Imperative programming

Arrays and pointers

Tamás Kozsik et al

Eötvös Loránd University

September 27, 2022

ELTE
EÖTVÖS LORÁND
TUDOMÁNYEGYETEM

Table of contents

1. Arrays
2. Pointers
3. Passing arrays as parameters

Concept of arrays

- Same type (size) elements in the memory next to each other
- Any element access is fast
- Fixed number of objects

```
int vector[4];
int matrix[5][3];      /* 15 elements sequentially */
```

Indexing from 0

- Address of `vector[i]`: vector's address + i * sizeof(int)
- Address of `matrix[i][j]`: matrix's address + (i * 3 + j) * sizeof(int)

Array indexing

- `int t[] = {1, 2, 3, 4}`
- Indexing from zero
- Length is unknown in runtime
- During compilation: `sizeof(t) / sizeof(t[0])`
- Bad index: undefined behavior

Examples

```c
int t[5] = {2, 6, 5, 9, 1};

int sum = 0;
for (int i = 0; i < 5; ++i)
  sum += t[i];

printf("Sum of elements: %d\n", sum);

int max = t[0];
for (int i = 0; i < 5; ++i)
  if (t[i] > max)
    max = t[i];

printf("The greatest element: %d\n", max);
```

Definition of C arrays

```
int a[4];                    /* 4 elements, uninitialized */
int b[] = {1, 5, 2, 8};      /* 4 elements */
int c[8] = {1, 5, 2, 8};     /* 8 elements, filled with zeros */
int d[3] = {1, 5, 2, 8};     /* 3 elements, unnecessary skipped */

int m[5][3];                 /* 15 elements sequentially */
int n[][3] = {{1,2,3},{2,3,4}}; /* Size is mandatory! */
int q[3][3][4][3];           /* 108 elements */

char s[] = "apple";
char z[] = {'a','p','p','l','e','\0'};
```

Text

```
char good[] = "good";
char  bad[] = {'b', 'a', 'd'};
char ugly[] = {'u', 'g', 'l', 'y', '\0'};
printf("%s %s %s", good, bad, ugly);
```

Pointers

- Stores the address of other objects: “points to them” (indirection)
- Type-safe

```
int i = 42;
int* p = &i;
printf("%d %d", i, *p);       /* 42 42 */
*p = 5;
printf("%d %d", i, *p);       /* 5 5 */
int j;                        /* Unspecified value */
p = &j;
*p = 10;
printf("%d", j);              /* 10 */
p = NULL;                     /* Points nowhere */
```

Connection of pointers and arrays

- Arrays convert to pointers
- They are not equivalent!

```
int arr[] = {1, 2, 3};
arr = {1, 2, 4};    /* Compile error */

int* ptr = arr;
int* q = &arr[0];

arr[1] = 5;
ptr[1] = 5;

int t[] = {3, 2, 1};
ptr = t;    /* ok */
arr = t;    /* Compile error */

printf("%d %d\n", sizeof(arr), sizeof(ptr));
```

Example

Conditional find

```
int t[] = {6, 2, 8, 7, 3};
int* p = NULL;

for (int i = 0; i < 5; ++i)
  if (t[i] % 2 == 1)
    p = &t[i];

if (p)
  printf("First odd number: %d\n", *p);
else
  printf("No odd number\n");
```

Pointer arithmetics

Stepping

```c
int v[] = {6, 2, 8, 7, 3};
int* p = v;        /* v: 6, 2, 8, 7, 3 */
int* q = p + 3;    /*    p        q    */

q = v + 3;         /* v converts   */

++p;               /* v: 6, 5, 8, 7, 3 */
*p = 5;            /*       p     q    */

p += 2;            /* v: 6, 5, 8, 1, 3 */
*q = 1;            /* v:          pq   */

q -= 2;            /* v: 6, 2, 8, 1, 3 */
*q = 2;            /*       q     p    */
```

Pinter arithmetics

Comparisons

```
int v[] = {6, 2, 8, 7, 3};
int* p = v;
int* q = v + 3;

if (p == q) { ... }
if (p != q) { ... }
if (p <  q) { ... }
if (p <= q) { ... }
if (p >  q) { ... }
if (p >= q) { ... }
```

Pointer arithmetics

Indexing

```
char str[] = "hello";

str[1] = 'o';
*(str + 1) = 'o';
*(1 + str) = 'o';

printf("%s\n", str + 3);
printf("%c\n", 3[str]);
```

Pointer arithmetics

Subtraction

```
int v[] = {6, 2, 8, 7, 3};

int* p = v;
int* q = v + 3;

int i = q - p;      /* 3 */
```

Example

Length of string

```
char str[] = "hello";
char* p = str;
char* q = str;

while (*q != '\0')
  ++q;

printf("Length of string: %d\n", q - p);
```

Passing arrays as parameters

Hardcoded value

```c
double distance(double a[3], double b[3]) {
  double sum = 0.0;
  unsigned int i;
  for (i = 0; i < 3; ++i) {    /* hardcoded value :-( */
    double delta = a[i] - b[i];
    sum += delta*delta;
  }
  return sqrt(sum);
}

int main() {
  double p[3] = {36, 8, 3}, q[3] = {0, 0, 0};
  printf("%f\n", distance(p, q));
  return 0;
}
```

Passing arrays as pointers

Fixed size in compile time

```
#define DIMENSION 3

double distance(double a[DIMENSION], double b[DIMENSION]) {
  double sum = 0.0;
  unsigned int i;
  for (i = 0; i < DIMENSION; ++i) {
    double delta = a[i] - b[i];
    sum += delta * delta;
  }
  return sqrt(sum);
  }

int main() {
  double p[DIMENSION] = {36, 8, 3}, q[DIMENSION] = {0, 0, 0};
  printf("%f\n", distance(p, q));
  return 0;
}
```

Passing arrays as parameters

Fixed size in compile time

```
double distance(double a[], double b[]) {
  double sum = 0.0;
  unsigned int i;
  for (i = 0; i < ???; ++i) {
    /* Size is not known */
    double delta = a[i] - b[i];
    sum += delta * delta;
  }
  return sqrt(sum);
}

int main() {
  double p[] = {3.0, 4.0}, q[] = {0.0, 0.0};
  printf("%f\n", distance(p, q));
  return 0;
}
```

Passing arrays as parameters

Bad approach

```
double distance(double a[], double b[]) {
  double sum = 0.0;
  unsigned int i;
  for (i = 0; i < sizeof(a) / sizeof(a[0]); ++i) {
    double delta = a[i] - b[i];
    sum += delta * delta;
  }
  return sqrt(sum);
}

int main() {
  double p[] = {3.0, 4.0}, q[] = {0.0, 0.0};
  printf("%f\n", distance(p, q));
  return 0;
}
```

Passing arrays as parameters

Correct

```
double distance(double a[], double b[], int dim) {
  double sum = 0.0;
  unsigned int i;
  for (i = 0; i < dim; ++i) {
    double delta = a[i] - b[i];
    sum += delta * delta;
  }
  return sqrt(sum);
}

int main() {
  double p[] = {3.0, 4.0}, q[] = {0.0, 0.0};
  printf("%f\n", distance(p, q, sizeof(p) / sizeof(p[0])));
  return 0;
}
```

Multi-dimensional arrays as parameters

```
double m[4][4] = {{1,2,3,4}, {1,2,3,4}, {1,2,3,4}, {1,2,3,4}};

transpose(m);

{
  int i, j;
  for (i = 0; i < 4; ++i) {
    for (j = 0; j < 4; ++j) {
      printf("%3.0f", m[i][j]);
    }
    printf("\n");
  }
}
```

Multi-dimensional arrays as parameters

Hardoced value

```
void transpose(double matrix[4][4]) {  /* double matrix[][4] */
  int size = sizeof(matrix[0]) / sizeof(matrix[0][0]);
  int i, j;
  for (i = 1; i < size; ++i) {
    for (j = 0; j < i; ++j) {
      double tmp = matrix[i][j];
      matrix[i][j] = matrix[j][i];
      matrix[j][i] = tmp;
    }
  }
}

double m[4][4] = {{1,2,3,4}, {1,2,3,4}, {1,2,3,4}, {1,2,3,4}};
transpose(m);
```

Multi-dimensional arrays as pointers

Sequential representation

```
void transpose(double* matrix, int size) {  /* size*size double */
  int i, j;
  for (i = 1; i < size; ++i) {
    for (j = 0; j < i; ++j) {
      int idx1 = i * size + j,   /* instead of matrix[i][j] */
          idx2 = j * size + i;   /* instead of matrix[j][i] */
      double tmp = matrix[idx1];
      matrix[idx1] = matrix[idx2];
      matrix[idx2] = tmp;
    }
  }
}

double m[4][4] = {{1,2,3,4}, {1,2,3,4}, {1,2,3,4}, {1,2,3,4}};
transpose(&m[0][0], 4);  /* transpose((double*)m, 4) */
```

Multi-dimensional arrays as parameters

Array of pointers

```c
void transpose(double* matrix[], int size) {
  int i, j;
  for (i = 1; i < size; ++i) {
    for (j = 0; j < i; ++j) {
      double tmp = matrix[i][j];
      matrix[i][j] = matrix[j][i];
      matrix[j][i] = tmp;
    }
  }
}

double m[4][4] = {{1,2,3,4}, {1,2,3,4}, {1,2,3,4}, {1,2,3,4}};
double* helper[4]; for (i = 0; i < 4; ++i) helper[i] = m[i];
transpose(helper, 4);
```

Command-line arguments

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

- argc: positive number
- argv[0]: program name
- argv[i]: command-line argument ($1 \leq i <$ argc)
    - Character array with ending \0
- argv[argc]: NULL

Command-line arguments

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

```
$ ./a.out one two three
0 -> ./a.out
1 -> one
2 -> two
3 -> three
```