Pointers and Arrays

CS 217

Pointers

• What is a pointer
  o A variable whose value is the address of another variable
  o \( p \) is a pointer to variable \( v \)

• Operations
  o \&: address of (reference)
  o \*: indirection (dereference)

• Declaration mimics use
  o \( \text{int } *p; \)
    \( p \) is the address of an \( \text{int} \) (dereference \( p \) is an integer)
  o \( \text{int } v; \)
    \( p = \&v; \)
    \( p \) stores the address of \( v \)

Pointer Operation Examples

• Examples of * and &
  \( \text{int } x, y, *p; \)
  \( p = \&x; \) /* \( p \) gets the address of \( x \) */
  \( y = *p; \) /* \( y \) gets the value point to by \( p \) */
  \( y = *(&x); \) /* same as \( y = x \) */

• Unary operators associate right to left
  \( y = *&x; \) /* same as \( y = *(\&x) \) */

• Unary operators bind more tightly than binary ones
  \( y = *p + 1; \) /* same as \( y = (*p) + 1; \) */
  \( y = *p++; \) /* same as \( y = *(p++); \) */

More Pointer Examples

• References (e.g., \(*p\)) are variables
  \( \text{int } x, y, *px, *py; \)

  \( px = \&x; \) /* \( px \) 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 */

• What about the following?
  \( ++*px \)
  \( *px++ \)
**Argument Passing**

- C functions pass arguments “by value”
- To pass arguments “by reference,” use pointers

```c
void swap(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int a = 3, b = 7; swap(a, b); printf("%d %d\n", a, b);
```

```c
void swap(int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
int a = 3, b = 7; swap(&a, &b); printf("%d %d\n", a, b);
```

**Pointers and Arrays**

- Pointers can “walk along” arrays

```c
int a[10], *p, x;
p = &a[0]; /* p gets the address of a[0] */
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] */
```

- What about the following?

```c
x = *p++;
x = ++*p;
```

**Pointers and Arrays, cont’d**

- Array names are constant pointers

```c
int a[10], *p, i;
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

```c
a[i], *(a+i), i[a] /* Legal and the same */
&a[i], a+i /* Legal and the same */
p = &a[0] /* &*(a+0) Æ &a Æ a */
```

- Pointers can walk arrays efficiently

```c
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

```c
long *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

```c
p - q /* number of elements between p and q */
```

- Does it make sense to add two pointers?

```c```

```c```
Pointer Arithmetic, cont’d

- Comparison operations for pointers
  - <, >, <=, >=, ==, !=
  - if (p < q) ...;
  - p and q must point to the same array
  - no runtime checks to ensure this

- An example

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

Pointer & 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”

```c
void f(T a[])
// is equivalent to
void f(T *a)
```

Pointers & Strings

- A C string is an array of “char” with NULL at the end
- String constants denote constant pointers to actual chars
  ```c
  char *msg = “now is the time”;
  char *msg = “now is the time”;
  char *msg = amsg;
  /* msg points to 1st character of “now is the time” */
  ```
- Strings can be used whenever arrays of chars are used
  ```c
  putchar(“0123456789”[i]);
  static char digits[] = “0123456789”;
  putchar(digits[i]);
  ```

An Example: String Copy

- Array version
  ```c
  void scopy(char s[], char t[]) {
    int i = 0;
    while ((s[i] = t[i]) != ‘\0’)
        i++;
  }
  ```
- Pointer version
  ```c
  void scopy(char *s, char *t) {
    while (*s = *t) {
        s++;
        t++;
    }
  }
  ```
- Idiomatic version
  ```c
  void scopy(char s[], char t[]) {
    while (*s++ = *t++)
        ;
  }
  ```
Arrays of Pointers

- Used to build tabular structures
- Indirection “*” has lower precedence than “[]”
- Declare array of pointers to strings
  ```c
  char *line[100];
  char *(line[100]);
  ```
- Reference examples
  ```c
  line[i]  /* refers to the i-th string */
  *line[i]  /* refers to the 0-th char of the i-th string */
  ```

Arrays of Pointers, cont’d

- Initialization example
  ```c
  char *month(int n) {
    static char *name[] = {
      "January", "February", "March", "April",
      "May", "June", "July", "August",
      "September", "October", "November", "December"
    };
    assert(n >= 1 && n <= 12);
    return name[n-1];
  }
  ```
- Another example
  ```c
  int a, b;
  int *x[] = {&a, &b, &b, &a, NULL};
  ```

Arrays of Pointers, cont’d

- An array of pointers is a 2-D array
  ```c
  int a[10][10];
  int *b[10];
  ```
- Array a:
  - 2-dimensional 10x10 array
  - Storage for 100 elements allocated at compile time
  - Each row of a has 10 elements, cannot change at runtime
  - a[6] is a constant
- 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
  - Each row of b can have a different length (ragged array)
  - b[6] is a variable; b[i] can change at runtime

More Examples

- Equivalence example
  ```c
  void f(int *a[10]);
  void f(int **a);
  ```
- Another equivalence example
  ```c
  void g(int a[][10]);
  void g(int (*a)[10]);
  ```
- Legal in both f and g:
  ```c
  **a = 1;
  ```
Command-Line Arguments

- By convention, `main()` is called with 2 arguments
  - `int main(int argc, char *argv[])`
  - `argc` is the number of arguments, including the program name
  - `argv` is an array of pointers to the arguments

- Example:
  ```
  % echo hello
  argc = 2
  argv[0] = "echo"
  argv[1] = "hello"
  argv[2] = 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’d

- Use an array of `void*` (generic pointers) to pass data
- `void*` is a placeholder
  - Dereferencing a `void *` requires a cast to a specific type
- Declaration syntax can be confusing:
  - `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’d

• 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, strcmp);

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);

Summary

• Pointers
  - “type *” (int *p) declares a pointer variable
  - * and & are the key operations

• Operation rules
  - Unary operations bind more tightly than binary ones
  - Pointer arithmetic operations consider size of the elements

• Pointers and arrays have a tight relationship
  - An array is a constant pointer pointing to the 1st element
  - A pointer can walk through elements of an array
  - An array of pointers is a 2-D array (1-D fixed and another variable)
  - Master how to get command-line arguments from main()

• Pointers to functions
  - Can be used to parameterize functions