Arrays & Pointers
Characters & strings

Jinyang Li
What we’ve learnt and today’s plan

• Bitwise operations
• Pointers
  – Pointers are addresses
  – With pointers arguments, a callee can modify local variables in the caller.
• Today’s lesson:
  – Array and its relationship with pointer
  – Pointer casting
  – Characters & strings
Array: a collection of contiguous objects with the same type
Array

```c
int a[3];
```
Array

int a[3] = {1, 2, 3};
Array

int a[3] = {1, 2, 3};
int a[3] = {1, 2, 3};
int *p;
p = &a[0]; // equivalent to p = a;
printf("%p\n", p); // output? 0x100
printf("%d\n", *p); // output? 1
int a[3] = {1, 2, 3};

int *p;
p = &a[0];
p = p + 1; //equivalent to p++

printf("%p\n", p); //output? 0x104

printf("%d\n", *p); //output? 2

Rule of pointer arithmetic:
p+i has address of i-th object after p, i.e. p+i’s value is p’s value plus i*sizeof(object)
int a[3] = {1, 2, 3};

int *p;
p = &a[0];

printf("%p\n", p+2);  // output? 0x108
printf("%d\n", *(p+2));  // output? 3

Rule of pointer arithmetic:

p+i has address of i-th object after p, i.e.
p+i’s value is p’s value plus i*sizeof(object)
**Pointer arithmetic**

```
char a[3] = {1, 2, 3};
char *p;
p = &a[0];
printf("%p\n", p+2); // output? 0x102
printf("%d\n", *(p+2)); // output? 3
```

Rule of pointer arithmetic:
p+i has address of i-th object after p, i.e.
p+i’s value is p’s value plus i*sizeof(object)
Array and pointer

```c
int a[10];
int *p;

p = &a[0]; // a is alias for &a[0];

for (int i = 0; i < 10; i++) {
    *(p+i) = 0; // p[i] is alias for *(p+i)
}
```
Array and pointer

```c
int a[10];
int *p;

p = a;     // a is alias for &a[0];

for (int i = 0; i < 10; i++) {
    p[i] = 0;   // p[i] is alias for *(p+i)
}
```
int a[3] = {1, 2, 3};

int *p;
p = &a[0];

printf("%p\n", p+3); //output? 0x10c
printf("%d\n", *(p+3)); //output?

??
can be any value
or segmentation fault

printf("%p\n", &a[3]); //output? 0x10c
printf("%d\n", a[3]); //output?
??
can be any value
or segmentation fault
Example

```c
#include <stdio.h>

int main() {
    int a[3] = {100, 200, 300};
    int *p = a;
    *p = 400;
    for (int i=0; i<3; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

Output? 400 200 300
```
Another Example

#include <stdio.h>

int main() {
    int a[3] = {100, 200, 300};
    int *p = a;
    p++;
    *p = 400;
    for (int i=0; i<3; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

Output?  100 400 300
Pass array to function via pointer

// multiply every array element by 2
void multiply2(int *a) {
    for (int i = 0; i < ???; i++) {
        a[i] *= 2;
    }
}

int main() {
    int a[2] = {1, 2};
    multiply2(a);
    for (int i = 0; i < 2; i++) {
        printf("a[%d]=%d", i, a[i]);
    }
}
Pass array to function via pointer

// multiply every array element by 2
void multiply2(int *a, int n) {
    for (int i = 0; i < n; i++) {
        a[i] *= 2; // (*a+i) *= 2;
    }
}

int main() {
    int a[2] = {1, 2};
    multiply2(a, 2);
    for (int i = 0; i < 2; i++) {
        printf("a[%d]=%d\n", i, a[i]);
    }
}
Pointer casting

```c
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
printf("%x\n", *c);
```

Output? (when running on Intel laptop)
Pointer casting

```c
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
```

Intel laptop is small endian
*c is 0x78

What is c+1? p+1?
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
Pointer casting

```c
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
```

assert(p+i == (char *)p + i*sizeof(*p))

sizeof(*p), or sizeof(int) is a C built-in that returns size of object/expression
**Pointer casting**

```c
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
```

```c
for (int i = 0; i < 4; i++) {
    printf("%x ", c[i]);
}
```

Output: 0x78 0x56 0x34 0x12

What about big endian?
Another example of pointer casting

bool is_normalized_float(float f)
{
}

Another example of pointer casting

```c
bool is_normalized_float(float f) {
    unsigned int i;
    i = *(unsigned int *)&f;

    unsigned exp = (i&0x7fffffff)>>23;
    return (exp != 0 && exp != 255);
}
```
function `sizeof`

`sizeof(type)`
- Returns size in bytes of the object representation of type

`sizeof(expression)`
- Returns size in bytes of the type that would be returned by expression, if evaluated.
**function `sizeof`**

<table>
<thead>
<tr>
<th><code>sizeof()</code></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sizeof(int)</code></td>
<td></td>
</tr>
<tr>
<td><code>sizeof(long)</code></td>
<td></td>
</tr>
<tr>
<td><code>sizeof(float)</code></td>
<td></td>
</tr>
<tr>
<td><code>sizeof(double)</code></td>
<td></td>
</tr>
<tr>
<td><code>sizeof(int *)</code></td>
<td></td>
</tr>
</tbody>
</table>

64 bits machine
function `sizeof`

<table>
<thead>
<tr>
<th><code>sizeof()</code></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sizeof(int)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>sizeof(long)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>sizeof(float)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>sizeof(double)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>sizeof(int *)</code></td>
<td>8</td>
</tr>
</tbody>
</table>

64 bits machine
## function `sizeof`

<table>
<thead>
<tr>
<th>expr</th>
<th><code>sizeof()</code></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int a = 0;</code></td>
<td><code>sizeof(a)</code></td>
<td></td>
</tr>
<tr>
<td><code>long b = 0;</code></td>
<td><code>sizeof(b)</code></td>
<td></td>
</tr>
<tr>
<td><code>int a = 0; long b = 0;</code></td>
<td><code>sizeof(a + b)</code></td>
<td></td>
</tr>
<tr>
<td><code>char c[10];</code></td>
<td><code>sizeof(c)</code></td>
<td></td>
</tr>
<tr>
<td><code>int arr[10];</code></td>
<td><code>sizeof(arr)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>sizeof(arr[0])</code></td>
<td></td>
</tr>
<tr>
<td><code>int *p = arr;</code></td>
<td><code>sizeof(p)</code></td>
<td></td>
</tr>
</tbody>
</table>

64 bits machine
function `sizeof`

<table>
<thead>
<tr>
<th>expr</th>
<th><code>sizeof()</code></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int a = 0;</code></td>
<td><code>sizeof(a)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>long b = 0;</code></td>
<td><code>sizeof(b)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>int a = 0; long b = 0;</code></td>
<td><code>sizeof(a + b)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>char c[10];</code></td>
<td><code>sizeof(c)</code></td>
<td>10</td>
</tr>
<tr>
<td><code>int arr[10];</code></td>
<td><code>sizeof(arr)</code></td>
<td><code>10 * 4 = 40</code></td>
</tr>
<tr>
<td></td>
<td><code>sizeof(arr[0])</code></td>
<td>4</td>
</tr>
<tr>
<td><code>int *p = arr;</code></td>
<td><code>sizeof(p)</code></td>
<td>8</td>
</tr>
</tbody>
</table>

64 bits machine
Undefined behavior

In computer programming, undefined behavior (UB) is the result of executing computer code whose behavior is not prescribed by the language specification.
Classic undefined behaviors

- Use an uninitialized variable
  
  ```c
  int a;
  int b = a + 1;
  ```

- out of bound array access
  
  ```c
  int a[2] = {1, 2};
  int *p = a
  *(p+3) = 3;
  ```

- Divide by zero
  
  ```c
  int a = 1 / 0;
  ```

- integer overflow
  
  ```c
  int a = 0x7fffffff
  int b = a + 1
  ```
Why does C have undefined behavior?

Simplify compiler’s implementation

Enable better performance
Classic undefined behaviors

• Use uninitialized variables
  – Avoid memory write

• Out-of-bound array access
  – Avoid runtime bound checking

• Divided by zero

• integer overflow
Classic undefined behaviors

At instruction set level, different architectures handle them in different ways:

Divided by zero
- X86 raises an exception
- MIPS and PowerPC silently ignore it.

Integer overflow
- X86 wraps around (with flags set)
- MIPS raises an exception.
Classic undefined behaviors

Assumption: Unlike Java, C compilers trust the programmer not to submit code that has undefined behavior.

The compiler optimizes this code under this assumption.

→ Compiler may remove the code or rewrite the code in a way that programmer did not anticipate.
Classic undefined behaviors

```c
#include <stdio.h>

void foo(int a) {
    if(a+100 < a) {
        printf("overflowed\n");
        return;
    }
    printf("normal is boring\n");
}

int main() {
    foo(100);
    foo(0x7fffffff);
}
```
#include <stdio.h>

void foo(int a) {
    if(a+100 <= a) {
        printf("overflowed\n");
        return;
    }
    printf("normal is boring\n");
}

int main() {
    foo(100);
    foo(0x7fffffff);
}
Characters
How to represent text characters?

• How to associate bit patterns to integers?
  – base 2
  – 2’s complement

• How to associate bit patterns to floats?
  – IEEE floating point representation (based on normalized scientific notation)

• How to associate bit patterns to characters?
  – by convention
  – ASCII, UTF
ASCII: American Standard Code for Information Exchange

- Developed in 60s, based on the English alphabet
- use one byte (with MSB=0) to represent each character
- How many unique characters can be represented?

128
# ASCII TABLE

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[NULL]</td>
<td>32</td>
<td>20</td>
<td>[SPACE]</td>
<td>64</td>
<td>40</td>
<td>@</td>
<td>96</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>[START OF HEADING]</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>[START OF TEXT]</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>[END OF TEXT]</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>[END OF TRANSMISSION]</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>[ENQUIRY]</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>[ACKNOWLEDGE]</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>[BEL]</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>[BACKSPACE]</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>[HORIZONTAL TAB]</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>[VERTICAL TAB]</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>[FORM FEED]</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>[CARRIAGE RETURN]</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>[SHIFT OUT]</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>[SHIFT IN]</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>G</td>
<td>[DATA LINK ESCAPE]</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>H</td>
<td>[DEVICE CONTROL 1]</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>I</td>
<td>[DEVICE CONTROL 2]</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>J</td>
<td>[DEVICE CONTROL 3]</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>K</td>
<td>[DEVICE CONTROL 4]</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>L</td>
<td>[NEGATIVE ACKNOWLEDGE]</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>M</td>
<td>[SYNCHRONOUS IDLE]</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>N</td>
<td>[ENG OF TRANS. BLOCK]</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>O</td>
<td>[CANCEL]</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>P</td>
<td>[END OF MEDIUM]</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>Q</td>
<td>[SUBSTITUTE]</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>R</td>
<td>[ESCAPE]</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>[</td>
<td>123</td>
<td>7B</td>
<td>{</td>
</tr>
<tr>
<td>28</td>
<td>S</td>
<td>[FILE SEPARATOR]</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
<td>124</td>
<td>7C</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>T</td>
<td>[GROUP SEPARATOR]</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
<td>125</td>
<td>7D</td>
<td>}</td>
</tr>
<tr>
<td>30</td>
<td>U</td>
<td>[RECORD SEPARATOR]</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>V</td>
<td>[UNIT SEPARATOR]</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>_</td>
<td>127</td>
<td>7F</td>
<td>[DEL]</td>
</tr>
</tbody>
</table>
C exercise 1: tolower

// tolower returns the corresponding lowercase character for c if c is an uppercase letter. Otherwise, it returns c.
char tolower(char c) {

}

int main() {
    char c = ‘A’;
    c = tolower(c);
    ...
}

C exercise 1: tolower

// tolower returns the corresponding
// lowercase character for c if c is an
// uppercase letter. Otherwise, it returns c.
char tolower(char c) {

    // test if c is an uppercase letter
    if (c < 'A' || c > 'Z') {
        return c;
    }

    }
}
C exercise 1: tolower

// tolower returns the corresponding
// lowercase character for c if c is an
// uppercase letter. Otherwise, it returns c.
char tolower(char c) {

    // test if c is an uppercase letter
    if (c < 'A' || c > 'Z') {
        return c;
    }

    return c + ('a' - 'A');
}

C’s standard library includes
tolower, toupper
C exercise 2: toDigit

// toDigit returns the corresponding integer for c
// if c is a valid digit character, e.g. '1', '2',
// Otherwise, it returns -1.
int toDigit(char c) {
}

int main() {
    int d = toDigit('8');
    printf("int is %d, multiply-by-2 %d\n", d, 2*d);
}
C exercise 2: toDigit

// toDigit returns the corresponding integer for c
// if c is a valid digit character, e.g ‘1’, ‘2’,
// Otherwise, it returns -1.
int toDigit(char c) {
    // test if c is a valid character
    if (c < '0' || c > '9') {
        return -1;
    }
}

int main() {
    int d = toDigit('8');
    printf("int is %d, multiply-by-2 %d\n", d, 2*d);
}

C exercise 2: toDigit

// toDigit returns the corresponding integer for c
// if c is a valid digit character, e.g. '1', '2',
// otherwise, it returns -1.
int toDigit(char c) {
    // test if c is a valid character
    if (c < '0' || c > '9') {
        return -1;
    }
    return c - '0';
}
int main() {
    int d = toDigit('8');
    printf("int is %d, multiply-by-2 %d\n", d, 2*d);
}
The Modern Standard: UniCode

• ASCII can only represent 128 characters
  – How about Chinese, Korean, all of the worlds languages? Symbols? Emojis?

• Unicode standard represents >135,000 characters

<table>
<thead>
<tr>
<th>Code</th>
<th>Emoticon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+1F600</td>
<td>😊</td>
<td>grinning face</td>
</tr>
<tr>
<td>U+1F601</td>
<td>😊</td>
<td>beaming face with smiling eyes</td>
</tr>
<tr>
<td>U+1F602</td>
<td>😂</td>
<td>face with tears of joy</td>
</tr>
<tr>
<td>U+1F923</td>
<td>😂</td>
<td>rolling on the floor laughing</td>
</tr>
<tr>
<td>U+1F603</td>
<td>😊</td>
<td>grinning face with big eyes</td>
</tr>
</tbody>
</table>
UTF-8

• UTF-8 is one encoding form for Unicode
  – use 1, 2, or 4 byte to represent a character
  – Unicode for ASCII characters have the same ASCII value ➔ UTF-8 one byte code is the same as ASCII
• C has no primitive support for Unicode
What we’ve learnt and today’s plan

• Bitwise operations
• Pointers & arrays
• ASCII characters
• Today’s lesson:
  – String
  – Struct
  – Malloc
C Strings
Strings

• String is represented as an array of chars.
  – Array has no space to encode its length.
• How to determine string length?
  – explicitly pass around an integer representing length

// tolower_string turns every character in character array s
// into lower case
void tolower_string(char *s, int len) {
    for (int i = 0; i < len; i++) {
        s[i] = tolower(s[i]);
    }
}
Strings

• String is represented as an array of chars.
  – Array has no space to encode its length.

• How to determine string length?
  – explicitly pass around an integer representing length
  – C string stores a NULL character to mark the end (by convention)

```c
void tolower_string(char *s) {
}
```
Strings

• String is represented as an array of chars.
  – Array has no space to encode its length.

• How to determine string length?
  – explicitly pass around an integer representing length
  – C string stores a NULL character to mark the end (by convention)

```c
void tolower_string(char *s) {
    int i = 0;
    while (s[i] != ‘\0’) {
        s[i] = tolower(s[i]);
        i++;
    }
}
```
Copying string?

does this make a copy of “hi”?

```c
char s[3] = {'h', 'i', '\0'};
char *h;
h = s;
h[0] = 'H';
printf("s=%s  h=%s\n", s, h);
```

![Diagram showing memory allocation and values for s and h]
Copying string?

does this make a copy of “hi”?

```c
char s[3] = {'h', 'i', '\0'};
char h[3];
h = s;
h[0] = 'H';

printf("s=%s h=%s\n", s, h);
```
```c
void strcpy(char *dst, char *src)
{

}

int main()
{
    char s[3] = {‘h’, ‘i’, ‘\0’};
    char h[3];
    strcpy(h, s);
    h[0] = ‘H’;

    printf("s=%s h=%s\n", s, h);
}
```
void strcpy(char *dst, char *src) {
    int i = 0;
    while (src[i] != '\0') {
        dst[i] = src[i];
        i++;
    }
}

int main() {
    char s[3] = {'h','i','\0'};
    char h[3];
    strcpy(h, s);
    h[0] = 'H';

    printf("s=%s h=%s\n", s, h);
}
void strcpy(char *dst, char *src) {
    int i = 0;
    while (src[i] != '\0') {
        dst[i] = src[i];
        i++;
    }
}

int main() {
    char s[3] = {'h', 'i', '\0'};
    char h[2];
    strcpy(h, s);
    h[0] = 'H';

    printf("s=%s h=%s\n", s, h);
}
Copying string

```c
void strncpy(char *dst, char *src, int n) {
    int i = 0;
    while (src[i] != '\0' && i < n) {
        dst[i] = src[i];
        i++;
    }
}

int main() {
    char s[3] = {'h', 'i', '\0'};
    char h[2];
    strncpy(h, s, 2);
    h[0] = 'H';

    printf("s=%s h=%s\n", s, h);
}
```

strncpy is included in C std library.
A different way of initializing string

...
A different way of initializing string

char s1[3] = {'h', 'i', '\0'};
// equivalent to
// char s1[3] = “hi”;
char *s2 = “bye”;
s1[0] = ‘H’;
s2[0] = ‘B’;

printf(“s1=%s s2=%s\n”, s1, s2);
The Atoi function

// atoi returns the integer
// corresponding to the string of digits
int atoi(char *s)
{

}

int main()
{
  char *s = "123";
  printf("integer is %d\n", atoi(s));
}
The Atoi function

// atoi returns the integer
// corresponding to the string of digits
int atoi(char *s) {
    int result = 0;
    int i = 0;
    while (s[i] >= '0' && s[i] <= '9') {
        result = result * 10 + (s[i] - '0');
        i++;
    }
    return result;
}
Array of pointers

```c
char* names[3] = {
    "alice",
    "bob",
    "clark"
};

char **namep;
namep = names;

printf("name is %s", namep[1]);
```
The most commonly used array of pointers: argv

```c
int main(int argc, char **argv)
{
    for (int i = 0; i < argc; i++) {
        printf(“%s
”, argv[i]);
    }
}
```

```
$ ./a.out 1 2 3
./a.out 1 2 3
```

argv[0] is the name of the executable
Structs

Struct stores fields of different types contiguously in memory.
Structure

• Array: a block of n consecutive elements of the same type.

• Struct: a collection of elements of different types.
```c
struct student {
    int id;
    char *name;
};
```

Fields of a struct are allocated next to each other, but there may be gaps (padding) between them.
struct student {
    int id;
    char *name;
};

struct student t; /* define variable t with type “struct student” */
Structure

struct student {
    int id;
    char *name;
};

struct student t;

t.id = 1024;  // Access the fields of this struct

t.name = "alice";
typedef struct {
    int id;
    char *name;
} student;

struct student t;
typedef struct {
    int id;
    char *name;
} student;

student t = {1023, "alice"};
student *p = &t;

p->id = 1023;
p->name = "bob";
printf("%d %s\n", t.id, t.name);
Malloccs

Allocates a chunk of memory dynamically
Recall memory allocation for global and local variables

- **Global** variables are allocated space before program execution.

- **Local** variables are allocated when entering a function and de-allocated upon its exit.
Malloc

Allocate space dynamically and flexibly:
- `malloc`: allocate storage of a given size
- `free`: de-allocate previously malloc-ed storage

```c
void *malloc(size_t size);
```

A `void` pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be casted to any type.

```c
void free(void *ptr);
```
#include <stdlib.h>

int *newArray(int n) {
    int *p;
    p = (int*)malloc(sizeof(int) * n);
    return p;
}

Malloc is implemented as a C library
Conceptual view of a C program’s memory at runtime

• Separate memory regions for global, local, and malloc-ed.

We will refine this simple view in later lectures
Linked list in C: insertion

typedef struct {
    long val;
    struct node *next;
}node;

// insert val into linked list to the head of the linked
// list and return the new head of the list in *head
void
insert(node **head, long val) {
}

int main() {
    node *head = NULL;
    for (long i = 10; i < 13; i++)
        insert(&head, i);
}
void insert(node **headp, long val) {
}

int main() {
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
1st insert call

```c
void insert(node **headp, long val) {
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```
```c
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
void insert(node **headp, long val) {
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main() {
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
void insert(node **headp, long val) {
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main() {
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
void insert(node **headp, long val) {
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main() {
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}