

Function Pointers & ADTs

- **EXAM Mon evening March 5**

L-values in C

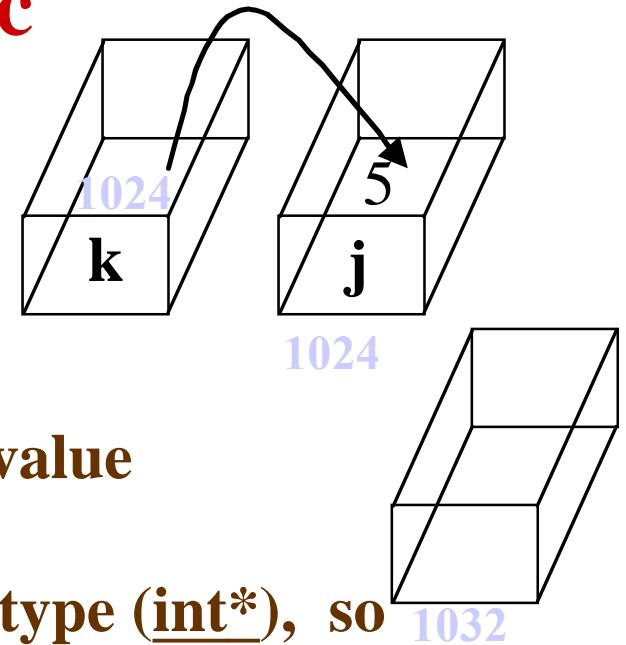
- names of arrays, functions, enumerations are not L-values; other names are.
- e.name is valid iff e is an L-value
- *e is always an L-value (whether or not e is an L-value; but type of e must be T*)
- e->name is always an L-value (...)
- e[k] is always an L-value (...)
- assignement and function calls do not produce L-values
- **&, ++, --, and left-side of =** require L-value

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**

Pointer Arithmetic

```
int j ;  
int *k;  
k=&j; /* k is legal L-value */
```



*k+2 means $((*k)+2) == 5+2 == 7$

$==(*k+2)$: legal R-value ($h = *k + 2;$), *illegal* L-value
(because its type is int)

k+2 $==1024 + 2*\text{sizeof } j == 1024+2*4 = 1032$ has type (int*), so 1032
it is a *legal* (but not meaningful) L-value

**/*need to know layout of storage to see to what (k+1)
points: the byte that is 8 bytes beyond the L-value of k,
since adding 2 is like adding storage for 2 int (4 bytes
each)*/**

*(k+2) $== *(1032) == ??$

has type int: so again an *illegal* L-value

k++ has same properties when used with *

Pointers and Arrays

- An array name is considered pointer to first element
 - **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 architecture
- An array name is not a variable
 - **a++** and **a = pa** are ILLEGAL!

Problems with Pointers

- **Uninitialized pointers**
- **Dangling pointers**
- **Storage leaks**

Casting

- **Safe uses of casting**

- **For pointers returned from *malloc***

```
p = (int *) malloc (4);
```

- **For simulating subtyping safely in C**

```
struct s{  
    int a;  
    int b;  
    double c;  
}
```

```
struct t{  
    int a;  
    int b;  
}
```

s is like a subtype of **t**
because it has same
fields as **t** plus an
extra field.

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 NULL pointer
 - So you ALWAYS have to check return the value
 - to allocate storage of a different type requires sending *malloc* the proper amount of bytes needed and casting the return pointer value appropriately
- ```
head = (listcell *) malloc(sizeof (listcell));
```

# newcasting.c

```
/*example due to satish chandra of bell labs
 this is a use of casting that is like subtyping*/
#include<stdio.h>
typedef struct{
 int x,y;
}point;
typedef enum{
 RED, BLUE
}color;
typedef struct{
 int x,y;
 color c;
}colorpoint;
void translateX(point *p, int dx){
 p ->x += dx; /*translates x co-ordinate by dx*/
}
```

# newcasting.c

```
main() {
 point p;
 colorpoint cp;
 /* initialize p to (0,0) and cp to (1,1) */
 p.x = 0;
 p.y = 0;
 cp.x = 1;
 cp.y = 1;
 cp.c = RED;
 printf(" p= %d,%d cp= %d,%d\n",p.x,p.y,cp.x,cp.y) ;
 /* move x co-ordinate by 1 for both points*/
 translateX(&p, 1);
 translateX((point *) &cp, 1);
 printf("after translation, p= %d,%d cp= %d,%d\n",
 p.x,p.y,cp.x,cp.y) ;
}
```

# Output

```
/* output resulting
20 scherzo!c> gcc newcasting.c
21 scherzo!c> a.out
 p= 0,0 cp= 1,1
after translation, p= 1,0 cp= 2,1
22 scherzo!c>
*/
```

# Pointers to Functions

- **int foo(int x) { ...}**  
**int (\*pf)( int x);** //pf points to a function of type **int (int)**  
**pf = &foo;**  
(beware **int \*pf (int x)** is parsed as **(int\*) pf (int x)** )
- **Function names are always converted to function pointers (!), so f(5), (\*pf)(5), and pf(5) all do the same thing.**
- **Typical use:**  
**Node \***  
**Search\_list(Node \*node, void const \*lookFor,**  
**int (\*compare) (void const \*, void const \*) ) {**  
**while (node!=NUL){**  
**if (\*compare)(&(amp;node-->value), lookFor) break;**  
**else node = node --> link; }**  
**return node;**  
**}**

# Pointers to Functions

- So if you define

```
int compare_ints (void const *a, void const *b) {
 if (* (int*) a < * (int*)b) return 0; else return 1; }
```

you can invoke

```
Node *where;
```

```
where = Search_list(head, desired_value, compare_ints)
```

- In general, use **void \*** when you want a *generic* pointer (because it is automatically cast into the type of the argument, IF it is also a pointer)

# Summary: why pointers in C?

- Call-by-value parameter passing means this is the only way you can affect params
- Pointers can be passed efficiently as arguments. Since arrays are pointers, it allows them to be passed in by value without copying. Also, this allows arrays to be returned.
- char pointers (arrays of characters) are the standard way of providing *strings*: a non-implemented data type
- **char\* array[10]** permits each entry (**array[i]**) to be of different length while **char array[10][20]** makes all strings have length 20
- Makes possible recursive structures
- Allows space to be allocated dynamically, at run time, usually for data structures that can grow or shrink

# Abstract Data Types

user-defined types  
with associated  
operations       $\Rightarrow$       data abstractions       $\Rightarrow$       OOPLs

***Data abstraction*** - a mechanism which encapsulates or hides the representation and operation details of a datatype from its users.

**in Modula, Ada, CLU, Smalltalk-80, C++, Java**

# Motivating Example

- **Pizza order entry system**
  - need to represent a set of toppings
  - initially allow cheese, extra cheese, or super cheese and mushrooms and anchovies
  - represent as a struct with 1 int field (cheese) = 0,1,2, and 2 booleans
- **Change: 57 flavors of pizza**
  - need new representation
  - need to fix code to use it

# Changing Pizza

- **If all parts of code use structure directly, must find and change all such uses**
- **If only part of code that depends on representation is a set of access procedures, much easier to change**
  - **also much easier to implement and debug in the first place**
- **But how can you be sure all access is through access procedures?**

# Stack ADT

- **Abstract specification - tells behavior**
  - **push: element, Stack  $\rightarrow$  Stack**
  - **pop: Stack  $\rightarrow$  Stack, element**
  - **isEmpty: Stack  $\Rightarrow$  boolean**
  - **Offers enough information for use of the ADT**
- **Concrete specification - actual implementation details**
  - **e.g., stack is stored as a list or an array**

# ADT

- **Advantages**
  - **Can modify the implementation without affecting users of the ADT**
  - **User's changes can't affect the ADT**
  - **Encourages modularity in programs**
- **Disadvantages(?)**
  - **Writer of ADT has to put more effort into design of a useful interface of operations**

# ADTs

- **ADTs are usually designed with some invariant of its behavior in mind**
  - **e.g., for Stack, top of stack is always the last element added**
    - **Need to initialize stack within the ADT**
    - **Must keep some variables private (top\_of\_stack)**
    - **Must keep some operations private to prevent illegal use (check isEmpty when doing a pop)**

# Stack in Ada

---ADT specification

**package STACKS is**

**type Stack is private;**

**procedure Push(S: in out Stack; Element: in Integer);**

**function Pop(S: in out Stack) returns Integer;**

**function Is\_empty(S: in Stack) returns Boolean;**

**private**

**type Stack is**

**record**

**space: array (1..100) of integer;**

**top: integer range 0..100 := 0;**

**end record;**

**end;**

# Stack in Ada

---ADT body (implementation)

**package body Stacks is**

**procedure Push (S: in out Stack; Element: in Integer);**

**begin**

**if S.top = 100 ---error endif;**

**S.top := S.top +1;**

**S.space(S.top) := Element;**

**end Push;**

**function Pop(S: in Stack) returns Boolean is**

**begin**

**if Is\_empty(S) --error endif;**

**S.top := S.top -1;**

**return S.space(S.top+1);**

**end Pop;**

**function Is\_empty(S: in Stack) returns Boolean is**

**begin**

**if S.top = 0 then return True**

**else return False;**

**end Is\_empty;**

**end Stacks;**

# Generics

- **What if the ADT could be parameterized by the kind of data it contains**
  - e.g., stacks of integers versus stacks of strings
    - C++ templates, Ada packages, not in Java yet, but coming!
- **What if we could write one sort routine that works on all kinds of arrays, as long as the partial ordering operator is passed in as an argument to sort?**

# Ada Generic Stack

**generic**

```
max : natural; ---a natural number
type Item is private; ---Item is the base type of the abstraction
package STACKS is
 type Stack is limited private --means that assignment
 ---and comparison operations are not defined
 ---unless they are given in the package body
 procedure Push (S: in out Stack; Element: in Item);
 function Pop(S: in out Stack) returns Item;
 function Is_empty(S: in Stack) returns Boolean;
 private
 type Stack is record
 space : array(1..max) of Item;
 top: integer range 0..max := 0;
 end record;
end;
```

an example of  
parametric  
polymorphism

# Ada Generic Stack

- **This is how we can define a particular instance of the generic.**

declare package My\_stack is **new Stack (100, Real);**---this instantiates  
---the Stack ADT to be a Stack of reals

- **Once we do this, we can use My\_stack as a stack of reals.**

Push (x, 2.5);

- **Same principle is used in almost all modern languages**

# Objects

- **What do objects have the ADTs don't?**
  - **Inheritance**
    - **Allows code sharing or reuse between related types**
  - **Control over visibility of members**
    - **Not necessary that everything is private**
  - **Some OOPs have generic classes**