

Arrays and Strings

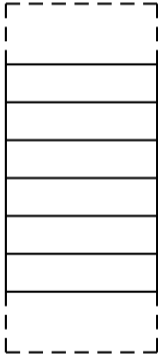
Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

October 7, 2024

- General memory model
- Definition and use of pointers
- Invalid pointers and common errors
- Arrays and pointers
- Strings
- The main function

computer memory

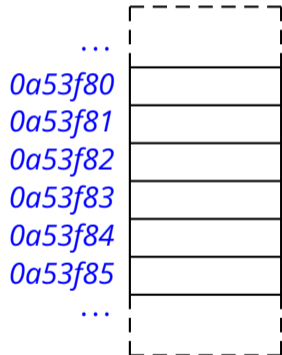


Memory Model

computer memory

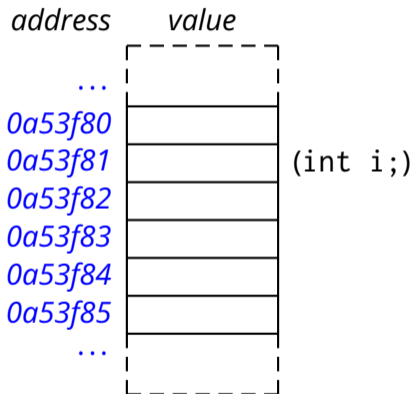
address

value



Memory Model

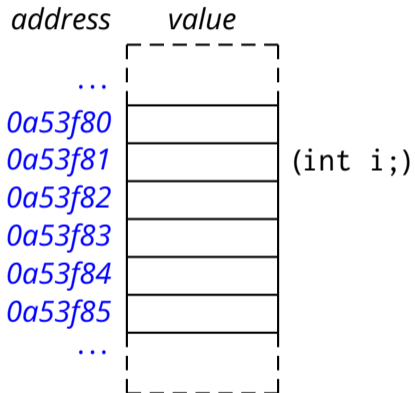
computer memory



```
/* an int variable */  
int i;
```

Memory Model

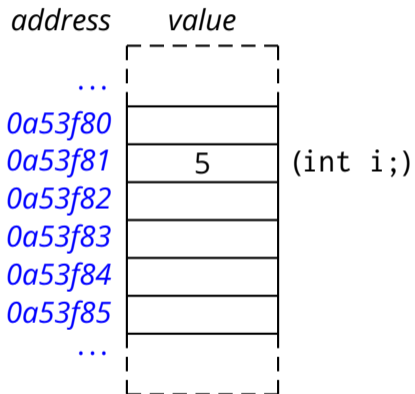
computer memory



```
/* an int variable */  
int i;  
i = 5;
```

Memory Model

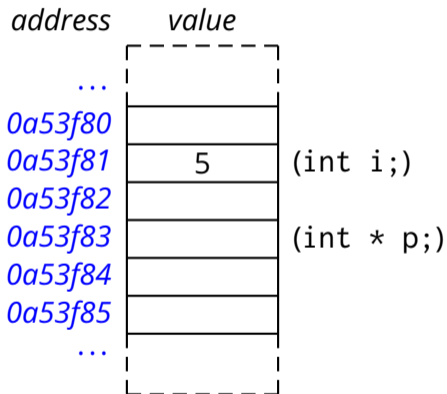
computer memory



```
/* an int variable */  
int i;  
i = 5;
```

Memory Model

computer memory



```
/* an int variable */
```

```
int i;
```

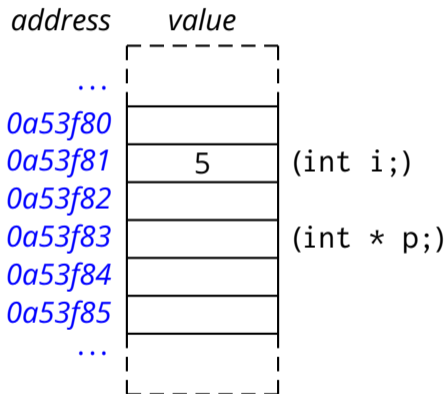
```
i = 5;
```

```
/* pointer to an int */
```

```
int * p;
```

Memory Model

computer memory



```
/* an int variable */
```

```
int i;
```

```
i = 5;
```

```
/* pointer to an int */
```

```
int * p;
```

```
/* pointer assignment */
```

```
p = &i;
```

Memory Model

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80		
0a53f81	5	(int i;)
0a53f82		
0a53f83	0a53f81	(int * p;)
0a53f84		
0a53f85		
...		

```
/* an int variable */
```

```
int i;
```

```
i = 5;
```

```
/* pointer to an int */
```

```
int * p;
```

```
/* pointer assignment */
```

```
p = &i;
```

Memory Model

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80		
0a53f81	5	(int i;)
0a53f82		
0a53f83	0a53f81	(int * p;)
0a53f84		
0a53f85		
...		

```
/* an int variable */  
int i;  
i = 5;
```

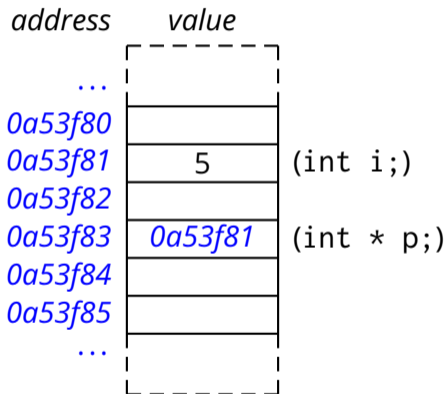
```
/* pointer to an int */  
int * p;
```

```
/* pointer assignment */  
p = &i;
```

```
/* pointer dereference */  
printf("%d\n", *p);
```

Memory Model

computer memory



```
/* an int variable */  
int i;  
i = 5;
```

```
/* pointer to an int */  
int * p;
```

```
/* pointer assignment */  
p = &i;
```

```
/* pointer dereference */  
printf("%d\n", *p);
```

```
*p = 13;
```

Memory Model

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80		
0a53f81	13	(int i;)
0a53f82		
0a53f83	0a53f81	(int * p;)
0a53f84		
0a53f85		
...		

```
/* an int variable */  
int i;  
i = 5;  
  
/* pointer to an int */  
int * p;  
  
/* pointer assignment */  
p = &i;  
  
/* pointer dereference */  
printf("%d\n", *p);  
  
*p = 13;
```

Memory Model

computer memory

address	value	
...		
0a53f80		
0a53f81	13	(int i;)
0a53f82		
0a53f83	0a53f81	(int * p;)
0a53f84		
0a53f85		
...		

address-of operator

```
/* an int variable */  
int i;  
i = 5;  
  
/* pointer to an int */  
int * p;  
  
/* pointer assignment */  
p = &i;  
  
/* pointer dereference */  
printf("%d\n", *p);  
  
*p = 13;
```

Memory Model

computer memory

address	value	
...		
0a53f80		
0a53f81	13	(int i;)
0a53f82		
0a53f83	0a53f81	(int * p;)
0a53f84		
0a53f85		
...		

address-of operator

object pointed-to by p

```
/* an int variable */
int i;
i = 5;

/* pointer to an int */
int * p;

/* pointer assignment */
p = &i;

/* pointer dereference */
printf("%d\n", *p);

*p = 13;
```

Objects and Pointers

- An ***object*** is an area in memory that holds a ***value*** of a certain ***type***
 - ▶ e.g., an `int`, a `char`, a "structure", an array of `int`, and array of struct, ...

"object" \approx "variable"

- An **object** is an area in memory that holds a **value** of a certain **type**
 - ▶ e.g., an **int**, a **char**, a "structure", an array of **int**, and array of struct, ...

"object" \approx "variable"

- An **object** is an area in memory that holds a **value** of a certain **type**
 - ▶ e.g., an **int**, a **char**, a "structure", an array of **int**, and array of struct, ...
- A **pointer** is an **object** whose value is the **memory address of another object**

"object" \approx "variable"

- An **object** is an area in memory that holds a **value** of a certain **type**
 - ▶ e.g., an `int`, a `char`, a "structure", an array of `int`, and array of struct, ...
- A **pointer** is an **object** whose value is the **memory address of another object**
- Pointers are **typed** according to the object they point to

“object” \approx “variable”

- An **object** is an area in memory that holds a **value** of a certain **type**
 - ▶ e.g., an **int**, a **char**, a “structure”, an array of **int**, and array of struct, ...
- A **pointer** is an **object** whose value is the **memory address of another object**
- Pointers are **typed** according to the object they point to

```
int * p;    /* pointer to an int */
int ** pp;  /* pointer to a pointer to an int */
char c;     /* a char variable */
int i;      /* an int variable */

p = &c;     /* type mismatch! */
p = &i;     /* okay */
pp = &i;    /* type mismatch! */
pp = &p;    /* okay */
```

- A pointer is like any other variable

- A pointer is like any other variable
 - ▶ it can be assigned a value (of a compatible *pointer type*)
 - ▶ it can be “dereferenced” to read the “pointed” value
 - ▶ it can be “dereferenced” to write the “pointed” value
 - ▶ more generally, it can be used in arbitrary expressions

- A pointer is like any other variable
 - ▶ it can be assigned a value (of a compatible *pointer type*)
 - ▶ it can be “dereferenced” to read the “pointed” value
 - ▶ it can be “dereferenced” to write the “pointed” value
 - ▶ more generally, it can be used in arbitrary expressions

■ Example:

```
int main() {
    int i = 123;
    int * p;    /* pointer declaration */
    p = &i;    /* address-of operator */
    *p = 345;  /* dereference operator */
    printf("i=%d\n", i);
    printf("*p=%d\n", *p);
}
```


- Pointers cause *side-effects*
 - ▶ they should be used with special care
 - ▶ at the same time they are *indispensable*

...but first we need to talk about parameters passing

Pop quiz: What is the output of this program?

```
#include <stdio.h>

void sign(int x) {
    if (x > 0)
        x = 1;
    else if (x < 0)
        x = -1;
}

int main() {
    int i = 7;
    sign(i);
    printf("i = %d\n", i);
}
```

Pop quiz: What is the output of this program?

```
#include <stdio.h>

void sign(int x) {
    if (x > 0)
        x = 1;
    else if (x < 0)
        x = -1;
}

int main() {
    int i = 7;
    sign(i);
    printf("i = %d\n", i);
}
```

Answer: `i = 7`

Pop quiz: What is the output of this program?

```
#include <stdio.h>

void sign(int x) {
    if (x > 0)
        x = 1;
    else if (x < 0)
        x = -1;
}

int main() {
    int i = 7;
    sign(i);
    printf("i = %d\n", i);
}
```

Answer: `i = 7`

in C *parameters are always passed by value* ("call by value" semantics)

Parameters, Arguments, and their Semantics

Parameters, Arguments, and their Semantics

```
int maximum(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}

int set_temp(int t) {
    int new_t = maximum(t, 36);
    printf("new temperature is %d\n", new_t)
}
```

Parameters, Arguments, and their Semantics

function definition

```
int maximum(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}

int set_temp(int t) {
    int new_t = maximum(t, 36);
    printf("new temperature is %d\n", new_t)
}
```

Parameters, Arguments, and their Semantics

function definition

```
int maximum(int a, int b) {  
    if (a > b)  
        return a;  
    else  
        return b;  
}
```

parameters (a.k.a., "formal" parameters)

```
int set_temp(int t) {  
    int new_t = maximum(t, 36);  
    printf("new temperature is %d\n", new_t)  
}
```

Parameters, Arguments, and their Semantics

function definition

```
int maximum(int a, int b) {  
    if (a > b)  
        return a;  
    else  
        return b;  
}
```

parameters (a.k.a., "formal" parameters)

function invocation

```
int set_temp(int t) {  
    int new_t = maximum(t, 36);  
    printf("new temperature is %d\n", new_t)  
}
```

Parameters, Arguments, and their Semantics

function definition

```
int maximum(int a, int b) {  
    if (a > b)  
        return a;  
    else  
        return b;  
}
```

parameters (a.k.a., "formal" parameters)

function invocation

```
int set_temp(int t) {  
    int new_t = maximum(t, 36);  
    printf("new temperature is %d\n", new_t)  
}
```

arguments (a.k.a., "actual" parameters)

Parameters, Arguments, and their Semantics

function definition

```
int maximum(int a, int b) {  
    if (a > b)  
        return a;  
    else  
        return b;  
}
```

parameters (a.k.a., "formal" parameters)

function invocation

```
int set_temp(int t) {  
    int new_t = maximum(t, 36);  
    printf("new temperature is %d\n", new_t)  
}
```

arguments (a.k.a., "actual" parameters)

runs maximum with
int a = t;
int b = 36;

Parameters, Arguments, and their Semantics

function definition

```
int maximum(int a, int b) {  
    if (a > b)  
        return a;  
    else  
        return b;  
}
```

parameters (a.k.a., "formal" parameters)

function invocation

```
int set_temp(int t) {  
    int new_t = maximum(t, 36);  
    printf("new temperature is %d\n", new_t)  
}
```

arguments (a.k.a., "actual" parameters)

runs maximum with
int a = t;
int b = 36;

A function call **initializes the parameters with the values of the arguments**

Parameters, Arguments, and their Semantics

A function call *initializes* each *parameter* with the *value of the corresponding argument*

Parameters, Arguments, and their Semantics

A function call *initializes* each *parameter* with the *value of the corresponding argument*

- The parameters are fresh, new variables
- The parameters are totally independent from the arguments, *and vice-versa!*
 - ▶ (apart from the initialization)

Parameters, Arguments, and their Semantics

A function call *initializes* each *parameter* with the *value of the corresponding argument*

- The parameters are fresh, new variables
- The parameters are totally independent from the arguments, *and vice-versa!*
 - ▶ (apart from the initialization)

So, how do we get information *out* of a function, other than through its return value?

- Pointers cause *side-effects*
 - ▶ they should be used with special care
 - ▶ at the same time they are *indispensable*

- Pointers cause *side-effects*
 - ▶ they should be used with special care
 - ▶ at the same time they are *indispensable*

Question: how do we get data out of a function?

- Pointers cause *side-effects*
 - ▶ they should be used with special care
 - ▶ at the same time they are *indispensable*

Question: how do we get data out of a function?

Answer: by letting the function take *pointers* as parameters

- Pointers cause *side-effects*
 - ▶ they should be used with special care
 - ▶ at the same time they are *indispensable*

Question: how do we get data out of a function?

Answer: by letting the function take *pointers* as parameters

■ Example:

```
int main() {
    int i;
    printf("How old are you? ");
    scanf("%d", &i);
    printf("You look a little older than %d\n", i);
}
```

- Write (1) a C function called swap that swaps two integer variables and (2) a small C program that uses the swap function

- Write (1) a C function called `swap` that swaps two integer variables and (2) a small C program that uses the `swap` function

Hint: the `swap` function takes two *pointers* as parameters

- Write (1) a C function called `swap` that swaps two integer variables and (2) a small C program that uses the `swap` function

Hint: the `swap` function takes two *pointers* as parameters

- Example

```
void swap(int * p1, int * p2) {
    *p1 ^= *p2;
    *p2 ^= *p1;
    *p1 ^= *p2;
}
int main() {
    int i = 7;
    int j = 13;
    swap(&i,&j);
    printf("i=%d j=%d\n", i, j);
}
```

- Special *pointer type* compatible with any other pointer type (only in C)
 - ▶ i.e., can be assigned to/from any pointer type

```
#include <stdio.h>

int main() {
    int i;
    void * ptr = &i;
    int * i_ptr = ptr;          /* in C++: i_ptr = static_cast <int*> (ptr) */
    int * i_ptr_2 = &i;

    if (i_ptr != i_ptr_2) {
        /* should never be the case */
        printf("Your system is broken!\n");
    }
}
```

Pointer to void (Example, only in C)

```
int i;
char c;
void * int_or_char(int type) {
    if (type == 0)
        return &i;          /* (void *) <-- (int *) */
    else
        return &c;          /* (void *) <-- (char *) */
}
int main() {
    int t;
    scanf("%d", &t);
    if (t == 0) {
        int * p = int_or_char(t); /* (int *) <-- (void *) */
        *p = 123;
    } else {
        char * p = int_or_char(t); /* (char *) <-- (void *) */
        *p = 'a';
    }
}
```

- The special “null” *pointer value*

■ The special “null” *pointer value*

- ▶ integer constant expression with value `0`
- ▶ or integer constant expression with value `0` cast to `void *`
- ▶ or the `NULL` macro defined in `<stddef.h>`
- ▶ usable with any pointer type
 - ▶ `0` and `NULL` convert to any pointer type
 - ▶ a null pointer of any type compares *equal* to `0` or `NULL`
- ▶ guaranteed to *never compare equal to any valid pointer*
- ▶ in C++ and since C23, also `nullptr` pointer literal value (preferred)

- Pointers are “dangerous” because they can take a ***restricted set of valid values***
 - ▶ values set by the platform
 - ▶ values themselves are *meaningless to the application*
 - ▶ in general, you can not check whether a pointer is valid

- Pointers are “dangerous” because they can take a ***restricted set of valid values***
 - ▶ values set by the platform
 - ▶ values themselves are *meaningless to the application*
 - ▶ in general, you can not check whether a pointer is valid

■ Example

```
int * p;  
*p = 345; /* dereference on invalid pointer */
```

- Pointers are “dangerous” because they can take a ***restricted set of valid values***
 - ▶ values set by the platform
 - ▶ values themselves are *meaningless to the application*
 - ▶ in general, you can not check whether a pointer is valid

■ Example

```
int * p;  
*p = 345; /* dereference on invalid pointer */
```

- Dereferencing an invalid pointer causes ***undefined behavior***

- Pointers are “dangerous” because they can take a ***restricted set of valid values***
 - ▶ values set by the platform
 - ▶ values themselves are *meaningless to the application*
 - ▶ in general, you can not check whether a pointer is valid

■ Example

```
int * p;  
*p = 345; /* dereference on invalid pointer */
```

- Dereferencing an invalid pointer causes ***undefined behavior***
- In fact, *any* use of an invalid pointer value invokes undefined behavior

■ Use of invalid pointers

- ▶ uninitialized pointer value
- ▶ pointer to an object that no longer exists
- ▶ pointer incremented beyond properly allocated boundaries
- ▶ “uninitialized” \neq NULL

- Uninitialized pointer

- Uninitialized pointer

```
int * p;  
*p = 345; /* p was not initialized! */
```

- Uninitialized pointer

```
int * p;  
*p = 345; /* p was not initialized! */
```

- Pointer to an object that no longer exists

- Uninitialized pointer

```
int * p;  
*p = 345; /* p was not initialized! */
```

- Pointer to an object that no longer exists

```
int * new_intp(int i) {  
    int result = i;  
    return &result;  
}  
  
int main() {  
    int * p = new_intp(100);  
    *p = 345;    /* what is p pointing to?! */  
}
```

Uninitialized Pointers

computer memory

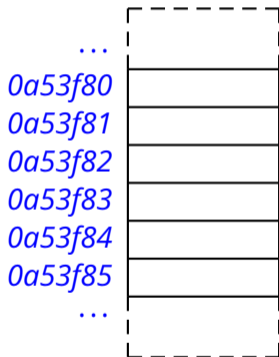
<i>address</i>	<i>value</i>
...	
0a53f80	
0a53f81	
0a53f82	
0a53f83	
0a53f84	
0a53f85	
...	

Uninitialized Pointers

computer memory

address

value



(int * p;)

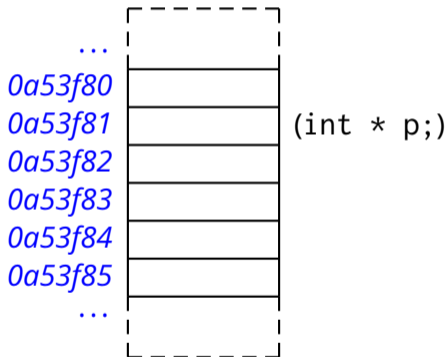
```
/* pointer to an int */  
int * p;
```

Uninitialized Pointers

computer memory

address

value



```
/* pointer to an int */
```

```
int * p;
```

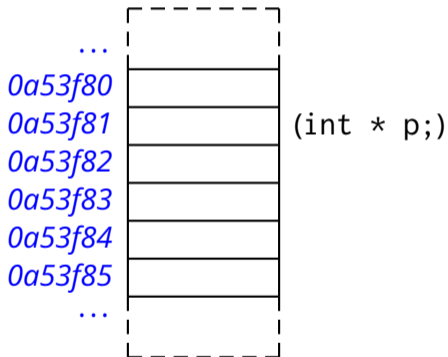
```
*p = 13;
```

Uninitialized Pointers

computer memory

address

value



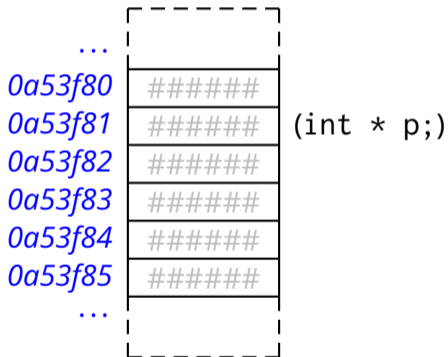
```
/* pointer to an int */  
int * p;  
  
*p = 13; /* Undefined behavior! */
```

Uninitialized Pointers

computer memory

address

value



```
/* pointer to an int */
```

```
int * p;
```

```
*p = 13; /* Undefined behavior! */
```

Pointer to a Dead Object

computer memory

<i>address</i>	<i>value</i>
...	
0a53f80	
0a53f81	
0a53f82	
0a53f83	
0a53f84	
0a53f85	
...	

Pointer to a Dead Object

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80	5	(int i;)
0a53f81		
0a53f82		
0a53f83		
0a53f84		
0a53f85		
...		

```
int i = 5;
```

Pointer to a Dead Object

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80	5	(int i;)
0a53f81		(int * p;)
0a53f82		
0a53f83		
0a53f84		
0a53f85		
...		

```
int i = 5;
```

```
int * p;
```

Pointer to a Dead Object

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80	5	(int i;)
0a53f81	0a53f80	(int * p;)
0a53f82		
0a53f83		
0a53f84		
0a53f85		
...		

```
int i = 5;  
int * p;  
p = &i;
```

Pointer to a Dead Object

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80	5	(int i;)
0a53f81	0a53f80	(int * p;)
0a53f82		
0a53f83		
0a53f84		
0a53f85		
...		

```
int i = 5;
int * p;
p = &i;

if (i > 0) {
```

Pointer to a Dead Object

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80	5	(int i;)
0a53f81	0a53f80	(int * p;)
0a53f82		(int j;)
0a53f83		
0a53f84		
0a53f85		
...		

```
int i = 5;
int * p;
p = &i;

if (i > 0) {
    int j;
```

Pointer to a Dead Object

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80	5	(int i;)
0a53f81	0a53f82	(int * p;)
0a53f82		(int j;)
0a53f83		
0a53f84		
0a53f85		
...		

```
int i = 5;
int * p;
p = &i;

if (i > 0) {
    int j;
    p = &j;
}
```

Pointer to a Dead Object

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80	5	(int i;)
0a53f81	0a53f82	(int * p;)
0a53f82		
0a53f83		
0a53f84		
0a53f85		
...		

```
int i = 5;
int * p;
p = &i;

if (i > 0) {
    int j;
    p = &j;
}
```

Pointer to a Dead Object

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80	5	(int i;)
0a53f81	0a53f82	(int * p;)
0a53f82		
0a53f83		
0a53f84		
0a53f85		
...		

```
int i = 5;
int * p;
p = &i;

if (i > 0) {
    int j;
    p = &j;
}
*p = 13;
```

Pointer to a Dead Object

computer memory

<i>address</i>	<i>value</i>	
...		
0a53f80	5	(int i;)
0a53f81	0a53f82	(int * p;)
0a53f82		
0a53f83		
0a53f84		
0a53f85		
...		

```
int i = 5;
int * p;
p = &i;

if (i > 0) {
    int j;
    p = &j;
}

*p = 13; /* Undefined behavior! */
```


- An array of type T is a sequence of consecutive objects of type T
 - ▶ supports random access by an *index* (starting at 0)

- An array of type T is a sequence of consecutive objects of type T
 - ▶ supports random access by an *index* (starting at 0)

Example:

```
int main() {
    int v[100];
    int i;
    for (i = 0; i < 100; ++i) {
        v[i] = getchar();
        if (v[i] == EOF) break;
    }
    while (i >= 0) {
        putchar(v[i]);
        --i;
    }
}
```

- A *string* in C is a *zero-terminated* array of chars

- A *string* in C is a *zero-terminated* array of chars

Example:

```
int main() {
    char s[100];
    int i;
    for (i = 0; i < 99; ++i) {
        s[i] = getchar();
        if (s[i] == EOF || s[i] == '\n') break;
    }
    s[i] = 0;

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

- A *string* in C is a *zero-terminated* array of chars

Example:

```
int main() {
    char s[100];
    int i;
    for (i = 0; i < 99; ++i) {
        s[i] = getchar();
        if (s[i] == EOF || s[i] == '\n') break;
    }
    s[i] = 0;

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

- A string is represented by a pointer to its first character

- We have already seen many string literals in this course.

- We have already seen many string literals in this course.

```
#include <stdio.h>

int main() {
    printf("Ciao!\n");
}
```

- We have already seen many string literals in this course.

```
#include <stdio.h>

int main() {
    printf("Ciao!\n");
}
```

- A slightly more explicit example

```
#include <stdio.h>

int main() {
    char * format;
    char * name;
    name = "James Bond";
    format = "My name is Bond--%s.\n";
    printf(format, name);
}
```

Strings in Memory

computer memory

address

value

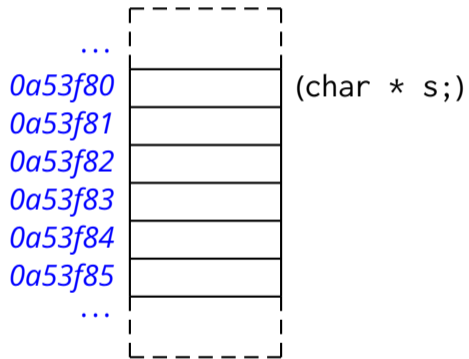
...	
0a53f80	
0a53f81	
0a53f82	
0a53f83	
0a53f84	
0a53f85	
...	

Strings in Memory

computer memory

address

value



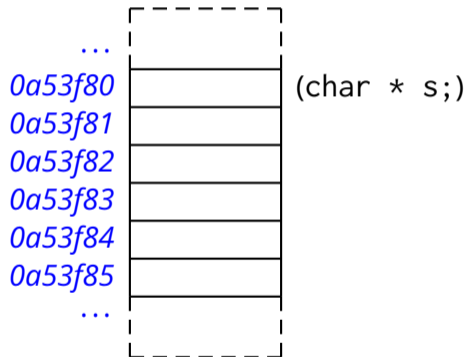
```
/* a pointer to char */  
char * s;
```

Strings in Memory

computer memory

address

value



```
/* a pointer to char */  
char * s;  
  
s = "bla";
```

Strings in Memory

computer memory

<i>address</i>	<i>value</i>
...	
0a53f80	
0a53f81	
0a53f82	'b'
0a53f83	'l'
0a53f84	'a'
0a53f85	0
...	

(char * s;)

```
/* a pointer to char */  
char * s;  
  
s = "bla";
```

Strings in Memory

computer memory

<i>address</i>	<i>value</i>
...	
0a53f80	0a53f82 (char * s;)
0a53f81	
0a53f82	'b'
0a53f83	'l'
0a53f84	'a'
0a53f85	0
...	

```
/* a pointer to char */  
char * s;  
  
s = "bla";
```

Strings in Memory

computer memory

<i>address</i>	<i>value</i>
...	
0a53f80	0a53f82 (char * s;)
0a53f81	
0a53f82	'b'
0a53f83	'l'
0a53f84	'a'
0a53f85	0
...	

```
/* a pointer to char */  
char * s;  
  
s = "bla";  
  
while (*s != 0) {  
    putchar(*s);  
    ++s;  
}
```

- Implement a string comparison function `stringequal`
- `stringequal` takes two strings as pointers to characters and returns true if and only if the two strings are equal

```
int stringequal(char * s1, char * s2);
```

- Input/Output of array of bytes and strings

A Bit More Standard I/O

- Input/Output of array of bytes and strings

(what's the difference?)

- Input/Output of array of bytes and strings (what's the difference?)
- Reading an array of bytes with `fgets`

```
#include <stdio.h>

int main() {
    char buffer[100];
    while(fgets(buffer, 100, stdin))
        printf("I just read this: %s\n", buffer);
}
```

- Input/Output of array of bytes and strings (what's the difference?)
- Reading an array of bytes with `fgets`

```
#include <stdio.h>

int main() {
    char buffer[100];
    while(fgets(buffer, 100, stdin))
        printf("I just read this: %s\n", buffer);
}
```

- `stdin` is the predefined input stream
- `fgets` produces a C string (i.e., terminated by 0)
- `fgets` reads the input up to EOF, end of line, or the given number of characters (e.g., 100), whichever comes first

- Reading an array of bytes with fread

```
#include <stdio.h>

int main() {
    char buffer[100];
    size_t size; /* size_t is an integer type */
    size = fread(buffer, 1, 100, stdin);
    if (size == 0) {
        fprintf(stderr, "Error or end of input\n");
    } else {
        fprintf(stdout, "%zu bytes read\n", size);
    }
}
```

A Bit More Standard I/O (2)

- Reading an array of bytes with fread

```
size = fread(buffer, 1, 100, stdin);
```

- fread reads end of line bytes as every other byte
- fread reads up to 100 elements of size 1 (byte)
- fread does not append a 0 (byte) at the end
- stdout and stderr are the predefined output and error streams

- Reading numbers and other elements with scanf

```
#include <stdio.h>
int main() {
    unsigned int x, y;
    char battlefield[20][20];
    /* ... */
    puts("coordinates? ");
    if (scanf("%ud%ud", &x, &y)==2 && x < 20 && y < 20) {
        switch(battlefield[x][y]) {
            case 'S': /* ship ... */
            case 'w': /* water ... */
                }
        } else {
            puts("bad input!\n");
        }
    }
}
```

- scanf reads a number of fields according to the given format
- scanf returns the number of successfully read fields

- Arrays are made of *contiguous elements in memory*
 - ▶ given the address of the first element, we can point to all other elements

- Arrays are made of *contiguous elements in memory*
 - ▶ given the address of the first element, we can point to all other elements

■ Example:

```
int main() {
    int v[100];
    int * p;
    for(p = &(v[0]); p != &(v[100]); ++p)
        if ((*p = getchar()) == EOF) {
            --p;
            break;
        }
    while (p != v)
        putchar(*--p);
}
```

- Another example

■ Another example

```
void printchar_string(const char * s) {
    for (;*s != '\0'; ++s)
        putchar(*s);
}

int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; ++i) {
        printchar_string("Argument: ");
        printchar_string(argv[i]);
        printchar_string("\n");
    }
    return 0;
}
```

Arrays and Pointers (3)

- The *name* of an array can be used (in an expression) as a pointer to the array
 - ▶ i.e., a pointer to the first element

Arrays and Pointers (3)

- The *name* of an array can be used (in an expression) as a pointer to the array
 - ▶ i.e., a pointer to the first element
- Given a declaration

```
int A[100];
```

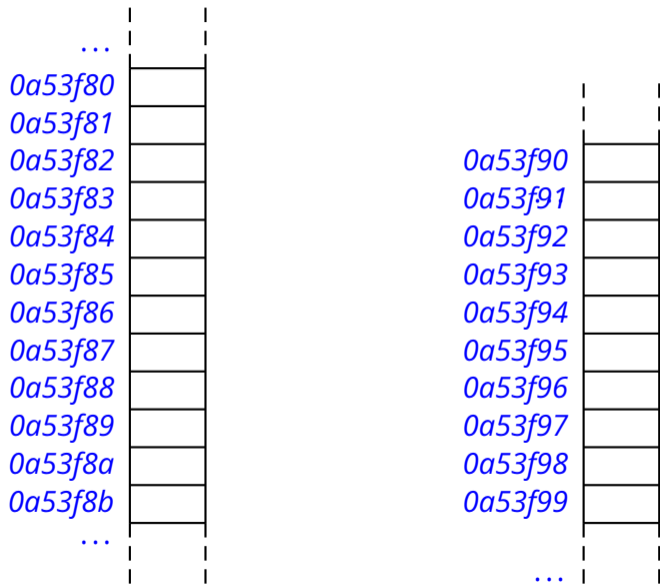
The following expressions are equivalent

```
int * p = A;
```

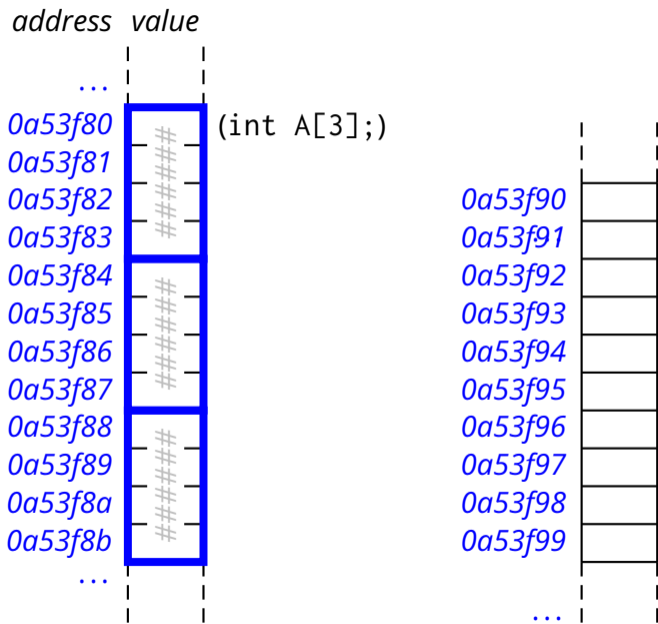
```
int * p = &(A[0]);
```

Pointer Arithmetic

address *value*

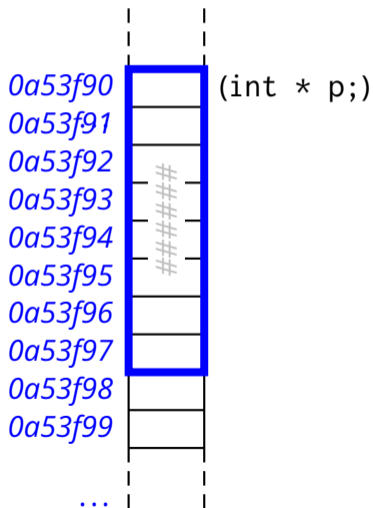
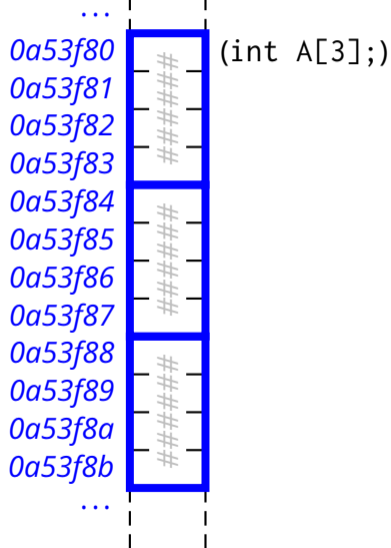


Pointer Arithmetic



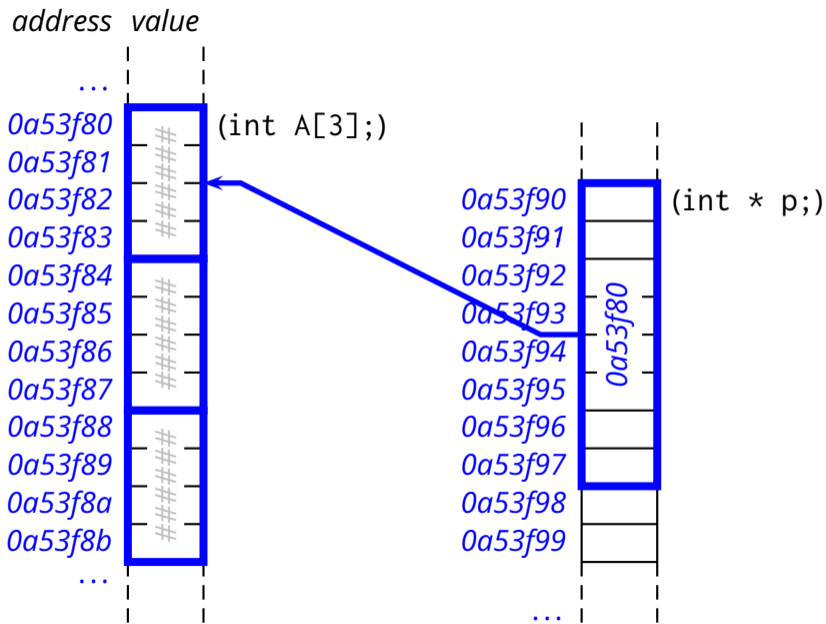
Pointer Arithmetic

address value



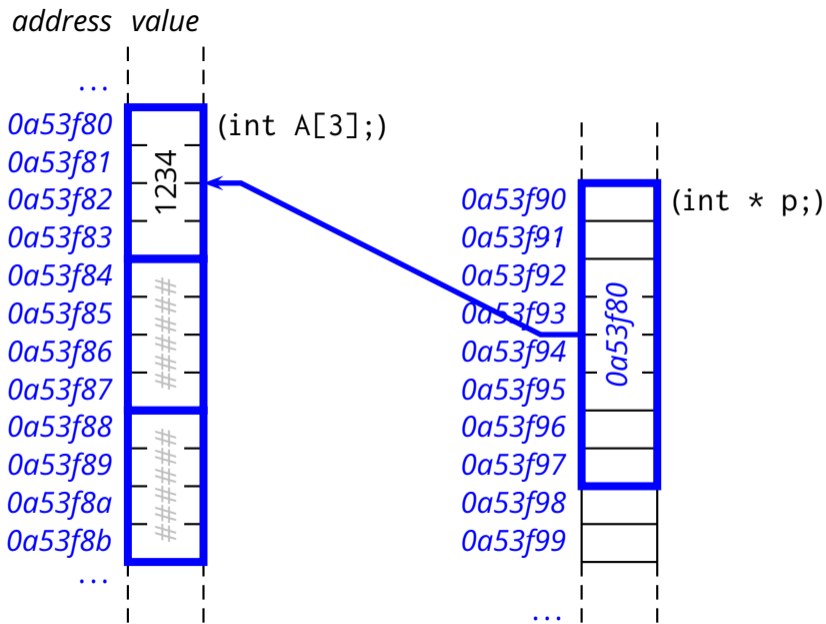
```
int A[3];
int * p;
```

Pointer Arithmetic



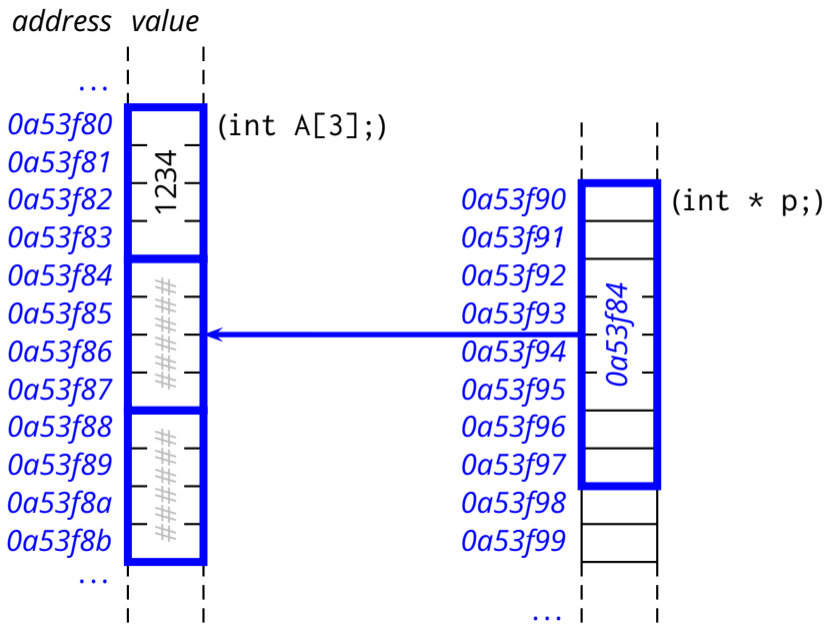
```
int A[3];  
int * p;  
p = A;
```

Pointer Arithmetic



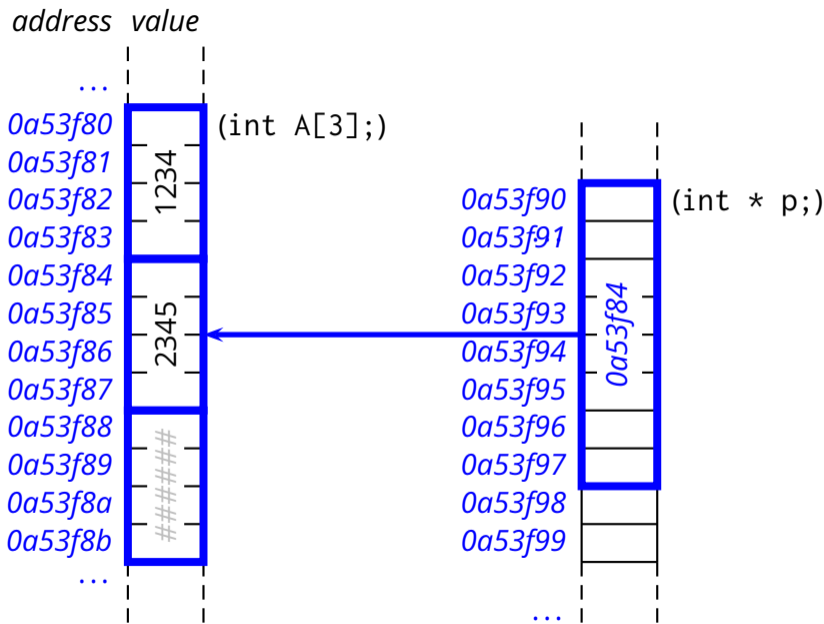
```
int A[3];  
int * p;  
p = A;  
*p = 1234;
```

Pointer Arithmetic



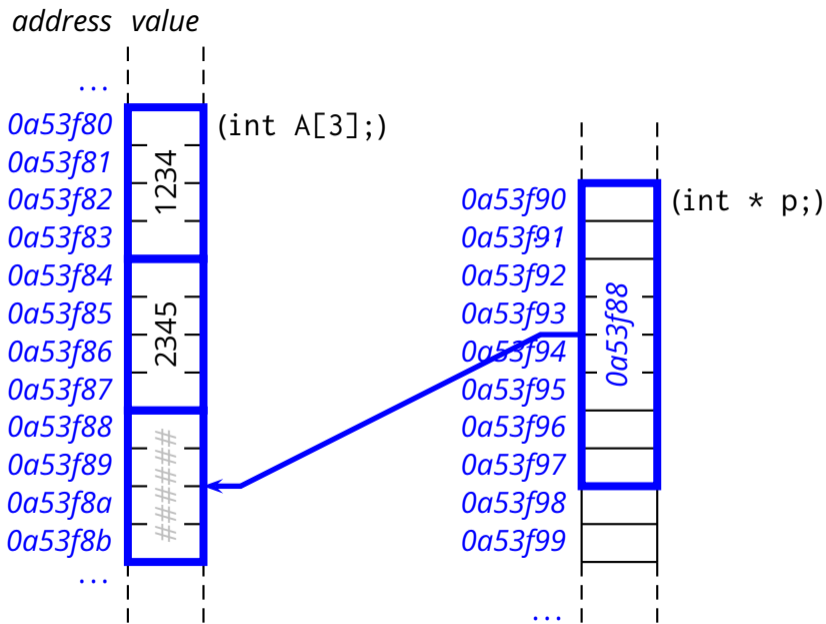
```
int A[3];  
int * p;  
p = A;  
*p = 1234;  
p = p + 1;
```

Pointer Arithmetic



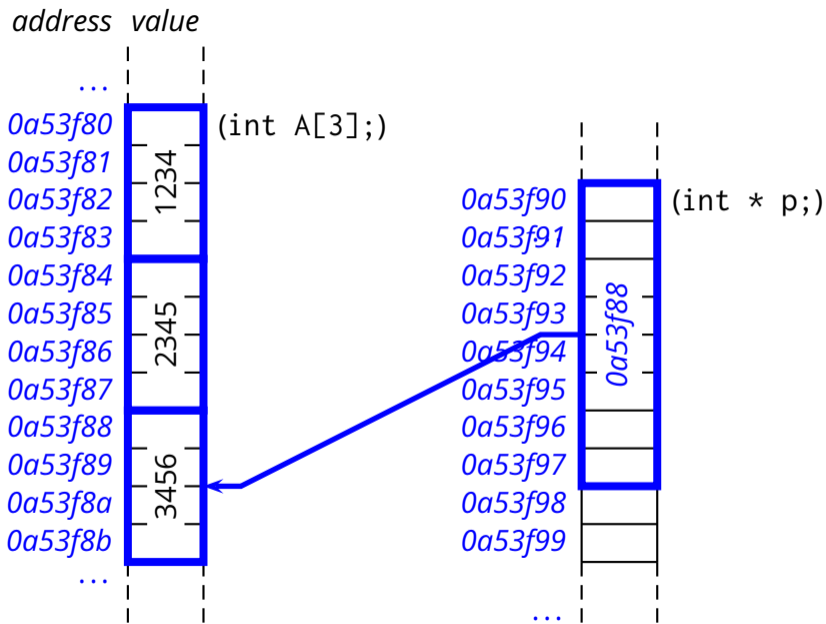
```
int A[3];
int * p;
p = A;
*p = 1234;
p = p + 1;
*p = 2345;
```

Pointer Arithmetic



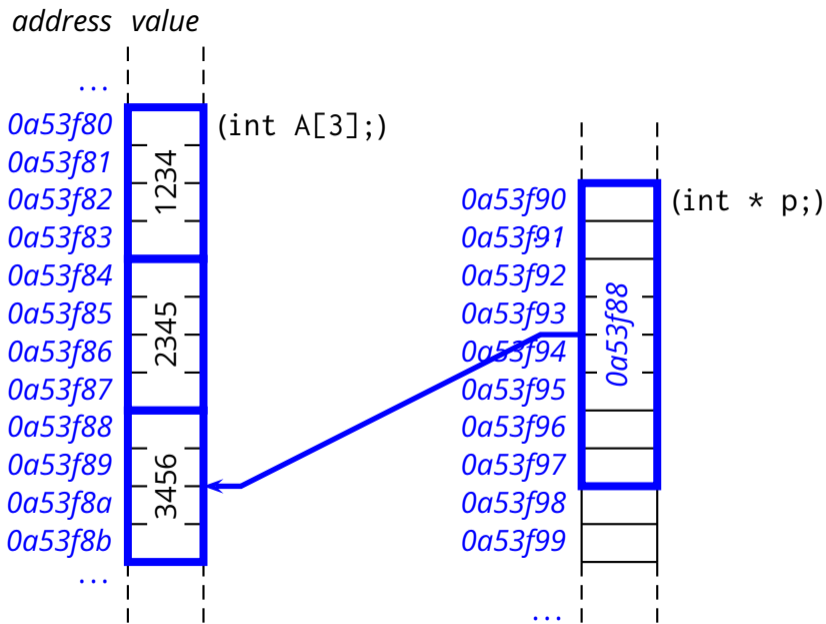
```
int A[3];
int * p;
p = A;
*p = 1234;
p = p + 1;
*p = 2345;
p = p + 1;
```

Pointer Arithmetic



```
int A[3];
int * p;
p = A;
*p = 1234;
p = p + 1;
*p = 2345;
p = p + 1;
*p = 3456;
```

Pointer Arithmetic



```
int A[3];
int * p;
p = A;
*p = 1234;
p = p + 1;
*p = 2345;
p = p + 1;
*p = 3456;
```

- Write a program that reads a text file from standard input in which lines are at most 1000 characters long, and outputs each line in reverse. A *line* is terminated by the newline character or by the end of file. Use the tests available on-line here:
https://www.inf.usi.ch/carzaniga/edu/sysprog_exercises/flipline_tests.tgz

- The `const` keyword means that the value can not be modified

- The `const` keyword means that the value can not be modified
 - ▶ which value?

```
void printchar_string(const char * s) {  
    while (*s != '\0') {  
        putchar(*s); /* no modifications here? */  
        ++s;         /* definitely a modification. */  
    }  
}
```

- The **const** keyword means that the value can not be modified
 - ▶ which value?

```
void printchar_string(const char * s) {  
    while (*s != '\0') {  
        putchar(*s); /* no modifications here? */  
        ++s;        /* definitely a modification. */  
    }  
}
```

What about this example?

```
void clear_string(const char * s) {  
    while (*s != '\0') {  
        *s = ' ';  
        ++s;  
    }  
}
```

- Anatomy of a function definition

■ Anatomy of a function definition

```
int http_request(const char * method,  
                int pcount,  
                const char *args[]) {  
    /* ... */  
}
```

■ Anatomy of a function definition

```
int http_request(const char * method,  
                int pcount,  
                const char *args[]) {  
    /* ... */  
}
```

- ▶ return type

■ Anatomy of a function definition

```
int http_request(const char * method,  
                int pcount,  
                const char *args[]) {  
    /* ... */  
}
```

- ▶ return type
- ▶ function name

■ Anatomy of a function definition

```
int http_request(const char * method,  
                int pcount,  
                const char *args[]) {  
    /* ... */  
}
```

- ▶ return type
- ▶ function name
- ▶ formal parameters

■ Anatomy of a function definition

```
int http_request(const char * method,  
                int pcount,  
                const char *args[]) {  
    /* ... */  
}
```

- ▶ return type
- ▶ function name
- ▶ formal parameters
- ▶ body

Declaring Functions

- A function must be *declared* before it is used

(like all symbols in C)

- A function must be *declared* before it is used

(like all symbols in C)

```
int http_request(const char * method,
                int pcount,
                const char *args[]); /* no body */

int main() {
    /* ... */
    http_request("GET", 0, NULL);
    /* ... */
}

int http_request(const char * method,
                int pcount,
                const char *args[]) {
    /* function definition is here! */
}
```

- Implement a function `twotimes` that reads a word of up to 1000 characters from the standard input and returns *true* if the given string consists of the concatenation of two identical substrings
- Test this function by writing a little program that reads a word of up to 1000 characters from the standard input and outputs "YES" or "NO" according to the result of the `twotimes` function applied to the input word

- What is the output of the following program?

```
void f(char * s) {
    char p;
    unsigned int c;
    while(*s != 0) {
        c = 1;
        p = *s;
        for(++s; *s == p; ++s) {
            ++c;
        }
        printf(" %d", c);
    }
    putchar('\n');
}

int main() {
    f("mamma, ciaaaaao!");
    /* ... */
}
```

The main Function

- The main function takes two parameters

- The main function takes two parameters

```
int main(int argc, char *argv[]) {  
    int i;  
    printf("You gave me %d parameters:\n", argc);  
    for (i = 0; i < argc; ++i)  
        printf("argv[%d] = %s\n", i, argv[i]);  
    return 0;  
}
```

- The main function takes two parameters

```
int main(int argc, char *argv[]) {  
    int i;  
    printf("You gave me %d parameters:\n", argc);  
    for (i = 0; i < argc; ++i)  
        printf("argv[%d] = %s\n", i, argv[i]);  
    return 0;  
}
```

- argv is an array of strings
- argc is the length of the array
- main returns an integer value
 - ▶ in general a 0 return value means "completed successfully"

- Write a program called `sortlines` that reads one line at a time from the standard input, and outputs the sequence of words in each line sorted in lexicographical order. A *word* is a (maximal) contiguous sequence of alphabetic characters as defined by the `isalpha` library function. The output sequence for each line should be printed on a single line with each word separated by one space. An input line is guaranteed to be at most 1000 characters.

Homework Assignment

- Implement a program that takes a string as a command-line parameter, reads the standard input, and returns 0 if the given string is found in the input stream.

Homework Assignment

- Implement a program that takes a string as a command-line parameter, reads the standard input, and returns 0 if the given string is found in the input stream.
- **More interesting variant:** Implement a program that takes one or more strings as a command-line parameters, reads the standard input, and returns 0 if all the given strings are found in the input stream.