Basics of programming

Balázs Goldschmidt

Indirect reference: pointers

Problem: family tree

- Let's declare a struct for a human
- Store
  - age
  - height
  - father and mother

```
struct human;
struct human {
   int age;
   double height;
   struct human father, mother;
};
```

- Size?
  - |***human***| = |*int*| + |*double*| + 2|*human*|
  - |*human*| = -(|*int*|+|*double*|)

Solution: family tree w/ pointers

- Let's store just a **pointer** to parents
- Store
  - age
  - height
  - ptr to parents

```
struct human;
struct human {
   int age;
   double height;
   struct human *father, *mother;
};
```

- Size?
  - |***human***| = |*int*| + |*double*| + 2|*human_ptr*|
    = 20 bytes

Handling pointers

- Pointer definition: type with *
  - `int*`, `double*`, etc.
  - `int *i, j, *k; /* i and k are ptrs */`
- Variables and operators
  - *: value pointed to by pointer (dereferencing)
  - &: address of the variable (referencing)

```
int i=1, j=2, k=5;
int *ip, *jp;
ip = &i;
jp = &jp;
k = (*ip)+(*jp);
(*ip) = 12;
```

```
printf("%d\n", *ip);

ip = jp;
(*jp) += 3;
printf("%d\n", *ip);
```


Parameter passing

- Function parameters can be pointers
  - less overhead
  - data may be corrupted

```
int getMax(int* p, int n) {
   int max = p[0], i;
   for (i = 1; i < n; i++) {
      if (max < p[i]) p[i] = max;
      if (max < p[i]) max = p[i];
   }
   return max;
}
```

Return values

```
int* foo(int* p) {
   return p;
}
```

```
int a = 13, *b;
b = foo(&a);
*b = 10;
printf("%d\n", a);
```

```
int* bar() {
   int q;
   return &q;
}
```

```
int *b;
b = bar();
*b = 10;
```

Address to local variables **mustn't** be returned!

Pointer arithmetic

- Operators
  - +, -, etc.
  - adding *pointer* and *int* results *pointer*
  - eg.: `p1 = p0+13;`
- Stepping is type dependent
  - *p+1* points to next element (not next byte)
  - size is deduced from type of *p*

Arrays and pointers

- Array variables are considered pointers to first element

```
int a[3], *p;
a[0] = 10;
a[1] = 20;
a[2] = 30;
```

```
p = a;
printf("%d\n", p);
*p = 2;
```

- Array expressions use pointer arithmetic
  - `a[x] ≡ *(a+x)`

```
p[2] = 13;
*(p+1) = 50;
*(a+2) = 10;
```

Dynamic memory handling

- Allocate single element of type T
  - `T *p = (T*) malloc (sizeof(T));`
  - e.g.: *T* is *int*
    `int *p = (int*) malloc (sizeof(int));`
- Allocate array of *T* having *n* elements
  - `T *p = (T*) malloc (n*sizeof(T));`
  - e.g.: *T* is *int, 10* elements
    `int *p = (int*) malloc (10*sizeof(int));`
- If out of memory, return NULL
  - ***NULL*** is the pointer that points nowhere

Lifetime and deallocation

- All dynamically allocated values exist until deallocated
  - lifetime is longer than a single function call
- Deallocation
  - **`free(p);`**
  - both for single and array elements

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

```
int *p = getN(10);
p[3] = 12;
free(p);
```


Resizing dynamic arrays

- Realloc

```
int *tmp, *p = ...;
tmp = realloc(p, 10*sizeof(int));
if (tmp != NULL) p = tmp;
```

  - has more space after orig: simple allocation
    - pointer remains same
  - no more space after orig: reallocation
    - return pointer is new
    - data copied, original freed
  - out of memory: returns NULL

Complex types: strings

String

- Strings are arrays of characters
  - `"abcde"` -> `{'a','b','c','d','e','\0'}`
  - last character is always `'\0'`
  - string type is `char[]` or `char*`
- String handling functions: *string.h*
  - strcmp, strcpy, strcat, etc.
  - strncmp, strncpy, strncat, etc.
  - strlen

String exercise

- Implement standard functions
  - `int strlen(char* s1)`
    - indexing
    - increment
  - `int strcmp(char* s1, char* s2)`
    - return value
      - `-1 if s1<s2`
      - `0 if s1==s2`
      - `+1 if s1>s2`