Arrays

An *array* is a collection of elements all of the same type.

Arrays

An *array* is a collection of elements all of the same type.

- Declared with *type name*[*size*].

Arrays

An *array* is a collection of elements all of the same type.

- Declared with *type name*`[`*size*`]`.

  Example: `double x[100];`

Arrays

An *array* is a collection of elements all of the same type.

- Declared with *type name*`[`*size*`]`.

  Example: `double x[100];`

- The elements are numbered `0`, `1`, `2`, `3`, . . . , *size*`-1`

Arrays

An *array* is a collection of elements all of the same type.

- Declared with *type name*`[`*size*`]`.

  Example: `double x[100];`
- The elements are numbered `0`, `1`, `2`, `3`, . . . , *size*`-1`
- Once an array is created it has a fixed size.

Arrays

An *array* is a collection of elements all of the same type.

- Declared with *type name*`[`*size*`]`.

  Example: `double x[100];`
- The elements are numbered `0`, `1`, `2`, `3`, . . . , *size*`-1`
- Once an array is created it has a fixed size.
- Individual elements are accessed using the *index operator* `[]`.

Arrays

An *array* is a collection of elements all of the same type.

- Declared with *type name*`[`*size*`]`.

  Example: `double x[100];`
- The elements are numbered `0`, `1`, `2`, `3`, . . . , *size*`-1`
- Once an array is created it has a fixed size.
- Individual elements are accessed using the *index operator* `[]`.

  Example: `x[i] = (x[i-1] + x[i+1])/2.;`

Arrays

An *array* is a collection of elements all of the same type.

- Declared with *type name*`[`*size*`]`.

  Example: `double x[100];`
- The elements are numbered `0`, `1`, `2`, `3`, . . . , *size*`-1`
- Once an array is created it has a fixed size.
- Individual elements are accessed using the *index operator* `[]`.

  Example: `x[i] = (x[i-1] + x[i+1])/2.;`
- Arrays are allocated in contiguous memory.

Arrays cont.

Arrays cont.

- Arrays can be initialised in the declaration.

  Examples:

```
int a[10] = {1, 2, 3, 4};
char c[] = {'h', 'e', 'l', 'l', 'o'};
```

  `a` will be of size 10 and `c` of size 5.

Arrays cont.

- Arrays can be initialised in the declaration.

  Examples:

  ```
  int a[10] = {1, 2, 3, 4};
  char c[] = {'h', 'e', 'l', 'l', 'o'};
  ```

  `a` will be of size 10 and `c` of size 5.

- The "array constructors" can **only** be used declarations.

Arrays cont.

- Arrays can be initialised in the declaration.

  Examples:

  ```
  int a[10] = {1, 2, 3, 4};
  char c[] = {'h', 'e', 'l', 'l', 'o'};
  ```

  `a` will be of size 10 and `c` of size 5.

- The "array constructors" can **only** be used declarations.
- Arrays can **not** be moved/copied using the assignment operator `=`. Use iterations!

Arrays cont.

- Arrays can be initialised in the declaration.

  Examples:

  ```
  int a[10] = {1, 2, 3, 4};
  char c[] = {'h', 'e', 'l', 'l', 'o'};
  ```

  `a` will be of size 10 and `c` of size 5.

- The "array constructors" can **only** be used declarations.
- Arrays can **not** be moved/copied using the assignment operator `=`. Use iterations!

  Example:

  ```
  for (int i = 0; i < 10; i++) {
      b[i] = a[i];
  }
  ```

Example: Read an array

```
double v[8];

printf("Enter double values. End with a letter: ");
int counter = 0;
while(scanf("%lf", &v[counter])==1) {
  counter++;
}
```

Example: Read an array

```
double v[8];

printf("Enter double values. End with a letter: ");
int counter = 0;
while(scanf("%lf", &v[counter])==1) {
  counter++;
}
```

Note:

Example: Read an array

```
double v[8];

printf("Enter double values. End with a letter: ");
int counter = 0;
while(scanf("%lf", &v[counter])==1) {
  counter++;
}
```

Note:

1. the `lf` format code,

Example: Read an array

```
double v[8];

printf("Enter double values. End with a letter: ");
int counter = 0;
while(scanf("%lf", &v[counter])==1) {
  counter++;
}
```

Note:

1. the `lf` format code,
2. the address operator `&` and

Example: Read an array

```
double v[8];

printf("Enter double values. End with a letter: ");
int counter = 0;
while(scanf("%lf", &v[counter])==1) {
  counter++;
}
```

Note:

1. the `lf` format code,
2. the address operator `&` and
3. that the code is unsafe.

Example: A print function for an array

```
void print(double v[],
           int n) {
  printf("[");
  for (int i=0; i<n; i++) {
    printf("%4.1lf", v[i]);
    if (i < n-1) {
      printf(", ");
    } else {
      printf("]\n");
    }
  }
}
```

Example: A print function for an array

```
void print(double v[],
           int n) {
  printf("[");
  for (int i=0; i<n; i++) {
    printf("%4.1lf", v[i]);
    if (i < n-1) {
      printf(", ");
    } else {
      printf("]\n");
    }
  }
}
```

Note:

Example: A print function for an array

```
void print(double v[],
           int n) {
  printf("[");
  for (int i=0; i<n; i++) {
    printf("%4.1lf", v[i]);
    if (i < n-1) {
      printf(", ");
    } else {
      printf("]\n");
    }
  }
}
```

Note:

- The declaration of the array parameter.

Example: A print function for an array

```
void print(double v[],
           int n) {
  printf("[");
  for (int i=0; i<n; i++) {
    printf("%4.1lf", v[i]);
    if (i < n-1) {
      printf(", ");
    } else {
      printf("]\n");
    }
  }
}
```

Note:

- The declaration of the array parameter.
- The used size must be passed to the function.
  The is *no way* for the function to know neither how much we have used nor the declared size of the array!

Example: A print function for an array

```
void print(double v[],
           int n) {
  printf("[");
  for (int i=0; i<n; i++) {
    printf("%4.1lf", v[i]);
    if (i < n-1) {
      printf(", ");
    } else {
      printf("]\n");
    }
  }
}
```

Note:

- The declaration of the array parameter.
- The used size must be passed to the function.
  The is *no way* for the function to know neither how much we have used nor the declared size of the array!

Link to demo program.

Example: An array as both input and output parameter

Example: An array as both input and output parameter

```
void normalize(double v[], int n) {
  double max = maxNorm(v, n);      // maxNorm defined below
  for (int i= 0; i<n; i++) {
    v[i] = v[i]/max;
  }
}
```

Example: An array as both input and output parameter

```
void normalize(double v[], int n) {
  double max = maxNorm(v, n);      // maxNorm defined below
  for (int i= 0; i<n; i++) {
    v[i] = v[i]/max;
  }
}
```

Why does the code work when parameters are passed by value???

Example: An array as both input and output parameter

```
void normalize(double v[], int n) {
  double max = maxNorm(v, n);      // maxNorm defined below
  for (int i= 0; i<n; i++) {
    v[i] = v[i]/max;
  }
}
```

Why does the code work when parameters are passed by value???

```
double maxNorm(double v[], int n) {
  double result = 0;
  for (int i=0; i<n; i++) {
    result = fmax(result, fabs(v[i]));
  }
  return result;
}
```

Example: An array as both input and output parameter

```
void normalize(double v[], int n) {
  double max = maxNorm(v, n);      // maxNorm defined below
  for (int i= 0; i<n; i++) {
    v[i] = v[i]/max;
  }
}
```

Why does the code work when parameters are passed by value???

```
double maxNorm(double v[], int n) {
  double result = 0;
  for (int i=0; i<n; i++) {
    result = fmax(result, fabs(v[i]));
  }
  return result;
}
```

Note: `fmax` and `fabs`

Example: An array as both input and output parameter

```
void normalize(double v[], int n) {
  double max = maxNorm(v, n);      // maxNorm defined below
  for (int i= 0; i<n; i++) {
    v[i] = v[i]/max;
  }
}
```

Why does the code work when parameters are passed by value???

```
double maxNorm(double v[], int n) {
  double result = 0;
  for (int i=0; i<n; i++) {
    result = fmax(result, fabs(v[i]));
  }
  return result;
}
```

Note: `fmax` and `fabs`

Link to demo program

Strings

Strings

- There is no data type for strings in C.
  They are handled as *array of chars*.

Strings

- There is no data type for strings in C.
  They are handled as *array of chars*.
- The only syntactic support in the language is that they can be written enclosed by quotation marks. Example: `"Hello"`

Strings

- There is no data type for strings in C.
  They are handled as *array of chars*.
- The only syntactic support in the language is that they can be written enclosed by quotation marks. Example: `"Hello"`
- By convention, strings are terminated by a null character. Thus, the declaration

  ```
  char c[] = "Hello";
  ```

  is equivalent to

  ```
  char c[] = {'H', 'e', 'l', 'l, 'o', '\0'};
  ```

Strings

- There is no data type for strings in C.
  They are handled as *array of chars*.
- The only syntactic support in the language is that they can be written enclosed by quotation marks. Example: `"Hello"`
- By convention, strings are terminated by a null character. Thus, the declaration

  ```
  char c[] = "Hello";
  ```

  is equivalent to

  ```
  char c[] = {'H', 'e', 'l', 'l, 'o', '\0'};
  ```

- Can **not** use operators like =, ==, <, . . . on them

Example: The printBase program revisited

```
char myDigits[] = "0123456789abcdef";

void printb(int n, int b) {
  assert(b >=2 && b <= 16);
  if (n < 0) {
    putchar('-');
    printb(-n, b);
  } else if (n < b) {
    putchar(myDigits[n]);
  } else {
    printb(n/b, b);
    printb(n%b, b);
  }
}
```

Example: The printBase program revisited

```
char myDigits[] = "0123456789abcdef";

void printb(int n, int b) {
  assert(b >=2 && b <= 16);
  if (n < 0) {
    putchar('-');
    printb(-n, b);
  } else if (n < b) {
    putchar(myDigits[n]);
  } else {
    printb(n/b, b);
    printb(n%b, b);
  }
}
```

- A *global* array with digits.

Example: The printBase program revisited

```
char myDigits[] = "0123456789abcdef";

void printb(int n, int b) {
  assert(b >=2 && b <= 16);
  if (n < 0) {
    putchar('-');
    printb(-n, b);
  } else if (n < b) {
    putchar(myDigits[n]);
  } else {
    printb(n/b, b);
    printb(n%b, b);
  }
}
```

- A *global* array with digits.

  Why global?

Example: The printBase program revisited

```
char myDigits[] = "0123456789abcdef";

void printb(int n, int b) {
  assert(b >=2 && b <= 16);
  if (n < 0) {
    putchar('-');
    printb(-n, b);
  } else if (n < b) {
    putchar(myDigits[n]);
  } else {
    printb(n/b, b);
    printb(n%b, b);
  }
}
```

- A *global* array with digits.

  Why global?

- `putchar` instead of `printf`

Example: The printBase program revisited

```
char myDigits[] = "0123456789abcdef";

void printb(int n, int b) {
  assert(b >=2 && b <= 16);
  if (n < 0) {
    putchar('-');
    printb(-n, b);
  } else if (n < b) {
    putchar(myDigits[n]);
  } else {
    printb(n/b, b);
    printb(n%b, b);
  }
}
```

- A *global* array with digits.

  Why global?

- `putchar` instead of `printf`

Link to a demo program

Example: A string comparison

```
int equals(char s[], char t[]) {
    int i = 0;
    while (s[i] == t[i] && s[i] != '\0') {
        i++;
    }
    return s[i] == t[i];
}
```

Example: A string comparison

```
int equals(char s[], char t[]) {
    int i = 0;
    while (s[i] == t[i] && s[i] != '\0') {
        i++;
    }
    return s[i] == t[i];
}
```

Note:

Example: A string comparison

```
int equals(char s[], char t[]) {
   int i = 0;
   while (s[i] == t[i] && s[i] != '\0') {
      i++;
   }
   return s[i] == t[i];
}
```

Note:

1. Since strings are null terminated we do not need to pass the string length (array size) to the function.

Example: A string comparison

```
int equals(char s[], char t[]) {
   int i = 0;
   while (s[i] == t[i] && s[i] != '\0') {
      i++;
   }
   return s[i] == t[i];
}
```

Note:

1. Since strings are null terminated we do not need to pass the string length (array size) to the function.
2. You don't have to write this function – use `strcmp` in the string libraray!

The string library

```
int  strlen (char s[]);
void strcpy (char s1[], char s2[]);
void strncpy(char s1[], char s2[], int n);
void strcat (char s1[], char s2[]);
void strncat(char s1[], char s2[], int n);
int  strcmp (char s1[], char s2[]);
...
```

(Somewhat simplified regarding parameter types and return values.)

Include `stdio.h` to use!

Example: Reading and writing strings

```
/* Find the longest word in standard input */
#include <stdio.h>
#include <string.h>

int main() {
  char word[100];
  char longest[100] = "";
  int length = 0;
  while(scanf("%s", word) == 1) {
    printf("%s \n", word);
    if (strlen(word) > strlen(longest)) {
      strcpy(longest, word);
    }
  }
  printf("\n\nLongest 'word': %s\n", longest);
}
```

Link to readWords.c

Pointer examples

Pointer examples

```
int  m = 1;        // An ordinary variable
```

Pointer examples

```
int  m = 1;        // An ordinary variable
int *p = &m;       // A pointer to m
```

Pointer examples

```
int  m = 1;        // An ordinary variable
int *p = &m;       // A pointer to m
int *q = p;        // Another pointer to m
```

Pointer examples

```
int  m = 1;        // An ordinary variable
int *p = &m;       // A pointer to m
int *q = p;        // Another pointer to m
int  n = *p;       // n will be 1
```

Pointer examples

```
int  m = 1;       // An ordinary variable
int *p = &m;      // A pointer to m
int *q = p;       // Another pointer to m
int  n = *p;      // n will be 1
*p     = 2;       // m will be 2
```

Pointer examples

```
int  m = 1;       // An ordinary variable
int *p = &m;      // A pointer to m
int *q = p;       // Another pointer to m
int  n = *p;      // n will be 1
*p     = 2;       // m will be 2
q      = &n;      // Points to n
```

Pointer examples

```
int  m = 1;        // An ordinary variable
int *p = &m;       // A pointer to m
int *q = p;        // Another pointer to m
int  n = *p;       // n will be 1
*p     = 2;        // m will be 2
q      = &n;       // Points to n
(*q)--;            // n will be 0
```

Pointer examples

```
int  m = 1;        // An ordinary variable
int *p = &m;       // A pointer to m
int *q = p;        // Another pointer to m
int  n = *p;       // n will be 1
*p     = 2;        // m will be 2
q      = &n;       // Points to n
(*q)--;            // n will be 0
*q--;              // Meaningless! No effect!
```

Pointer examples

```
int  m = 1;        // An ordinary variable
int *p = &m;       // A pointer to m
int *q = p;        // Another pointer to m
int  n = *p;       // n will be 1
*p     = 2;        // m will be 2
q      = &n;       // Points to n
(*q)--;            // n will be 0
*q--;              // Meaningless! No effect!
--*q;              // n will be -1
```

Pointer examples

```
int  m = 1;        // An ordinary variable
int *p = &m;       // A pointer to m
int *q = p;        // Another pointer to m
int  n = *p;       // n will be 1
*p     = 2;        // m will be 2
q      = &n;       // Points to n
(*q)--;            // n will be 0
*q--;              // Meaningless! No effect!
--*q;              // n will be -1
p      = NULL;     // Pointer constant
```

Pointer examples

```
int  m = 1;        // An ordinary variable
int *p = &m;       // A pointer to m
int *q = p;        // Another pointer to m
int  n = *p;       // n will be 1
*p     = 2;        // m will be 2
q      = &n;       // Points to n
(*q)--;            // n will be 0
*q--;              // Meaningless! No effect!
--*q;              // n will be -1
p      = NULL;     // Pointer constant
*p     = 99;       // Illegal!
```

Pointers and arrays

Pointers and arrays

```
int  a = {1, 2, 3, 4, 5};
```

Pointers and arrays

```
int  a = {1, 2, 3, 4, 5};
int *p = &a[0];          // A pointer to first element
```

Pointers and arrays

```
int  a = {1, 2, 3, 4, 5};
int *p = &a[0];          // A pointer to first element
int *q = a;              // Also a pointer to the first element
```

Pointers and arrays

```
int  a = {1, 2, 3, 4, 5};
int *p = &a[0];          // A pointer to first element
int *q = a;              // Also a pointer to the first element
printf("%d", *(p+2));  // Prints the third element (3)
```

Pointers and arrays

```
int  a = {1, 2, 3, 4, 5};
int *p = &a[0];          // A pointer to first element
int *q = a;              // Also a pointer to the first element
printf("%d", *(p+2));    // Prints the third element (3)
*p     = -1;             // same as a[0] = -1;
```

Pointers and arrays

```
int  a = {1, 2, 3, 4, 5};
int *p = &a[0];          // A pointer to first element
int *q = a;              // Also a pointer to the first element
printf("%d", *(p+2));    // Prints the third element (3)
*p     = -1;             // same as a[0] = -1;
p[1]      = -2;          // same as *(p+1) = -2;
```

Pointers and arrays

```
int  a = {1, 2, 3, 4, 5};
int *p = &a[0];          // A pointer to first element
int *q = a;              // Also a pointer to the first element
printf("%d", *(p+2));    // Prints the third element (3)
*p     = -1;             // same as a[0] = -1;
p[1]      = -2;          // same as *(p+1) = -2;


q = a + 4;                          // Last position
for (int *r = a; r <= q; r++) {     // Iterate over the array
   printf("%d ", *r);
}
```

Arrays revisited

Here is a link to the program with array functions where some of the array notations have replaced by pointer notations.

It shows that there is very little difference between an array and a pointer.

Arrays revisited

Here is a link to the program with array functions where some of the array notations have replaced by pointer notations.

It shows that there is very little difference between an array and a pointer.

It also contains an example of the *conditional operator* in the print function:

Arrays revisited

Here is a link to the program with array functions where some of the array notations have replaced by pointer notations.

It shows that there is very little difference between an array and a pointer.

It also contains an example of the *conditional operator* in the print function:

Instead of

```
if (i < n-1) {
    printf(", ");
} else {
    printf("]\n");
}
```

Arrays revisited

Here is a link to the program with array functions where some of the array notations have replaced by pointer notations.

It shows that there is very little difference between an array and a pointer.

It also contains an example of the *conditional operator* in the print function:

Instead of

```
if (i < n-1) {
    printf(", ");
} else {
    printf("]\n");
}
```

we can write

```
i < n-1 ? printf(", ") : printf("]\n");
```