Computer Programming

Pointers 2

Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

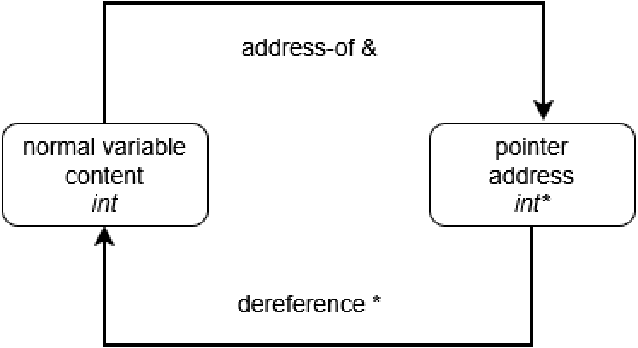
Course 7

Contents

Pointers recap

- pointers are meant to hold addresses of variables
- an address of a variable is a natural number, which shows its position in the memory byte array (simplified view)
- if the data occupies multiple bytes, its address is given by the lowest byte (little-endian order)
- the address can be obtained using the *address-of* operator &
- the content from a given address can be retrieved using the *dereference* operator *

Diagram for pointer operators



Higher order pointers

- higher order pointers appear naturally in applications
- 2nd or 3rd order pointers are common, but pointers can be of any order
- syntax for declaration uses multiple * modifiers:

```
type** pp_name;
type*** ppp_name;
```

- pp_name is a double pointer to *type*, or pointer to pointer
- ppp_name is a triple pointer to *type*
- the number of stars indicate how many times it needs to be dereferenced until we obtain useful content

Program 7.1 - 2nd order pointer

```c
#include <stdio.h>

int main(){
  char c = 'a';
  char* pc = &c; //alfa
  char** ppc = &pc; //beta
  printf("%p %p %c",
    ppc, *ppc, **ppc);
  return 0;
}
```

- c is a normal char variable
- pc is a pointer to char
- ppc is a pointer to pointer to char
- we can dereference ppc two times

Utility

- creating and maintaining multidimensional arrays that have rows of different sizes (non-rectangular/non-cuboid)
  - this is done using dynamic memory allocation
- changing the values of pointers inside functions
  - the address of the pointer should be sent
  - this has the type of double pointer (or higher order)

Program 7.2 - Multidimensional arrays

```
#include <stdio.h>

int main(){
  int a[2] = {1, 2};
  int b[2] = {3, 4};
  int c[3] = {5, 6, 7};
  int* pp[] = {a, b};
  int** q = pp;
  for(int i=0; i<2; i++)
    for(int j=0; j<2; j++)
      printf("%d ", *(*(pp+i)+j));
  q[1] = c;
  printf("%d %d %d",
    q[1][0], q[1][1], q[1][2]);
  return 0;
}
```

- pp contains arrays
- each element is itself a pointer
- both pointer indexing with * and array indexing with [] is valid

Contents

Memory zones

- Stack
    - stores local variables, function arguments, call return addresses
    - maintained automatically
    - life-cycle coincides with function execution
- Global
    - stores code and global variables
- Heap
    - zone destined for large data
    - maintained manually using functions
    - life-cycle determined by the programmer

Heap

- Advantages
  - + variable life controlled by user
  - + we can return arrays from functions
  - + can handle large data (limited only by the physical memory size)
- Disadvantages
  - - additional work and overhead for maintenance
  - - source of bugs
  - - memory leak = unused memory zones from the heap which are not deallocated, especially in applications that run for a long time

Functions for memory management - stdlib.h

```
void* calloc(size_t num, size_t size);
```

- function attempts to allocate a memory block of $num * size$ bytes, with each byte set to 0
- the type `size_t` is equivalent to `unsigned long`
- input parameters
  - `num` = number of elements
  - `size` = the size of a single element in bytes, obtainable with the `sizeof` operator/function
- return value
  - void pointer to the start of the memory zone
  - NULL pointer in case allocation fails

Functions for memory management - stdlib.h

```
void* malloc(size_t size);
```

- function attempts to allocate a memory block of *size* bytes, without initializing it (will likely contain random values)
- the type `size_t` is equivalent to `unsigned long`
- input parameters
  - `size` = the size of the whole block in bytes
- return value
  - void pointer to the start of the memory zone
  - NULL pointer in case allocation fails

Functions for memory management - stdlib.h

```
void* realloc(void* old_block, size_t new_size);
```

- function attempts to resize an already allocated memory block from `old_block` to have size equal to `new_size`
- the type `size_t` is equivalent to `unsigned long`
- input parameters
  - `old_block` = pointer to the existing block allocated previously with calloc, malloc or realloc
  - `new_size` = the size of the new block in bytes
- return value
  - void pointer to the start of the memory zone
  - may return the same location if resize is possible in-place
  - NULL pointer in case allocation fails

Functions for memory management - stdlib.h

```
void free(void* block);
```

- function attempts to deallocate the memory zone from the `block` pointer
- input parameters
  - `block` = pointer to the memory block allocated previously with calloc, malloc or realloc
- possible errors:
  - invalid pointer / block not created via stdlib functions
  - zone already unallocated
  - accessing unallocated zones
- heap memory is deallocated upon program exit

Program 7.3 - 1D dynamic allocation template

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
  int n = 1e6;
  float* a = calloc(n, sizeof(float));
  if (a == NULL){
    puts("error at allocation");
    return -1;
  }
  for(int i=0; i<n; i++)
    a[i] = 1.f/(1+i);
  a = realloc(a, 10);
  if (a == NULL){
    puts("error at reallocation");
    return -2;
  }
  for(int i=0; i<10; i++)
    printf("%f ", a[i]);
  free(a);
  return 0;
}
```

- 1e6 = 10^6 as a double
- void* from allocation functions is implicitly converted to float*
- terminate to program on error
- attempt to resize the array
- free the zone after it is not needed

1D dynamic allocation

- both static allocation on the stack and dynamic allocation of an array produce a pointer to the first element
- the two allocation methods are compatible
- we can send dynamically allocated pointers to functions and we can also return them safely
- returning a dynamically allocated pointer assumes that the parent function will be responsible for freeing the memory zone = source of memory leaks

Example:

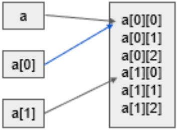
```
int* return_pointer(int n){
    return malloc(n * sizeof(int));
}
```

2D dynamic allocation - overview

- in case of two-dimensional arrays, static allocation on the stack and dynamic allocation on the heap produce incompatible pointers
  - they are of different types
  - the memory layout is also different

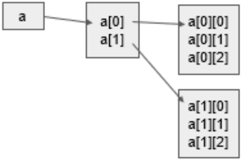
Static allocation (on stack)

```c
int a[2][3];
```



Dynamic allocation (on heap)

```c
int** a = malloc(2 * sizeof(int*));
a[0] = malloc(3 * sizeof(int));
a[1] = malloc(3 * sizeof(int));
```



Program 7.4 - 2D dynamic allocation template

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


int main(){
  int n = 2, m = 3;
  char** a = calloc(n, sizeof(char*));
  for(int i=0; i<n; i++)
    a[i] = calloc(m, sizeof(char));

  for(int i=0; i<n; i++)
    for(int j=0; j<m; j++)
      a[i][j] = '#';
  *(*(a+1)+1) = '.';

  for(int i=0; i<n; i++)
    free(a[i]);
  free(a);
  return 0;
}
```

- first we allocate space for row pointers
- the we allocate arrays for each row
- both array indexing with [] and pointer indexing with * work
- deallocation proceeds in reverse order: first the rows, then the array with row pointers

Contents

Pointers to functions

- a function pointer holds the address where the code of the function is located
- permits us to send functions as parameters to other functions
- typical syntax for declaring a function pointer:

```
type (*function_pointer_name) (type1, type2, ...);
```

- `type` is the type returned by the function
- `type1, type2, ...` are input parameter types of the function
- the * operator must be applied first on the `function_pointer_name`

Function pointer declaration examples

- pointer to a function that doesn't return anything and has no input parameters

```
void (*f)();
```

- pointer to a function that returns int and has two input parameters of type int

```
int (*f)(int, int);
```

- pointer to a function that returns a pointer to int and has two input parameters, one pointer to int and one int

```
int* (*f)(int*, int);
```

Operations on function pointers

- function pointers can be changed, sent to functions, saved in arrays
- to call a function pointed by a pointer:
  - we can use the pointer directly, standing in for the name of the function

    ```
    func_pointer(params)
    ```

  - we can dereference the pointer, to highlight that we are using a function pointer

    ```
    (*func_pointer)(params)
    ```

Program 7.5 - Function pointer example

```c
#include <stdio.h>
int add(int a, int b){ return a+b; }
int multi(int a, int b){ return a*b; }
int accum(int* a, int n, int (*f)(int, int)){
  int ret = a[0];
  for(int i=1; i<n; i++)
    ret = f(ret, a[i]);
  return ret;
}

int main(){
  int a[] = {1, 2, 4};
  int n = sizeof(a)/sizeof(a[0]);
  int s = accum(a, n, add);
  int p = accum(a, n, multi);
  printf("sum = %d\n", s);
  printf("product = %d\n", p);
  return 0;
}
```

Program 7.6 - Sorting with custom comparator

```c
#include <stdio.h>
int comp_leq(int a, int b){ return a<=b; }
int comp_geq(int a, int b){ return a>=b; }
int comp_swap(int a, int b){ //91 <= 82
    int a2 = a%10 * 10 + a/10;
    int b2 = b%10 * 10 + b/10;
    return a2 <= b2;
}
void sortby(int* a, int n, int (*cmp)(int, int)){
    for(int i=0; i<n; i++){
        for(int j=i+1; j<n; j++){
            if (cmp(a[j], a[i])){
                int t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}
```

- define three different comparator functions
- each returns true when the first parameter is less than the second based on some rule
- sortby orders using the comparator

Program 7.6 - Sorting with custom comparator

```c
int main(){
  int a[] = {11, 91, 82, 13, 14, 7, 0};
  int n = sizeof(a) / sizeof(int);
  int (*comps[3])(int, int) = {
    comp_leq, comp_geq, comp_swap
  };

  for(int i=0; i<3; i++){
    sortby(a, n, comps[i]);
    for(int j=0; j<n; j++)
      printf("%02d ", a[j]);
    puts("");
  }
  return 0;
}
```

- comps is an array with 3 function pointers
- sort using each of the three comparator functions

Study problem - Root find

- you are given a mathematical function
  - continuous, well-behaved
  - in general any polynomial, exponential, trigonometric function
- find a root of the function in the interval [a, b]
  - it is guaranteed that there is a single root inside
- find the answer with a precision of x digits after the decimal point
  - note, that even for equations which have a closed-form solution (like $\sqrt{2}$), only an approximate value can be stored in memory

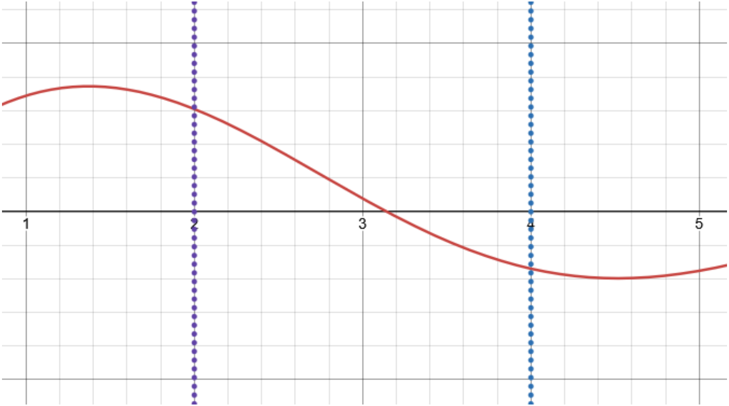
Study problem - Root find

- we will apply the Bisection Method
- repeatedly reduce the interval length by a factor of 2
- since f crosses the Ox axis exactly once between a and b we have:

$$f(a)f(b) <= 0$$

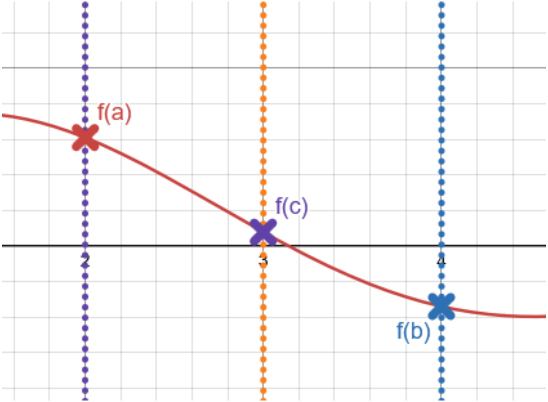
- we want to maintain this *invariant*
- check the sign of f in the midpoint $c = (a + b)/2$
- continue with interval *[a, c]* if $f(a)f(c) <= 0$
- otherwise continue with interval *[c, b]*
- the root will always be inside the interval

Study problem - Root find



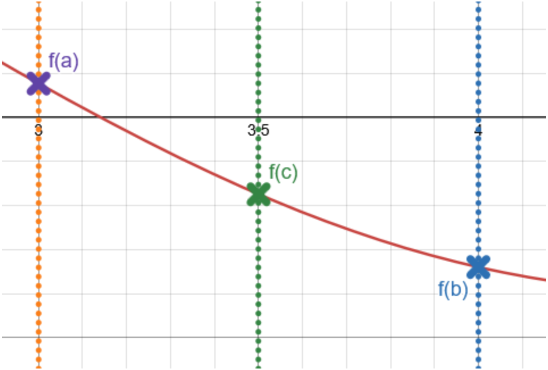
Input - function and [a, b] interval

Study problem - Root find



Step 1 - test midpoint c and set a = c

Study problem - Root find



Step 2 - test midpoint c and set b = c

Program 7.7 - Bisection root find

```c
#include <stdio.h>
#include <math.h>
double f(double x){
  return x*x*x - x - 2;
}
double g(double x){
  return exp(x)*x - 10;
}
double bisection(double (*f)(double),
    double a, double b, int prec){
  double eps = pow(10, -prec);
  while(b-a > eps){
    double c = (a+b)/2;
    if (f(a)*f(c) <= 0)
      b = c;
    else
      a = c;
  }
  return a;
}
int main(){
  printf("%.8f\n",
    bisection(f, 1, 2, 7));
  printf("%.8f\n",
    bisection(g, 1, 2, 7));
  return 0;
}
```

- the bisection method works on any compatible function
- is the equality check important?
- what happens if a and b are large?