Principles of Programming Languages

Topic: Imperative Programming II

Professor Lou Steinberg
Fall 2004
Review

• Imperative programming:
  – based on underlying machine: RAM, thread of control
  – variable: a memory location
  – assignment

• L-values and R-values

• Pointers
  – * and &

• #INCLUDE
  – #DEFINE STACKSIZE = 10;

• printf
## L- and R-values

```c
int a; int [10] b; int *p
```

<table>
<thead>
<tr>
<th></th>
<th>L-value</th>
<th>R-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>illegal</td>
<td>12</td>
</tr>
<tr>
<td>a</td>
<td>address of variable a</td>
<td>contents of variable a</td>
</tr>
<tr>
<td>&amp;a</td>
<td>illegal</td>
<td>address of variable a</td>
</tr>
<tr>
<td>b[2]</td>
<td>address of 3rd element of b</td>
<td>contents of 3rd element of b</td>
</tr>
<tr>
<td>b</td>
<td>illegal</td>
<td>address of 1st element of b</td>
</tr>
<tr>
<td>*b</td>
<td>illegal</td>
<td>contents of 1st element of b</td>
</tr>
<tr>
<td>p</td>
<td>address of p</td>
<td>contents of p</td>
</tr>
<tr>
<td>*p</td>
<td>contents of p</td>
<td>contents of variable that p points to</td>
</tr>
</tbody>
</table>
public class List extends Object {
protected Object element;
protected List subList;
/**
 * Create a new List, initially empty.
 */
public List() {
    element = null;
    subList = null;
}

// cons operation
public List(Object newElement, List oldList) {
    element = newElement;
    subList = oldList;
}
}
How List building works?

List p = new List();

List q = new List("a", p);

List("b", q);
/*sample program to write a linear linked list of integers built like in Java, adding new elements on the front*/

#include<stdio.h>

/*this makes these definitions and variables global*/

void *malloc();

typedef struct cell listcell;

struct cell{
    int num;
    listcell *next;
};

listcell *head, *ele, *p;
```c
void main(void)
{
    int j;
    /*create first node in list*/
    head = (listcell *) malloc(sizeof (listcell));
    head->next = NULL;

    /*now create entries in list of numbers from 1 to 10*/
    head->num = 1;
    for (j=2; j<11; j++)
    {
        ele = (listcell *) malloc(sizeof (listcell));
        ele->num = j;
        ele->next = head; /**<
        head = ele;
    }

    /*now traverse the list and print the elements*/
    for (p=head; p!=NULL; p=p->next)
        printf("%d ",p->num);
    printf("\n");

    (*head).next is same as head->next
```
head->num = 1;

for (j=2; j<11; j++)
{
    ele = *(new listcell*)
    ele->num = j;
    ele->next = head; //***/
    head = ele;
}

head->num = 1;
list2.c output

scherzo!c> gcc list2.c
scherzo!c> ./a.out
10 9 8 7 6 5 4 3 2 1
scherzo!c>
Review: Stack vs Heap

- Procedure activations, statically allocated local variables, parameter values
- Lifetime same as block in which variables are declared
- Stack frame with each invocation of procedure

- Dynamically allocated data structures, whose size may not be known in advance
- Lifetime extends beyond block in which they are created
- Must be explicitly freed or garbage collected
Heap Storage

`void *malloc (size_t n)`

- returns pointer to block of contiguous storage of `n` bytes (chars), if possible
- if not enough memory left for allocation, `malloc` returns a NULL pointer
  - So you **ALWAYS** have to check the return value
- to allocate storage of a different type requires sending `malloc` the proper amount of bytes needed and casting the return pointer value appropriately

```c
head = (listcell *) malloc(sizeof (listcell));
```
listwithfree.c

/*sample program to write a linear linked list of integers built like in Java, adding new elements on the front*/
#include<stdio.h>
/*this makes these definitions and variables global*/
void *malloc();
void free();
typedef struct cell listcell;
struct cell{
    int num;
    listcell *next;
};
listcell *head, *ele, *p;

Almost same declarations as list2.c
main(void)
{
    int j;
    /*create first node in list*/
    head = (listcell *) malloc(sizeof (listcell));
    /*now create other entries in list of numbers from 1 to 10*/
    head->num = 1;
    for (j=2; j<11; j++)
    {
        ele = (listcell *) malloc(sizeof (listcell));
        ele->num = j;
        ele->next = head; /****/
        head = ele;
    }
    printf(" values in the list: ");
    /*now traverse the list and print the elements*/
    for (p=head; p!=NULL; p=p->next)
    {
        printf("%d ",p->num);
    }
    printf("\n");
}

Same building of the list and printing it out as list2.c
/*now delete the first 2 elements of the list and free their storage */
   ele = head->next;    /*1*/
   free (head); /* free 1st list element storage*/
   head = ele;
   ele = head->next;    /*2*/
   free (head); /* free 2nd list element storage*/
   head = ele;  /*3*/

/*now traverse the list and print the elements*/
   printf(" values in list after two free's: ");
   for (p=head; p!=NULL; p=p->next)
      printf(" %d ",p->num);
   printf(" \n");
   }

Trace

After `ele = head->next; /*1*/`

After `free (head);
head = ele;
ele = head->next; /*2*/`
listwithfree.c, cont.

```c
/* output
221 remus!c> a.out
values in the list: 10 9 8 7 6 5 4 3 2 1
values in list after two free's: 8 7 6 5 4 3 2 1
222 remus!c> */
```

After `free (head);`

head = ele; /*3*/

```
head - - - - - - - - -
8  

ele - - - - - - - - -

head - - - - - - - - -
8

head - - - - - - - - -
9 - - - - - - - - -

8

8 - - - - - - - - -
...```

head

ele

head

ele

After

```c
free (head);
head = ele; /*3*/
```
Functions

• Prototype:
  ```
  int mult(int, int); int square(int n);
  ```
  – often in a *.h or “header” file
  – used by compiler for type checking

• Definition:
  ```
  int square(int n) {return n*n;}
  ```

• Invocation:
  ```
  int b = 3; c = square(b);
  ```
Functions

• Parameter Passing uses “Call by Value”
  – Value of actual argument is copied into formal parameter
  – Example:

    ```c
    void swap(int a, int b)
    {
        int temp; temp = a; a = b; b = temp;
    }
    ...
    int x = 3; int y = 5;
    swap(x, y);
    printf(“%d %d”, x, y);
    --> 3 5 !?!
    – Why?
    ```
Functions

• To get the effect of “Call by Reference” use pointers
  – Example:

    ```c
    void swap(int *a, int *b)
    {
        int temp; temp=*a; *a=*b; *b=temp;
    }
    ...
    int x=3;
    int y=5;
    swap( &x, &y );
    printf("%d %d",x,y);  --> 5 3
    ```
int i;
double x;
scanf("%i %lf", &i, &x);

remus> man scanf
Structures

- **Structure**: Aggregate object with named fields
  - Declaration:
    ```c
    struct EMPLOYEE {
        int age;
        double payrate;
    }
    joe, mary;
    struct EMPLOYEE bill = {35, 65000.00};
    ```
  - Accessed as:
    ```c
    joe.age = 45;
    bill.payrate *= 1.1;
    ```
Structures

- Structure may contain actual fields, as opposed to references

```c
struct STACK {
    int top;
    int contents[5];
} stack1;
```

```c
struct STACK {
    int top;
    int *contents;
} stack1;
```
Structures

- Defining new types:
  
  ```cpp
typedef float Celsius;
typedef float Fahrenheit;

enum {BROWN, BLOND} hisHair, herHair;

typedef enum

  {BLACK, BROWN, RED, BLOND, GRAY}

  HairColors;
  ```
Structures

- Defining new structure types:

```c
typedef struct {
    float x-coord;
    float y-coord;
} Point;

struct POLYGON {
    Point vertices[10];
    int N;
} p;
```
Structures

• Pointers to Structures:

```c
typedef struct {
    int age;
    double payrate;
} Employee;
```

```c
Employee joe, *maryp;
```

• Accessed as:

```c
(*maryp).payrate = 75000.00;
(*maryp).payrate += 5000.00;
maryp->payrate += 5000.00;
```
Recursive Structures

• This is legal:

```c
typedef struct NODE {
    int data;
    struct NODE *next;
} Cell;
```

• These are not legal:

```c
typedef struct NODE {
    int data;
    struct NODE next;
} Cell;
```
Memory Allocation

• Remember the Stack and the Heap?
• Parameters and local variables can be allocated on the stack, as part of the stack frame:
  
  ```
  int n, *pn;
  pn = &n;
  ```

• But consider:
  
  ```
  Employee *maryp;
  ```

If you want to create the Employee structure that is pointed to by maryp, where does it go?

On the Heap!
Memory Allocation

• This is similar to the storage of lists in Scheme, except that you have to do it yourself!

• Standard allocation procedure:

   ```c
   Employee *maryp;
   maryp =
      (Employee *) malloc( sizeof(Employee) );
   Note: malloc returns the NULL pointer if it cannot find enough space.
   ```

• To release memory:

   ```c
   free( maryp );
   ```
Example: list.c

```
Example: list.c
```

```
head

1

2

3

at /***/ in 1st iteration

head

2

1

at /***/ in 1st iteration

head

3

2

1

at /***/ in 2nd iteration
```
Example: listwithfree.c

After `ele = head->next; /*1*/`

After `free (head);`  
`head = ele;`  
`ele = head->next; /*2*/`
Example: listwithfree.c

After `free (head);`
`head = ele; /*3*/`

/* output
59 scherzo!c> a.out
10 9 8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
60 scherzo!c> */
public class List extends Object {
protected Object element;
protected List subList;
/**
 * Create a new List, initially empty.
 */
public List() {
    element = null;
    subList = null;
}

/* cons operation */
public List(Object newElement, List oldList) {
    element = newElement;
    subList = oldList;
} ...
}
Compare: List in Java

List p = new List();

List q = new List("a", p);

List("b", q);
Pointers vs. References

- Cell and Cell* are different
- Explicit dereference operator
- Pointers needed for recursive definitions
- Casting results in type conversion:
  \[ \text{int } x = (\text{int}) (&w); \]
- Memory management is usually performed by user
- Everything is a reference
- Dereference is implicit
- Recursive definitions are automatic
- Casting just satisfies the type checker
- Memory management is usually automatic
Pointers to Functions

• Syntax:
  ```
  int foo(int x){ … }
  int (*pf)(int x);
  pf = &foo;
  ```

• Motivation: “Polymorphic” Processing
  – e.g., sorting an array of integers in increasing vs. decreasing order
  – e.g., sorting an array of integers vs. an array of floats vs. an array of Employee structs vs. ...

• How can we do this?
Pointers to Functions

• Suppose `Cell` is defined as before, but the `data` field could be an `int`, a `char`, a `struct`, or ...

• Use a “callback” function:

```c
Cell *
searchlist(Cell *c, void *value,
    int (*compare)(void *, void *)) {
    for ( ; c!=NULL; c=c->next){
        if ((*compare)(&(c->data), value))
            break;
    }
    return c;
}
```
Arrays

• Array: Typed collection of values indexed by nonnegative integers.
• Type and size must be known at compile time, except in the case of formal parameters:
  ```c
  int a[20]; void sort(int b[], int n);
  ```
• Arrays in C are one-dimensional:
  ```c
  piece chess[8][8];
  ```
• Strings are arrays of characters:
  ```c
  char msg[21] = "Enter your password:"
  ```
Arrays and Pointers

• An array name is considered to be a pointer to its first element. Thus, given the declarations:
  ```c
  int A[20], *ip;
  – A is a pointer to A[0]
  – ip = A; and ip = &A[0]; have the same meaning
  • Array subscripting is defined in terms of “pointer arithmetic”
    – *(A + 3) and A[3] have the same meaning
  • Array names are not variables, but constants
    – A++ and A = ip are illegal
• Pointer Arithmetic:

\[ \text{int } j = 5; \text{ int } *k = \&j; \]

\((*k+2)\) means \(((\ast k)+2)\): 
legal R-value, illegal L-value

\((k+2)\) : legal R-value, legal (but not necessarily meaningful) L-value

\(* (k+2)\) : legal R-value, illegal L-value

• General Rule:

Given declaration \( T \ast ptr; \)
\((ptr + k)\) points to location \( ptr + k \sim \text{sizeof}(T) \)
Arrays and Pointers

-**Two ways to process the array** A[20]
  - with subscripts:
    ```c
    int k;
    for(k=0; k < 20; k++){
        A[k] = 0;
    }
    ```
  - with pointers:
    ```c
    int *ap;
    for(ap=A; ap < A+20; ap++){
        *ap = 0;
    }
    ```
Strings and Pointers

• The library package `string.h` assumes that a string is an array of characters terminated by `\0`.
• Thus:
  ```c
  char name[5] = "Mary";
  ```
• Example:
  ```c
  int strlen(char const *str){
    int n=0;
    for( ; *str != '\0'; str++) { n+=1;}
    return n;
  }
  ```
Strings and Pointers

• Example:

```c
char *strcpy(char *dst, char const *src){
    char *str = dst;
    for( ; *src != '\0' ; src++, dst++)
    {*dst = *src;}
    *dst = '\0';
    return str;
}
```

• Caution: User must guarantee that `dst` has been allocated enough memory to store `src`
Pointers to Functions

• To search a list of integers using “less than”, define:
  
  ```c
  int
  lessthanints(void *a, void *b){
      if ( *(int *)a < *(int *)b )
          return 1;
      else return 0;
  }
  ```

• Then call:
  
  ```c
  Cell *foundit; int k=5;
  foundit = searchlist(head, &k, lessthanints);
  ```
Pointers to Functions

• To search a list of characters using “equals”, define:

```c
int equalchars(void *a, void *b){
    if ( *(char *)a == *(char *)b )
        return 1;
    else return 0;
}
```

• Then call:

```c
Cell *foundit; char ch='a';
foundit = searchlist(head, &ch, equalchars);
```