Pointers and Arrays

CS 217

Pointers

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

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

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

More Pointer Examples

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

House address: pointer to the variable: \&x, p
House: name of the variable: x
You: the value contained in the variable x, like 58

int \( x; \) build a house of type int and name \( x \)
int *\( p; \) \( p \) can contain an address to any int-type house (decl)
\( p = &x; \) \( p \) is now the address of house \( x \) (init)
\( x = 58; \) the person 58 moves into house \( x \)
\( *p = 58; \) the person 58 moves into the house at address \( p \) (use)

\( \&x \) and \( p \) are equivalent (\& returns address of house)
\( x \) and \( *p \) are equivalent (\* gets to house at address)
More Pointer Examples

- References (e.g., *p) are variables
  
  ```c
  int x, y, *px, *py;
  ```

- ```
px = &x; /* px is the address of x */
```
More Pointer Examples

• References (e.g., *p) are variables

```c
int x, y, *px, *py;
```

```c
px = &x;       /* px is the address of x */
*px = 0;        /* sets x to 0 */
py = px;        /* py also points to x */
```

![Diagram](image1)

```c
int x, y, *px, *py;
```

```c
px = &x;       /* px is the address of x */
*px = 0;        /* sets x to 0 */
py = px;        /* py also points to x */
```

![Diagram](image2)

```c
int x, y, *px, *py;
```

```c
px = &x;       /* px is the address of x */
*px = 0;        /* sets x to 0 */
py = px;        /* py also points to x */
```

```c
*py += 1;      /* increments x to 1 */
```

![Diagram](image3)
More Pointer Examples

- References (e.g., \(^*p\)) are variables
  
  ```c
  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 */
  
  py = px; /* sets y to 1, x to 2 */
  ```

- References (e.g., \(^*p\)) are variables
  
  ```c
  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 */
  ```
Operator Precedents

- Unary operators associate right to left
  
  \[ y = \&\& x; \quad /* \text{same as} \quad y = \& (\& x) */ \]

- Unary operators bind more tightly than binary ones
  
  \[ y = * p + 1; \quad /* \text{same as} \quad y = (* p) + 1; */ \]

- More examples
  
  \[ y = * p + 1; \quad /* \text{same as} \quad y = * p; \quad p + 1; */ \]

- When in doubt, liberally use parentheses

Argument Passing

- C functions pass arguments “by value”

```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);
```

```plaintext
a 3
b 7
```
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
int a = 3, b = 7;
swap(&a, &b);
printf("%d %d\n",a,b);
```

\[\begin{array}{cc}
x & 3 \\
y & 7 \\
a & 3 \\
b & 7 \\
\end{array}\]

\[\begin{array}{cc}
x & 7 \\
y & 3 \\
a & 3 \\
b & 7 \\
\end{array}\]

Formatted Input: scanf

- Example
  - `double v; scanf( "%lf", &v );`
  - `int day, month, year; scanf( "%d/%d/%d", &month, &day, &year );`
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] */
```

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] */
```
Pointers and Arrays, cont’d

- Array names are constant pointers
  ```c
  int a[10], *p, i;
  p = a;    /* p points to a[0] */
  a = p;    /* Illegal; can’t change a constant */
  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) \rightarrow &a \rightarrow a */
  ```

- Pointers can walk arrays efficiently
  ```c
  p = a;
  for (i = 0; i < 10; i++)
    printf( "%d\n", *p++ );
  ```

```
   1 0 0
```

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 (longs, or chars) between p and q */
  ```

- Does it make sense to add two pointers?

```
   s p
```

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;
  }
  ```

```
   1 0 0 0 2 0 0 0 3 0 0 4 0 0 0
```

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”

- Declaration:
  ```c
  void f(int a[])
  ```
  ```c
  is equivalent to
  ```
  void f(int *a)
  ```
  ```c
  Call:
  ```
  int a[10];
  ```
  ```c
  f(a);
  ```
Pointers & Strings

• A C string is an array of “char” with NULL at the end

now is the time

• String constants denote constant pointers to actual chars
char *msg = "now is the time";
char amsg[] = "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
static char digits[] = "0123456789";
putchar("0123456789\[i\]);
putchar(digits[i]);

• Pointers and arrays are essentially the same thing => Pointers to chars and arrays of chars are the same

An Example: String Copy

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

• 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

• Declare array of pointers to strings
char *line[100];
char *(line[100]); /* same as above */
char *(line[100]); /* never used */

• Reference examples
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
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
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
  - 2-d arrays are stored in “row-major” order

- 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]); /* known number of rows */
  void f(int **a);
  ```

- Another equivalence example
  ```c
  void g(int a[][10]); /* known number of columns */
  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:
  ```c
  % echo hello
  argc = 2
  argv[0] = "echo"
  argv[1] = "hello"
  argv[2] = NULL
  ```

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

Pointers to Functions

```c
#include <stdio.h>
int add(int x, int y) {
  return x + y;
}

int mul(int x, int y) {
  return x * y;
}

main() {
  int a[5] = {1, 2, 3, 4, 5};
  int sum = doArray(a, 5, 0, add);
  int prod = doArray(a, 5, 1, mul);
  printf("sum = %d, product = %d\n", sum, prod);
}
```
Points to Functions, cont’d

• Declaration syntax can be confusing:
  ○ `int (*op)(int, int)`
    declares `op` to be a “pointer to a function that takes two `int` arguments and returns an `int`”
  ○ `int *op(int, int)`
    declares `op` to be a “function that takes two `int` arguments and returns a pointer to an `int`”

• Invocation syntax can also confuse:
  ○ `(*op)(x, y)`
    calls the function pointed to by `op` with the arguments `x` and `y`, equivalent to `op(x, y)`
  ○ `*op(x, y)`
    calls the function `op` with arguments `x` and `y`, then dereferences the value returned

• Function call has higher precedence than dereferencing

Points to Functions, cont’d

• A function name itself is a constant pointer to a function (like an array name)
  ```c
  int add(int x, int y) {...}
  int mul(int x, int y) {...}
  int sum = doArray(a, 5, 0, add);
  int prod = doArray(a, 5, 1, mul);
  ```

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