Programming I

Pointers

Dimitrios Michail

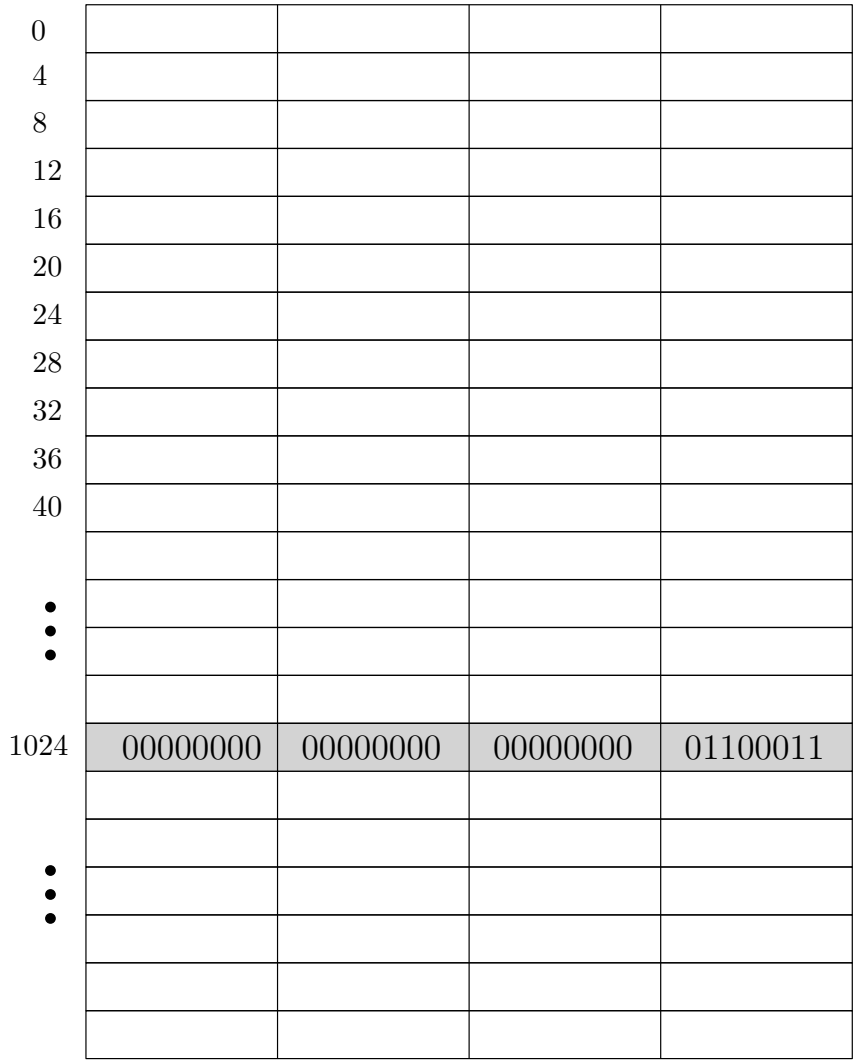
Dept. of Informatics and Telematics
Harokopio University of Athens

What is a pointer

A pointer is a variable which contain a memory address.

Recall that memory is a one-dimensional array.

In the example, the memory location 1024 contains the value 99.



Defining a Pointer in C

The "*" is used in order to define a pointer.

```
int *p;
```

The above definition create a variable of type pointer to integer.

Defining a Pointer in C

The "*" is used in order to define a pointer.

```
int *p;
```

The above definition create a variable of type pointer to integer.

The name of the variable is `p`.

Defining a Pointer in C

The "*" is used in order to define a pointer.

```
int *p;
```

The above definition create a variable of type pointer to integer.

The name of the variable is `p`.

The value of the variable is a memory address.

Setting the value of a pointer

Like all other local variables we must initialize a pointer.

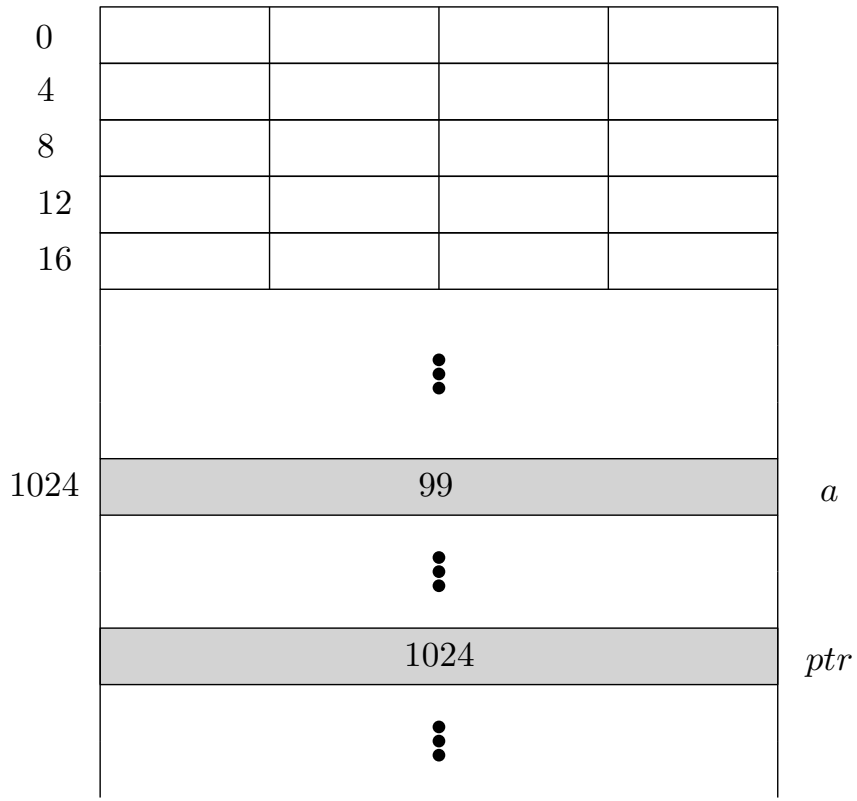
In order to find the memory address of a variable we can use the **address-of** operator `&`.

```
int main() {
    int a;
    int *ptr;

    a = 99;
    ptr = &a;

    return 0;
}
```

Setting the value of a pointer

In order to find the memory address of a variable we use the **address-of** operator `&`.

```
int main() {
    int a;
    int *ptr;

    a = 99;
    ptr = &a;

    return 0;
}
```



The pointer `ptr` **"points"** to variable `a`.

The value of variable `ptr` is the memory address of variable `a`.

Value 0 and Value NULL

In order for a pointer to not point somewhere we need to assign the value 0. The C library defines a constant named NULL in order to designate that a pointer does not point to anything.

```
int main() {
    int *ptr = 0;

    // ...

    return 0;
}
```

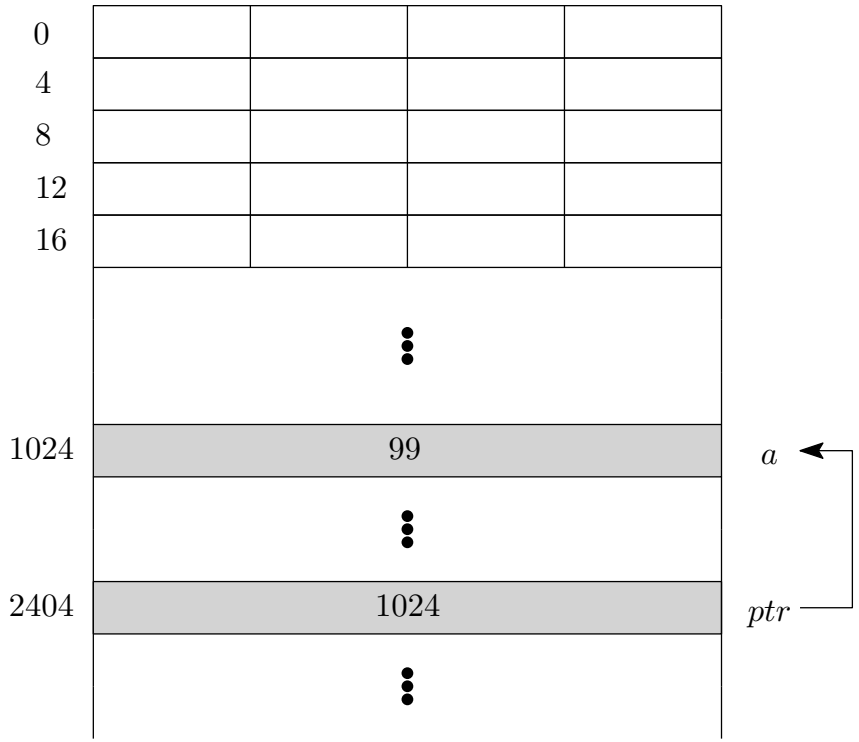
or

```
#include <stdio.h>

int main() {
    int *ptr = NULL;

    // ...

    return 0;
}
```

Indirection/dereferencing operator

Following a pointer in order to locate the variable that it points to can be done using the **indirection/dereferencing operator**, "*";

```
int main() {
    int a;
    int *ptr;

    a = 99;
    ptr = &a;

    printf("%d\n", *ptr);

    return 0;
}
```



The expression `*ptr` *returns the variable which is located at the memory location which is stored in* `ptr`.

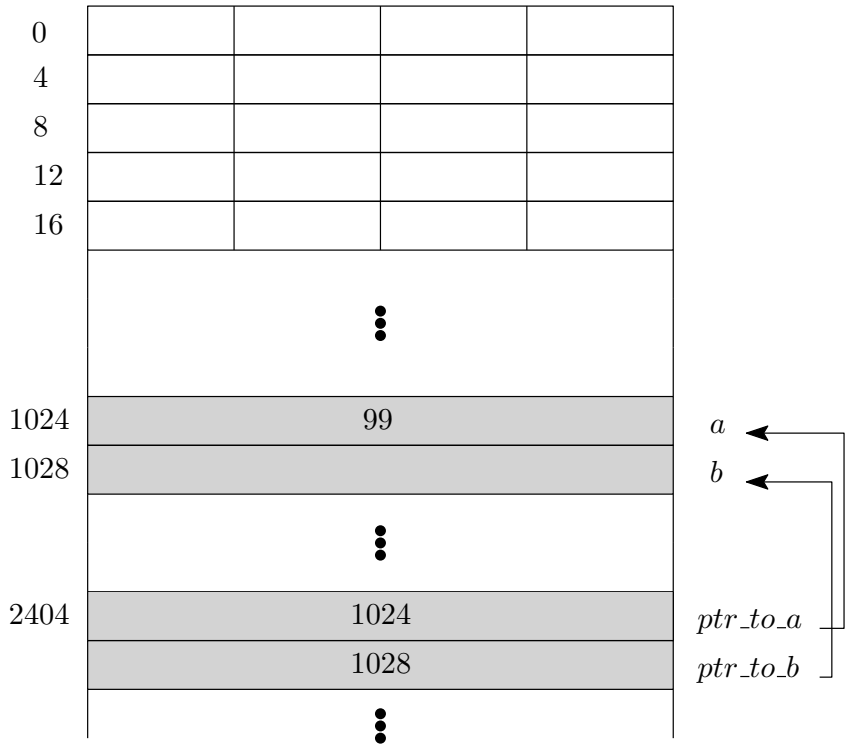
In this case variable a.

Operators Precedence

1. **parentheses:** (), `[ ]`, `expr++`, `expr--`
   Calculated first, from left to right. Nested are calculated first.
2. **unary operators:**
   +, −, `++expr`, `--expr`, `!`, `*`, `&` Calculated from right to left.
3. **multiplication, division and remainder:** $*$, /, or %
   Calculated from left to right.
4. **addition, subtraction:** + or −
   If there are many, calculated from left to right.
5. **relational:** $<$, $>$, $<=$, $>=$
   Calculated from left to right.
6. **equality:** $==$, $!=$
   Calculated from left to right.
7. **boolean AND:** `&&` Calculated from left to right.
8. **boolean OR:** `||` Calculated from left to right.
9. **assignment:** $=$, $+=$, $-=$, $*=$, $/=$, $\%=$
   From right to left.

Indirection/dereferencing operator

```
1  int main() {
2      int a, b;
3      int *ptr_to_a, *ptr_to_b;
4
5      a = 99;
6      ptr_to_a = &a;
7      ptr_to_b = &b;
8
9      *ptr_to_b = *ptr_t_a;
10     printf("%d\n", *ptr_to_b);
11
12     return 0;
13 }
```



Line 9 is practically:

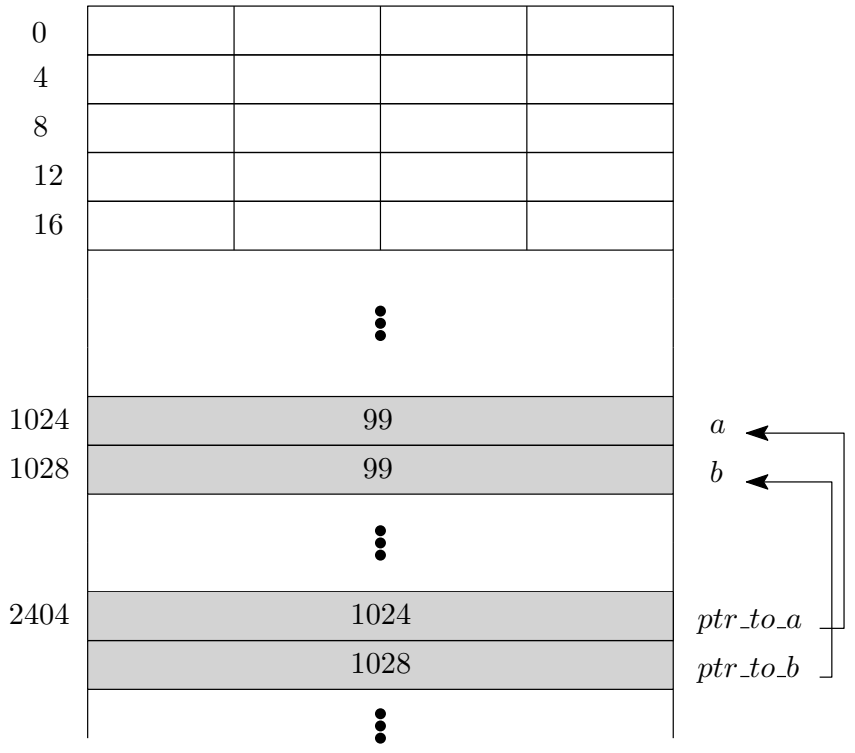
```
b = a;
```

while line 10 is:

```
printf("%d\n", b);
```

Indirection/dereferencing operator

```
1  int main() {
2      int a, b;
3      int *ptr_to_a, *ptr_to_b;
4
5      a = 99;
6      ptr_to_a = &a;
7      ptr_to_b = &b;
8
9      *ptr_to_b = *ptr_t_a;
10     printf("%d\n", *ptr_to_b);
11
12     return 0;
13 }
```



Line 9 is practically:

```
b = a;
```

while line 10 is:

```
printf("%d\n", b);
```

Pointers to other types

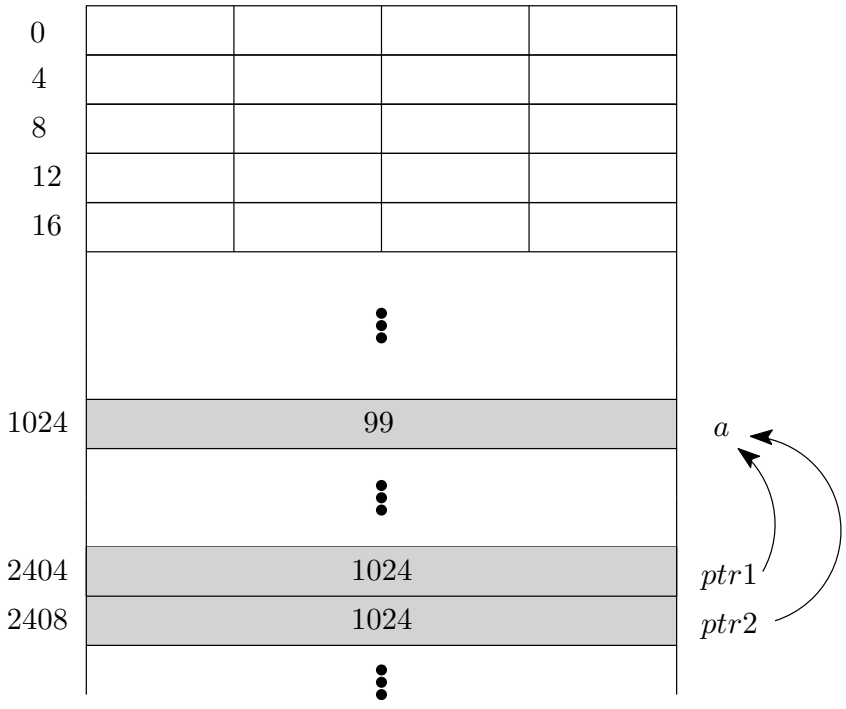
In C we may have pointers to any variable. We need, however, to correctly define it based on its type.

```c
#include <stdio.h>

int main() {
    double pi;
    double *ptr = &pi;

    *ptr = 3.14159265;
    printf("%lf\n", pi);

    return 0;
}
```

Similarly with the other types.

Multiple Pointers to a Single Variable

```
1   int main() {
2       int a;
3       int *ptr1, *ptr2;
4
5       a = 99;
6       ptr1 = &a;
7       ptr2 = &a;
8
9       (*ptr1)++;
10      (*ptr2)++;
11
12      printf("%d\n", a);
13      return 0;
14  }
```

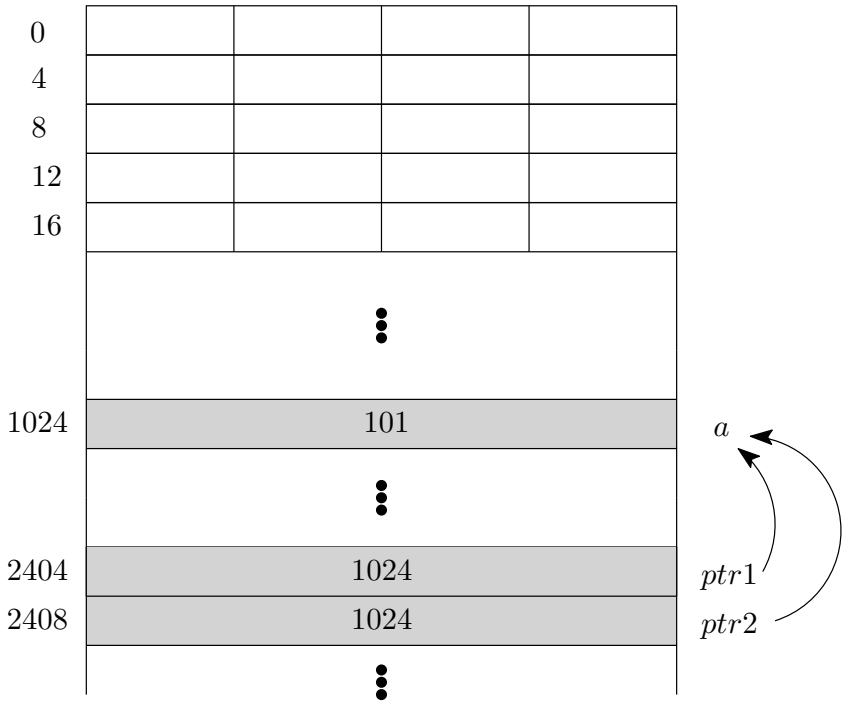


Lines 9 and 10 are practically line:

```
a++;
```

Multiple Pointers to a Single Variable

```
1   int main() {
2       int a;
3       int *ptr1, *ptr2;
4
5       a = 99;
6       ptr1 = &a;
7       ptr2 = &a;
8
9       (*ptr1)++;
10      (*ptr2)++;
11
12      printf("%d\n", a);
13      return 0;
14  }
```



Lines 9 and 10 are practically line:

a++;

The program prints 101.

Printing the Value of a Pointer

The value of a pointer is a memory address. In order to print the value of a pointer, function `printf()` provides the option `%p`.

```c
#include <stdio.h>

int main() {
    int a = 5;
    int *ptr = &a;

    printf("address of a is %p\n", ptr);

    return 0;
}
```

At the speaker's system this prints:

```
address of a is 0x7fff5bc2d5cc
```

Common Mistakes

Declaring Multiple Pointers

Care must be taken when defining multiple pointers in the same line.

```
int* ptr1, ptr2, ptr3;
```

The "*" operation belongs to the name and not the type.

The above line defines 1 pointer to int with name `ptr1` and two ints with names `ptr2` and `ptr3`.

In order to define 3 pointer to int we must write:

```
int *ptr1, *ptr2, *ptr3;
```

Common Mistakes

Operators Precedence

```
1   #include <stdio.h>
2
3   int main() {
4       int x, *p;
5
6       p = &x;          /* initialise pointer */
7       *p = 0;          /* set x to zero */
8       printf("x is %d\n", x);
9       printf("*p is %d\n", *p);
10
11      *p += 1;         /* increment what p points to */
12      printf("x is %d\n", x);
13
14      (*p)++; /* increment what p points to */
15      printf("x is %d\n", x);
16
17      return 0;
18  }
```

Notice that line 14 needs parentheses since the postfix ++ operator has larger priority than the unary operator *.

Simulating Call-By-Reference

Using pointers we can perform call-by-reference in C.

Let us make a small detour and explain the way function calling works.

Function Calls

Call Stack

In order to support functions, the compiler produces code which uses the "Call Stack".

The call stack is responsible for storing various kind of information such as:

- the return address of a function,
- the formal parameters of the function,
- the local variables of the function,
- etc.

Function Calls

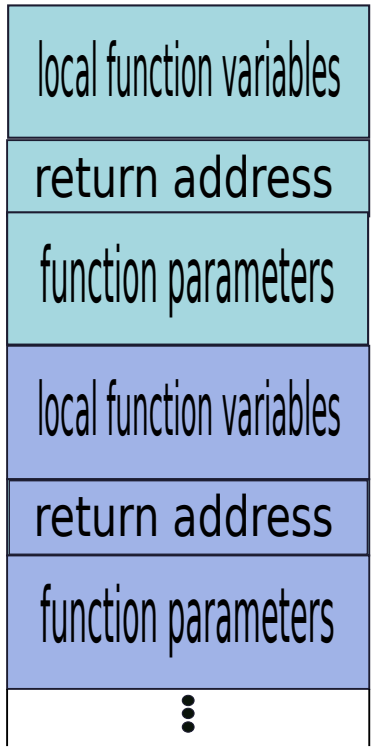
Stack Frame

Every time a function is called the compiler add one frame to the call stack which contains:

- the local variables of the function
- the return address of the function
- the formal parameters of the function
- etc

When the function execution ends the compiler removes the last frame from the stack, and thus destroyes all local variables and formal parameters.

The code for creating and destroying the stack frames is produced by the compiler and injected at the appropriate places of the program, before and after every function call.



Call-By-Reference

In order to simulate call-by-reference we use pointers.

- We give as parameter to a function a pointer to the variable that we wish to pass to the function.
- Inside the function, we use the pointer to read and/or write to the variable.
- When the function finishes, the formal parameter (our pointer) will be destroyed, but the value of our original variable will be updated.

Call-By-Reference

```c
#include <stdio.h>

void increase(int *x) {
    (*x)++;
}

int main() {
    int a = 1;

    increase(&a);
    printf("%d\n", a);

    return 0;
}
```

The above program prints 2.

- Variable `x` points to the variable which we would like to pass as a parameter.
- Inside the function we access our variable using the pointer by writing `*x`.
- After the end of the function, the local variable `x` is lost, but the changes in the value of a have been performed.

Call-By-Reference

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main() {
    int x = 1,
        y = 2;

    swap(&x, &y);

    printf("x = %d\n", x);
    printf("y = %d\n", y);

    return 0;
}
```

The program prints

```
x = 2
y = 1
```

Pointers and `const`

Recall that the reserved word `const` tells the compiler that something should not change.

But when we are dealing with pointer have 2 items:

1. a pointer, and
2. where the pointer points

The use of `const` is more complicated.

Pointers and `const`

There are 3 uses:

1. `const int *ptr;`

   Pointer to a constant integer. The pointer can change values, the integer cannot.

2. `int *const ptr;`

   Constant pointer to an integer. The pointer cannot change value, but the integer can.

3. `const int *const ptr;`

   Constant pointer to a constant integer. Neither the pointer nor the integer can change values.

It is easy to remember by looking left or right from the asterisk for the reserved word `const`.

Pointers and `const`

Example

```
1   #include <stdio.h>
2
3   int main() {
4       const int y = 5;
5       const int x = 3;
6
7       const int *ptr = &y;
8
9       (*ptr)++;     // NO!
10
11      ptr = &x; // ok
12  }
```

The compiler does not allow writing line 9.

Pointers and `const`

Example

```
1  #include <stdio.h>
2
3  int main() {
4      int y = 5;
5      int x = 3;
6
7      int *const ptr = &y;
8
9      (*ptr)++;  // ok
10
11     ptr = &x;  // NO!
12 }
```

The compiler does not allow writing line 11.

Pointers and `const`

Example

```
1   #include <stdio.h>
2
3   int main() {
4       const int y = 5;
5       const int x = 3;
6
7       const int *const ptr = &y;
8
9       (*ptr)++;  // NO!
10
11      ptr = &x;  // NO!
12  }
```

The compiler does not allow writing both lines 9 and 11.

Mistakes and Casts

Be careful with casting.

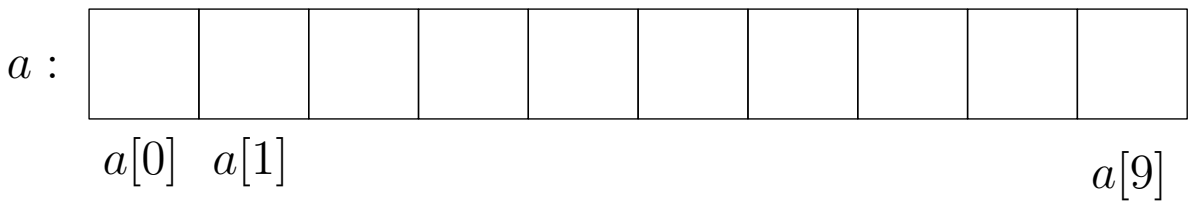
```c
#include <stdio.h>

int main() {
    int i;
    const int ci = 123;

    const int *cpi; // pointer to const
    int *ncpi;

    cpi = &ci;
    ncpi = &i;

    cpi = ncpi;      // allowed

    // this needs a cast - usually BIG MISTAKE
    ncpi = (int *)cpi;

    *ncpi = 0; // UNDEFINED BEHAVIOR !!
}
```

Pointers and Arrays

Arrays and pointer are very close connected in C.

In order to define an array with 10 integers named a we write:

```
int a[10];
```



Pointers and Arrays

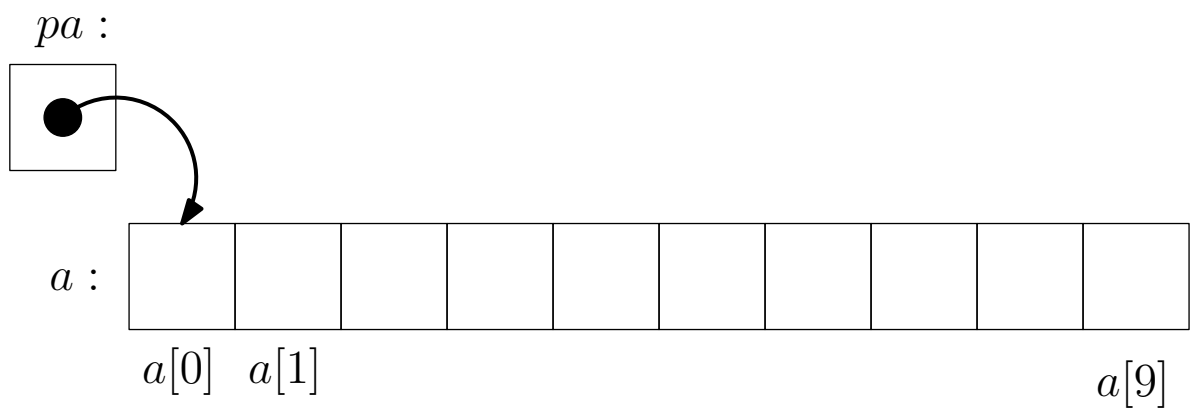
If pa is a pointer to an int

```
int *pa;
```

then the assignment

```
pa = &a[0];
```

sets pa to point to the 0-th element of array a.



Pointers and Arrays

From the definition, the value of a variable or an expression of array type is the address of the 0-th element of the array. Thus, after the assignment

```
pa = &a[0];
```

point `pa` and `a` have the exact same value.

Since the name of array is a synonym for the location of the 0-th array element, this assignment can also be written as

```
pa = a;
```

Pointers and Arrays

Since the name of array is a synonym for the location of the 0-th array element, we can also deference it.

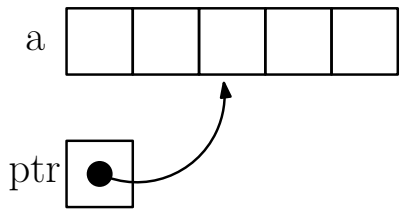
```c
#include <stdio.h>

int main() {
    int a[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};

    printf("%d\n", *a);
    *a = 11;
    printf("%d\n", a[0]);

    return 0;
}
```

The program above prints

```
9
11
```

Pointers and Arrays

A pointer can point at any element of an array.



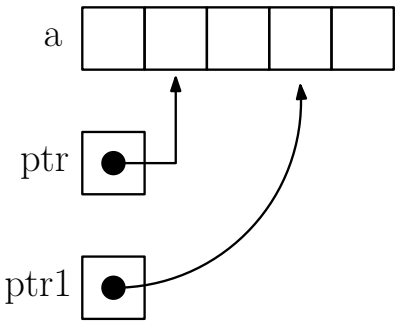

```c
#include <stdio.h>

int main() {
    int a[] = {9, 8, 7, 6, 5};
    int *ptr = &a[2];

    printf("%d\n", *ptr);

    return 0;
}
```

The above program prints

7

Pointers Arithmetic

Given a pointer to an array element, we can perform *pointer arithmetic*.




```c
#include <stdio.h>

int main() {
    int a[] = {9, 8, 7, 6, 5};

    int *ptr  = &a[1];
    int *ptr1 = ptr + 2;

    printf("%d\n", *ptr);
    printf("%d\n", *ptr1);

    return 0;
}
```

The above program prints

8
6

Pointers Arithmetic

When doing pointer arithmetic we can either add or subtract pointers.

When we write `p+1` where `p` is a pointer, the result is also a pointer which points one position after the one that `p` points.

How far `p+1` is from `p` depends on the type where `p` points.

If `p` is a pointer to int, then the memory location `p+1` will be 4 bytes after `p`. If, however, `p` points to char then it will only be 1 byte after.

Pointers and Arrays

When we write `a[i]` in C, it automatically gets translated to `*(a+i)`. The two forms are equivalent.

Using the `&` operator and the two forms we get that

- `&a[i]`
- `a+i`

are also equivalent. `a+i` is the address of the *i*-th element after *a*.

Similarly given pointer `pa = &a[0]`, the expression `pa[2]` is equivalent to `*(pa + 2)`. Thus, we can use the `[]` operator also with pointers.

Pointers and Arrays

Since `a[i]` in C gets translated to `*(a+i)`, the same happens with `i[a]`.

```
a[i]
*((a) + (i))     (definition)
*((i) + (a))     (commutative addition)
i[a]             (definition)
```

Pointers and Arrays

Since `a[i]` in C gets translated to `*(a+i)`, the same happens with `i[a]`.

```
a[i]
*((a) + (i))     (definition)
*((i) + (a))     (commutative addition)
i[a]             (definition)
```

This fact allows us to write "strange" code at various "strange" code competitions.

For example the expression

```
5["abcdef"]
```

is correct and translates to the character `'f'`.

Pointer Arithmetic

Copying Strings

```
void mystrcpy(char *dest, const char *src) {
    char *dp = &dest[0],
         *sp = &src[0];

    while(*sp != '\0')
        *dp++ = *sp++;

    *dp = '\0';
}
```

The expression `*dp++` first dereferences the pointer `dp` and then performs `dp=dp+1`. In order to increase the variable that `dp` points, it should have been written as `(*dp)++`.

Pointer Arithmetic

Copying Strings

When performing pointer arithmetic we need to be careful not to get out of bounds.

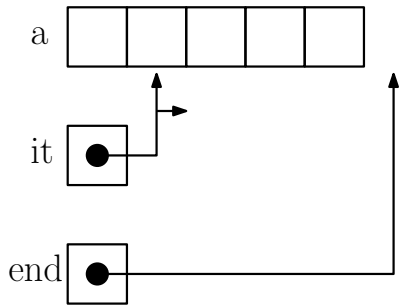
```c
#include <stdio.h>

int main() {
    int a[] = {1, 2, 3, 4, 5};
    int *p = &a[2];

    printf("%d\n", *(p + 4));

    return 0;
}
```

Otherwise, the result will be undefined.

Pointer Arithmetic

A small exception

When performing pointer arithmetic we can use for comparison (but never deference) the position that follows just after the end of an array.

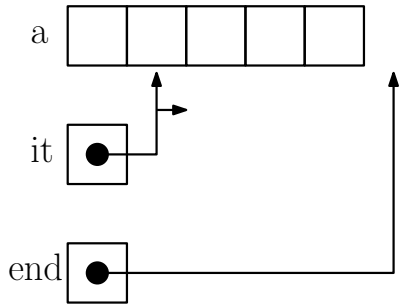



```c
#include <stdio.h>

int main() {
    int a[] = {1, 2, 3, 4, 5};

    int *it  = &a[0],
        *end = it + 5;

    while(it < end) {
        printf("%d\n", *it++);
    }

    return 0;
}
```

Pointer Arithmetic

A small exception

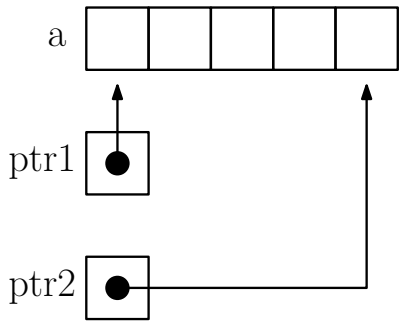
```c
#include <stdio.h>

int main() {
    int a[] = {1, 2, 3, 4, 5};

    int *it  = &a[0],
        *end = it + 5;

    while(it < end) {
        printf("%d\n", *it++);
    }

    return 0;
}
```



In the above code which prints all the elements of an array, pointer `end` points immediately after the end of the array. It is allowed to do the comparison `it < end`. The code is correct as long as we do not ever perform `*end`.

Pointer Arithmetic

We can also substract pointers.




```c
#include <stdio.h>

int main() {
    int a[] = { 1, 2, 3, 4, 5 };
    int *ptr1 = &a[0],
        *ptr2 = &a[4];

    printf("%d", ptr2 - ptr1);

    return 0;
}
```

What does the above code print?

Pointers and Arrays

There is a difference between the name of an array and a pointer which we must always remember.

A pointer is a variable and thus expressions like

```
pa = a;
```

or

```
pa++;
```

are perfectly valid.

The name of an array is not, however, a variable and thus the corresponding expressions

```
a = pa;
```

or

```
a++;
```

are not allowed.

Pointers, Arrays and Functions

When an array is a formal parameter to a function, only the location of the first array element is passed to the function. Inside the function, the formal parameter is a local variable of pointer type.

The following two declarations are equivalent

```
int strlen(char s[]);
```

and

```
int strlen(char *s);
```

so it is better to use the second way which clearly explains to the programmer that the parameter `s` is a pointer and not an array.

Pointers, Arrays and Functions

It is possible to pass only part of an array to a function by passing a pointer to the first element of the sub-array.

For example if `a` is an array, then

```
f(&a[2])
```

and

```
f(a+2)
```

pass to the function `f` the address of the sub-array which starts at `a[2]`.

Pointers, Arrays and Functions

The functions can also be written as

```
f(int arr[]) { ... }
```

or

```
f(int *arr) { ... }
```

The function `f` does not need to be changed at all.

If someone is certain that the elements exist, can also use negative array indices such as `arr[-1]` or `arr[-2]`. The result will be undefined if these array elements do not exist.

Array Pointers

The same way we use array of various types, we can also define arrays of pointers. The syntax is more complicated.

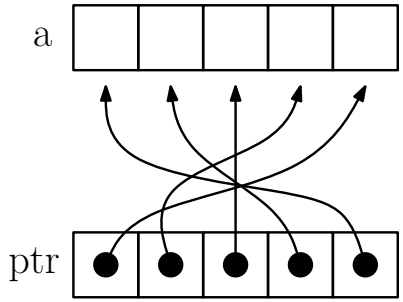
The following statement defines an array with 10 pointers to integers.

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

Array Pointers

Example

```c
#include <stdio.h>

int main() {
    int i;
    int a[5] = {0, 1, 2, 3, 4};
    int *ptr[5];

    for(i = 0; i < 5; i++)
        ptr[i] = &a[4-i];

    for(i = 0; i < 5; i++)
        printf("%2d", *ptr[i]);

    return 0;
}
```



The code prints

```
4 3 2 1 0
```

Function Pointers

The C language allows us to use pointers to functions. This is allowed since functions are also loaded in memory.

In order to declare a pointer to a function, the type of the pointer must describe the types of the parameters and the type of the returned result of the function.

```
int (*ptr)(int, double);
```

The above statement defines a pointer to a function. The name of the pointer is `ptr`. The pointer can point to functions than accepts two parameters, one `int` and one `double`, and returns an `int`.

Function Pointers

In the following code we define a pointer named `ptr` which can point to functions that accept an `int` parameter and return `void`.

```c
#include <stdio.h>

void print(int x) {
    printf("%d\n", x);
}

int main() {
    void (*ptr)(int);

    ptr = &print;

    return 0;
}
```

- In order to assign a value to the pointer, we use the address-of operator `&`. For example `ptr = &print`.
- The C language allows us to also write `ptr = print` with the same result.

Function Pointers

```c
#include <stdio.h>

void print(int x) {
    printf("%d\n", x);
}

int main() {
    void (*ptr)(int);

    ptr = &print;
    (*ptr)(5);

    return 0;
}
```

- In order to use a function pointer, we dereference in order to get the function and then call the function, e.g. `(*ptr)(5)`.
- We cal also directly write `ptr(5)` with the same result.

Function Pointers

```c
#include <stdio.h>

void foo(int x) {
    printf("foo %d\n", x);
}

void bar(int y) {
    printf("bar %d\n", y);
}

int main() {
    void (*ptr)(int);

    ptr = &foo;
    (*ptr)(5);

    ptr = &bar;
    (*ptr)(5);

    return 0;
}
```

This code call 2 different functions `foo()` and `bar()` using a function pointer.

Function Pointers as Parameters to Other Fucntions

```c
#include <stdio.h>

void print(int x, void (*ptr)(int)) {
    (*ptr)(x);
}

void foo(int x) {
    printf("foo %d\n", x);
}

void bar(int y) {
    printf("bar %d\n", y);
}

int main() {
    print(5, &foo);
    print(5, &bar);

    return 0;
}
```

We can pass a function pointer as parameter to another function.

.

The code first calls `foo()` and then `bar()`.

Returning a Function Pointer

The syntax is somewhat complicated, but we can write the following:

```
float (*getFunction(const char code))(float, float)
{
    // code here
}
```

which defines a function named `getFunction`. The function accepts one paramater of type `const char` and returns a function pointer which can point to a function with 2 `float` parameters which returns `float`.

Returning a Function Pointer

Example

```c
#include <stdio.h>

float minus(float f1, float f2) { return f1-f2; }

float plus(float f1, float f2) { return f1+f2; }

float (*getFunction(const char code))(float, float) {
    if (code == '+')
        return &plus;
    else if (code == '-')
        return &minus;
    return NULL;
}

int main() {
    float f;
    f = (*getFunction('+'))(2.0, 1.0);
    f = (*getFunction('-'))(f, 1.0);
    printf("%f\n", f);
    return 0;
}
```

Arrays of Function Pointers

We can also have arrays of function pointers. Again the syntax is somewhat complicated.

In the statement below we define an array with 10 elements called `funcArr` where each element is a pointer to function that returns `int` and accepts 3 parameters with types `float`, `char` and `char`.

```
int (*funcArr[10])(float, char, char);
```

Arrays of Function Pointers

```c
#include <stdio.h>

void foo1(int x) { printf("foo1 %d\n", x); }
void foo2(int x) { printf("foo2 %d\n", x); }
void foo3(int x) { printf("foo3 %d\n", x); }
void foo4(int x) { printf("foo4 %d\n", x); }

int main() {
    int i, j;
    void (*farray[4])(int) = {NULL};

    farray[0] = &foo1;
    farray[1] = &foo2;
    farray[2] = &foo3;
    farray[3] = &foo4;

    for(i = 0; i < 100; i++)
        for(j = 0; j < 4; j++)
            (*farray[j])(i);

    return 0;
}
```

Pointers to `void`

In the program that follows

```c
#include <stdio.h>

int main() {
    int c;
    double *ptr;

    ptr = &c;

    return 0;
}
```

the compiler warns us

```
test.c: In function ‘main’:
test.c:8: warning: assignment from incompatible pointer type
```

Pointer to `void`

The C language provides a pointer to `void` which can point anywhere.

```c
#include <stdio.h>

int main() {
    int c;
    double k;
    void *ptr;

    ptr = &c;
    ptr = &k;

    return 0;
}
```

We will see more example in next lectures. The downside is that we need to perform a lot of casts.

Pointers to Pointers

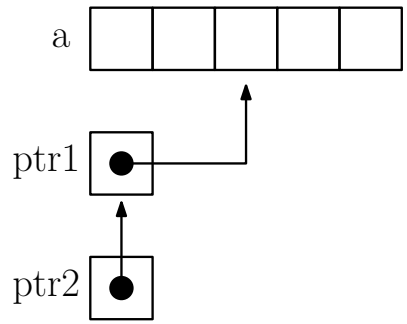
In order to define a pointer to another pointer we need to follow the following syntax.

```
int **x;
```

It might be clearer if we wrote `int* *x` in order to understand that `x` is a pointer which points to a pointer to int.

Pointers to Pointers

```c
#include <stdio.h>

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    int *ptr1  = &a[2];
    int **ptr2 = &ptr1;

    printf("%d\n", **ptr2);

    return 0;
}
```



The code prints

```
3
```