# Programs as Data 6
## Imperative languages, environment and store, micro-C

Peter Sestoft

Monday 2013-09-30*

Note by Baris Aktemur:
These slides have been adapted from the originals available at http://www.itu.dk/courses/BPRD/E2013/.
I thank Peter Sestoft for making the PPT's available.

# C pointer basics

- A pointer `p` refers to a storage location
- The dereference expression `*p` means:
  - *the content of the location* as in `*p + 4`
  - *the storage location itself* as in `*p = x+4`
- The pointer that points to `x` is `&x`
- Pointer arithmetics:
  - `*(p+1)` is the content of the location just after `*p`
- If `p` equals `&a[0]`
  then `*(p+i)` equals `p[i]` equals `a[i]`,
  so an array is a pointer
- Strange fact: `a[2]` can be written `2[a]` too

# Using pointers for return values

- Example ex5.c, computing square(x):

```
void main(int n) {
   ...
   int r;
   square(n, &r);
   print r;
}

void square(int i, int *rp) {
   *rp = i * i;
}
```

for input

for return value: a pointer to where to put the result

# Recursion and return values

- Computing factorial with MicroC/ex9.c

```
void main(int i) {
  int r;
  fac(i, &r);
  print r;
}

void fac(int n, int *res) {
  if (n == 0)
    *res = 1;
  else {
    int tmp;
    fac(n-1, &tmp);
    *res = tmp * n;
  }
}
```

- `n` is input parameter
- `res` is output parameter: a pointer to where to put the result
- `tmp` holds the result of the recursive call
- `&tmp` gets a pointer to `tmp`

# Lvalue and rvalue of an expression

- Rvalue is “normal” value, right-hand side of assignment: `17`, `true`
- Lvalue is “location”, left-hand side of assignment: `x`, `a[2]`
- In assignment `e1=e2`, expression `e1` must have lvalue

| | Has lvalue | Has rvalue |
|---|---|---|
| `x` | yes | yes |
| `a[2]` | yes | yes |
| `*p` | yes | yes |
| `x+2` | no | yes |
| `&x` | no | yes |

- Where else must an expression have lvalue in C#? In C?

# Operators &x and *p are inverses

- The address-of operator `&e`
  - evaluates `e` to its lvalue
  - returns the lvalue (address) as if it were an rvalue
- The dereferencing operator `*e`
  - evaluates `e` to its rvalue
  - returns the rvalue as if it were an lvalue

- It follows
  - that `&(*e)` equals `e`
  - that `*(&e)` equals `e`, provided `e` has lvalue

# C variable declarations

| Declaration | Meaning |
|---|---|
| int n | n is an integer |
| int *p | p is a pointer to integer |
| int ia[3] | ia is array of 3 integers |
| int *ipa[4] | ipa is array of 4 pointers to integers |
| int (*iap)[3] | iap is pointer to array of 3 integers |
| int *(*ipap)[4] | ipap is pointer to array of 4 pointers to ints |

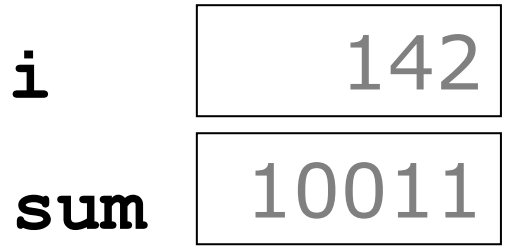
Unix program **`cdecl`** or www.cdecl.org may help:

```
cdecl> explain int *(*ipap)[4]
declare ipap as pointer to array 4 of pointer to int
cdecl> declare n as array 7 of pointer to pointer to int
int **n[7]
```

# A naive-store imperative language

- **Naive** store model:
  - a variable name maps to an integer value
  - so store is just a runtime environment
- Executing a statement gives a new store
- Assignment **`x=e`** updates the store

```
i = 1;
sum = 0;
while sum < 10000 do begin
  sum = sum + i;
  i = 1 + i;
end;
```

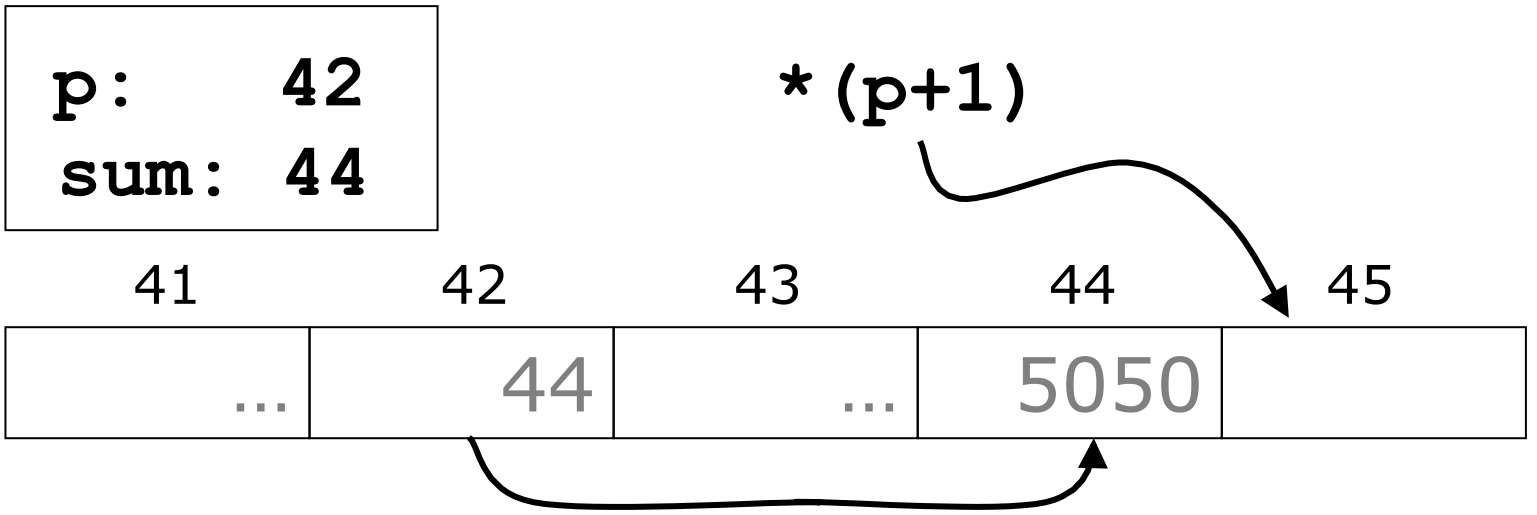


# Environment and store, micro-C

- The naive model cannot describe *pointers* and *variable aliasing*
- A more realistic store model:
  - *Environment* maps a variable name to an address
  - *Store* maps address to value

# The essence of C: Pointers

- Main innovations of C (1972) over Algol 60:
  - Structs, as in COBOL and Pascal
  - Pointers, pointer arithmetics, pointer types, array indexing as pointer indexing
  - Syntax: { } for blocks, as in C++, Java, C#




- Very different from Java and C#, which have no pointer arithmetics, but garbage collection

# Call-by-value and call-by-reference, C#

```
int a = 11;
int b = 22;
swapV(a, b);
swapR(ref a, ref b);
```

```
a: 41
b: 42
```

addresses

by value

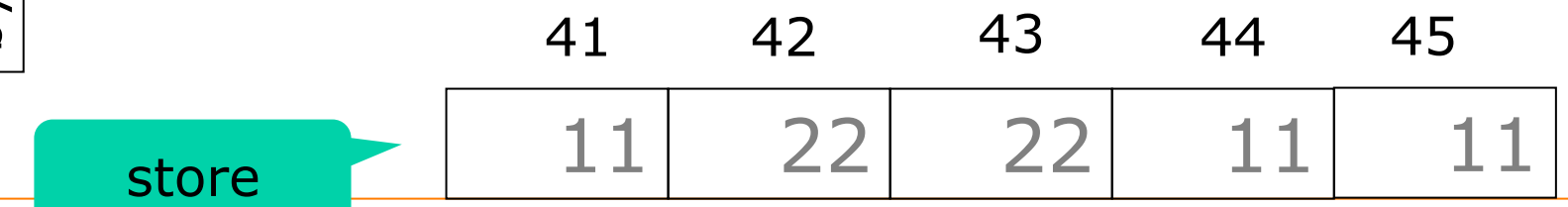
```
static void swapV(int x, int y) {
  int tmp = x; x = y; y = tmp;
}
```

```
x:   43
y:   44
tmp: 45
```

by reference

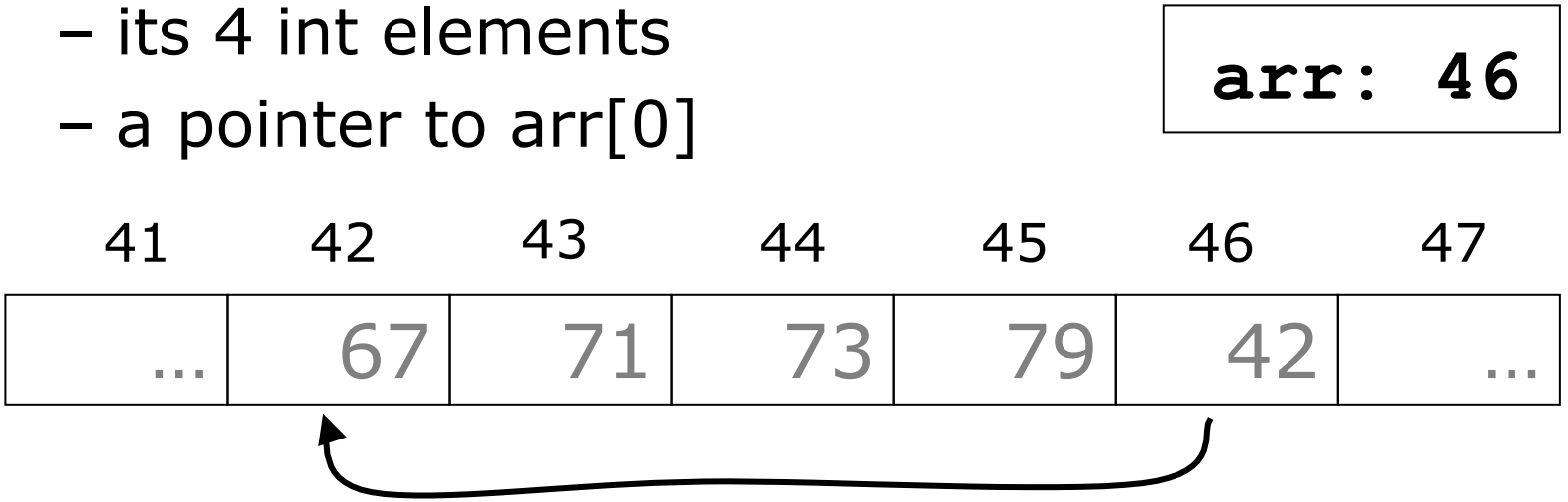
```
static void swapR(ref int x, ref int y) {
  int tmp = x; x = y; y = tmp;
}
```

```
x:   41
y:   42
tmp: 43
```



# Micro-C array layout

- An array **`int arr[4]`** consists of
  - its 4 int elements
  - a pointer to arr[0]




- This is the uniform array representation of B
- Real C treats array parameters and local arrays differently; complicates compiler

# Expression statements in C, C++, Java and C#

- The "assignment statement"

  `x = 2+y;`

  is really an expression

  `x = 2+y`

  followed by a semicolon

Value: none
Effect: change x

Value: 2+y
Effect: change x

- The semicolon means: ignore value