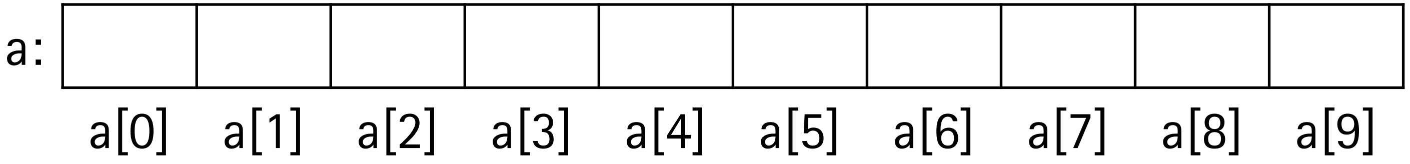
Preview

- Pointers and arrays
  - Strong relationship between pointers and arrays, pointer arithmetic
- Command line arguments
  - Input to a program on the command line: myprog.exe –s hello
- String and character functions
  - typedef char* String;
- Dynamic memory allocation
  - Automatic storage (call stack), static storage, dynamic heap
  - malloc/calloc, xmalloc/xcalloc, free
  - Memory allocation errors
- Recursive types and algorithms
  - Operations on lists

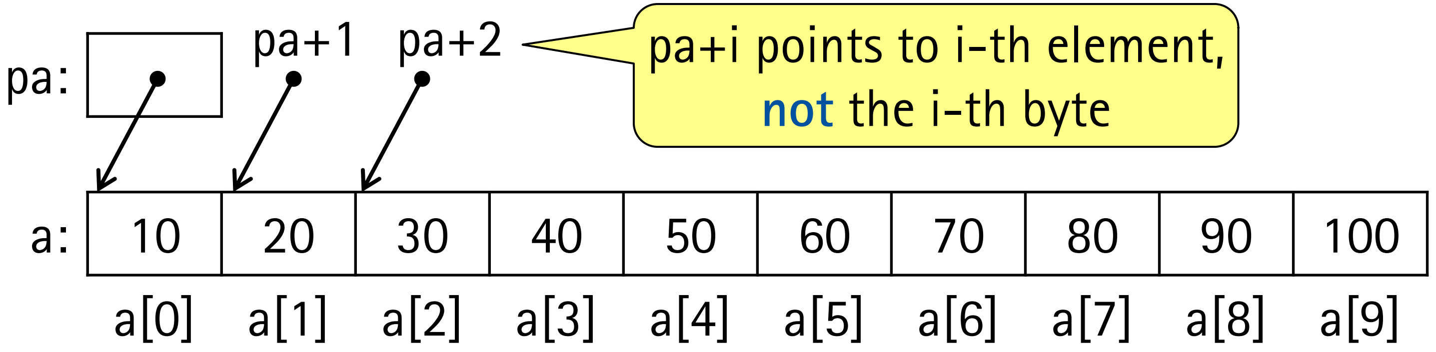
POINTERS AND ARRAYS

Pointers and Arrays

- Strong relationship between pointers and arrays
- int a[10]; defines an array of 10 consecutive integers



- int *pa = &a[0]; // address of first array element



*pa == 10 *(pa+1) == 20 *(pa+2) == 30

Pointer Arithmetic

- Pointer arithmetic uses byte size of type pointed to
- Let p be a pointer to an array element
  - **p++** points to next element
  - **p += i** points i elements beyond current element
- Pointers can be compared to 0 (NULL)
  - Operators: ==, !=
- Pointers can be compared to pointers of the same type
  - Operators: ==, !=, <, <=, >, >=, etc.
- Let p and q point to elements of the same array
  - **p < q** is true if p points to an earlier array member than q
  - **q – p** gives the number of elements (not bytes!) of between p and q

Pointers and Arrays

- Value of array a is address of first element
  - Example on the right: p == a, a == q
- p = &a[0] can be written as p = a
- a[i] can be written as *(a+i)
- &a[i] can be written as a+i
- p+i can be written as &p[i]
- p = a and p++ are legal
- a = p and a++ are illegal

- When an array is passed to a function only the address of the first element is passed (by value, copied)
  - Hence, no array length information is passed to function

```
int a[10];
int *p = &a[0];
int *q = a;
```

Pointers and Arrays, Example: String Length

- Returns length of null-terminated string

```
int strlen2(char *s) {
    int n = 0;
    for (; *s != '\0'; s++) {
        n++;
    }
    return n;
}

char greeting[ ] = "hello";
int n = strlen2(greeting); // argument greeting is an array
```

argument degenerates to a pointer, even if it is an array

point to next character

leave loop if s points to null-character

Limitations of C-Arrays: Length not Stored

- sizeof operator does not work for parameters

```
void f(int a[]) {
    printf("%lu\n", sizeof(a) / sizeof(int)); // output: 2!
}
int main(void) {
    int a[] = { 1, 2, 3, 4, 5 };
    printf("%lu\n", sizeof(a) / sizeof(int)); // output: 5
    f(a);
    return 0;
}
```

because, array decays into an int*, which has 64 bits

- void f(int a[]) ... actually is... void f(int* a)
- Solution: Supply array length as an additional parameter

Character Arrays and Character Pointers

- A modifiable (mutable) character array
  char messageArray[ ] = “now is the time”;

  messageArray: | now is the time\0 |

- A pointer to a fixed (immutable) array
  char *messagePointer = “now is the time”;

  messagePointer: | • | → | now is the time\0 |

- Assignment assigns memory address, does not copy contents
  messagePointer = “another message”;

Character Arrays and Character Pointers

- Example

```
char messageArray[ ] = "now is the time";  // mutable character array
char *messagePointer = "now is the time"; // immutable character array
printsln(messageArray);
printsln(messagePointer);

messageArray[0] = 'N';    // fine
messagePointer[0] = 'N'; // bus error, crash!
printsln(messageArray);
printsln(messagePointer);
```

Comparing Pointers vs. Comparing Content

- Operators == and != with pointer operands
  compare memory addresses, not content
- Use specific functions to compare content
  - Example: s_equals (prog1lib) or strcmp (stdlib) to compare strings
- Example

```
char *s = "hello";
char *t = "hello";
printbln(s == t); // output: true, same address
char *u = "hello";
char v[] = "hello";
printbln(u == v); // output: false, different addresses
printbln(s_equals(u, v)); // output: true, same content, prog1lib
printiln(strcmp(u, v)); // output: 0 (means equal), standard c library
```

Pointers and Arrays, Example: String Copy

- String copy with pointers

```
void strcpy2(char *dst, char *src) {
    while ((*dst = *src) != '\0') {
        dst++;
        src++;
    }
}
```

In C, an assignment is an expression and has a value (the value that was assigned)

- while-loop can be "simplified" to

```
    while (*dst++ = *src++);
```

Often used idiom. Difficult to read.

Pointers to Structures

- Passing large structures to a function is inefficient
  - Because the whole structure has to be pushed onto the call stack
  - More efficient to a pass pointer to the structure
- Declaring a pointer to a structure

```
struct Point { int x, int y};   // structure type declaration
struct Point p = { 100, 200 };   // p is a structure variable
struct Point *pp;   // pp is a pointer to a structure variable
pp = &p;   // pp now points to p (pp contains the address of p)
```

- Member operator **->** for pointers to structures

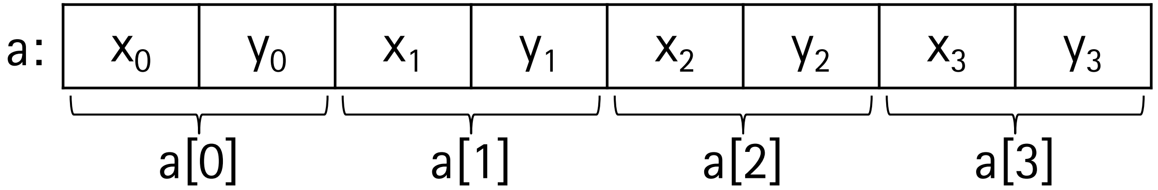
```
printf("x = %d, y = %d\n", pp->x, pp->y);
```

- Which is a shorthand for

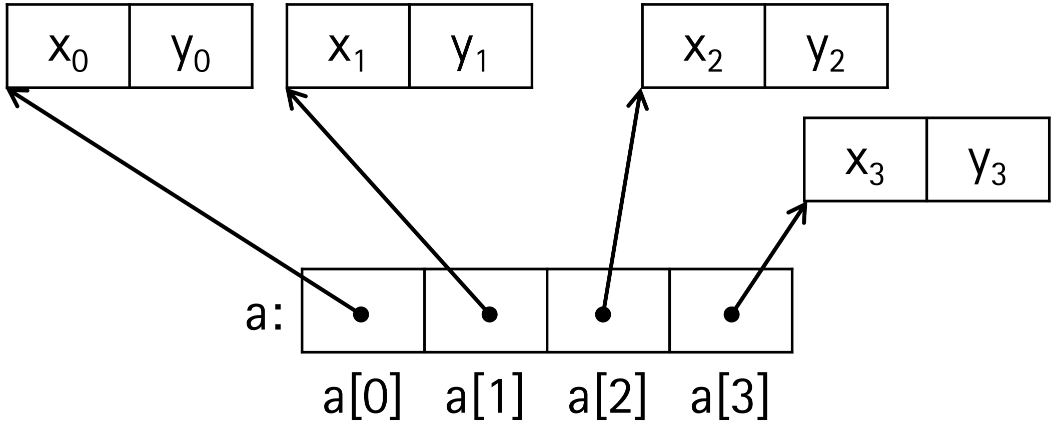
```
printf("x = %d, y = %d\n", (*pp).x, (*pp).y);
```

Array of Structures vs. Array of Pointers to Structures

- Given struct Point { int x, int y};
- Array of Point structures: Point a[4];

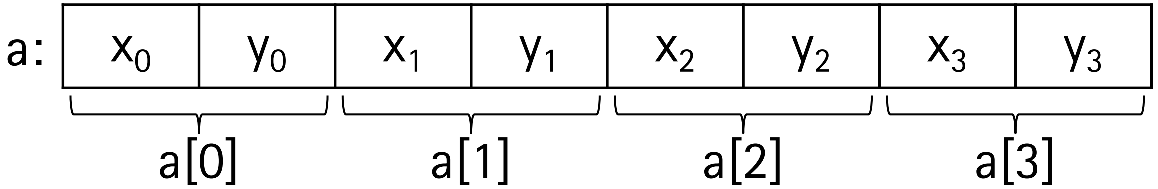


- Array of pointers to Point structures: Point * a[4];

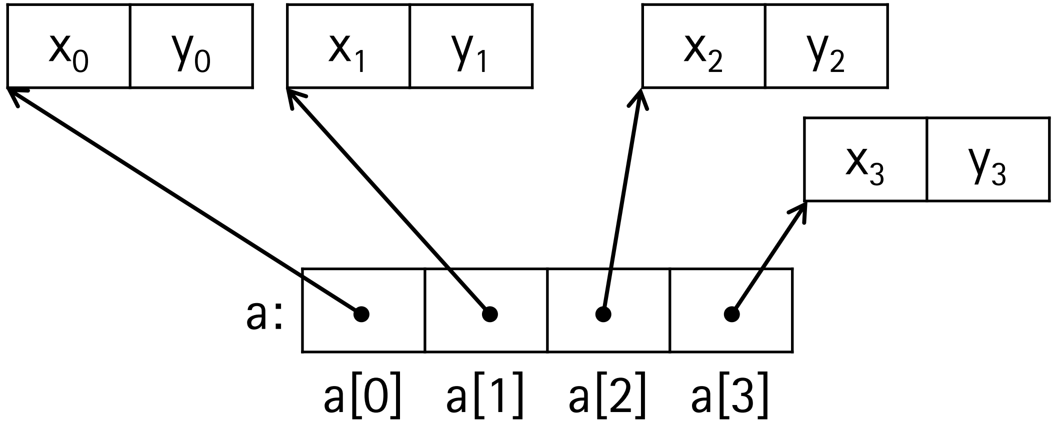


Array of Structures vs. Array of Pointers to Structures

- Array of Point structures: Point a[4];
  Accessing x-coordinate of i^{th} element: a[i].x



- Array of pointers to Point structures: Point * a[4];
  Accessing x-coordinate of i^{th} element: a[i]->x



Two-Dimensional Arrays

- Two-dimensional array: char a[4][5]; // 4 rows, 5 columns
  - a is a 4-element array whose elements are 5-element char arrays
  - 2D array of 20 chars
  - Contiguous in memory
  - a[row][col] is a char
  - a[row] is a char[5]

```
char a[4][5];
println(sizeof(a));        // 20 bytes
println(sizeof(a[0]));     //  5 bytes
println(sizeof(a[0][0]));  //  1 byte
```

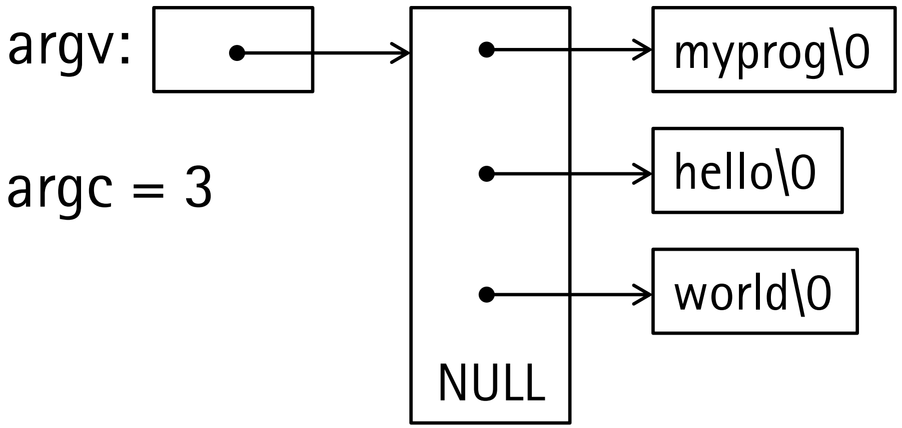
- Two-dimensional arrays are organized by row
  - int *p = (int*) a; // pointer to first element of a
  - These all access the same int element:
  - a[row][col]
  - p[5*row+col]
  - *(p + 5*row+col)

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

COMMAND LINE ARGUMENTS

Command Line Arguments

- Function main optionally takes arguments
  - int main(int argc, String argv[]) { ...}   // or: char* argv[]   or: char** argv
  - argc: argument count (an integer)
  - argv: argument vector (a pointer to an array of strings)
  - argv[0] is typically the program name, and argv[argc] == NULL
- Example:
  - Command line: `myprog.exe hello world`



Command Line Arguments

```c
int main(int argc, char** argv) {
    for (int i = 1; i < argc; i++) {
        printf("[%s] ", argv[i]);
    }
    printf("\n");
    return 0;
}
```

- `./myargv 1 "2 3" 4`
- `[1] [2 3] [4]`

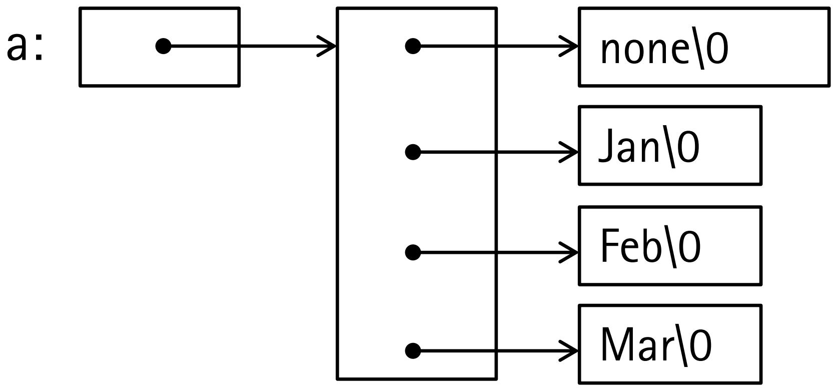
Command Line Arguments

```
int main(int argc, char** argv) {
    for (int i = 1; i < argc; i++) {
        printf("[%s] ", argv[i]);
    }
    printf("\n");
    return 0;
}
```

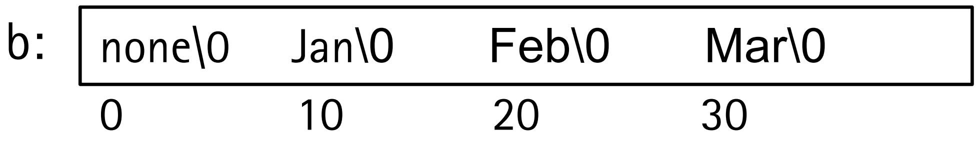
- `./myargv hello\ world`
- `[hello world]`

Arrays of Pointers vs. Two-Dimensional Arrays

- Array of pointers
  - char * a[ ] = { "none", "Jan" , "Feb" , "Mar"};



- Two-dimensional array
  - char b[ ][10] = { "none", "Jan" , "Feb" , "Mar"};



STRING AND CHARACTER FUNCTIONS

String is defined as char*

- A String is defined as a pointer to its first character
  - typedef char* String;   (see basedefs.h)
  - char* is the usual string representation in C
  - Standard C library functions expect char*

- End of string marked by character '\0'
  + Simplicity: Any point in memory may be interpreted as a string
  – Danger: A missing '\0' character results in hard to find bugs
  – Efficiency: Determining the length of a string requires finding '\0'

- Ambiguity: char* may also be a pointer to byte-sized binary data,
  or a pointer to a single char

Implementation of s_get and s_set (prog1lib)

```
typedef char* String;

char s[ ] = "abc";          // modifiable character array

char c = s_get(s, 1);       // get character at index 1 ('b'), also checks bounds
char d = s[1];              // get character at index 1 ('b')

s_set(s, 1, 'x');           // set character at index 1 to 'x', also checks bounds
s[1] = 'x';                 // set character at index 1 to 'x'
```

Implementation of s_get (prog1lib)

Documentation tool: https://www.doxygen.nl

```
/**
Returns character at index i.
@param[in] s input string
@param[in] i index of character to return
@return character at index i
@pre "index in range", i >= 0 && i < length
*/
char s_get(String s, int i) {
  require_not_null(s);
  int n = strlen(s);
  require_x("index in range", i >= 0 && i < n, "index == %d, length == %d", i, n);
  return s[i];
}
```

Implementation of s_set (prog1lib)

Documentation tool: https://www.doxygen.nl

```
/**
Sets s element at index i to value v.
@param[in,out] s input string
@param[in] i index of character to set
@param[in] c character to set
@pre "index in range", i >= 0 && i < length
*/
void s_set(String s, int i, char v) {
  require_not_null(s);
  int n = strlen(s);
  require_x("index in range", i >= 0 && i < n, "index == %d, length == %d", i, n);
  s[i] = v;
}
```

Pointers and Arrays: String Compare

- return <0 if s lexicographically smaller than t
- return 0 if string s is equal to string t
- otherwise return a value >0

```c
int strcmp2(char *s, char *t) {
    require_not_null(s);
    require_not_null(t);
    int i;
    for (i = 0; s[i] == t[i]; i++) {
        if (s[i] == '\0') return 0;
    }
    return s[i] - t[i];
}
```

Pointers and Arrays: String Compare

- return <0 if s lexicographically smaller than t
- return 0 if string s is equal to string t
- otherwise return a value >0

```c
int strcmp3(char *s, char *t) {
    require_not_null(s);
    require_not_null(t);
    for ( ; *s == *t; s++, t++) {
        if (*s == '\0') return 0;
    }
    return *s - *t;
}
```

The GNU C Library – String Operations

- #include <string.h>
- char * s = ...; contains a null-terminated string
- char * t = ...; contains a null-terminated string
- char c = ...; int n = ...;
- strcat(s, t) append t to end of s
- strcmp(s, t) <0 if s<t, ==0 if s==t, >0 if s>t
- strcpy(s, t) copy t to s (overwrites s)
- strncpy(s, t, n) copy first n chars from t to s (overwrites s)
- strlen(s) length of s (not counting terminating '\0')
- strchr(s, c) find c in s
- strrchr(s, c) find last c in s

http://www.gnu.org/software/libc/manual/html_node/String-and-Array-Utilities.html

The GNU C Library – Character Class Testing

- #include <ctype.h>
- char c;
- isalpha(c) non-zero if c is alphabetic
- isupper(c) non-zero if c is upper case
- islower(c) non-zero if c is lower case
- isdigit(c) non-zero if c is a digit
- isalnum(c) non-zero if c is alphabetic or a digit
- isspace(c) non-zero if c is whitespace (blank, tab, newline, etc.)
- toupper(c) convert c to upper case
- tolower(c) convert c to lower case

http://www.gnu.org/software/libc/manual/html_node/Classification-of-Characters.html

The GNU C Library – Math Functions

- #include <math.h>
- double x, y
- sin(x) sine of x, x in radians
- cos(x) cos of x, x in radians
- atan2(y,x) arctangent of y/x in radians
- exp(x) e^x
- log(x) $\ln(x)$, $x > 0$
- log10(x) $\log_{10}(x)$, $x > 0$
- pow(x,y) x^y
- sqrt(x) square root of x ($x \geq 0$)
- fabs(x) $|x|$

http://www.gnu.org/software/libc/manual/html_node/Mathematics.html

DYNAMIC MEMORY ALLOCATION

Dynamic Memory Allocation

- Sometimes, variable amounts of memory are required
- Example:
  - Implement function `String stars(int n)`, which produces a string of n stars
  - stars(2) → `"**"`, stars(8) → `"********"`, etc.
- Solution: Dynamic memory: Request memory as needed
- Example:
  - Request memory for n + 1 characters (for ASCII: n + 1 bytes)
  - Fill memory with `'*'`, terminate with `'\0'`
- How to request and release memory dynamically?
  - Request memory: `void* malloc(int n)`
  - Release memory: `free(void* p)`

Dynamic Stars

```c
#include "base.h"

// typedef char* String;

// Returns a string of n stars.
String stars(int n) {
    require("not negative", n >= 0);
    char *s = xmalloc(n + 1);
    for (int i = 0; i < n; i++) {
        s[i] = '*';
    }
    s[n] = '\0'; // end-marker for string
    return s;
}
```

```c
void stars_test(void) {
    String s;
    test_equal_s(s = stars(0), ""); // dynamically allocates 1 byte
    free(s);
    test_equal_s(s = stars(1), "*"); // dynamically allocates 2 bytes
    free(s);
    test_equal_s(s = stars(3), "***"); // dynamically allocates 4 bytes
    free(s);
}
int main(void) {
    report_memory_leaks(true);
    stars_test();
    return 0;
}
```

checks if allocated memory is released

Acquiring Dynamic Memory with malloc/calloc

- Allocate n bytes of memory:
  void* malloc(int n); // #include <stdlib.h> (or "base.h")
  - Allocates a block of n bytes
  - Returns pointer to start of block
  - Does not clear memory (may contain "garbage")
  - Returns NULL if no memory available
- Allocate and clear n elements, each of size s:
  void* calloc(int n, int s); // #include <stdlib.h> (or "base.h")
  - Allocates n items, each of size s bytes (n * s bytes)
  - Returns pointer to start block
  - Clears each byte to 0
  - Returns NULL if no memory available

Acquiring Dynamic Memory with xmalloc/xcalloc (prog1lib)

- Allocate n bytes of memory:
  void* xmalloc(int n); // #include "base.h"
  - Allocates a block of n bytes
  - Returns pointer to start of block
  - Does not clear memory (may contain "garbage")
  - Stops the program with an error message if no memory available
- Allocate and clear n elements, each of size s:
  void* xcalloc(int n, int s); // #include "base.h"
  - Allocates n items, each of size s bytes (n * s bytes)
  - Returns pointer to start block
  - Clears each byte to 0
  - Stops the program with an error message if no memory available

malloc vs. xmalloc

- malloc returns NULL if not enough memory
  - Very rare case, but needs to be handled in production code

```
int * p;
if ((p = malloc(100 * sizeof(int))) == NULL) {
    // handle error
}
// use allocated memory
```

- xmalloc exits the program if not enough memory
  - xmalloc is not standard → use it for debugging, but not in production code
  - allows defining a function that gets called on error before exiting, so error handling is possible
  - xmalloc defined by prog1lib to keep track of allocations

Releasing Dynamic Memory with free

- Release memory area when no longer needed
- Each allocation needs a corresponding call to free

- Release dynamic memory:
  void free(void* p); // #include <stdlib.h> (or "base.h")

- Manual memory management is complex and error prone
- Failing to release memory → memory leaks

Memory Leaks Example

```c
1.  #include "base.h"
2.  int main(void) {
3.      report_memory_leaks(true); // turn on checking memory leaks
4.      char *s = xmalloc(1); // allocates 1 byte
5.      s[0] = '\0';
6.      char *t = xmalloc(2); // allocates 2 bytes
7.      t[0] = 'x';
8.      t[1] = '\0';
9.      return 0;
10. }
```

Output:

```
   2 bytes allocated in main (leak.c at line 6) not freed
   1 bytes allocated in main (leak.c at line 4) not freed
2 memory leaks, 3 bytes total
```

Memory Leaks Example

```
1.  #include "base.h"
2.  int main(void) {
3.      report_memory_leaks(true); // turn on checking memory leaks
4.      char *s = xmalloc(1); // allocates 1 byte
5.      s[0] = '\0';
6.      free(s); // free
7.      char *t = xmalloc(2); // allocates 2 bytes
8.      t[0] = 'x';
9.      t[1] = '\0';
10.     return 0;
11. }
```

Output:

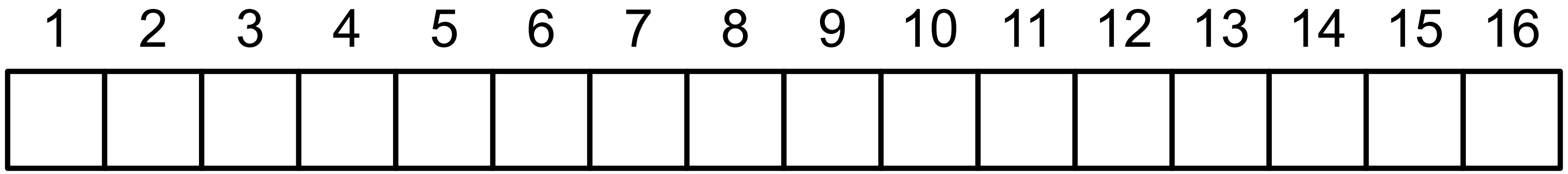
```
    2 bytes allocated in main (leak.c at line 7) not freed
1 memory leak, 2 bytes total
```

Memory Leaks Example

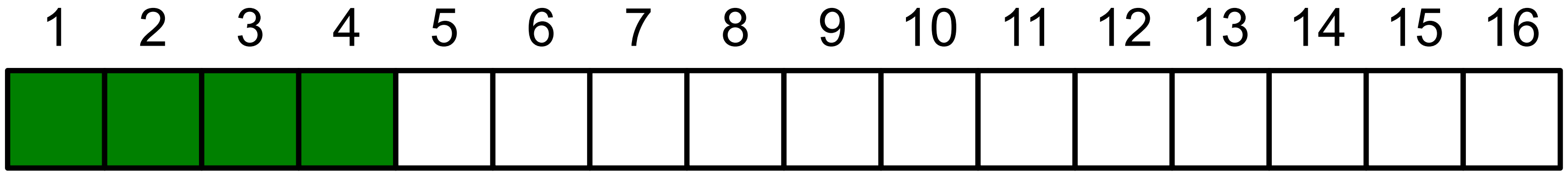
```
#include "base.h"
int main(void) {
    report_memory_leaks(true); // turn on checking memory leaks
    char *s = xmalloc(1); // allocates 1 byte
    s[0] = '\0';
    free(s); // free
    char *t = xmalloc(2); // allocates 2 bytes
    t[0] = 'x';
    t[1] = '\0';
    free(t); // free
    return 0;
}
```

Output:

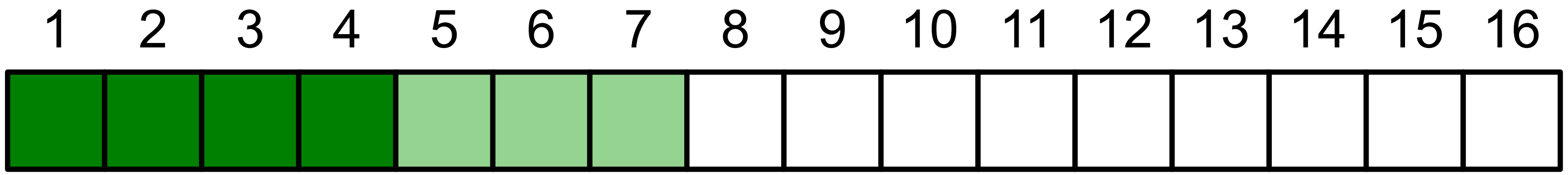
Dynamic Storage (Heap)



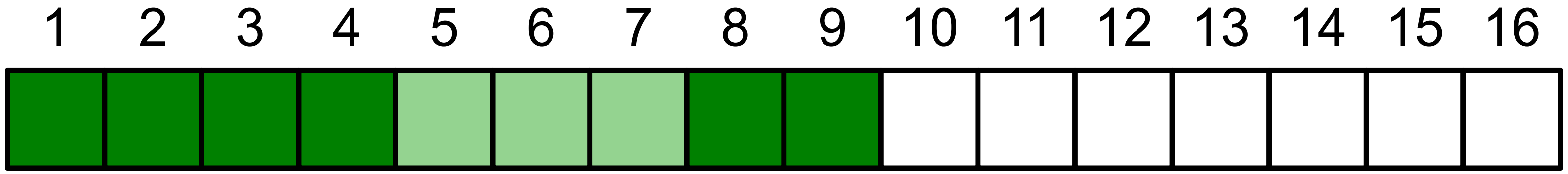
Byte *p1 = xmalloc(4); → p1 == 1 // address of first byte



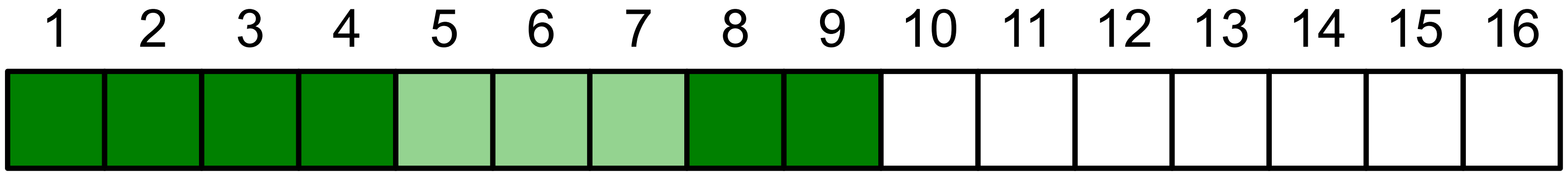
Byte *p2 = xmalloc(3); → p2 == 5



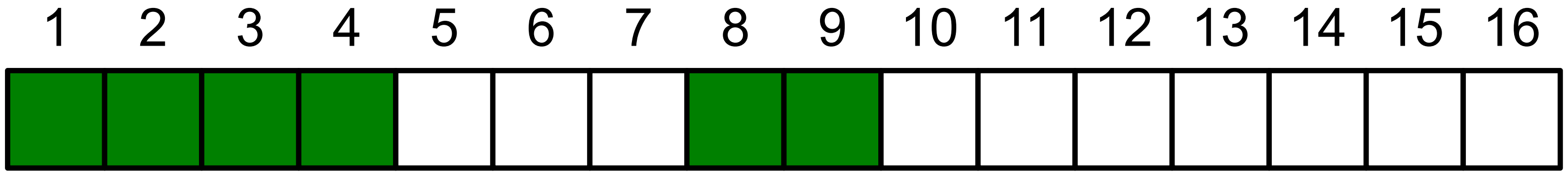
Byte *p3 = xmalloc(2); → p3 == 8



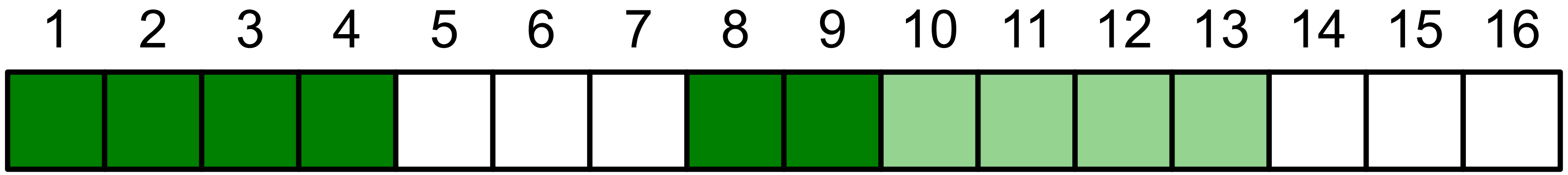
Dynamic Storage (Heap)



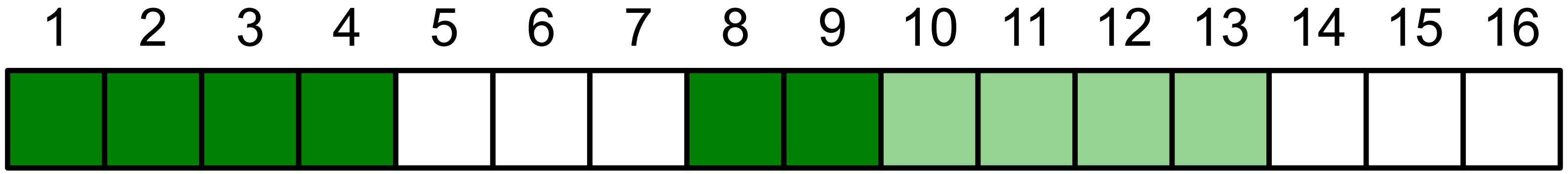
free(p2); (p2 == 5)



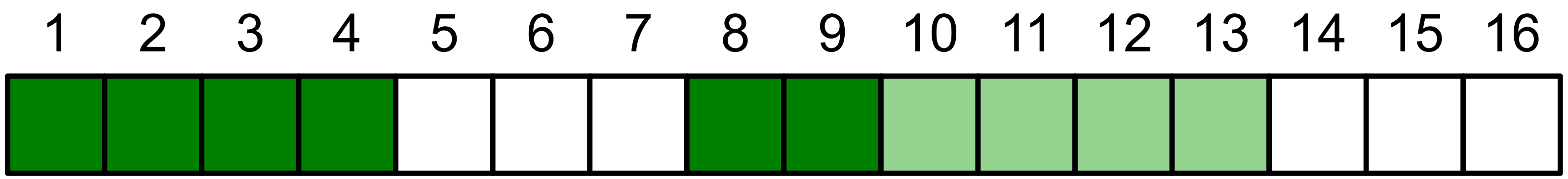
Byte *p4 = xmalloc(4); → p4 = 10



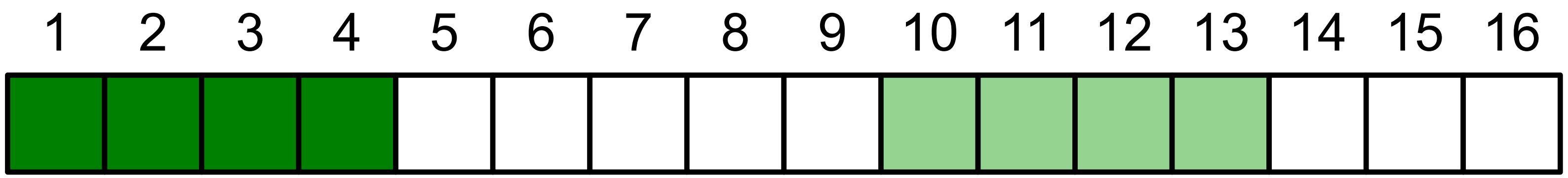
Byte *p5 = malloc(4); → p5 == NULL



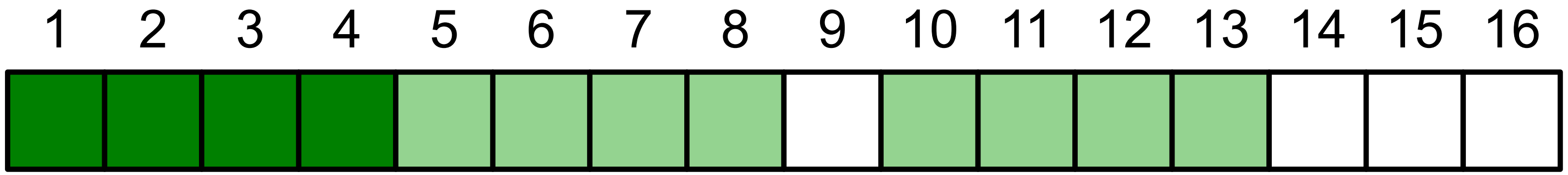
Dynamic Storage (Heap)



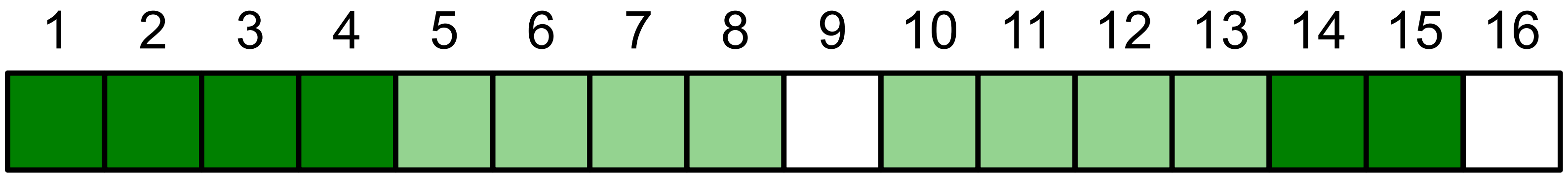
free(p3); (p3 == 8)



Byte *p6 = xmalloc(4); → p6 == 5

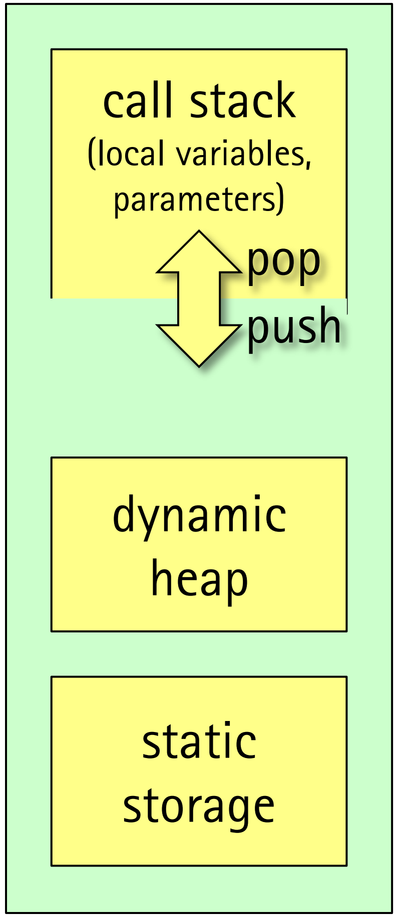


Byte *p7 = xmalloc(2); → p7 == 14



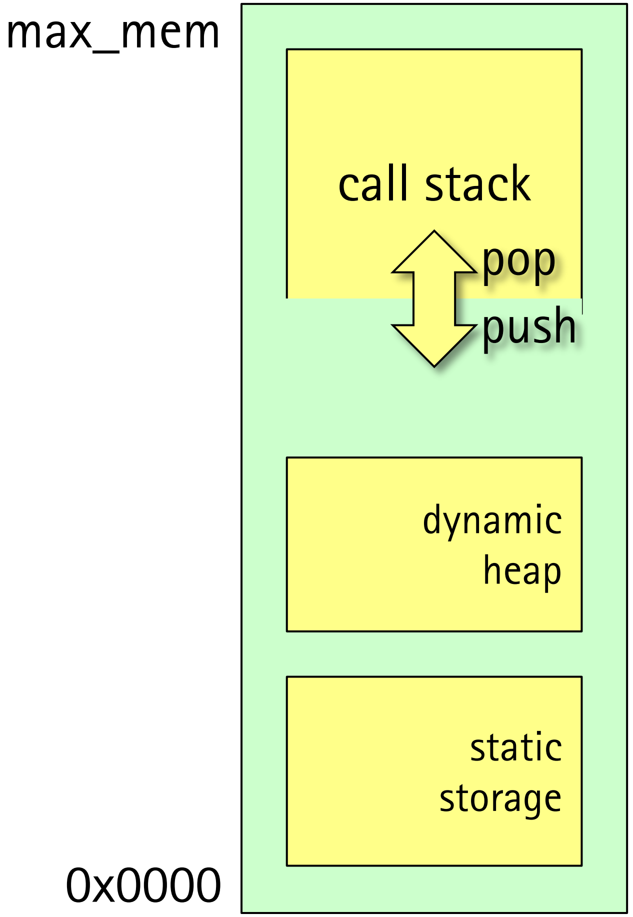
Memory in C: Automatic, Static, Dynamic

- Automatic storage (call stack)
  - Automatic-local variables and function parameters
  - Lifetime linked to block, enables recursion
- Static storage
  - File-level and static-local variables
  - Lifetime equals program execution time
  - Size known at compile time
- Dynamic storage (heap)
  - Memory blocks allocated with (x)malloc / (x)calloc
  - Lifetime controllable (between malloc and free)
  - Size known at runtime (e.g., depending on input)
  - Referenced through pointers



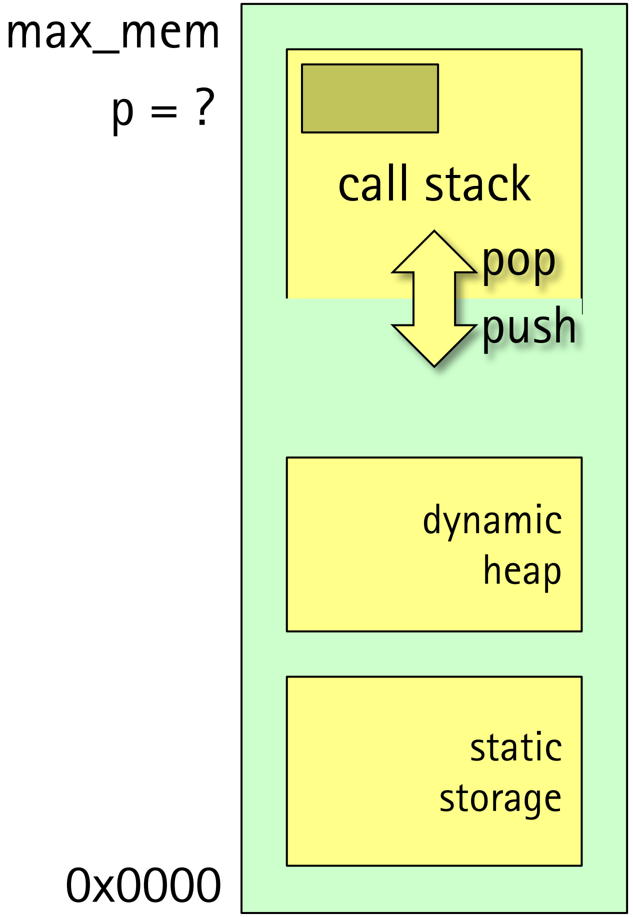
Memory in C: Automatic, Static, Dynamic

```c
#include "base.h"

int main(void)
{
    int *p;
    p = xmalloc(sizeof(int));
    *p = 123;
    printiln(*p);
    free(p);
    return 0;
}
```



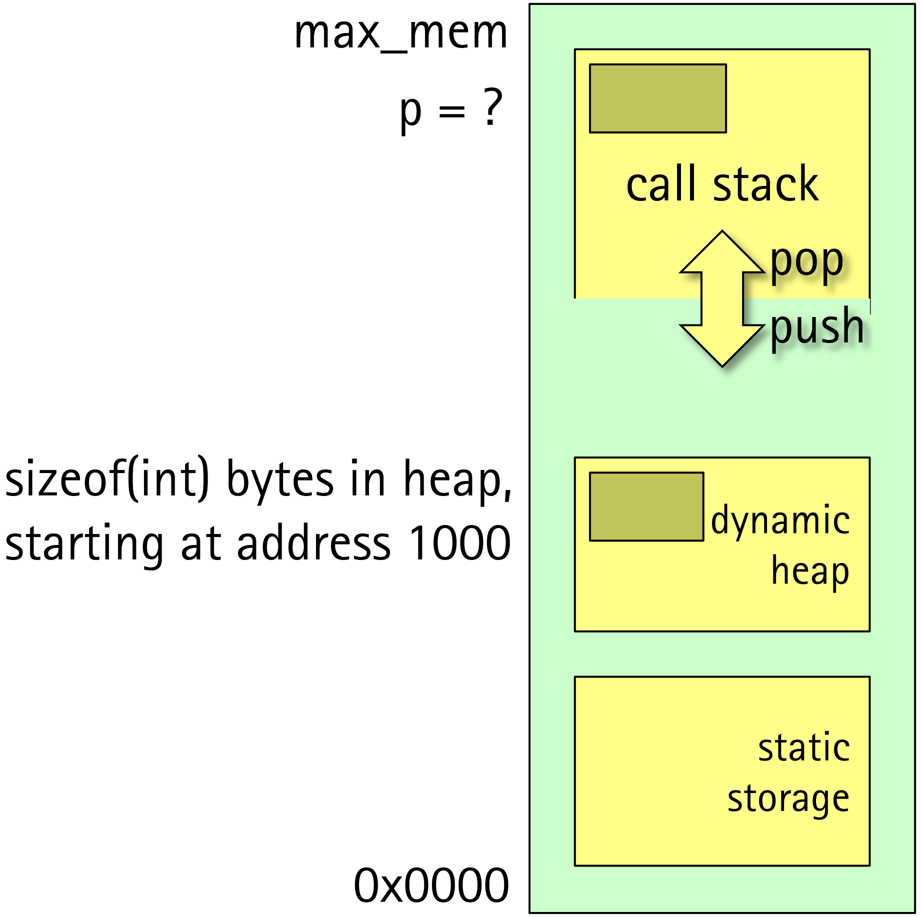
Memory in C: Automatic, Static, Dynamic

```c
#include "base.h"

int main(void)
{
    int *p;
    p = xmalloc(sizeof(int));
    *p = 123;
    printiln(*p);
    free(p);
    return 0;
}
```



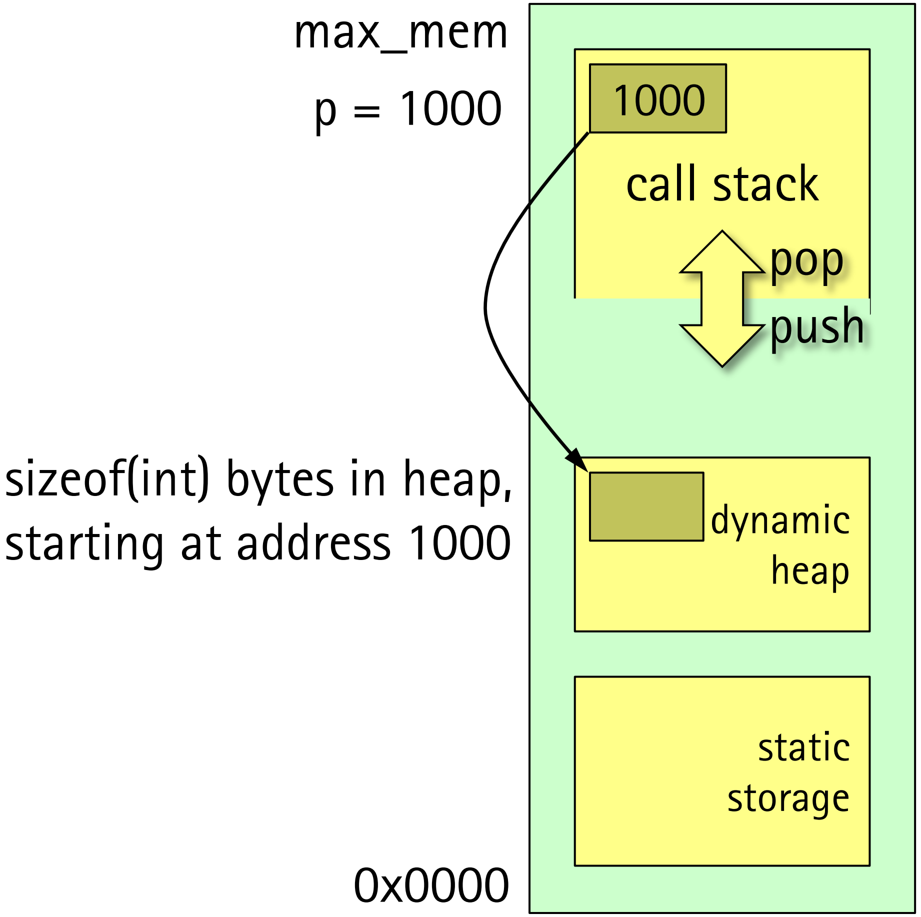
Memory in C: Automatic, Static, Dynamic

```
#include "base.h"

int main(void)
{
    int *p;
    p = xmalloc(sizeof(int));
    *p = 123;
    println(*p);
    free(p);
    return 0;
}
```



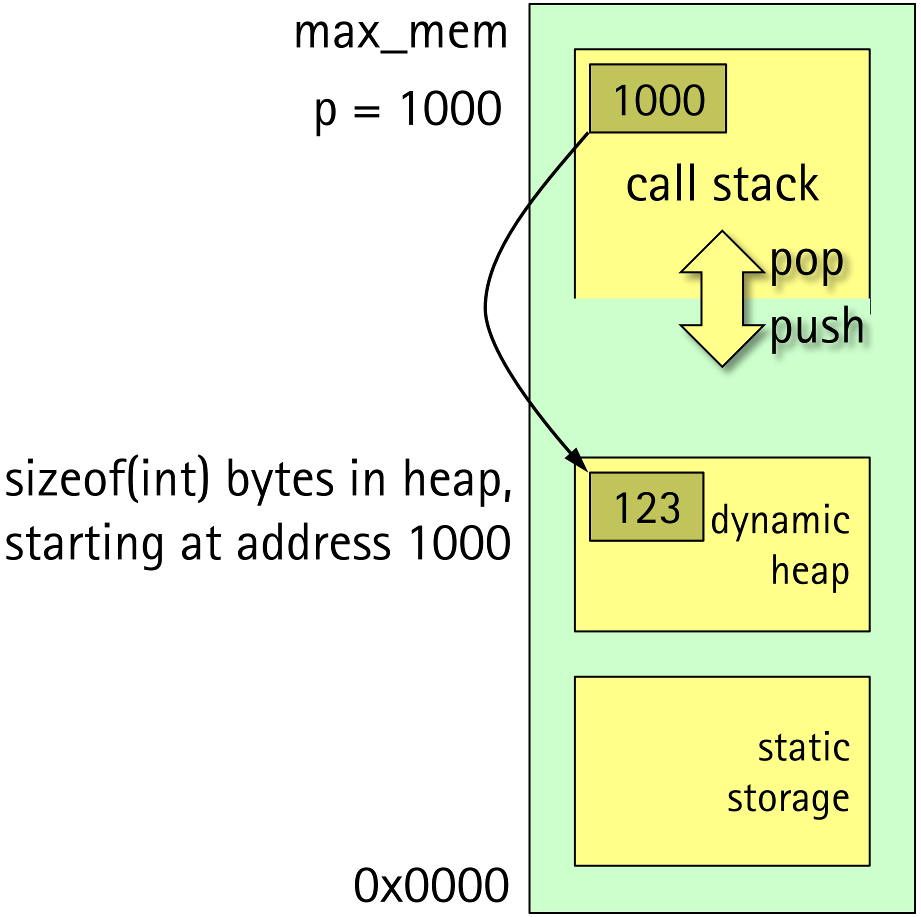
Memory in C: Automatic, Static, Dynamic

```
#include "base.h"

int main(void)
{
    int *p;
→   p = xmalloc(sizeof(int));
    *p = 123;
    println(*p);
    free(p);
    return 0;
}
```



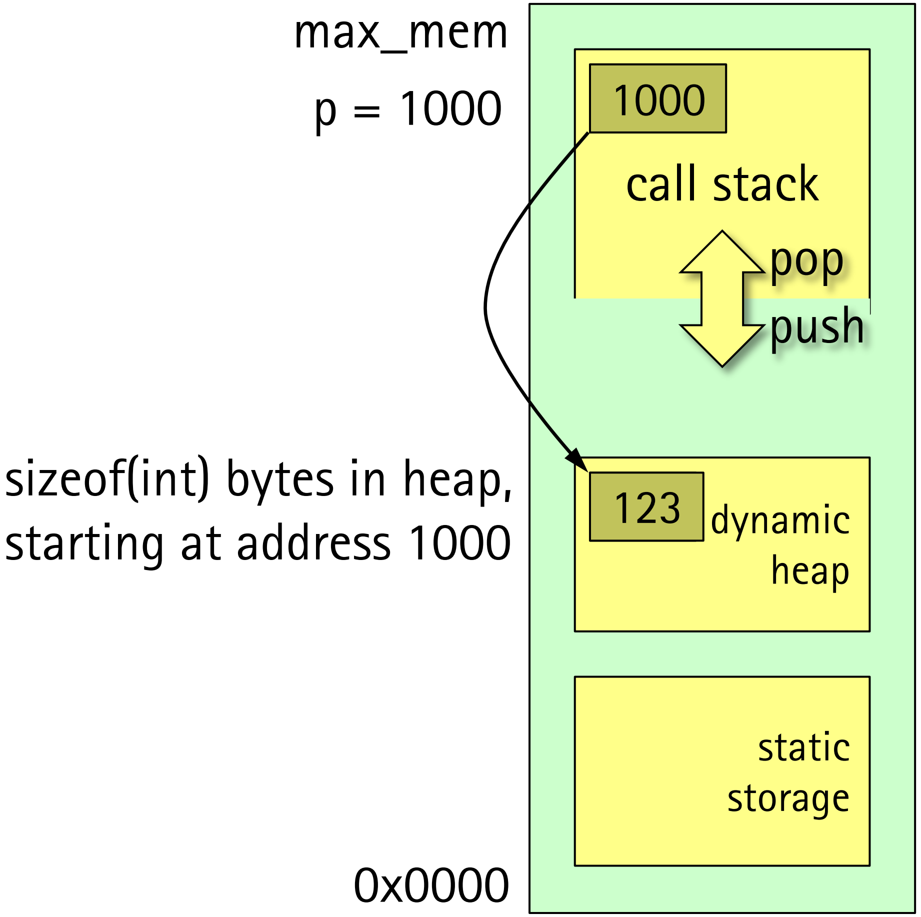
Memory in C: Automatic, Static, Dynamic

```
#include "base.h"

int main(void)
{
    int *p;
    p = xmalloc(sizeof(int));
→   *p = 123;
    println(*p);
    free(p);
    return 0;
}
```



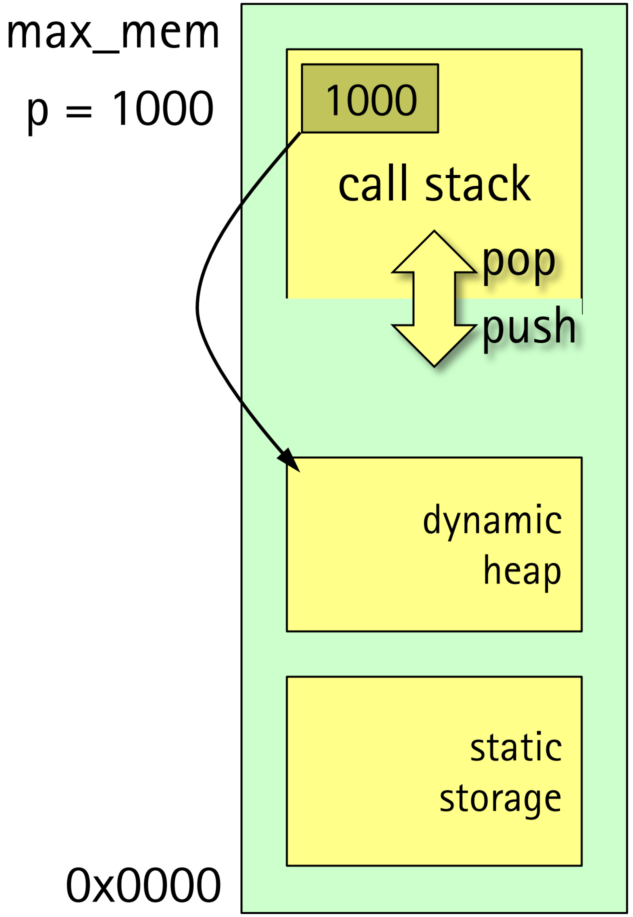
Memory in C: Automatic, Static, Dynamic

```
#include "base.h"

int main(void)
{
    int *p;
    p = xmalloc(sizeof(int));
    *p = 123;
→   println(*p);
    free(p);
    return 0;
}
```



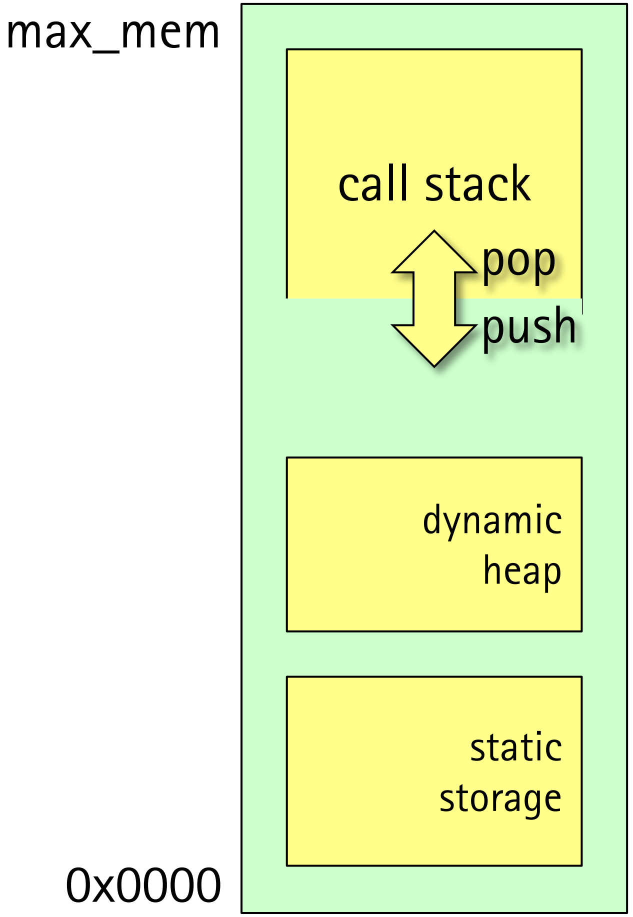
Memory in C: Automatic, Static, Dynamic

```
#include "base.h"

int main(void)
{
    int *p;
    p = xmalloc(sizeof(int));
    *p = 123;
    println(*p);
→   free(p);
    return 0;
}
```



Memory in C: Automatic, Static, Dynamic

```c
#include "base.h"

int main(void)
{
    int *p;
    p = xmalloc(sizeof(int));
    *p = 123;
    println(*p);
    free(p);
→   return 0;
}
```



Dynamic Memory Allocation Errors

```
int *p = xmalloc(3 * sizeof(int));
p[0] = 0; p[3] = 3; // use beyond allocated area
free(p); printf("%d", p[0]); // use memory that has been freed

p = xmalloc(sizeof(int));
p[0] = 1; free(p); free(p); // try to free already freed memory

int *q; // q on stack, not initialized
*q = 1; // try to dereference uninitialized pointer
q = xmalloc(sizeof(int));
free(q+1); // try to free address that was never allocated
```

errors marked in red

Dynamic Memory Allocation Errors

```
int a[5]; a[0] = 1; free(a);       // a was not dynamically allocated
int i = 1; int *p = &i;
free(p);                           // p was not dynamically allocated

p = xmalloc(sizeof(int));
p = &i;                            // memory leak, no reference to allocated memory

while (1) xmalloc(100000);         // will exhaust memory
```

Dynamic Memory Allocation Errors

```
int **p = xmalloc(3 * sizeof(int*));
p[0] = xmalloc(10 * sizeof(int));
p[1] = xmalloc(10 * sizeof(int));
p[2] = xmalloc(10 * sizeof(int));
free(p); // failed to free the elements p[0], p[1], p[2]
free(p[0]); free(p[1]); free(p[2]); free(p); // fix of previous line
```

Note: for each allocation there needs to be a corresponding call of free!

Dynamic Memory Allocation Errors

```
void f(void) {
    char *p = xmalloc(2 * sizeof(char));
    p[0] = 'x';
    p[1] = '\0';
    printf("%s", p);        // local variable is not freed before returning
}                           // p no longer accessible, memory leak!
                            // lost information, where allocated memory block begins

int main(void) {
    f();
    return 0;
}
```

Dynamic Memory Allocation Errors

```
char* f(void) {
    char *p = xmalloc(2 * sizeof(char));
    p[0] = 'x'; p[1] = '\0';
    printf("%s", p);
    return p; // now caller can free memory
}

int main(void) {
    char *s = f();                  // caller should free s
    return 0;                       // but fails to do so
}
```

Establish conventions of who is responsible to free dynamically allocated memory!

This is a hard problem!

RECIPE FOR SELF-REFERENTIAL DATA (RECURSIVE TYPES)

Recipe for Self-Referential Data (Recursive Types)

- Represent data that can take on one of different variants at least one of which is self-referential
  - A special case of variant data
  - Recursive types: The type to be defined is mentioned in its definition
- Self-referential data can represent information of arbitrary size
- Examples
  - A list is either empty (variant 1) or a value followed by a list (variant 2)
  - A binary tree is either empty (variant 1) or a left binary tree, a value, and a right binary tree (variant 2)
- Data definition
  - C structs can be used to represent self-referential data

Recursive Structs?

```
struct MyStruct {
    double x;
    ~~struct MyStruct s;~~
};
```

- Structs cannot contain themselves. Why?
  - sizeof(struct MyStruct)?
- Solution?
- Structs can contain pointers to themselves

```
struct MyStruct {
    double x;
    struct MyStruct * s; // self-reference
};
```

- sizeof(struct MyStruct)?

1. Problem Statement

- Write down the problem statement as a comment.
  - What information needs to be represented?
  - What should the function (to be implemented) do with the data?
  - What cases need to be considered?
- Example

```
/*
Computes the sum of the values of a list of integer numbers.
*/
```

2a. Data Definition

- How should domain information be represented as data in the program? How to interpret the data as real-world information?
- Data definition
  - Determine and name the variants (here: `Null` and `Pair`)
  - Identify self-references
  - Determine the types in each variant
    - `End`: empty parameter list, not self-referential
    - `Pair`: value is an integer number, rest is the self-reference to `List`

2a. Data Definition

```
struct Pair {
    int value;
    struct Pair* rest; // self-reference or NULL
};

struct IntList {
    Pair* first; // first element or NULL
};

typedef struct IntList IntList;
typedef struct Pair Pair;
```

pointer can be NULL, so represents both variants

IntList structure, not strictly necessary

2a. Data Definition: Constructor Functions

```
// Create a list node.
Pair* make_pair(int value, Pair* rest) {
    Pair* c = xcalloc(1, sizeof(Pair));
    c->value = value;
    c->rest = rest;
    return c;
}

// Create a list of integer values.
IntList make_list(void) {
    IntList l = { NULL };
    return l;
}
```

```
struct Pair {
    int value;
    struct Pair* rest;
};
```

2b. Example Values for Data Definition

- Create at least one example value per variant in the data definition
- Create examples that use the self-referential variant(s) more than once (i.e., create examples of different lengths)
- Examples
  - `Pair* list = NULL; // variant 1 (base case)`
  - `list = make_pair(10, NULL); // variant 2 and then variant 1`
  - `list = make_pair(10, make_pair(20, NULL)); // variants 2, 2, 1`
  - `list = make_pair(10, make_pair(20, make_pair(30, NULL))); // vars. 2, 2, 2, 1`

3. Function Name and Parameter List

- Find a good function name
  - Short, non-abbreviated, descriptive name that describes what the function does
- Find good parameter names
  - Short, non-abbreviated, descriptive name that describes what the parameter means
- Write function header
- Example

```
int sum(Pair* list);
```

4a. Function Stub

- Function stub returns an arbitrary value from the function's range
- The function stub can be executed
- Example

```
int sum(Pair* list) {
    return 0;
}
```

4b. Purpose Statement

- Briefly describes what the function does (not how!). Ideally as a single sentence. Multiple sentences may be necessary.
- Example

```
// Computes the sum of the values of the list.
int sum(Pair* list) {
    return 0;
}
```

5b. Examples and Expected Results (Test Function)

- Write several examples with expected results, at least one per variant in the data definition
  - Use the example values created before (in 2b)

```
void sum_test() {
    Pair* list = NULL; // empty list
    test_equal_i(sum(list), 0);
    list = make_pair(10, NULL); // one-element list: 10
    test_equal_i(sum(list), 10);
    list = make_pair(10, make_pair(20, NULL)); // two-element list: 10, 20
    test_equal_i(sum(list), 30);
    list = make_pair(10, make_pair(20, make_pair(30, NULL))); // 10, 20, 30
    test_equal_i(sum(list), 60);
}
```

6. Template

- Translate the data definition into a template
- Use if-else to handle the different variants
  - Conditions: Write one if-condition per variant
  - Actions: Access members relevant for the respective variant
  - Actions: Add one recursive call per self-reference

6. Template

- Data definition

```
struct Pair {
    int value;
    struct Pair* rest; // self-reference or NULL
};
```

self-reference

- Translate the data definition into a template

```
int sum(Pair* list) {
    if (list == NULL) {
        ...
    } else {
        ... list->value ... sum(list->rest) ...
    }
}
```

non-recursive base case

recursive case

self-reference

recursive call on self-reference

6. Function Body

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
// Computes the sum of the values of the list.
int sum(Pair* list) {
    if (list == NULL) {
        ...
    } else {
        ... list->value ...
        ... sum(list->rest) ...
    }
}
```

6. Function Body

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
// Computes the sum of the values of the list.
int sum(Pair* list) {
    if (list == NULL) {
        return 0;
    } else {
        ... list->value ...
        ... sum(list->rest) ...
    }
}
```

purpose statement

sum of empty list is 0 (base case)

assume that sum already does what the purpose statement says

then do induction step ("leap of faith")

6. Function Body

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
// Computes the sum of the values of the list.
int sum(Pair* list) {
    if (list == NULL) {
        return 0;
    } else {
        return list->value +
            sum(list->rest);
    }
}
```

purpose statement

sum of empty list is 0 (base case)

if sum(rest) = r (induction hypothesis), then sum(pair(v, rest)) = v + r (induction step)

7. Testing

- Call test function

```
int main(void) {
    sum_test();
    return 0;
}
```

- Test results

```
list.c, line 68: check passed
list.c, line 70: check passed
list.c, line 72: check passed
list.c, line 74: check passed
All 4 tests passed!
```

SELF-REFERENTIAL DATA, EXAMPLE 2

1. Problem Statement (Example 2)

- Write down the problem statement as a comment.
  - What information needs to be represented?
  - What should the function (to be implemented) do with the data?
  - What cases need to be considered?
- Example

```
/*
Write a function that determines whether
a value is present in a list of integer numbers.
 */
```

2. Data Definition

```
struct Pair {
    int value;
    struct Pair* rest; // self-reference or NULL
};
typedef struct Pair Pair;


struct IntList {
    Pair* first; // first element or NULL
};
typedef struct IntList IntList;
```

3. Function Name and Parameter List (Example 2)

- Find a good function name
  - Short, non-abbreviated, descriptive name that describes what the function does
- Find good parameter names
  - Short, non-abbreviated, descriptive name that describes what the parameter means
- Write function header
- Example

```
bool contains(Pair* list, int x);
```

4a. Function Stub (Example 2)

- Function stub returns an arbitrary value from the function's range
- The function stub can be executed
- Example

```
bool contains(Pair* list, int x) {
    return false;
}
```

4b. Purpose Statement (Example 2)

- Briefly describes what the function does (not how!). Ideally as a single sentence. Multiple sentences may be necessary.
- Example

```
// Returns true iff (if and only if) list contains x.
bool contains(Pair* list, int x) {
    return false;
}
```

5. Examples and Expected Results (Example 2)

- Write several examples with expected results, at least one per variant in the data definition
  - Use the example values created before (in 2b)

```
void containts_test(void) {
    Pair* list = NULL;
    test_equal_i(contains(list, 0), false); // empty list
    list = make_pair(10, NULL); // one-element list: 10
    test_equal_i(contains(list, 10), true);
    test_equal_i(contains(list, 11), false);
    list = make_pair(10, make_pair(20, NULL)); // 10, 20
    test_equal_i(contains(list, 20), true);
    test_equal_i(contains(list, 21), false);
}
```

6. Template (Example 2)

- Data definition

```
struct Pair {
    int value;
    struct Pair* rest; // self-reference or NULL
};
```

self-reference

- Translate the data definition into a template

```
bool contains(Pair* list,
    if (list == NULL) {
        ...
    } else {
        ... list->value ... contains(list->rest, x) ...
    }
}
```

non-recursive base case

recursive case

self-reference

recursive call on self-reference

6. Function Body (Example 2)

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
// Returns true iff list contains x.
bool contains(Pair* list, int x) {
    if (list == NULL) {
        ...
    } else {
        ... list->value ...
        ... contains(list->rest, x) ...
    }
}
```

6. Function Body (Example 2)

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
// Returns true iff list contains x.
bool contains(Pair* list, int x) {
    if (list == NULL) {
        return false;
    } else {
        ... list->value ...
        ... contains(list->rest, x) ...
    }
}
```

empty list does not contain anything (base case)

check whether x is first element or whether rest contains x

6. Function Body (Example 2)

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
// Returns true iff list contains x.
bool contains(Pair* list, int x) {
    if (list == NULL) {
        return false;
    } else {
        if (list->value == x) return true;
        else return contains(list->rest, x);
    }
}
```

empty list does not contain anything (base case)

check whether x is the first element or whether the rest contains x

7. Testing (Example 2)

- Call test function

```
int main(void) {
    containts_test();
    return 0;
}
```

- Test results

  list.c, line 91: check passed
  list.c, line 93: check passed
  list.c, line 94: check passed
  list.c, line 96: check passed
  list.c, line 97: check passed
  All 5 tests passed!

8. Review and Revise (Example 2)

- Review the products of the steps
  - Improve function name
  - Improve parameter names
  - Improve purpose statement
  - Improve and extend tests

- Improve / generalize the function
  - Simplify the conditions

8. Review and Revise (Simplify Conditions)

```
bool contains(Pair* list, int x) {
    if (list == NULL) {
        return false;
    } else {
        if (list->value == x) {
            return true;
        } else {
            return contains(list->rest, x);
        }
    }
}
```

simplify conditions

```
bool contains(Pair* list, int x) {
    if (list == NULL) return false;
    if (list->value == x) return true;
    return contains(list->rest, x);
}
```

Convert List to String

```
String list_to_string(Pair* list) {
    if (list == NULL) { // empty list
        return "";
    } else if (list->rest == NULL) { // one-element list
        return s_of_int(list->value);
    } else { // list has two or more elements
        String s = s_of_int(list->value);
        s = s_concat(s, " ");
        s = s_concat(s, list_to_string(list->rest));
        return s;
    }
}
```

Summary

- Pointers and arrays
  - Strong relationship between pointers and arrays, pointer arithmetic
- Command line arguments
  - Input to a program on the command line
- String and character functions
  - typedef char* String;
- Dynamic memory allocation
  - Automatic storage (call stack), static storage, dynamic heap
  - malloc/calloc, xmalloc/xcalloc, free
  - Memory allocation errors
- Recursive types and algorithms
  - Operations on lists


Auch DU kannst Programmieren lernen