Pointers

CS 217

Pointers

- Variables whose values are the addresses of other variables
- Operations
  - "address of" (reference) `*`
  - "indirection (dereference) `&`
- Declaration mimics use
  - `int *p;` `p` is an `int`, so `*p` is a pointer to an `int`

Pointers (cont)

- Suppose `x` and `y` are integers and `p` is a pointer to an integer…
  - `p = &x;` `p` gets the address of `x`
  - `y = *p;` `y` gets the value point to by `p`
  - `y = *(&x);` same as `y = x`
- Unary `*` and `&` bind more tightly than most
  - `y = *p + 1;` `y = (*p) + 1;`
  - `y = *p++;` `y = *(p++);`

Pointers (cont)

- References (e.g., **`*p`**) are variables

```
int x, y, *px, *py;
px = &x;          px is the address of x
*px = 0;          sets x to 0
py = px;          py also points to x
*py += 1;         increments x to 1
y = (*px)++;      sets y to 1, x to 2
```


Argument Passing

- Passing pointers to functions simulates passing arguments "by reference"

```
void swap(int x, int  y)
{
   int t;

   t = x;
   x = y;
   y = t;
}

int a = 1, b = 2;
swap(a, b);
printf("%d %d\n",a,b);
```

```
void swap(int *x, int  *y)
{
   int t;

   t = *x;
   *x = *y;
   *y = t;
}

int a = 1, b = 2;
swap(&a, &b);
printf("%d %d\n",a,b);
```


Pointers & Arrays

- Pointers can "walk along" arrays

```
int a[10], i, *p, x;
p = &a[0];     p is addr of 1st element of a
x = *p;        x gets a[0]
x = *(p+1);    x gets a[1]
p = p + 1;     p points to a[1]
p++;           p points to a[2];
```

Pointers/Arrays (cont)

- Array names are constant pointers

```
p = a;      p points to a[0]
a++;        illegal; can't change a constant
p++;        legal; p is a variable
```

- Subscripting is defined in terms of pointers

```
a[i]    *(a+i)    i[a] is legal too
&a[i]   a+i
p = &a[0] → &*(a+0) → &*a → a
```


Pointers/Arrays (cont)

- Pointers can walk arrays efficiently

```
p = a;
for (i = 0; i < 10; i++)
   printf("%d\n", *p++);
```


Pointer Arithmetic

- Pointer arithmetic takes into account the stride (size of) the value pointed to

```
T *p;
p += i;   increments p by i elements
p -= i;   decrements p by i elements
p++;      increments p by 1 element
p--;      decrements p by 1 element
```

- If **p** and **q** are pointers to same type **T**

```
p – q     number of elements between p and q
```

- Does it make sense to add two pointers?

Pointer Arithmetic (cont)

- Other ops: **`p < q;  <= == != >= >`**
  - **`p`** and **`q`** must point to the same array
  - no runtime checks to ensure this
- Example

```
int strlen(char *s) {
   char *p;
   for (p = s; *p; p++)
      ;
   return p – s;
}
```


Pointer & Array Parameters

- Array parameters:
  - formals are not constant; they are variables
  - passing an array passes a pointer to 1st element
  - arrays (and only arrays) are passed "by reference"

```
void f(T a[]) {. . .}
```

is equivalent to

```
void f(T *a) {. . .}
```


Pointers & Strings

- String constants denote constant ptrs to actual chars

```
char *msg = "now is the time";
```

and

```
char amsg[] = "now is the time";
char *msg = amsg;
```

`msg` points to 1st character of **`"now is…"`**

- Strings can be used whenever arrays of chars are used

```
putchar("0123456789"[i]);
```

and

```
static char digits[] = "0123456789";
putchar(digits[i]);
```

More on Parameters

- Copying strings

  `void scopy(char *s, char *t)`

  copies **`t`** to **`s`**

- Array version

```
void scopy(char s[], char t[]) {
   int i = 0;
   while ((s[i] = t[i]) != '\0')
      i++;
}
```


More on Parameters (cont)

- Pointer version

```
void scopy(char *s, char *t) {
   while (*s = *t) {
      s++;  t++;
   }
}
```

- Idiomatic version

```
void scopy(char s[], char t[]) {
   while (*s++ = *t++)
      ;
}
```


Arrays of Pointers

- Used to build tabular structures
- Indirection (**`*`**) has lower precedence than **`[]`**

  `char *line[100];`

  same as

  `char *(line[100]);`

  declares array of pointers to strings

  `*line[i]`

  refers to the 0th character of the ith string

Arrays of Pointers (cont)

- Can be initialized

```
char *month(int n) {
   static char *name[] = {
      "January",
      "February",
      . . .,
      "December" };
   assert(n >= 1 && n <= 12);
   return name[n-1];

int a, b;
int *x[] = {&a, &b, &b, &a, NULL};
```


Arrays of Pointers (cont)

- Similar to multi-dimensional arrays

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

both `a[i][j]` `b[i][j]` are legal references to **`int`**`s`

- Array **`a`**:
  - 2-dimensional 10x10 array
  - storage for 100 elements allocated at compile time
  - **`a[6]`** is a constant; **`a[i]`** cannot change at runtime
  - each row of **`a`** has 10 elements

Array of Pointers (cont)

- Array **`b`**:
  - an array of 10 pointers; each element could point to an array
  - storage for 10 pointers allocated at compile time
  - values of these pointers must be initialized at runtime
  - **`b[6]`** is a variable; **`b[i]`** can change at runtime
  - each row of **`b`** can have a different length (ragged array)

Array of Pointers (cont)

- Another example

```
void f(int *a[10]);
```

is the same as

```
void f(int **a);
```

and

```
void g(int a[][10];
```

is the same as

```
void g(int (*a)[10]);
```

`**a = 1;` is legal in both `f` & `g`

Command-Line Arguments

- By convention, **`main`** is called with 2 arguments

```
int main(int argc, char *argv[])
```

`argc` is the number of arguments

`argv` is an array of pointers to the arguments

- Example: **`echo hello world`**

```
argc = 3
argv[0] = "echo"
argv[1] = "hello"
argv[2] = "world"
argv[3] = NULL
```


Implementation of **echo**

```
int main(int argc, char *argv[]) {
   int i;
   for (i = 1; i < argc; i++)
      printf("%s%c",argv[i],
            (i < argc-1) ? ' ' : '\n');
   return 0;
}
```

Pointers to Functions

- Used to parameterize other functions

```
void sort(void *v[], int n,
     int (*compare)(void *, void *)) {
   . . .
   if ((*compare)(v[i],v[j]) <= 0) {
      . . .
   }
   . . .
}
```

- **sort** does not depend on the type of the object
  – such functions are called polymorphic

Pointers to Functions (cont)

- Use an array of **void *** (generic pointers) to pass data
- **void *** is a placeholder
  – dereferencing a **void *** requires a cast to a specific type

Pointers to Functions (cont)

- Declaration syntax can confuse:

```
int (*compare)(void *, void*)
```

declares **compare** to be a "pointer to a function that takes two **void *** arguments and returns an **int**"

```
int *compare(void *, void *)
```

declares **compare** to be a "function that takes two **void *** arguments and returns a pointer to an **int**"

Pointers to Functions (cont)

- Invocation syntax can also confuse:

```
(*compare)(v[i], v[j])
```

calls the function pointed to by **`compare`** with the arguments **`v[i]`** and **`v[j]`**

```
*compare(v[i], v[j])
```

calls the function **`compare`** with arguments **`v[i]`** and **`v[j]`**, then dereferences the value returned

- Function call has higher precedence than dereferencing

Pointers to Functions (cont)

- A function name itself is a constant pointer to a function (like an array name)

```
extern int strcmp(char *, char *);
main(int argc, char *argv[]) {
   char *v[VSIZE];
   . . .
   sort(v, VSIZE, strcmp);
   . . .
}
```

- Actually, both **`v`** and **`strcmp`** require a cast

```
sort((void **)v, VSIZE,
     (int (*)(void *, void*)strcmp);
```


Pointers to Functions (cont)

- Arrays of pointers to functions

```
extern int mul(int, int);
extern int add(int, int);
. . .
int (*operators[])(int, int) = {
   mul, add, . . .
};
```

- To invoke

```
(*operators[i])(a, b);
```

Closure

- Imagine a *string set* ADT (**`strset.h`**)

```
typedef struct Strset_T *T;
T Strset_new(void);
void Strset_free(T *set);
void Strset_insert(T set, char *str);
void Strset_delete(T set, char *str);
int Strset_memberof(T set, char *str);
void Strset_foreach(T set,
     void apply(char *str, void *cl),
     void *cl);
```


Closure (cont)

- User (client) defines the following function

```
void cardinality(char *str, void *cl) {
   int *p = cl;
   (*p)++;
}
```

- Client invokes **`Strset_foreach`** operation

```
Strset_foreach(set, cardinality);
```


Closure (cont)

- ADT implements **`Strset_foreach`**

```
void Strset_foreach(T set,
     void apply(char *str, void *cl),
     void *cl) {
   assert(set);
   assert(apply);
   while ((set = set->next) != NULL)
      apply(set->str, cl);
}
```