Lecture 12. Pointers

- Variables denote ***locations in memory*** that can hold values; arrays denote ***contiguous locations***

```
int i = 8, sum = -456;
float average = 34.5;
unsigned count[4];
```

Address	Value	Name
$09A8_{16}$	8	i
$09AC_{16}$	-456	sum
$09B0_{16}$		
$09B4_{16}$	34.5	average
	⋮	
$0F10_{16}$		count[0]
$0F14_{16}$		count[1]
$0F18_{16}$		count[2]
$0F1C_{16}$		count[3]

- The ***location*** of a variable is its ***lvalue*** or ***address***; the contents stored in that location is its ***rvalue***
- A ***pointer*** is a variable whose ***rvalue*** is the ***lvalue*** of another variable — the address of that variable
- Pointers are typed: a 'pointer to an `int`' may hold only the lvalue of an `int` variable

  If `p` points to `sum`, `q` points to `count[2]`:

```
int *p; unsigned *q;

p = &sum;
q = &count[2];
```

Address	Value	Name
$13A4_{16}$	09AC	p
$13A8_{16}$	0F18	q

  `p` and `q` ***cannot*** point to `average`

- The ***null pointer*** — denoted `NULL` — points to ***nothing***

```
p = NULL;
```

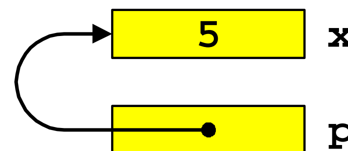
Pointer Operations

- **Two fundamental operations: *creating* pointers, *accessing* the values they point to**

  unary `&` 'address of' returns the address of its *lvalue* operand as an *rvalue*

  unary `*` 'indirection' returns the *lvalue* given by its *pointer* operand's *rvalue*

  Suppose `x` and `y` are `int`s, `p` is a pointer to an `int`



  `p = &x;` **`p` is assigned the address of `x`**

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

  `y = *(&x);` **same as `y = x`**

- **Declaration syntax for pointer types *mimics the use* of pointer variables in expressions**

```
int x, y;
int *p;
```

  `*p` **is an `int`, so `p` must be a pointer to an `int`**

- **Unary `*` and `&` have higher precedence than most other operators**

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

Indirection

- **Pointer indirection (e.g., `*p`) yields an *lvalue* — a *variable* — and pointer values can be manipulated like other values**

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

Code	Effect	Side effect
`px = &x;`	`px` is the address of `x`	no effect on `x`
`*px = 0;`	sets `x` to 0	no effect on `px`
`py = px;`	`py` also points to `x`	no effect on `px` or `x`
`*py += 1;`	increments `x` to 1	no effect on `px` or `py`
`y = (*px)++;`	sets `y` to 1, `x` to *2*	no effect on `px` or `py`

- **Passing pointer arguments *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);

1 2
```

```
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
```

Copyright ©1996 David R. Hanson

Pointers and Arrays

- **Pointers can ‘walk along’ arrays by pointing to each element in turn**

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

`p = &a[0];`	`p` **is assigned the address of the 1st element of** `a`
`x = *p;`	`x` **is assigned** `a[0]`
`x = *(p + 1);`	`x` **is assigned** `a[1]`
`p = p + 1;`	`p` **is assigned the address of** `a[1]`, ***by definition***
`p++;`	`p` **points to** `a[2]`

- **Pointer arithmetic: If `p` points to `a[i]`, `p + k` points to `a[i+k]`**

- **An array name is a *constant* pointer to the first element of the array**

`p = a;`	`p` **is assigned the address of** `a[0]`
`a++;`	***illegal*: can’t change a constant**
`p++;`	**legal: `p` is a variable**

- **The idiom `*p++` walks along the array pointed to by `p`**

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

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

Both loops print the same output, both are efficient, both are acceptable

Copyright ©1996 David R. Hanson

Pointers and Array Parameters

- **An array parameter type is identical to a pointer to the element type**

  Array parameters are *not* constants, they are *variables*

  Passing an array as an actual argument passes a *pointer* to the *first element*

  In effect, arrays — and *only* arrays — are passed *by-reference*

```
void print(int x[], int size) {
    int i;

    for (i = 0; i < size; i++)
        printf("%d\n", x[i]);
}
```

```
void print(int *x, int size) {

    while (size-- > 0)
        printf("%d\n", *x++);
}
```

- **A *string* is an *array of characters*; the name of a character array is thus a `char *`**
- **String functions can be written using arrays or pointers, but often *return pointers***

  `char *strcpy(char *dst, char *src)` copies `src` to `dst`, then returns `dst`

```
char *strcpy(char dst[], char src[]) {
    int i;

    for (i = 0; src[i] != '\0'; i++)
        dst[i] = src[i];
    dst[i] = '\0';
    return dst;
}
```

Copyright ©1996 David R. Hanson

Pointers and Array Parameters, cont'd

- **Pointer version**

```
char *strcpy(char *dst, char *src) {
    char *d = dst, *s = src;

    while (*d = *s) {                   while ((*d = *s) != '\0')
        d++;
        s++;
    }
    return dst;
}
```

- ***Idiomatic* version**

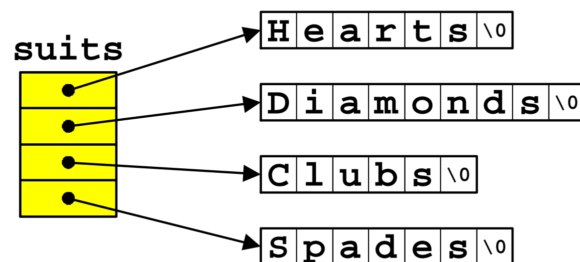
```
char *strcpy(char *dst, char *src) {
    char *d = dst;

    while (*dst++ = *src++)             while ((*dst++ = *src++) != '\0')
        ;
    return d;
}
```

- **Pointer versions *might* be faster, but strive for *clarity*, not microefficiency**

Copyright ©1996 David R. Hanson

Arrays of Pointers

- **Arrays of pointers help build tabular structures**




```
char *suits[] = {
     "Hearts", "Diamonds", "Clubs", "Spades"
};

char *faces[] = {
     "Ace", "2", "3", "4", "5", "6", "7", "8",
     "9", "10", "Jack", "Queen", "King"
};
```

Declare `suits` and `faces` each to be an 'array of pointers to characters,' *not* 'a pointer to an array of characters', and initialize them as shown

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

`char *suits[];` is the same as `char *(suits[]);`

Declaration mimics use: `*suits[i]` refers to the 0th character in the `i`th string

```
printsuit(int card) {
     printf("%c", *suits[card%13]);
}
```

- **A string constant is shorthand for the name of an array of characters**

```
print("0123456789ABCDEF"[n%b]);     char digits[] = "0123456789ABCDEF";
                                    print(digits[n%b]);
```

Copyright ©1996 David R. Hanson

Common Errors

- Only ***addresses*** can be assigned to pointers

```
int *p, i;
p = i;                      p = &i;
```

- Only addresses of variables of the ***correct types*** can be assigned to pointers

```
int *p;                     float *p;
float x;
p = &x;
```

- Only pointers can be used with ***indirection***

```
p = *i;                     i = *p; ?
```

- Pointers must be ***initialized*** to valid addresses ***before*** using indirection

```
                            p = &i;
*p = 5;
printf("%d\n", *p);
```

- The null pointer must ***not*** be dereferenced, because it points to ‘nothing’

```
p = NULL;                   p = &i;
*p = 6;
```

Copyright ©1996 David R. Hanson

Common Errors, cont'd

- **Pointers must point to variables that *exist*! See page 4-8**

```
int *SumPtr(int a, int b) {
    int sum = a + b;

    return &sum;
}

p = SumPtr(2, 5);
printf("%d\n", *p);
```

`sum` **does not exist!**

```
char *itoa(int n) {
    char buf[100];

    sprintf(buf, "%d", n);
    return buf;
}

char *s;
s = itoa(56);
printf("%s\n", s);
```

`buf` **does not exist!**

`sprintf` **is like** `printf`**, but stores the 'output' in a string**

- **When faced with bugs involving a pointer, ask: Is this pointer initialized? Does the memory it points to exist?**

Copyright ©1996 David R. Hanson