Imperative programming

Types

Tamás Kozsik et al

Eötvös Loránd University

November 23, 2022

ELTE
EÖTVÖS LORÁND
TUDOMÁNYEGYETEM

Table of contents

Constants

Constants

```
const int i = 3;
int const j = 3;

const int t[] = {1, 2, 3};

const int* p = &i;

int v = 3;
int* const q = &v;
```

Compile error

```
i = 4;
j = 4;

t[2] = 4;

*p = 4;


q = (int*)malloc(sizeof(int));
```

Not full safety

```
const int = 3;
const int* r = &i;    /* correct */
```

Not full safety

```
const int = 3;
const int* r = &i;    /* correct */

int j = 2;
r = &j;               /* correct */
```

Not full safety

```
const int = 3;
const int* r = &i;    /* correct */


int j = 2;
r = &j;               /* correct */


r = &i;
int* p = r;           /* only warning */
int* const q = &i;    /* only warning */

*p = *q = 4;
```

First occurrance of a character

```
char* strfind(char* str, int c) {
  int i = 0;
  while (str[i] != 0 && str[i] != c) ++i;
  return &str[i];
}
```

```
char p[] = "Hello";
char *q = strfind(p, 'e');
*q = 'o';                    /* ok: "Hollo" in p, "ollo" in q */
```

First occurrance of a character

```
char* strfind(char* str, int c) {
  int i = 0;
  while (str[i] != 0 && str[i] != c) ++i;
  return &str[i];
}
```

```
char p[] = "Hello";
char *q = strfind(p, 'e');
*q = 'o';                    /* ok: "Hollo" in p, "ollo" in q */
```

```
char* r = strfind("Hello", 'e'); /* read-only storage! */
*r = 'o';                        /* Segmentation fault :-( */
```

String literals in read-only storage

```
char* r = "Hello";
*r = 'h';                /* segmentation fault :-( */
```

String literals in read-only storage

```
char* r = "Hello";
*r = 'h';                /* segmentation fault :-( */

const char* cr = "Hello";
*cr = 'h';               /* compile error :-) */
```

Exclude illegal memory write!

```
const char* strfind(char* str, int c) {
  int i = 0;
  while (str[i] != 0 && str[i] != c) ++i;
  return &str[i];
}
```

```
const char* r = strfind("Hello", 'e');
*r = 'o';                          /* compile error :-) */
```

Exclude illegal memory write!

```
const char* strfind(char* str, int c) {
  int i = 0;
  while (str[i] != 0 && str[i] != c) ++i;
  return &str[i];
}
```

```
const char* r = strfind("Hello", 'e');
*r = 'o';                          /* compile error :-) */
```

```
char p[] = "Hello";
char *q = strfind(p, 'e');  /* warning :-( */
*q = 'o';
```

Exclude illegal memory write!

```
const char* strfind(char* str, int c) {
  int i = 0;
  while (str[i] != 0 && str[i] != c) ++i;
  return &str[i];
}
```

```
const char* r = strfind("Hello", 'e');
*r = 'o';                          /* compile error :-) */
```

```
char p[] = "Hello";
char *q = strfind(p, 'e');  /* warning :-( */
*q = 'o';
```

```
const char *r = "Hello";
*r = strfind(r, 'e');          /* warning :-( */
```

Works for constant?

```
const char* strfind(const char* str, int c) {
  int i = 0;
  while (str[i] != 0 && str[i] != c) ++i;
  return &str[i];
}
```

```
const char* r = "Hello";
r = strfind(r, 'e');
*r = 'o';                          /* compile error :-) */
```

Works for constant?

```
const char* strfind(const char* str, int c) {
  int i = 0;
  while (str[i] != 0 && str[i] != c) ++i;
  return &str[i];
}
```

```
const char* r = "Hello";
r = strfind(r, 'e');
*r = 'o';                          /* compile error :-) */
```

```
char p[] = "Hello";
char* q = strgind(p, 'e');
*q = 'o';                          /* warning :-( */
```

No polymorph solution on const

```
char* strfind(char* str, int c) {
  int i = 0;
  while (str[i] != 0 && str[i] != c) ++i;
  return &str[i];
}

const char* conststrfind(const char* str, int c) {
  int i = 0;
  while (str[i] != 0 && str[i] != c) ++i;
  return &str[i];
}
```

- Code duplication
- The first can still be used with string literal

The same in string.h library

```
char* strchr(const char* str, int c) {
  while (*str != 0 && *str != c) ++str;
  return str;
}
```

- Works in general
- Works with string literal

The same in string.h library

```
char* strchr(const char* str, int c) { ... }

const char* p = "Hello";
char* r = "Hello";
char q[] = "Hello";

p = strchr(p, 'e');
r = strchr(q, 'e');

*p = 'o';                 /* compile error :-) */
*r = 'o';                 /* ok :-) */

r = strchr(p, 'o');       /* vagy strchr("Hello", 'e') */
*r = 'e';                 /* segmentation fault :-( */
```

Type aliases

```
typedef double Element;

Element max(Element array[], int size) {
  Element m = array[0];
  while (size > 0) {
    --size;
    if (array[size] > 0)
      m = array[size];
  }
  return m;
}
```

- Type alias: multiple names for the same type
- It has esthetic role
- Maintainability (readability, modifiability)

Types, type constructions

- Simple types
  - Numeric types
    - Integer types (and character)
    - Floating point types
  - Enumeration types
  - Pointer types
- Compound types
  - Array
  - List (Python, Haskell)
  - Tuples (Python, Haskell)
  - Record (C `struct`, Haskell `record`)
  - Union type (C `union`, Haskell algebraic data type)
  - Class (C++)

Enumeration type

```
enum color { WHITE, GREEN, YELLOW, RED, BLACK };

const char* property(enum color code) {
  switch (code){
    case WHITE:  return "pure";
    case GREEN:  return "jealous";
    case YELLOW: return "envy";
    case RED:    return "angry";
    case BLACK:  return "sad";
    default:     return "?";
  }
}
```

enum and typedef

```
enum color { WHITE, GREEN, YELLOW, RED, BLACK };

typedef enum color Color;

const char* property(Color code) {
  switch (code){
    case WHITE:  return "pure";
    case GREEN:  return "jealous";
    case YELLOW: return "envy";
    case RED:    return "angry";
    case BLACK:  return "sad";
    default:     return "?";
  }
}
```

enum: it is in fact an integer type

```
enum color { WHITE, GREEN, YELLOW, RED, BLACK };

const char* const properties[] =
  {"pure", "jealous", "envy", "angry", "sad"};

const char* property(enum color code) {
  return properties[code];
}
```

Setting type values

```
enum color { WHITE = 1, GREEN, YELLOW, RED = 6, BLACK };

typedef enum color Color;
const char* property(Color code){ ... }
```

Cartesian-product types

Compound type of (potentially) different type elements

- Tuple
- Record, struct

C struct

```c
/* creating the type */
struct month { char *name; int days; };

/* creating a variable */
struct month jan = {"January", 31};

/* three-way comparison */
int compare_days_of_month(
  struct month left, struct month right)
{
  return left.days - right.days;
}
```

C struct

```
struct month { char* name; int days; };
struct month jan = {"January", 31};

struct date { int year; struct month* month; char day; };
struct person { char* name; struct date birthdate; };

typedef struct person Person;

int main() {
  Person pete = {"Pete", {1970, &jan, 28}};
  printf("%d\n", (*pete.birthdate.month).days);
  printf("%d\n", pete.birthdate.month->days);
  return 0;
}
```

Parameter passing

```
void one_day_forward(struct date* d) {
  if (d->day < d->month->days) ++d->day;
  else { ... }
}

struct date next_day(struct date d) {
  one_day_forward(&d);
  return d;
}

int main() {
  struct date new_year = {2019, &jan, 1};
  struct date sober;
  sober = next_day(new_year);
  return sober.day != 2;
}
```

Union type

Type values from any of its members

C union

```c
/* name and days */
struct month { char* name; int days; };
/* either of them */
union name_or_days { char* name; int days; };

/* now it contains a name */
union name_or_days brrr = {"Pete"};
printf("%s\n", brrr.name);  /* fine */
printf("%d\n", brrr.days);  /* prints rubbish */

/* now it contains an int */
brrr.days = 42;
printf("%d\n", brrr.days);  /* fine */
printf("%s\n", brrr.name);  /* probably segmentation fault */
```

C array decarations

```
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 thrown */

extern int e[];
extern int f[10];               /* size ignored */

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

int m[5][3];                    /* 15 elements, sequentially */
int n[][3] = {{1,2,3},{2,3,4}}; /* size needed! */
int q[3][3][4][3];              /* 108 elements */
```

Pointer declarations

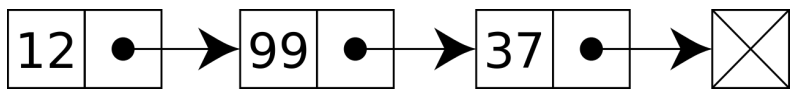
```
int i = 42;
int* p = &i;
int** pp = &p;          /* Pointer on pointer */
int* ps[10];            /* Array of pointers */
int (*pt)[10];          /* Pointer on array */

char* str = "Hello!";

void* foo = str;        /* points to anything */

int* p, q;              /* pointer and int */
int s, t[5];            /* int and array */
int* f();               /* function returns int* */
int (*f)(void);         /* pointer on a function returning int */
```

Data structures

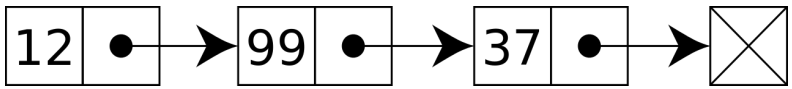
- "Many" elements
- Efficient access, manipulation
- Basic methods
  - Array representation (indexing)
  - Linked data structures
  - Hashing

Linked data structures

- Sequence: linked list
- Tree, e.g. search tree
- Graph



Linked list




```
struct node
{
  int data;
  struct node* next;
};

struct node* head;
head = (struct node*)malloc(sizeof(struct node));
head->data = 12;
head->next = (struct node*)malloc(sizeof(struct node));
head->next->data = 99;
head->next->next = (struct node*)malloc(sizeof(struct node));
head->next->next->data = 37;
head->next->next->next = NULL;
```

Insertion to an ordered list (Haskell)

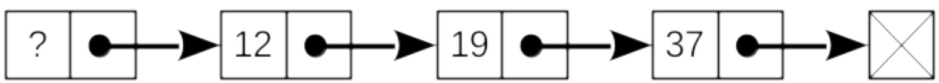
```
insert [] y = [y]
insert (x:xs) y
  | x < y      = x:(insert xs y)
  | otherwise = y:x:xs
```

Insertion to an ordered list (C)

```
typedef struct node* list_t;
list_t insert(list_t list, int value) {
  list_t new = (list_t)malloc(sizeof(struct node));
  if (NULL == new) return NULL;
  new->data = value;
  if (NULL == list || value < list->data) {
    new->next = list;
    return new;
  } else {
    list_t current = list, next = current->next;
    while (next != NULL && next->data < value) {
      current = next;
      next = current->next;
    }
    new->next = next;
    current->next = new;
    return list;
  }
}
```

Linked list with head

- An extra `node` at the beginning of the list
- Stored value not used in it
- No need to deal with the case of empty list



Insertion to a list with head

```
void insert(list_t list, struct node* new_node) {
  list_t current = list, next = current->next;
  while (next != NULL && next->data < new_node->data) {
    current = next;
    next = current->next;
  }
  new_node->next = next;
  current->next = new_node;
}
```

Equality check and copy

With base types

```
int a = 5;
int b = 7;

if (a != b)
{
  a = b;
}
```

Equality check and copy

With pointers

```
int n = 4;
int* a = (int*)malloc(sizeof(int));
int* b = &n;

if (a != b)
{
  a = b;
}
```

Equality check and copy

With arrays

```
int a[] = {5, 2};
int b[] = {5, 2};

if (a != b)
{
  a = b;    /* compile error */
}
```

Equality check and copy

With arrays

```
#define SIZE 3

int is_equal(int a[], int b[]) {
  for (int i = 0; i < SIZE; ++i)
    if (a[i] != b[i])
      return 0;
  return 1;
}

void copy (int a[], int b[]) {
  for (int i = 0; i < SIZE; ++i)
    a[i] = b[i];
}

int a[SIZE] = {5, 2}, b[SIZE] = {7, 3, 0};
...
if (!is_equal(a, b))
  copy(a, b);
```

Equality check and copy

With structs

```
struct pair { int x, y; };

struct pair a, b;
a.x = a.y = 1;
b.x = b.y = 2;

if (a != b)    /* compile error */
{
  a = b;
}
```

Equality check and copy

With structs

```
struct pair { int x, y; };

int is_equal (struct pair a, struct pair b)
{
  return (a.x == b.x) && (a.y == b.y);
}

struct pair a, b;
a.x = a.y = 1;
b.x = b.y = 2;

if (is_equal(a, b))
{
  a = b;
}
```

Equality check and copy

With linked lists

```
struct node
{
  int data;
  struct node* next;
};

struct node *a, *b;
...

if (a != b)
{
  a = b;
}
```

Equality check and copy

With linked lists

Shallow solution (not good here)

```
struct node
{
  int data;
  struct node *next;
};

int is_equal(struct node* a, struct node* b) {
  return (a->data == b->data) && (a->next == b->next);
}

void copy(struct node* a, const struct node* b) {
  *a = *b;
}
```

Equality check and copy

With linked lists

Deep equality check

```
struct node
{
  int data;
  struct node* next;
};

int is_equal(struct node* a, struct node* b)
{
  if (a == b) return 1;
  if ((NULL == a) || (NULL == b)) return 0;
  if (a->data != b->data) return 0;
  return is_equal(a->next, b->next);
}
```

Equality check and copy

With linked lists

Deep copy

```
struct node {
  int data;
  struct node* next;
};

struct node* copy(const struct node* b) {
  if (NULL == b) return NULL;
  struct node* a = (struct node*)malloc(sizeof(struct node));
  if (NULL != a) {
    a->data = b->data;
    a->next = copy(b->next);
  } /* else error message! */
  return a;
}
```

Motivation

```
int* find(int arr[], int size) {
  for (int i = 0; i < size; ++i)
    if (arr[i] % 2 == 0)
      return &arr[i];
  return NULL;
}
```

```
int* find(int arr[], int size) {
  for (int i = 0; i < size; ++i)
    if (arr[i] > 100000)
      return &arr[i];
  return NULL;
}
```

Higher order functions

With function pointers

```
/* Pointer to a function with int parameter returning bool */
bool (*fp)(int);

/* A function that expects an array, its size, and
   a function with a bool value and an int parameter */
int* find(int arr[], int size, bool (*f)(int));
```

Find

```
int* find(int arr[], int size, bool (*f)(int)) {
  for (int i = 0; i < size; ++i)
    if (f(arr[i]))
      return &arr[i];
  return NULL;
}

bool even(int n) { return n % 2 == 0; }
bool huge(int n) { return n > 100000; }

int main() {
  int arr = {1, 9, 3, 6, 7, 2};
  int size = sizeof(arr) / sizeof(arr[0]);
  int* first_even = find(arr, size, even);

  if (first_even)
    printf("The first even number: %d\n", *first_even);
  else
    printf("There is no even number\n");
}
```

Function pointers

Some notes

```
bool even(int n) { return n % 2 == 0; }

bool (*f)(int) = &inc;
f = even;
f(3) && (*f)(3);

bool (*g)() = even;
g(3, 4); g();
```