Pointers

· Pointers are variables whose values are the addresses of other variables

· Basic operations

ªaddress ofº (reference)
ªindirectionº (dereference)

· Suppose \( x \) and \( y \) are integers, \( p \) is a pointer to an integer:

\[
\begin{align*}
\text{p} &= & & \&x; & & \text{p} & \text{gets the address of } x \\
\text{y} &= & & *p; & & \text{y} & \text{gets the value pointed to by } p \\
\text{y} &= & & *(\&x); & & \text{y} & \text{gets the address of } x \\
\end{align*}
\]

· Declaration syntax mimics use of variables in expressions

\[
\begin{align*}
\text{int } *p; & & \text{p} & \text{is an } \text{int} & & \text{so } p & \text{is a pointer to an } \text{int} \\
\text{Unary } * & & \text{bind more tightly than most other operators} \\
\end{align*}
\]

· Pointer references (e.g. \(*p\) ) are variables

\[
\begin{align*}
\text{int } x, y, \ast px, \ast py; & & \text{px} & \text{is the address of } x \\
\text{px} &= & & \&x; & & \text{px} & \text{sets } x \text{ to 0} \\
\text{\ast px} &= & & 0; & & \text{\ast px} & \text{sets } x \text{ to 0} \\
\text{py} &= \text{px}; & & \text{py} & \text{also points to } x \\
\text{\ast py} &= & & 1; & & \text{\ast py} & \text{increments } x \text{ to 1} \\
\text{y} &= &(\text{\ast px})++; & & y & \text{sets } y \text{ to 1, } x \text{ to 2} \\
\end{align*}
\]

· Passing pointers to functions simulates passing arguments ªby referenceº

\[
\begin{align*}
\text{void swap(int } x, \text{ int } y) & & \text{t} & \text{= x; } \\
& & \text{t} & \text{= y; } \\
& & \text{y} & \text{= t; } \\
\end{align*}
\]

\[
\begin{align*}
\text{int } a = 1, b = 2; & & \text{swap}(a, b); & & \text{printf}(\text{"%d %d\n"}, a, b); \\
\end{align*}
\]

Pointers & Arrays

· Pointers can ªwalk alongº arrays

\[
\begin{align*}
\text{int } a[10], i, \ast p, x; & & \text{p} & \text{is the address of the } 1\text{st element of } a \\
\text{x} &= \ast p; & & \text{x} & \text{gets } a[0] \\
\text{x} &= \ast (p + 1); & & \text{x} & \text{gets } a[1] \\
\text{p} &= p + 1; & & \text{p} & \text{points to } a[2], \text{by definition} \\
\text{p} &= p++; & & \text{p} & \text{points to } a[3] \text{, by definition} \\
\end{align*}
\]

· Array names are constant pointers

\[
\begin{align*}
\text{p} &= a; & & \text{p} & \text{points to } a[0] \\
\text{a} &= a++; & & \text{a} & \text{is illegal; can't change a constant} \\
\text{p} &= p++; & & \text{p} & \text{is a variable} \\
\end{align*}
\]

· Subscripting, for any type, is defined in terms of pointers

\[
\begin{align*}
\text{a}[i] & & \text{is legal, too!} \\
\&a[i] & & \text{is the address of the } i\text{th element of } a \\
\&*(a + i) & & \text{is the address of the } i\text{th element of } a \\
\end{align*}
\]

· Pointers can walk along arrays efficiently

\[
\begin{align*}
\text{p} &= a; & & \text{for (i = 0; i < 10; i++)} \\
& & \text{printf}(%d\n\text{"}, \ast p++); \\
\end{align*}
\]

Pointer Arithmetic

· Pointer arithmetic takes into account the size of the value pointed to

\[
\begin{align*}
&\pm x & & \text{is legal, but } x & & \text{must point to the same array} \\
&\pm \text{variable} & & \text{must point to the same type} \\
\text{t} &= & & \pm d; & & \text{t} & \text{get } d+1 \text{th element} \\
\text{t} &= & & \pm d; & & \text{t} & \text{get } d-1 \text{th element} \\
\text{t} &= & & \pm 1; & & \text{t} & \text{get } d \text{th element} \\
\text{t} &= & & \pm x; & & \text{t} & \text{get } x \text{th element} \\
\text{t} &= & & \pm \text{variable} & & \text{must point to the same type} \\
\text{t} &= & & \pm \text{variable} & & \text{must point to the same type} \\
\text{if } & & \text{ptr1} & = & & \text{ptr2} & & \text{ptr1} & \text{and } \text{ptr2} & \text{must point to the same array} \\
\end{align*}
\]

· Other operators:

\[
\begin{align*}
\& & & \text{refers to the address of } d \\
\&\& & & \text{is the address of the 1st element of } a \\
\pm & & & \text{is legal, but } x & & \text{must point to the same array} \\
\pm & & & \text{variable} & & \text{must point to the same type} \\
\end{align*}
\]

· If \( p \) and \( q \) are pointers to the same type

\[
\begin{align*}
\text{p} &= & & \pm q; & & \text{p} & \text{equals } q \text{ plus } d \text{ elements} \\
\end{align*}
\]

· Does it make sense to add two pointers?

\[
\begin{align*}
\text{\&} & & \text{refers to the address of } d \\
\end{align*}
\]

· Other operations:

\[
\begin{align*}
\pm & & \text{is legal, but } x & & \text{must point to the same array} \\
\pm & & \text{variable} & & \text{must point to the same type} \\
\end{align*}
\]

· Example

\[
\begin{align*}
\text{int } \text{strlen(char } *s) & & \text{char } *p; \\
& & \text{for (p = s; } *p; p++) \\
& & \text{return } p - s; \\
\end{align*}
\]

· Pointer references (e.g. \(*p\) ) are variables whose values are the addresses of other variables
Pointers & Array Parameters

- Array parameters:
  - Array formal parameters are not constants, they are variables.
  - Passing an array passes a pointer to the 0th element.
  - Arrays (and only arrays) are automatically passed by reference.

\[ \text{void \( f(A[]) \)} \] is equivalent to \[ \text{void \( f(A[]) \)} \]

- String constants denote constant pointers to the actual characters.

\[ \text{char \*msg = \"now is the time\";} \]
\[ \text{char amsg[\]} = \"now is the time\";} \]
\[ \text{\textbf{char \*msg = amsg;}} \]
\[ \text{msg points to the first character of \"now is \...\";} \]

- Strings can be used wherever arrays of characters are used.

\[ \text{putchar(digits[i]);} \]
\[ \text{putchar(digits[i]);} \]

- Is there any difference between \[ \text{extern char x[];} \] and \[ \text{extern char \*x;} \]?

Pointers & Arrays

- Arrays of pointers can be used wherever arrays of characters are used.

\[ \text{String \( s \)} \]
\[ \text{Siests \( s \)} \]

- Arrays of pointers help build tabular structures.

Pointers & Arrays, cont'd

- Arrays of pointers can be used to build tabular structures.

\[ \text{void scopy(char \*s, char \*t)} \]
\[ \text{copies \( t \)} \]
\[ \text{to \( s \)} \]

- Arrays of pointers help build tabular structures.

Pointers & Arrays, cont'd

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

\[ \text{int \( a[10][10] \)} \]
\[ \text{int \( b[10] \)} \]
\[ \text{are legal references to \textbf{int}s;} \]
\[ \text{each row of \( a \) has 10 elements;} \]
\[ \text{storage for 100 elements allocated at compile time;} \]
\[ \text{\( a[6] \) is a constant;} \]
\[ \text{\( a[i] \) cannot change during execution;} \]
\[ \text{each row of \( b \) can have a different length;} \]
\[ \text{\( b[6] \) is a variable;} \]
\[ \text{\( b[i] \) can change during execution;} \]
\[ \text{storage for 10 pointer elements allocated at compile time;} \]
\[ \text{values of these pointers must be initialized during execution;} \]
\[ \text{an array of 10 pointers; each element \textbf{could} point to an array;} \]
\[ \text{An array \( a \):} \]
\[ \text{2-dimensional 10x10 array;} \]
\[ \text{storage of 100 elements allocated at compile time;} \]
\[ \text{\( a[6] \) is a constant;} \]
\[ \text{\( a[i] \) cannot change during execution;} \]
\[ \text{An array \( b \):} \]
\[ \text{an array of 10 pointers; each element \textbf{could} point to an array;} \]
\[ \text{An array \( b \):} \]
\[ \text{2-dimensional 10x10 array;} \]
\[ \text{storage of 10 pointer elements allocated at compile time;} \]
\[ \text{values of these pointers must be initialized during execution;} \]
\[ \text{an array of 10 pointers; each element \textbf{could} point to an array;} \]
\[ \text{An array \( b \):} \]
\[ \text{2-dimensional 10x10 array;} \]
\[ \text{storage of 10 pointer elements allocated at compile time;} \]
\[ \text{values of these pointers must be initialized during execution;} \]
\[ \text{an array of 10 pointers; each element \textbf{could} point to an array;} \]

Pointers & Arrays, cont'd

- Arrays of pointers help build tabular structures.

Pointers & Arrays, cont'd

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

\[ \text{int \( a[10][10] \)} \]
\[ \text{int \( b[10] \)} \]
\[ \text{are legal references to \textbf{int}s;} \]
\[ \text{each row of \( a \) has 10 elements;} \]
\[ \text{storage for 100 elements allocated at compile time;} \]
\[ \text{\( a[6] \) is a constant;} \]
\[ \text{\( a[i] \) cannot change during execution;} \]
\[ \text{each row of \( b \) can have a different length;} \]
\[ \text{\( b[6] \) is a variable;} \]
\[ \text{\( b[i] \) can change during execution;} \]
\[ \text{each row of \( b \) can have a different length;} \]
\[ \text{\( b[6] \) is a variable;} \]
\[ \text{\( b[i] \) can change during execution;} \]

Pointers & Arrays, cont'd

- Arrays of pointers help build tabular structures.

Pointers & Arrays, cont'd

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

\[ \text{int \( a[10][10] \)} \]
\[ \text{int \( b[10] \)} \]
\[ \text{are legal references to \textbf{int}s;} \]
\[ \text{each row of \( a \) has 10 elements;} \]
\[ \text{storage for 100 elements allocated at compile time;} \]
\[ \text{\( a[6] \) is a constant;} \]
\[ \text{\( a[i] \) cannot change during execution;} \]
\[ \text{each row of \( b \) can have a different length;} \]
\[ \text{\( b[6] \) is a variable;} \]
\[ \text{\( b[i] \) can change during execution;} \]

Pointers & Arrays, cont'd

- Arrays of pointers help build tabular structures.

Pointers & Arrays, cont'd

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

\[ \text{int \( a[10][10] \)} \]
\[ \text{int \( b[10] \)} \]
\[ \text{are legal references to \textbf{int}s;} \]
\[ \text{each row of \( a \) has 10 elements;} \]
\[ \text{storage for 100 elements allocated at compile time;} \]
\[ \text{\( a[6] \) is a constant;} \]
\[ \text{\( a[i] \) cannot change during execution;} \]
\[ \text{each row of \( b \) can have a different length;} \]
\[ \text{\( b[6] \) is a variable;} \]
\[ \text{\( b[i] \) can change during execution;} \]

Pointers & Arrays, cont'd

- Arrays of pointers help build tabular structures.
Command-Line Arguments

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

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

- `argc` (argument count) is the number of command-line arguments.
- `argv` (argument vector) 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, defined to be 0.

Declarations:

- Pointers to functions help parameterize other functions.
- `void *` is a placeholder.
- Dereferencing a `void *` cast to a specific type fails.
- Use an array of `void *` (generic pointers) to pass data.
- Such functions are called polymorphic or generic functions, as can a set array of pointers to any type.
- `void *` does not depend the type of the objects it is storing.

Pointers to functions help parameterize other functions.

```
void sort(void *v[], int n, int (*compare)(void *, void *)) {
    ...
    ...
    ...
}
```

`sort` does not depend the type of the objects it is sorting.

- Example:
  ```c
generic void f(int *a[10]);
generic void g(int a[][10]);
void g(int (*a)[10]);
**a = 1;
```

**`a` is legal in both `f` and `g`; what gets changed in each?**

- See H&S for more on `void` and `void *`.
- Pointers to functions help parameterize other functions.
- Declarations can also confuse:
  ```c
defines a function that takes `void *` arguments and returns it.
  ...
  ...
  ...
```

Invocation syntax can also confuse:

```c
defines a function that takes four `void *` arguments and returns it.
  ...
  ...
  ...
```

Example:

```c
void (*v)(int, int); v = &f;
```

- By convention, `main` is called with 2 arguments (actually 3!).
- Function call has higher precedence than dereferencing.
- The pointer value returned calls the function `void (int, int) (*v)(int, int)`.
- Invocation syntax:
  ```c
  (*v)(int, int)
  ```
- Dereferencing syntax:
  ```c
  v(int, int)
  ```

**Function call has higher precedence than dereferencing.**

```
Another (less clear) implementation of `echo`:
```
Pointers to Functions, cont'd

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

```c
#include <string.h>
extern int strcmp(char *, char *);

main(int argc, char *argv[]) {
    char *v[VSIZE];
 ...
    sort(v, VSIZE, strcmp);
 ...
    for (int i = 0; i < VSIZE; i++)
        if ([i] == VSIZE) {
            continue;
        }
        if (strcmp([i], VSIZE) == 0)
            break;
    }
}
```

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

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

- Arrays of pointers to functions:

```c
extern int mul(int, int), add(int, int), sub(int, int), ...
int (*operators[])(int, int) = {
    mul, add, sub, ...
};
to call the i-th function:

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