Pointers

- Pointers are variables whose ***values*** are the ***addresses*** of other variables
- Basic operations

  “address of” (reference)

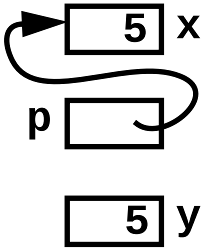
  “indirection” (dereference)

- Suppose **`x`** and **`y`** are integers, **`p`** is a pointer to an integer:

  `p = &x;`   **`p`** gets the address of **`x`**

  `y = *p;`   **`y`** gets the value pointed to by **`p`**

  `y = *(&x);`



- Declaration syntax mimics use of variables in expressions

  `int *p;`   **`*p`** is an **`int`**, so **`p`** is a pointer to an **`int`**

- Unary **`*`** and **`&`** bind more tightly than most other operators

```
y = *p + 1;   y = (*p) + 1;
y = *p++;     y = *(p++);
```

Pointer References

- Pointer 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
```

- 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);

2 1
```

Pointers & Arrays

- Pointers can “walk along” arrays

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

`p = &a[0];`	**p** is the address of the 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]**, ***by definition***
`p++;`	**p** points to **a[2]**

- 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, for any type, 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 can walk along 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`	increment **`p`** by **`i`** elements
`p -= i`	decrement **`p`** by **`i`** elements
`p++`	increment **`p`** by **`1`** element
`p--`	decrement **`p`** by **`1`** element

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

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

- Does it make sense to add two pointers?

- Other operations: **`p < q`**; **`<= == != >= >`**

  `p` and **`q`** ***must*** point to the ***same*** array; ***no runtime checks*** to insure this

- Example

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

Pointers & Array Parameters

- Array parameters:

  array formal parameters are not constants, they are ***variables***

  passing an array passes a ***pointer*** to the ***first element***

  arrays (and ***only*** arrays) are automatically passed “by reference”

  void f(*T* a[]) {...} is equivalent to **void f(*T* *a) {...}**

- String constants denote constant pointers to the actual characters

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

  msg points to the first character of **"now is ..."**

- Strings can be used wherever arrays of characters are used

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

- Is there any difference between

```
extern char x[];           extern char *x;
```

Pointers & Array Parameters, cont’d

- 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++;
}
```

- ***Pointer*** version:

```
void scopy(char *s, char *t) {
    while (*s = *t) {                      while ((*s = *t) != 0)
        s++;
        t++;
    }
}
```

- ***Idiomatic*** version:

```
void scopy(char *s, char *t) {
    while (*s++ = *t++)                    while ((*s++ = *t++) != 0)
        ;
}
```

- **Which one is better and why?**

Arrays of Pointers

- Arrays of pointers help build tabular structures

- Indirection (`*`) has ***lower*** precedence than `[]`

  `char *line[100];` same as **`char *(line[100]);`**

  declares an array of pointers to char (strings); declaration mimics use:

  `*line[i]`

  refers to the 0th character in the **`i`**th string

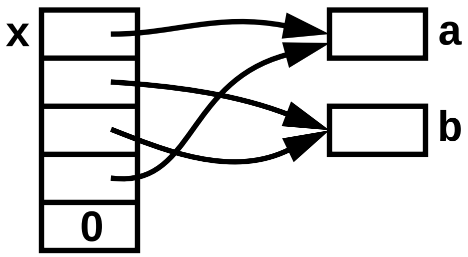
- Arrays of pointers can be ***initialized***

`name` is visible only within `month`;
allocated & initialized at *compile time*

```
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’d

- Arrays of pointers are ***similar*** to multi-dimensional arrays, but different

```
int a[10][10];     both    a[i][j]
int *b[10];                b[i][j]
                   are legal references to ints
```

- Array **`a`**:

  2-dimensional 10x10 array

  storage for 100 elements allocated at compile time

  `a[6]` is a ***constant***; **`a[i]`** ***cannot*** change during execution

  each row of **`a`** has 10 elements

- Array **`b`**:

  an array of 10 pointers; each element ***could*** point to an array

  storage for 10 pointer elements allocated at compile time

  values of these pointers must be initialized during execution

  `b[6]` is a ***variable*****`;`** **`b[i]`** ***can*** change during execution

  each row of **`b`** can have a different length; “ragged array”

Command-Line Arguments

- By convention, **`main`** is called with 2 arguments (actually 3!)

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

  `argc` ("***arg***ument ***c***ount") is the number of command-line arguments

  `argv` ("***arg***ument ***v***ector") is an array of pointers to the arguments

- For the command ***`echo hello, world`***

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

- **`NULL`** is the ***null pointer***, which points to nothing; defined to be 0

- Implementation of **`echo`**:

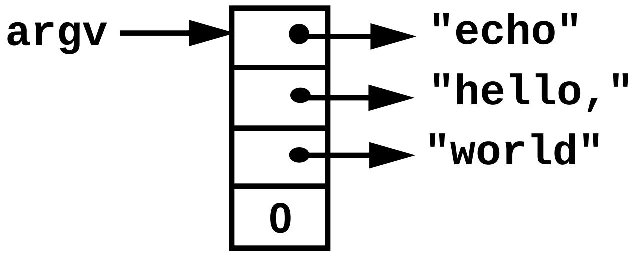
```
int main(int argc, char *argv[]) {
    int i;
    for(i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i < argc-1) ? ' ' : '\n');
    return 0;
}
```

More on argc and argv

- Another (less clear) implementation of **`echo`**:

```
int main(int argc, char **argv) {
    while (--argc > 0)
        printf("%s%c", *++argv, argc > 1 ? ' ' : '\n');
    return 0;
}
```

initially, **`argv`** points to the program name:



`*++argv` increments **`argv`** to point the cell that points to **`"hello,"`**, and indirection fetches that pointer (a **`char *`**)

- Example

`void f(int *a[10]);` is the same as **`void f(int **a);`**
`void g(int a[][10]);` **`void g(int (*a)[10]);`**

`**a = 1;` is legal in ***<u>both</u>*** **`f`** and **`g`**; what gets changed in each?

- See H&S for more

Pointers to Functions

- Pointers to functions help ***parameterize*** other functions

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

- **`sort`** does not depend the type of the objects it’s sorting

  it can sort arrays of pointers to ***any*** type

  such functions are called ***generic*** or ***polymorphic*** functions

- 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'd

- 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`**"

- 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 the arguments **`v[i]`** and **`v[j]`**, then ***dereferences*** the pointer 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 array name)

```
#include <string.h>  contains 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);
```

- Arrays of pointers to functions:

```
extern int mul(int, int), add(int, int), sub(int, int), ...;
```

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

to call the ***i***th function: **`(*operators[i])(a, b);`**