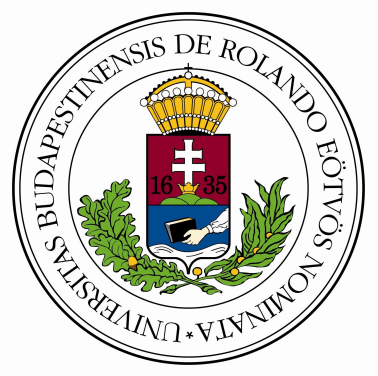
Imperative programming

12th Lecture



Kozsik Tamás

ELTE Eötvös Loránd Tudományegyetem

Outline

Compound Value Types

- Sequence
- Set
- Map
- Cartesian Product
- **Union**
- **Class**

Union Type

Value types are from one of multiple types.

C: union

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

union name_or_days brrr = {"Pete"};  /* now it contains a name */
printf("%s\n", brrr.name);   /* fine */
printf("%d\n", brrr.days);   /* prints rubbish */
brrr.days = 42;              /* now it contains a date */
printf("%d\n", brrr.days);   /* fine */
printf("%s\n", brrr.name);   /* probably segmentation fault */
```

Labeled Union

```
enum shapes { CIRCLE, SQUARE, RECTANGLE };
struct    circle { double radius; };
struct    square { int side; };
struct rectangle { int a; int b; };

struct shape
{
    int x, y;
    enum shapes tag;
    union csr
    {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};
```

Unified Usage

```
struct shape
{
    int x, y;
    enum shapes tag;
    union csr
    {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};

void move( struct shape *aShape, int dx, int dy ){
    aShape->x += dx;
    aShape->y += dy;
}
```

Usage with Case Separation

```
struct shape {
    int x, y;
    enum shapes tag;
    union csr {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};

double leftmost( struct shape aShape ){
   switch( aShape.tag ){
        case CIRCLE: return aShape.x - aShape.variant.c.radius;
        default:     return aShape.x;
   }
}
```

Secure Creation

```
struct shape {
    int x, y;
    enum shapes tag;
    union csr {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};

struct shape make_circle( int cx, int cy, double radius ){
    struct shape c;
    c.x = cx; c.y = cy; c.tag = CIRCLE;
    c.variant.c.radius = radius;
    return c;
}
```

Class

- Object-Oriented Languages
- Class: Record-like structure
  - Data Members (Fields)
  - Operations (Methods)
- Inheritance: labeled union

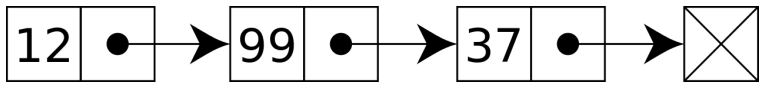
Outline

Data Structures

- Organizing „lot of” data
- Efficient access and manipulation
- Basic methods
  - Array-based representation (indexing)
  - Linked Data Structure
  - Hashing

Linked Data Structures

- Sequence: Linked List
- Tree, e.g. Search Trees
- Graph



Representing Sequences

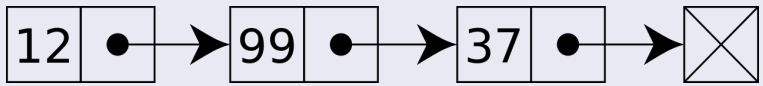
Array

- Getting and Writing arbitrary element N
- Insertion/Deletion?
  - Moving data
  - Reallocation

(example: http://gsd.web.elte.hu/lectures/imper/imper-lecture-10/)

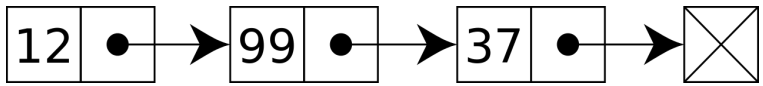
Linked List

- Getting and Writing elements (in order) by using traversal
- Insertion/Deletion during traversal
- Getting and Writing arbitrary element N?



Linked List

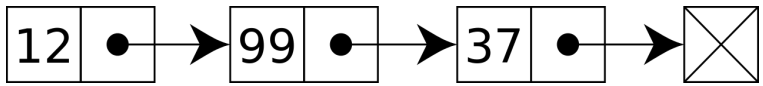
```
struct node
{
    int data;
    struct node *next;
};
```



Construction of 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;
```



Outline

Equality Check and Copy on Primitive Types

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

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

Using Pointers?

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

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

Using Arrays?

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

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

Using 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 );
```

Using Structures?

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

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

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

Using Structures!

```
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;
}
```

Using Linked List?

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

struct node *a, *b;
...

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

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;
}
```

Deep Equality Check

```c
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);
}
```

Deep Copying

```
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 give error! */
    return a;
}
```

Outline

Error Handling

- If something unexpected happens
  - For example `malloc` fails
- Robustness
- Language support: Python

Error handling in C

- Return value of function
  - Returning an error code (`int`)
  - Returning special („extremal”) value (e.g. `NULL`)
- Global variable: error code

Drawback of C Error Handling

- Too much branches and condition checking
  - Error handling can be 30-40% of the code
  - We loose the important part in the code
- Error handling not written - Dangerous!
- Error handling left out - Dangerous!

Exception Propagation

Passing of Control!

- Along the call chain of subprograms
- Execution stack
- From raising...
  - ... to handling
  - ... to program exit

Modules in C

- Compilation Units
- Source code: `.c` and `.h`
- `#include`
- Linking: static or dynamic

Header Files

- „header files”: `.h`
- Interface between Modules
  - `extern`
  - not `static`
- in the Module and in its Client `#include`
  - type correspondence
- Dependencies between Compilation Units
  - independent compilation
  - task for linking
  - compilation process: `make`

Include guard

vector.h (part)

```
#ifndef VECTOR_H
#define VECTOR_H

#define VEC_EOK       0
#define VEC_ENOMEM    1

struct VECTOR_S;
typedef struct VECTOR_S *vector_t;

extern int vectorErrno;

extern void *vectorAt( vector_t v, size_t idx);
extern void  vectorPushBack( vector_t v, void *src);

#endif
```