

---

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {
    char a[4] ; int i ;
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    for( i=1 ; i<4 ; i++ ) {
        a[i] = tolower( a[i] ) ;
    }
    return 0 ;
}
```

---

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {  
⇒ char a[4] ; int i ;  
  a[0] = 'B' ;  
  a[1] = 'L' ;  
  a[2] = 'A' ;  
  a[3] = 'H' ;  
  for( i=1 ; i<4 ; i++ ) {  
    a[i] = tolower( a[i] ) ;  
  }  
  return 0 ;  
}
```

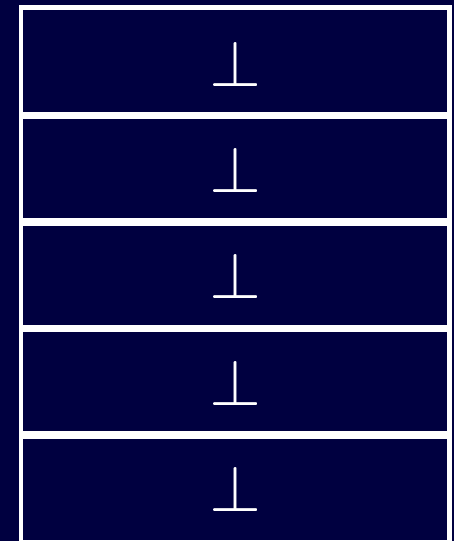
# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {  
    char a[4] ; int i ;  
⇒ a[0] = 'B' ;  
   a[1] = 'L' ;  
   a[2] = 'A' ;  
   a[3] = 'H' ;  
   for( i=1 ; i<4 ; i++ ) {  
       a[i] = tolower( a[i] ) ;  
   }  
   return 0 ;  
}
```

i:  
a:



# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {
    char a[4] ; int i ;
    a[0] = 'B' ;
    ⇒ a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    for( i=1 ; i<4 ; i++ ) {
        a[i] = tolower( a[i] ) ;
    }
    return 0 ;
}
```

i:		⊥
a:	a[0]	/ 'B'
	a[1]	⊥
	a[2]	⊥
	a[3]	⊥

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {
    char a[4] ; int i ;
    a[0] = 'B' ;
    a[1] = 'L' ;
    ⇒ a[2] = 'A' ;
    a[3] = 'H' ;
    for( i=1 ; i<4 ; i++ ) {
        a[i] = tolower( a[i] ) ;
    }
    return 0 ;
}
```

i:		⊥
a:	a[0]	/ 'B'
	a[1]	/ 'L'
	a[2]	⊥
	a[3]	⊥

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {  
    char a[4] ; int i ;  
    a[0] = 'B' ;  
    a[1] = 'L' ;  
    a[2] = 'A' ;  
⇒ a[3] = 'H' ;  
    for( i=1 ; i<4 ; i++ ) {  
        a[i] = tolower( a[i] ) ;  
    }  
    return 0 ;  
}
```

i:		⊥
a:	a[0]	⌊ 'B'
	a[1]	⌊ 'L'
	a[2]	⌊ 'A'
	a[3]	⊥

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {
    char a[4] ; int i ;
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    ⇒ for( i=1 ; i<4 ; i++ ) {
        a[i] = tolower( a[i] ) ;
    }
    return 0 ;
}
```

i:		⊥
a:	a[0]	<u>/</u> 'B'
	a[1]	<u>/</u> 'L'
	a[2]	<u>/</u> 'A'
	a[3]	<u>/</u> 'H'

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {
    char a[4] ; int i ;
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    for( i=1 ; i<4 ; i++ ) {
⇒     a[i] = tolower( a[i] ) ;
    }
    return 0 ;
}
```

i:	<u>/</u>	1
a: a[0]	<u>/</u>	'B'
a[1]	<u>/</u>	'L'
a[2]	<u>/</u>	'A'
a[3]	<u>/</u>	'H'

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {  
    char a[4] ; int i ;  
    a[0] = 'B' ;  
    a[1] = 'L' ;  
    a[2] = 'A' ;  
    a[3] = 'H' ;  
⇒ for( i=1 ; i<4 ; i++ ) {  
    a[i] = tolower( a[i] ) ;  
    }  
    return 0 ;  
}
```

i:	<u>  </u>	<b>1</b>
a: a[0]	<u>  </u>	<b>'B'</b>
a[1]	<u>  </u>	<b>'l'</b>
a[2]	<u>  </u>	<b>'A'</b>
a[3]	<u>  </u>	<b>'H'</b>

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {
    char a[4] ; int i ;
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    for( i=1 ; i<4 ; i++ ) {
⇒     a[i] = tolower( a[i] ) ;
    }
    return 0 ;
}
```

i:	<u>  </u> 2 <u>  </u>
a: a[0]	<u>  </u> 'B'
a[1]	<u>  </u> 'L'
a[2]	<u>  </u> 'A'
a[3]	<u>  </u> 'H'

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {  
    char a[4] ; int i ;  
    a[0] = 'B' ;  
    a[1] = 'L' ;  
    a[2] = 'A' ;  
    a[3] = 'H' ;  
⇒ for( i=1 ; i<4 ; i++ ) {  
        a[i] = tolower( a[i] ) ;  
    }  
    return 0 ;  
}
```

i:		<u>1</u>	<b>2</b>	<u>3</u>
a:	a[0]	<u>1</u>	<b>'B'</b>	
	a[1]	<u>1</u>	<b>'l'</b>	
	a[2]	<u>1</u>	<b>'a'</b>	
	a[3]	<u>1</u>	<b>'H'</b>	

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {  
    char a[4] ; int i ;  
    a[0] = 'B' ;  
    a[1] = 'L' ;  
    a[2] = 'A' ;  
    a[3] = 'H' ;  
    for( i=1 ; i<4 ; i++ ) {  
⇒     a[i] = tolower( a[i] ) ;  
    }  
    return 0 ;  
}
```

i:

a: a[0]

a[1]

a[2]

a[3]

	3	2	1
	✓	✓	✓
	✓	'B'	
	✓	'l'	
	✓	'a'	
	✓	'H'	

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {
    char a[4] ; int i ;
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    ⇒ for( i=1 ; i<4 ; i++ ) {
        a[i] = tolower( a[i] ) ;
    }
    return 0 ;
}
```

i:		3	/	2	/
a:	a[0]	/		'B'	
	a[1]	/		'l'	
	a[2]	/		'a'	
	a[3]	/		'h'	

# Arrays

An array is a data structure that can hold a number of values of the same type

- Any array element is selected using [ ].

```
int main( void ) {  
    char a[4] ; int i ;  
    a[0] = 'B' ;  
    a[1] = 'L' ;  
    a[2] = 'A' ;  
    a[3] = 'H' ;  
    for( i=1 ; i<4 ; i++ ) {  
        a[i] = tolower( a[i] ) ;  
    }  
⇒ return 0 ;  
}
```

i:	4	3	2	1
a: a[0]				'B'
a[1]				'l'
a[2]				'a'
a[3]				'h'

Notice: a[4] has cells from 0 to 3

---

# Array types

Arrays can hold any type:

```
int a[10] ;    /* Array of 10 integers, a[0]..a[9] */
double b[40] ; /* Array of 40 doubles, b[0]..b[39] */
char c[2] ;    /* Array of 2 characters, c[0], c[1] */
short d[5] ;   /* Array of 5 shorts, d[0]..d[4] */
```

What do they represent?

What is an array of characters?

- `'B''L''A''H'` looks like a word to me.

Array of characters is known as a string, "BLAH"

- Strings have a maximum length (eg `char s[20]`).
- An *End-marker*, `'\0'`, is used to denote the end of a string.

⇒ A string of  $i$  characters needs  $i+1$  at least cells

---

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}
int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a) ;
    return 0 ;
}
```

---

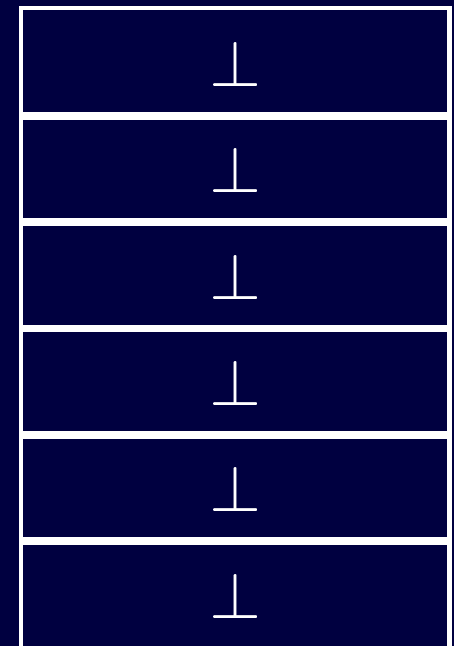
# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}
int main( void ) {
⇒ char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a) ;
    return 0 ;
}
```

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}
int main( void ) {
    char a[6] ; /* Max Length: 6 */
⇒ a[0] = 'B' ;
  a[1] = 'L' ;
  a[2] = 'A' ;
  a[3] = 'H' ;
  a[4] = '\0' ;
  printf( "%d: %s\n", strlen(a), a) ;
  return 0 ;
}
```

a:



# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}
int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    ⇒ a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a) ;
    return 0 ;
}
```

a: a[0]	<u>/</u> 'B'
a[1]	⊥
a[2]	⊥
a[3]	⊥
a[4]	⊥
a[5]	⊥

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}
int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    ⇒ a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a ) ;
    return 0 ;
}
```

a: a[0]	/ 'B'
a[1]	/ 'L'
a[2]	⊥
a[3]	⊥
a[4]	⊥
a[5]	⊥

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}

int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    ⇒ a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a) ;
    return 0 ;
}
```

a: a[0]	⌊ 'B'
a[1]	⌊ 'L'
a[2]	⌊ 'A'
a[3]	⊥
a[4]	⊥
a[5]	⊥

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}

int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    ⇒ a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a) ;
    return 0 ;
}
```

a: a[0]	<u>/</u> 'B'
a[1]	<u>/</u> 'L'
a[2]	<u>/</u> 'A'
a[3]	<u>/</u> 'H'
a[4]	⊥
a[5]	⊥

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}

int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
⇒ printf( "%d: %s\n", strlen(a), a) ;
    return 0 ;
}
```

a: a[0]	<u>/</u> 'B'
a[1]	<u>/</u> 'L'
a[2]	<u>/</u> 'A'
a[3]	<u>/</u> 'H'
a[4]	<u>/</u> '\0'
a[5]	⊥

# String: array of char

```
⇒int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}

int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a) ;
    return 0 ;
}
```

a: a[0]	<u>/</u> 'B'
a[1]	<u>/</u> 'L'
a[2]	<u>/</u> 'A'
a[3]	<u>/</u> 'H'
a[4]	<u>/</u> '\0'
a[5]	⊥

# String: array of char

```
int strlen( char s[] ) {  
⇒ int i = 0 ;  
  while( s[i] != '\0' ) {  
    i++ ;  
  }  
  return i ;  
}  
int main( void ) {  
  char a[6] ; /* Max Length: 6 */  
  a[0] = 'B' ;  
  a[1] = 'L' ;  
  a[2] = 'A' ;  
  a[3] = 'H' ;  
  a[4] = '\0' ;  
  printf( "%d: %s\n", strlen(a), a ) ;  
  return 0 ;  
}
```

a: a[0]	<u>/</u> 'B'
a[1]	<u>/</u> 'L'
a[2]	<u>/</u> 'A'
a[3]	<u>/</u> 'H'
a[4]	<u>/</u> '\0'
a[5]	⊥

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    ⇒ while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}

int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a ) ;
    return 0 ;
}
```

i:

a: a[0]

a[1]

a[2]

a[3]

a[4]

a[5]

	0
<u>/</u>	'B'
<u>/</u>	'L'
<u>/</u>	'A'
<u>/</u>	'H'
<u>/</u>	'\0'
<u> </u>	

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
⇒     i++ ;
    }
    return i ;
}
int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a) ;
    return 0 ;
}
```

i:

a: a[0]

a[1]

a[2]

a[3]

a[4]

a[5]

	0
	<u>/</u> 'B'
	<u>/</u> 'L'
	<u>/</u> 'A'
	<u>/</u> 'H'
	<u>/</u> '\0'
	⊥

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    ⇒ while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}

int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a ) ;
    return 0 ;
}
```

i:

a: a[0]

a[1]

a[2]

a[3]

a[4]

a[5]

	1
<u>/</u>	'B'
<u>/</u>	'L'
<u>/</u>	'A'
<u>/</u>	'H'
<u>/</u>	'\0'
<u> </u>	

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
⇒     i++ ;
    }
    return i ;
}
int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a) ;
    return 0 ;
}
```

i:

a: a[0]

a[1]

a[2]

a[3]

a[4]

a[5]

	1
<u>/</u> 'B'	
<u>/</u> 'L'	
<u>/</u> 'A'	
<u>/</u> 'H'	
<u>/</u> '\0'	
<u> </u>	

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    ⇒ while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}

int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a ) ;
    return 0 ;
}
```

i:

a: a[0]

a[1]

a[2]

a[3]

a[4]

a[5]

2

/ 'B'

/ 'L'

/ 'A'

/ 'H'

/ '\0'

⊥

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
⇒     i++ ;
    }
    return i ;
}
int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a) ;
    return 0 ;
}
```

i:	2
a: a[0]	<u>/</u> 'B'
a[1]	<u>/</u> 'L'
a[2]	<u>/</u> 'A'
a[3]	<u>/</u> 'H'
a[4]	<u>/</u> '\0'
a[5]	⊥

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    ⇒ while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}

int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a ) ;
    return 0 ;
}
```

i:

a: a[0]

a[1]

a[2]

a[3]

a[4]

a[5]

3

/ 'B'

/ 'L'

/ 'A'

/ 'H'

/ '\0'

⊥

# String: array of char

```
int strlen( char s[] ) {  
    int i = 0 ;  
    while( s[i] != '\0' ) {  
⇒     i++ ;  
    }  
    return i ;  
}  
int main( void ) {  
    char a[6] ; /* Max Length: 6 */  
    a[0] = 'B' ;  
    a[1] = 'L' ;  
    a[2] = 'A' ;  
    a[3] = 'H' ;  
    a[4] = '\0' ;  
    printf( "%d: %s\n", strlen(a), a) ;  
    return 0 ;  
}
```

i:

a: a[0]

a[1]

a[2]

a[3]

a[4]

a[5]

	3
<u>/</u> 'B'	
<u>/</u> 'L'	
<u>/</u> 'A'	
<u>/</u> 'H'	
<u>/</u> '\0'	
<u> </u>	

# String: array of char

```
int strlen( char s[] ) {  
    int i = 0 ;  
⇒ while( s[i] != '\0' ) {  
        i++ ;  
    }  
    return i ;  
}  
int main( void ) {  
    char a[6] ; /* Max Length: 6 */  
    a[0] = 'B' ;  
    a[1] = 'L' ;  
    a[2] = 'A' ;  
    a[3] = 'H' ;  
    a[4] = '\0' ;  
    printf( "%d: %s\n", strlen(a), a ) ;  
    return 0 ;  
}
```

i:	4
a: a[0]	<u>/</u> 'B'
a[1]	<u>/</u> 'L'
a[2]	<u>/</u> 'A'
a[3]	<u>/</u> 'H'
a[4]	<u>/</u> '\0'
a[5]	⊥

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
        i++ ;
    }
    ⇒ return i ;
}

int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a ) ;
    return 0 ;
}
```

i:

a: a[0]

a[1]

a[2]

a[3]

a[4]

a[5]

4

/ 'B'

/ 'L'

/ 'A'

/ 'H'

/ '\0'

⊥

# String: array of char

```
int strlen( char s[] ) {
    int i = 0 ;
    while( s[i] != '\0' ) {
        i++ ;
    }
    return i ;
}
int main( void ) {
    char a[6] ; /* Max Length: 6 */
    a[0] = 'B' ;
    a[1] = 'L' ;
    a[2] = 'A' ;
    a[3] = 'H' ;
    a[4] = '\0' ;
    printf( "%d: %s\n", strlen(a), a) ;
⇒ return 0 ;
}
```

a: a[0]	<u>/</u> 'B'
a[1]	<u>/</u> 'L'
a[2]	<u>/</u> 'A'
a[3]	<u>/</u> 'H'
a[4]	<u>/</u> '\0'
a[5]	⊥

---

# Strings library

<code>strncpy(t, s, n)</code>	copies <code>s</code> to <code>t</code> , copies at most <code>n</code> chars
<code>strncat(t, s, n)</code>	concatenates <code>s</code> to <code>t</code> , copies at most <code>n</code> chars
<code>strcmp(t, s)</code>	compare <code>t</code> and <code>s</code> , returns $< 0$ , (t before s) 0, (t equals s) or $> 0$ (t after s)
<code>strncmp(t, s, n)</code>	like <code>strcmp</code> , but stops if they are still equal after <code>n</code> characters.
<code>strlen(s)</code>	returns length of <code>s</code> (excluding <code>'\0'</code> ).
<code>strcpy, strcat</code>	no bound checks. Don't use these, unless if you can proof that bounds will not be violated.

---

# String program

```
int main( void ) {
    char s[20] ;
    strncpy( s, "Yes ", 20 ) ;
    strncat( s, "No!", 20 - strlen( s ) ) ;
    printf( "%s: %d, %d, %d\n", s,
            strcmp( s, "yes no!" ),
            strcmp( s, "Aargh..." ),
            strncmp( s, "Yes No", 6 ) ) ;
    for( i=0 ; s[i] != '\0' ; i++ ) {
        printf( "'%c'", s[i] ) ;
    }
    return 0 ;
}
```

- Prints

```
"Yes No!: -1, 1, 0; 'Y''e''s'' ''N''o''!' " ) ;
```

---

# Multi dimensional arrays

- `double x[184]` is a one-dimensional array. A vector (a single column).
- `double y[184][155]` is a two-dimensional array. A matrix (row of columns).
- See it as an array of arrays, `double (y[184])[155]...`
- Can be used to store, for example, an image.



- Each pixel has a position  $(0,0) .. (183,154)$ , stored in `y[0][0] .. y[183][154]`, store a numeric (grey) value.

---

# Using arrays

Some common uses of arrays:

- Two dimensional arrays `double image[184][155] ;`
- Three dimensional arrays `int v [1024][256][256] ;`
- One dimensional arrays `int a [4410000] ;`
- Two dimensional arrays, second dimension is 8?  
`int s [4410000][8] ;`
- Multimedia unit (second year CS) will deal with all of those...

Other uses

- Matrices, Vectors.
- Lists of numbers, cars, ...
- Chessboard (`c[8][8]`)

---

# Using arrays

Some common uses of arrays:

- Two dimensional arrays `double image[184][155] ;`
- Three dimensional arrays `int video[1024][256][256] ;`
- One dimensional arrays `int a [4410000] ;`
- Two dimensional arrays, second dimension is 8?  
`int s [4410000][8] ;`
- Multimedia unit (second year CS) will deal with all of those...

Other uses

- Matrices, Vectors.
- Lists of numbers, cars, ...
- Chessboard (`c[8][8]`)

---

# Using arrays

Some common uses of arrays:

- Two dimensional arrays `double image[184][155] ;`
- Three dimensional arrays `int video[1024][256][256] ;`
- One dimensional arrays `int audio[4410000] ;`
- Two dimensional arrays, second dimension is 8?  
`int s [4410000][8] ;`
- Multimedia unit (second year CS) will deal with all of those...

Other uses

- Matrices, Vectors.
- Lists of numbers, cars, ...
- Chessboard (`c[8][8]`)

---

# Using arrays

Some common uses of arrays:

- Two dimensional arrays `double image[184][155] ;`
- Three dimensional arrays `int video[1024][256][256] ;`
- One dimensional arrays `int audio[4410000] ;`
- Two dimensional arrays, second dimension is 8?  
`int surround[4410000][8] ;`
- Multimedia unit (second year CS) will deal with all of those...

Other uses

- Matrices, Vectors.
- Lists of numbers, cars, ...
- Chessboard (`c[8][8]`)

---

# Parameter passing, by value

Passing parameters in C normally *copies* a value

⇒ If a function assigns another value to the parameter, then this has no consequences for the caller:

```
void assign( int x ) {
    x = 3 ;
}

int main( void ) {
    int y = 1 ;
    assign( y ) ;
    printf( "y is %d\n", y ) ;
    return 0 ;
}
```

---

# Parameter passing, by value

Passing parameters in C normally *copies* a value

⇒ If a function assigns another value to the parameter, then this has no consequences for the caller:

```
void assign( int x ) {  
    x = 3 ;  
}
```

```
int main( void ) {  
    int y = 1 ;  
    assign( y ) ;  
    printf( "y is %d\n", y ) ; 1  
    return 0 ;  
}
```

---

# Parameter passing, pointers

So what if we *want* to change a value.

Suppose we want a function that returns the time in hours minutes and seconds:

```
gettime( int h, int m, int s ) {  
    ... /* set h, m and s */  
}  
  
int main( void ) {  
    int hours, minutes, seconds ;  
    gettime( hours, minutes, seconds ) ;  
    return 0 ;  
}
```

Will not work:

- `gettime` can only modify the `h`, `m` and `s`.

---

# Parameter passing, pointers

So what if we *want* to change a value.

Suppose we want a function that returns the time in hours minutes and seconds:

```
gettime( int *h, int *m, int *s ) {  
    ... /* set *h, *m and *s */  
}  
  
int main( void ) {  
    int hours, minutes, seconds ;  
    gettime( &hours, &minutes, &seconds ) ;  
    return 0 ;  
}
```

Solution, pass a pointer to the variables

– A pointer is the address of a variable...

---


# Remember the computer's organisation

- Memory is a big bucket full of bytes.
- Each byte has an address.
- Each variable is mapped onto a particular (collection of) byte(s)
- Each variable has an address.
- By telling a function the address of a variable, the function can modify its contents.
- Yuck.

# Parameter passing, pointers...

```
gettime( int *h, int *m, int *s ) {  
    *h = 13 ;  
    *m = 30 ;  
    *s = 59 ;  
}
```


```
⇒ int main( void ) {  
    int hours, minutes, seconds ;  
    gettime( &hours, &minutes, &seconds ) ;  
    printf( "%2d:%2d:%2d\n", hours, minutes, seconds ) ;  
    return 0 ;  
}
```



# Parameter passing, pointers...

```
gettime( int *h, int *m, int *s ) {  
    *h = 13 ;  
    *m = 30 ;  
    *s = 59 ;  
}
```

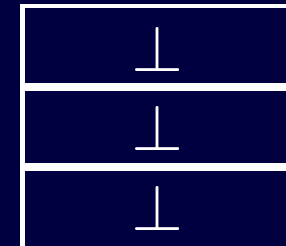
```
⇒ int main( void ) {  
    int hours, minutes, seconds ;  
    gettime( &hours, &minutes, &seconds ) ;  
    printf( "%2d:%2d:%2d\n", hours, minutes, seconds ) ;  
    return 0 ;  
}
```



# Parameter passing, pointers...

```
gettime( int *h, int *m, int *s ) {  
    *h = 13 ;  
    *m = 30 ;  
    *s = 59 ;  
}
```

seconds:  
minutes:  
hours:

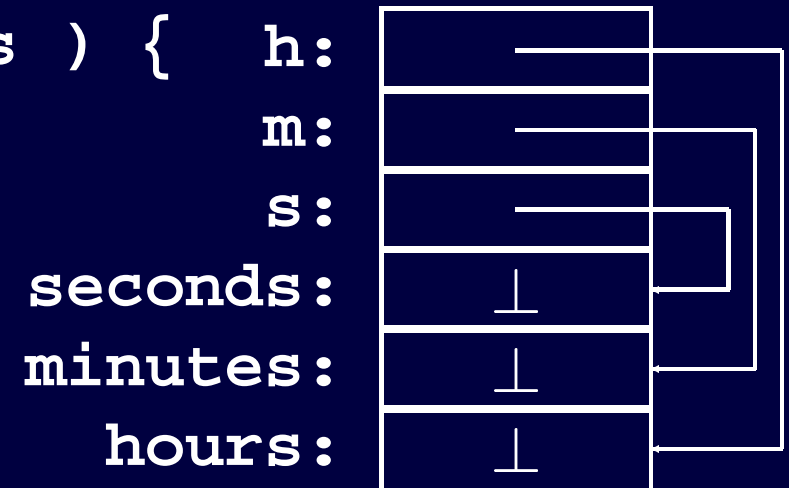


```
⇒ int main( void ) {  
    int hours, minutes, seconds ;  
    gettime( &hours, &minutes, &seconds ) ;  
    printf( "%2d:%2d:%2d\n", hours, minutes, seconds ) ;  
    return 0 ;  
}
```



# Parameter passing, pointers...

```
⇒ gettime( int *h, int *m, int *s ) {  
    *h = 13 ;  
    *m = 30 ;  
    *s = 59 ;  
}
```

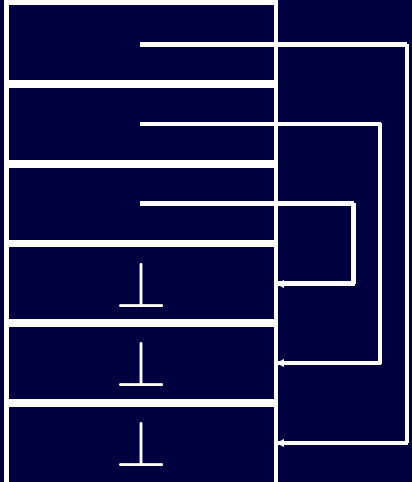


```
int main( void ) {  
    int hours, minutes, seconds ;  
    gettime( &hours, &minutes, &seconds ) ;  
    printf( "%2d:%2d:%2d\n", hours, minutes, seconds ) ;  
    return 0 ;  
}
```




# Parameter passing, pointers...

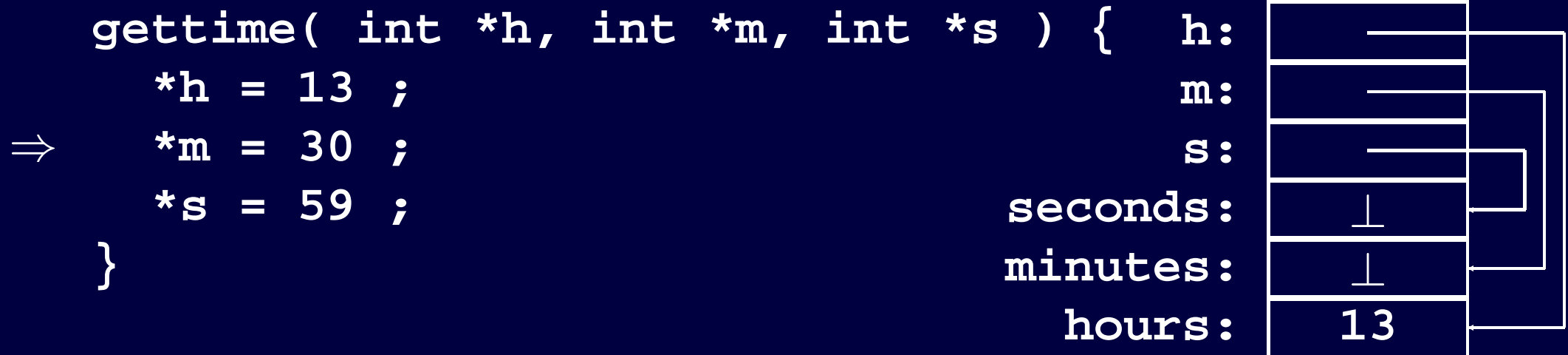
```
⇒ gettime( int *h, int *m, int *s ) {
    *h = 13 ;
    *m = 30 ;
    *s = 59 ;
}
```

h: 

```
int main( void ) {
    int hours, minutes, seconds ;
    gettime( &hours, &minutes, &seconds ) ;
    printf( "%2d:%2d:%2d\n", hours, minutes, seconds ) ;
    return 0 ;
}
```

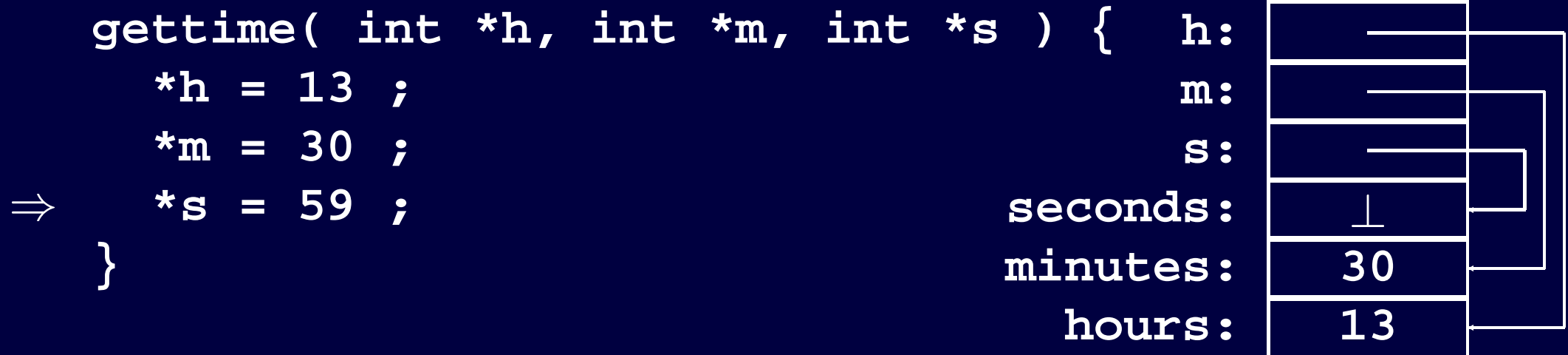


# Parameter passing, pointers...



```
int main( void ) {  
    int hours, minutes, seconds ;  
    gettime( &hours, &minutes, &seconds ) ;  
    printf( "%2d:%2d:%2d\n", hours, minutes, seconds ) ;  
    return 0 ;  
}
```

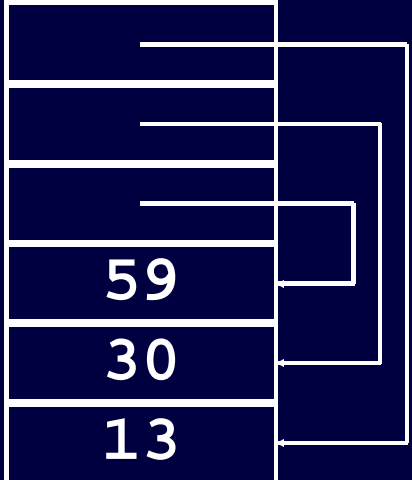
# Parameter passing, pointers...




```
int main( void ) {  
    int hours, minutes, seconds ;  
    gettime( &hours, &minutes, &seconds ) ;  
    printf( "%2d:%2d:%2d\n", hours, minutes, seconds ) ;  
    return 0 ;  
}
```

# Parameter passing, pointers...

```
gettime( int *h, int *m, int *s ) {  
    *h = 13 ;  
    *m = 30 ;  
    *s = 59 ;  
⇒ }  
seconds: 59  
minutes: 30  
hours: 13
```



```
int main( void ) {  
    int hours, minutes, seconds ;  
    gettime( &hours, &minutes, &seconds ) ;  
    printf( "%2d:%2d:%2d\n", hours, minutes, seconds ) ;  
    return 0 ;  
}
```



# Parameter passing, pointers...

```
gettime( int *h, int *m, int *s ) {  
    *h = 13 ;  
    *m = 30 ;  
    *s = 59 ;  
}
```

seconds:

59

minutes:

30

hours:

13

```
int main( void ) {  
    int hours, minutes, seconds ;  
    gettime( &hours, &minutes, &seconds ) ;  
⇒ printf( "%2d:%2d:%2d\n", hours, minutes, seconds ) ;  
    return 0 ;  
}
```



# Parameter passing, pointers...

```
gettime( int *h, int *m, int *s ) {  
    *h = 13 ;  
    *m = 30 ;  
    *s = 59 ;  
}
```

seconds:

59

minutes:

30

hours:

13

```
int main( void ) {  
    int hours, minutes, seconds ;  
    gettime( &hours, &minutes, &seconds ) ;  
    printf( "%2d:%2d:%2d\n", hours, minutes, seconds ) ;  
⇒ return 0 ;  
}
```

13:30:59

---

# Arrays and pointers...

An array is nothing else than a pointer to a block of memory.

- The `[n]` refers to the  $n$ -th element of the block
- When passed: only the pointer is passed (not like an int)
- When returned: only the pointer is returned (not like an int)
- You cannot assign arrays
- Array bounds are not checked...
- Array is a second class citizen  
(• ints, structs etc are first class citizen).

Other imperative languages (Modula, Pascal) have first class arrays.

# Strings are pointers...

Consequence: strings are second class citizens:

- Strings are passed by reference
- Need to allocate space for strings yourself
- That is why we need a library of functions...

```
char *makehello( void ) {
    char s[16] ;
    s = "Hello"           /*WRONG*/ ;
    s = s ++ " World"    /*WRONG*/ ;
    printf( "\"%s\"\n", s ) ;
    return s             /*WRONG*/ ;
}
```

Assignment not allowed, must use `strncpy`

---

# Strings are pointers...

Consequence: strings are second class citizens:

- Strings are passed by reference
- Need to allocate space for strings yourself
- That is why we need a library of functions...

```
char *makehello( void ) {  
    char s[16] ;  
    strncpy( s, "Hello", 16 ) ;  
    s = s ++ " World"      /*WRONG*/ ;  
    printf( "\"%s\"\n", s ) ;  
    return s              /*WRONG*/ ;  
}
```

Concatenation ? Use `strncat`

---

# Strings are pointers...

Consequence: strings are second class citizens:

- Strings are passed by reference
- Need to allocate space for strings yourself
- That is why we need a library of functions...

```
char *makehello( void ) {  
    char s[16] ;  
    strncpy( s, "Hello", 16 ) ;  
    strncat( s, " World", 11 ) ;  
    printf( "\"%s\"\n", s ) ;  
    return s          /*WRONG*/ ;  
}
```

Return would only return pointer, must use `strdup`

---

# Strings are pointers...

Consequence: strings are second class citizens:

- Strings are passed by reference
- Need to allocate space for strings yourself
- That is why we need a library of functions...

```
char *makehello( void ) {
    char s[16] ;
    strncpy( s, "Hello", 16 ) ;
    strncat( s, " World", 11 ) ;
    printf( "\"%s\"\n", s ) ;
    return strdup( s ) ;
}
```

Must use `strncpy`, `strncat`, and `strdup`.

---

# Concluding arrays and pointers

An array is something holding a row of values.

A string is a row of characters

⇒ An array of characters.

A pointer is something holding the address of a variable.

- An array is a pointer to a number of variables.

Pointers can be used to update values:

- Parameters passed by value
- Programmer can pass pointers, allows updates

Next lecture: Input and Output

- Read chapter 8 (background), “Reading and Displaying information” in chapter 30 (details)

---

# Input / Output

So far:

- All programs were generating something (a number, a string, something)
- They did not read any input.
- Instead, the input was encoded in the program.
- Input is what makes a program useful.

This lecture:

- How to read some input, for example, from the keyboard, general:
- Input (Keyboard, File)  $\Rightarrow$  Program  $\Rightarrow$  Output (Screen, File)

---

# Input / Output II

Do not confuse Input / Output with function parameters

- A function has parameters and produces a result
- These are “input” to and “output” from a function

I/O reads from / writes to something *outside* the program

- A program must perform I/O
- A function might perform I/O
- I/O is a global activity.

---

# Output

Types of output:

- Output that interacts with a user
  - May be in the form of text
  - May be graphical, audio, ...
- Output to a file.
  - May be text
  - May be in a binary format (write bit patterns to a file)

Languages traditionally only support non graphical I/O.

- Portable. Every computer has text
- Graphics are a library (eg, Quickdraw, OpenGL)

Java is an exception: graphics is part of the language

---

# Input

## Types of input

- Input from a device connected to a user
  - May be textual, a keyboard
  - May be graphical, a mouse, or audio input...
- Input from a file
  - Text or binary.
  - Used to read permanent data in your program, data that lives forever, even though your program will be restarted now and then.

Again, built-in support is traditionally text based.

---

# Output in C

We have seen output:

```
printf( "%d students, %d staff\n", stud, staff ) ;
```

printf prints its first argument, which must be a string. %...are substituted:

- %d: next parameter as an integer
- %f: next parameter as a floating point number
- %s: next parameter as a string
- %c: next parameter as a character
- %%: prints a single %
- %6d: next parameter as an integer using 6 places (right aligned)
- %7.2f: next parameter as a floating point number, 7 places with 2 behind the decimal point

There are 300 other possibilities. RTFM.

---

# Input in C: characters

`getchar()` returns the next character that is typed.

```
int main( void ) {
    char c ;
    while( 1 ) {
        c = getchar() ;
        if( c == '\n' ) { break ; }
        if( c >= 'a' && c <= 'z' ) {
            putchar( c - 'a' + 'A' ) ;
        } else {
            putchar( c ) ;
        }
    }
    return 0 ;
}
```

---

# Input in C: characters

`getchar()` returns the next character that is typed.

```
#include <ctype.h>          /* standard header file */
int main( void ) {
    char c ;
    while( 1 ) {
        c = getchar() ;
        if( c == '\n' ) { break ; }
        if( islower(c) ) {
            putchar( toupper(c) ) ;
        } else {
            putchar( c ) ;
        }
    }
    return 0 ;
}
```

---

# Return value of `getchar`

`getchar` actually returns an integer:

- one of the character codes if there is a next character, or
- the number 'EOF' (End of File) if there are no more characters.

EOF is not a character.

You can convert the value of `getchar()` to a character once you know it is not EOF

```
int i ;
while( (i=getchar()) != EOF ) {
    char c = i ;
    ... ;
}
```

People do not bother with the conversion (is implicit anyway; a `char` is a small `int`).

---

# Return value of `getchar`

`getchar` actually returns an integer:

- one of the character codes if there is a next character, or
- the number 'EOF' (End of File) if there are no more characters.

EOF is not a character.

You can convert the value of `getchar()` to a character once you know it is not EOF

```
char c ;
while( (c=getchar()) != EOF ) {
    ... ;
}
```

People do not bother with the conversion (is implicit anyway; a `char` is a small `int`).

---

# Reading integers and reals

And now for input:

```
int stud ;
int staff ;
double money ;
scanf( "%d %d %lf\n", &stud, &staff, &money ) ;
```

`scanf` is the input companion of `printf`, it uses its first argument to decide what to input:

- `%d`: read an integer and store it via an integer **pointer**
- `%lf`: read a floating point number and store it via a double **pointer**

**Do not forget the &-s**

---

# Reading strings

Lazy programmers will write:

```
char astring[20] ;
```

```
scanf( "%s", astring ) ; /*(where is the & ?)*/
```

(%s reads anything up to a space/newline)

What's wrong?

- If the user types a string of more than 20 characters?
- Excess characters will be stored in characters `s[20]`, `s[21]`,...
- Will overwrite valuable parts of the memory
- (• This was the leak exploited in the 'Internet Worm')

You should only read a string with %s if you can proof that it will fit.

---

# Other ways to read a string

Limit scanf to a maximum field:

`scanf( "%20s", s ) ;` reads no more than 21 characters ('`\0`!).

Read the string character by character

- Choose to
  - Either dispose of excess characters, or
  - Allocate extra space for the string.

You can read the string

- into an array passed to the function (watch out with the maximum length)
- into a local array, and return a `strduped` version

---

# File I/O in C

File I/O consists of three phases:

- You have to *open* a file
- You may then perform a number of I/O operations
- Finally you have to close a file.

```
int main( void ) {  
    FILE *fd ;  
  
    fd = fopen( "Somefile", "w" ) ;  
    fprintf( fd, "Bla %s %d %d\n", "World", 1, 13 ) ;  
    fclose( fd ) ;  
    return 0 ;  
}
```

---

# File I/O in C

File I/O consists of three phases:

- You have to *open* a file
- You may then perform a number of I/O operations
- Finally you have to close a file.

```
int main( ) {  
    FILE *fd ;  
    int i, j ;  
    fd = fopen( "Somefile", "r" ) ;  
    fscanf( fd, "%d %d\n", &i, &j ) ;  
    fclose( fd ) ;  
    return 0 ;  
}
```

---

# Peculiarities of IO in C

## Output:

- Output in C is buffered (i.e., characters are not printed until a newline)

## Input:

- Input in C is often line buffered (i.e., getchar does not read until you hit return, then it will read the whole line)
- 💣 Do not forget the `&` in a scanf
- 💣 Do not forget to use `"%lf"` when scanning a variable of type `double`
- 💣 Do not use `%s` unless if you can proof correctness.

---

# Complications

```
int main( void ) {  
  
    if( getchar() > getchar() ) {  
        printf("Yes!\n" ) ;  
    }  
    return 0 ;  
}
```

- I run the program with input 06. Will it print 'Yes'?

---

# Complications

```
int main( void ) {
    char c, d ;
    c = getchar() ; d = getchar() ;
    if( c > d ) {
        printf("Yes!\n" ) ;
    }
    return 0 ;
}
```

- I run the program with input 06. Will it print 'Yes'?

---

# Complications

```
int main( void ) {
    char c, d ;
    c = getchar() ; d = getchar() ;
    if( c > d ) {
        printf("Yes!\n" ) ;
    }
    return 0 ;
}
```

- I run the program with input 06. Will it print 'Yes'?

⇒ C I/O is not defined precisely.

- In other languages (like Haskell) you will be obliged to specify the order:
- This will make it much more complicated, but is unambiguous.

---

# Next lectures:

- Next slot is a problem class on Arrays, Sets, memory models.
- Next week, lectures on Datastructures  
(– last week on programming basics)
  - Structures Chapter 12.1-12.7
  - Dynamic memory, 8.5-8.8, 9.4
  - Lists 12.8-12.12
- Week after that break of hardcore programming:
  - Grammars
  - Ethics
  - Debugging programs.