

G22.2110 Programming Languages

6/14/2007
Low-Level Languages (C)

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

1

Low-level languages

- What are low-level languages?
 - Expose machine details (e.g., raw memory)
 - Statically compiled and weakly typed
- Motivation
 - “Systems programming”
 - Predictability
 - Performance
- Problems
 - Easy to make mistakes / hard to debug
 - Lower programmer productivity

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

2

Outline

- Concepts
- Uses of pointers
- Arrays
- Pointer arithmetic
- Multidimensional arrays

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

3

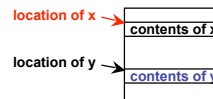
Lvalues and rvalues

What does an assignment do?

$x = y$

Store at **location of x** ... the **contents of y**.

Variables (x,y) are names for storage locations that can hold some contents, e.g., integers.



M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

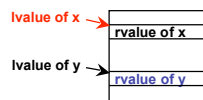
4

Lvalues and rvalues (cont.)

$x = y$

Store at **location of x** ... the **contents of y**.

lvalue = location (value of x when it occurs on left hand side of assign)
rvalue = contents (value of y when it occurs on right hand side of assign)



M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

5

Lvalues and rvalues (cont.)

- All expressions denote an rvalue:
 - x, 42, arr[4], x+1, obj.f, sin(2.5), ...
- Some expressions only denote an rvalue, but no lvalue:
 - 42, x+1, sin(2.5), ...
 - Can not appear on left-hand side of assign
 - Don't have a defined storage location
 - Not modifiable

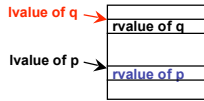
M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

6

Objects and pointers

- Object = has both lvalue and rvalue
 - Different definition from OOP (object-oriented programming): in OOP, an object is an instance of a class
- Pointer = reference to object, usually implemented by address
 - For example: when rvalue of p is the same as lvalue of q, then p is a pointer to q



Outline

- Concepts
- Uses of pointers
- Arrays
- Pointer arithmetic
- Multidimensional arrays

Pointers to stack objects

```
void swap(int* x, int* y) {
    int z = *x;
    *x = *y;
    *y = z;
}

int a = 2, b = 5;
swap(&a, &b);
```

Annotations:

- type pointer to int (pointing to `int* x`)
- dereference operator: use rvalue as lvalue (pointing to `*x`)
- "*" and "&" cancel each other out: `*(&a) == a` (pointing to `&a`)
- address-of operator: use lvalue as rvalue (pointing to `&a`)

- Motivation: Emulate call-by-reference using pointers and call-by-value

Dangling references

```
int* f(int x) {
    return &x;
}

int* g(float y) {
    return &y;
}

int a = 2;
int* p = f(a);
float* q = f(3.1);

... *p ...
```

Annotations:

- Dangling reference = pointer whose lifetime ends after the lifetime of its target
- Tough bugs:
 - Reading through a dangling reference yields undefined results
 - Writing through a dangling reference may corrupt unrelated objects

Pointers to heap objects

```
struct list {
    int _car;
    struct list* _cdr;
};

struct list* cons(int car, struct list* cdr) {
    struct list* result =
        (struct list*)malloc(sizeof(struct list));
    result->_car = car;
    result->_cdr = cdr;
    return result;
}

struct list* x = cons(2, cons(5, cons(1, NULL)));
```

Annotations:

- recursive data type: refers to itself (pointing to `struct list*`)
- number of required bytes (pointing to `sizeof(struct list)`)
- allocate new heap object (pointing to `malloc`)
- cast pointer from void* to specific type (pointing to `(struct list*)`)
- shorthand for `(*result) . cdr` (pointing to `_cdr`)
- pointer to no object (pointing to `NULL`)

Pointers to heap objects (cont.)

```
struct list* x = cons(2, cons(5, cons(1, NULL)));
...
struct list* i;
for (i = x; i != NULL; i = i->_cdr)
    printf("%d\n", i->_car);
```

Annotations:

- identity comparison (pointing to `i != NULL`)

- Motivation: gain most of the advantages of structured programming, while retaining most of the advantages of assembly language

Heap object deallocation

- C uses explicit memory management


```
while (NULL != x) {
    struct list* y = x;
    x = x->_cdr;
    free(y);
}
```
- Memory leak = malloc without free
 - Program may run out of memory
 - May lead to crash after hours of deployment
- Dangling reference: use (read/write) after free

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

13

C features for pointers thus far

Description	Examples
declaration	T* p;
assignment	p = q
dereference	*p; p->f
address-of	&x
cast (interpret as type T*)	(T*)x
size of type; size of variable	sizeof(T); sizeof(*x)
allocation; deallocation	malloc; free
pointer to nothing	NULL
identity comparison	==; !=

Address-of and dereference cancel each other out: &(*x)==x

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

14

Outline

- Concepts
- Uses of pointers
- Arrays
- Pointer arithmetic
- Multidimensional arrays

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

15

Stack-allocated arrays

```
int x[5];
int i;
for (i=0; i<5; i++)
    x[i] = i * i;
```

- Assume sizeof(int) == 4, and the array starts at address 100
- lvalue(x[i]) == 100 + 4 * i
- rvalue(x[i]) == i * i
- x itself does not have an lvalue!

(addr)	...
100	0
104	1
108	4
112	9
116	16
120	...

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

16

Arrays as pointers

- Subscript operator is syntactic sugar

$$x[i] == *(x + i) == i[x]$$
- "&" and "*" (address-of and dereference) cancel each other out

$$\&(*e) == e$$

$$\&(x[i]) == \&*(x+i) == x+i$$
- What is the value of x?

$$x == x+0 == \&*(x+0) == \&(x[0])$$
 ⇒ the rvalue of x is the lvalue of x[0]

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

17

Arrays as pointers (cont.)

- What does the "+" operator do for pointers?

$(\text{uint})(x+i)$
Assume uint is unsigned int type with same number of bits as pointer

$$\begin{aligned}
 &== (\text{uint})(\&*(x+i)) \\
 &== (\text{uint})(\&(x[i])) \\
 &== (\text{uint})(\&(x[0]) + \text{sizeof}(\text{int}) * i) \\
 &== (\text{uint})x + \text{sizeof}(\text{int}) * i
 \end{aligned}$$

- In general:

$$\begin{aligned}
 &(\text{uint})(p+i) \\
 &== (\text{uint})p + \text{sizeof}(\text{baseType}(p)) * i
 \end{aligned}$$

(addr)	...
100	0
104	1
108	4
112	9
116	16
120	...

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

18

C pointer features: addendum

Description	Examples
declaration	T* p;
assignment	p = q
dereference	*p; p->f
address-of	&x
cast	(T*)x
size of type; size of variable	sizeof(T); sizeof("x")
allocation; deallocation	malloc; free
pointer to nothing	NULL
identity comparison	==, !=
subscript	x[i]
arithmetic	+, -, ++, --
magnitude comparison	<, <=, >=, >

treating pointers like arrays

treating pointers like integers

Outline

- Concepts
- Uses of pointers
- Arrays
- Pointer arithmetic
- Multidimensional arrays

Pointer arithmetic

- Assume declarations "T* p₁; T* p₂; int i;"
- ptr + int → ptr
 - add multiple of size of pointed-to object
 - p₁ + i == (T*)((uint)p₁ + i * sizeof(T))
- ptr – int → ptr
 - subtract multiple of size of pointed-to object
 - p₁ – i == (T*)((uint)p₁ – i * sizeof(T))
- ptr – ptr → int
 - return difference in number of objects
 - p₁ – p₂ = ((uint)p₁ – (uint)p₂) / sizeof(T)

C strings

- C string = null-terminated array of char


```
char* x = "hello";
char* i;
for (i=x; '\0' != *i; i++)
    printf("%c\n", *i);
```
- The above code uses pointer arithmetic, would be clearer with array subscripts

C pointers illustrate usefulness of PL knowledge

- Emulating features in a language that does not directly support them
 - Call-by-reference
 - Recursive data types
- Reading obscure code
 - Treating arrays as pointers
 - Loops with pointer arithmetic

Arrays vs. pointers

- Arrays and pointers are almost the same
 - &*(x[i]) == x+i
- But it still matters how you declare them

Declaration	int x[5];	int* x;
Allocates	5 integers	1 pointer
sizeof(x)	5 * sizeof(int)	sizeof(int*)
Has lvalue?	No	Yes

- int x[]: size determined by initialization

Multidimensional arrays

Declaration	int x[5][5];	int** x;
Allocates	25 integers	1 pointer
sizeof(x)	25 * sizeof(int)	sizeof(int**)
Has lvalue?	No	Yes

- int[5][5] is array with contiguous layout
- int** may point to array with contiguous layout, or to row-pointer (array of arrays)

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

25

Outline

- Concepts
- Uses of pointers
- Arrays
- Pointer arithmetic
- Multidimensional arrays

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

26

How to read C definitions

- Rule of thumb:
 - Start at innermost identifier
 - Look right until “)” or end
 - Look left until “(“ or start
 - Pop out one layer of “(“...” at a time
- Examples:

int *x[3]	Array of 3 pointers to int.
int (*x)[3]	Pointer to array of 3 int.
int (*x)()	Pointer to function returning int.
int (**x)[]	Pointer to function returning array of unspecified number of pointers to int.

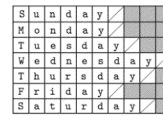
M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

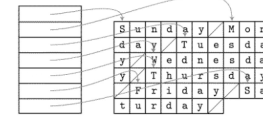
27

Multidimensional arrays

Scott Fig. 7.10: “char days[][10]” vs. “char* days[]”



Contiguous array allocation
True 2-dimensional array
7 * 10 * sizeof(char)



Row pointer
Array of pointers to arrays
7 * sizeof(char*) + 57 * sizeof(char)

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

28

Pointer arithmetic (cont.)

- int x[5][5]
 - Since x is array: x == &(x[0])
 - Since x[i] is array: x[i] == &(x[i][0])
- Assume (uint)x == 200, sizeof(int) == 4, what is the rvalue for each expression?
 - sizeof(*x), *x, x+2, *x+2
 - x[2]+3, (x+2)-x, x[2]-x[0],
 - x<=x, x[2]>x[1]

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

29

Discussion

- Low-level languages are good for:
 - Device drivers
 - Memory managers
 - Network protocols
 - ...
- Low-level languages have problems:
 - Less error checking
 - More verbose code
 - Missing high-level features

M.Hirzel, A.Guria

G22.2110 NYU 6/14/2007

30