

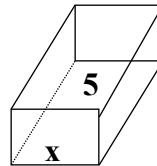
Announcements

- **Project 1 due tonight**
 - Extra office hours tonight 7-9 pm in ARC lab
- **Midterm is Friday, 2:50-4:10pm, Van Dyck 211**
 - read the **Examination Policy** before the exam (the link is off of the *Academic Integrity* page); please arrive early

1

Imperative Programming: C

- **Assignment as main operation**
 - Names \Leftrightarrow Locations \Leftrightarrow Values
 - L-value: name labeling memory location
 - R-value: contents of memory location
- **State of a computation**
 - M: Locations \rightarrow Values
 - Remaining input
 - Output so far



2

Random Access Machine

- **A universal computing device, similar to a Turing Machine**
- **Components:**
 - **Program:** sequence of instructions
 - **Memory:** sequence of locations
 - **Control:** current location in program
 - **Input file:** sequence of values
 - **Output file:** sequence of values
- **Imperative programming languages have primitives close to the machine instructions (e.g., assignment, branch)**

3

Random Access Machine

- **Normal control “flows” from one instruction to the next.**
 - *Thread of computation:* sequence of program points reached as execution flows through the program
- **Control flow directs the thread without changing the state.**
- **Data flow (through assignment) affects the state without directly affecting the thread.**

4

Bird's Eye View: C vs. Java

- Types:
int, double, char
- *Pointer (to a value)*
- Aggregates: array, *struct*
- Control flow: if-else, switch, while, break, continue, for, return, *goto*
- Logic operators: || && !
- Logical comparisons: == !=
- Numeric comparisons:
<> <= >=
- string as char * array
- Primitive types:
int, double, char, *boolean*
- *Reference (to objects)*
- Aggregates: array, object
- Control flow: if-else, switch, while, break, continue, for, return
- Logic operators: || && !
- Logical comparisons: == !=
- Numeric comparisons:
<> <= >=
- *String* as an object

5

Control Statements

- Choice:

```
if ( <expr> ) <stmt> [ else <stmt> ]  
switch ( <expr> ) { <statement-list> }
```

```
switch (grade){  
    case 'A': printf("doing very well");  
              break;  
  
    case 'D': ← missing  
    case 'F': printf("doing poorly"); ← break  
    default: printf..  
};
```

6

Control Statements

- **Iteration:**

```
while ( <expr> ) <stmt>
do <stmt> while ( <expr> );
for ( <expr>; <expr>; <expr> ) <stmt>
```

```
while ((c = getchar()) != eof) putchar(c);
    Embedded side effects in expressions allowed!
```

```
for (j=0; s[j] == ' '; j++);
    body is empty statement
```

```
for (k=0; k<n; k++) {
    if (a[k]<0) break;
    ...
}
```

7

Control Statements

- **Unstructured Branch:** `goto beyond-the-end;`
- **Debate in the 1970's:**
 - “Go To Statement Considered Harmful” (Dijkstra, 1968)
- **Structured Programming:**
 - Control flow should be obvious from syntactic structure
 - Single-entry, single-exit loops preferred
 - Eliminate the `goto` statement!
- **Theorem (Boehm & Jacopini, 1966):**
 - Any control structure can be expressed as a combination of composition, conditionals and while loops.

8

Example: hello.c

```
/* sample program to print hello world,
   the numbers from 1 to 10, and the even ones */
#include<stdio.h>
main()
{   int j, n;

    printf(" hello world\n");
    n = 10;
    for (j = 0; j <= n; j++)
        printf(" %d",j);
    printf("\n the even numbers are: ");
    for (j = 0; j <= 10; j++)
        if ((j%2) == 0) printf(" %d",j);

    printf("\n");
}
```

9

Formatted Print Statements

```
printf(<format-string>,arg1,arg2,...);
```

format-string can include, e.g.,

- %d** - decimal integer
- %g** - floating point
- %10.2f** - ten digits, 2 after decimal point
- %c** - character
- %s** - character string

to indicate where arguments go, and how they should be displayed

```
printf("here are two characters: %c and %c\n",'q','$');
```

10

Example Output

When `hello.c` is run it looks like this:

```
% gcc hello.c
% a.out
hello world
0 1 2 3 4 5 6 7 8 9 10
the even numbers are: 0 2 4 6 8 10
%
```

11

Running a C program

- `gcc <filename>` calls the GNU C compiler to compile the file.
 - This produces an executable `a.out` in the same directory
 - Type `a.out` at the prompt to run the C program
- You often use the `-o` option to give the executable a name (in this case, there are 2 `.c` files):
 - `% gcc -o pgm test.c des.c`
puts the executable translation of the program in the files `test.c` and `des.c` in the file `pgm`
 - `% pgm` runs the program

12

L-Values and R-Values

- Think about simple examples:

$y = 5; y = b; y = w + 3; y = y + 2; \dots$

location in which result is stored

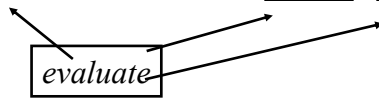
evaluate

- These statements would be illegal:

$5 = y; w + 3 = y; \dots$

- But the left-hand side can also be evaluated:

$age[k] = y + 2; a[b[m]] = f[n]; \dots$



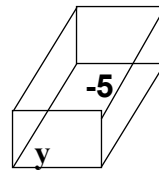
13

L-Values and R-Values

- R-Value:

– *value* of an expression

$y = -5;$



1022

- L-Value:

– *address* of memory location

– but this may be the result of evaluating an expression

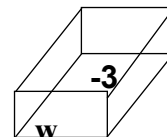
- Names:

– refer to locations containing values

– symbol table associates names

with physical addresses

$w = y+2;$



538

14

C Data Types

- **Primitive:** char, int, double (and others...)
- **Without boolean type, any nonzero value is *true*; zero is *false***
- **Aggregate: arrays, structs**

- **Homogeneous arrays**

```
char a[10];
```

```
int b[2][10];
```

- **Heterogeneous structs**

```
struct employee{  
    int age;  
    double payrate;  
}
```

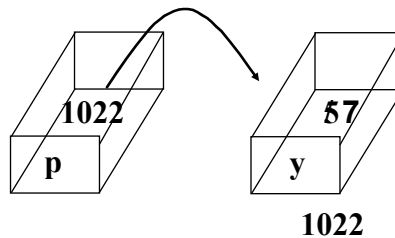
```
struct rectangle{  
    struct point p1;  
    struct point p2;  
}
```

15

Pointers

- **Pointer: A variable whose value is an L-value.**
 - **declaration:** `int *p;`
 - **address-of operator & :** returns L-value of variable
 - **dereference operator for a pointer * :** obtains R-value at that address value

```
int y = 5;  
int *p;  
p = &y;  
*p = 7;
```

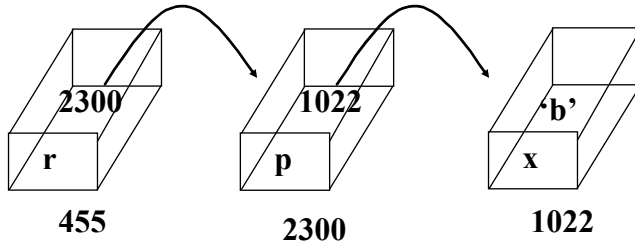


16

Pointers

- Pointers can point to pointer variables (multi-level).

```
char x;
char *p;
char **r;
p = &x;
*p = 'a';
r = &p;
**r = 'b';
```

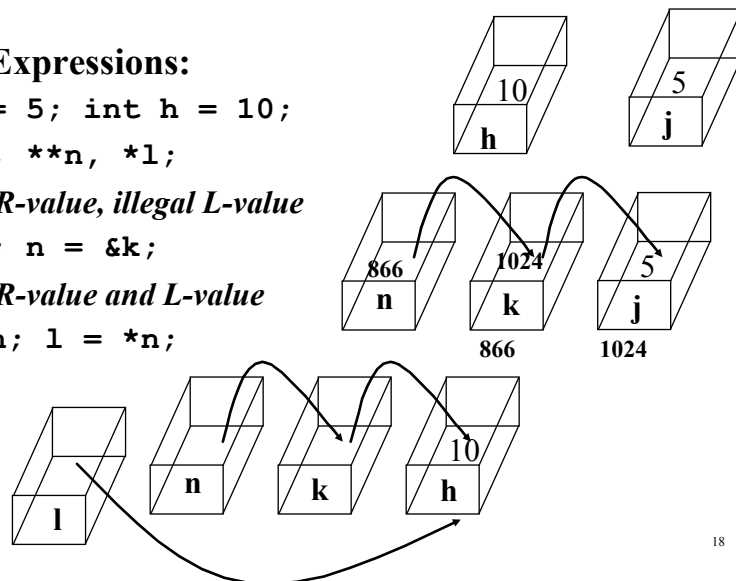


- Assignment Rule: `<lhs> = <rhs>`
 - lhs must be an L-value; rhs is evaluated, with all L-values dereferenced once (unless blocked by `&`)

Pointers

- Pointer Expressions:


```
int j = 5; int h = 10;
int *k, **n, *l;
&j, legal R-value, illegal L-value
k = &j; n = &k;
*n, legal R-value and L-value
*n = &h; l = *n;
```



Pointers and Arrays

- **An array name is considered pointer to first element: `int a[5];`**
 - `a` is pointer to `a[0]`
 - `pa = &a[0]` and `pa = a` mean the same thing
 - `a+1` means L-value of `a[0]` plus as many bytes as are needed to store value of elements of `a`'s type
 - Pointer arithmetic is an address calculation with respect to the underlying representation
- **An array name is a constant pointer**
 - `a++` and `a = pa` are illegal