

C Pointers

An Advanced Introduction to Unix/C Programming



Dennis
Ritchie



Ken
Thompson



Linus
Torvalds



Richard
Stallman



Brian
Kernighan

John Dempsey

COMP-232 Programming Languages
California State University, Channel Islands

Pointer Advantages

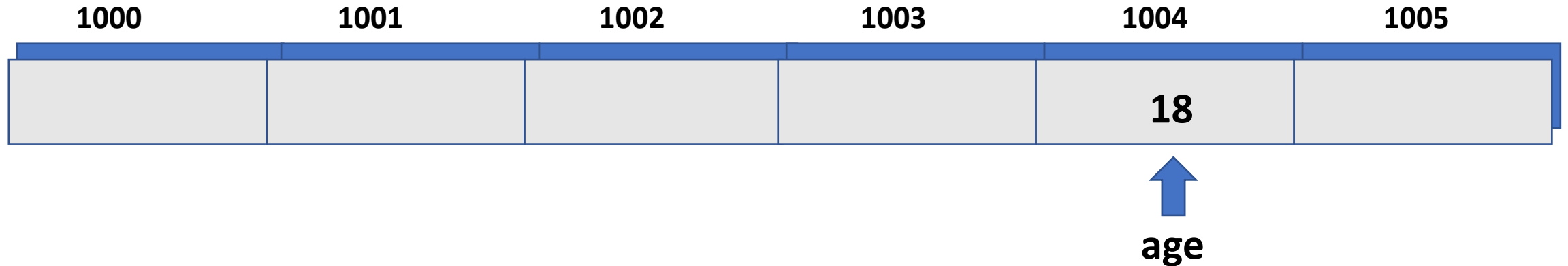
- Pointers are important!
- Pointers can reduce the size of and increase the execution speed of a program.
- Pointers allow you to allocate/deallocate memory while the program is running.
- Pointers allow you to save memory by passing only the address of an object instead of copying all of the data contained in the object.
- There is a closer association between pointers and the underlying hardware. In many engineering applications, low-level hardware interactions should be as close as possible.

Pointer Disadvantages

- Pointers add complexity to the code.
- Segmentation violations will occur if pointers are not initialized.
- Using pointers set to the wrong location can cause crazy, hard to debug problems and corrupt memory.
- Memory leaks may occur if memory is constantly being allocated, but never freed. If left unchecked, you'll run out of memory and program will crash.
- It's your responsibility to set pointers carefully and manage allocated memory.

Pointers

```
int    age = 18;
```



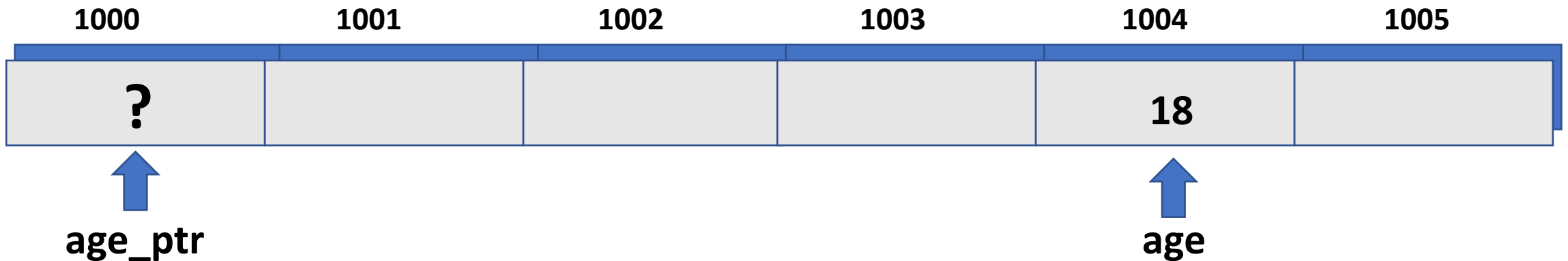
- When you declare a variable, like “int age;”, the compiler allocates memory for the variable with a unique address to store the variable. You don’t know, nor need to know, the variable’s actual address.
- The compiler associates the memory address with the variable’s name.
- When the variable is used, the program accesses the memory location in order to read or write the variable’s value.

Pointers

```
int age = 18;
```

```
int *age_ptr;
```

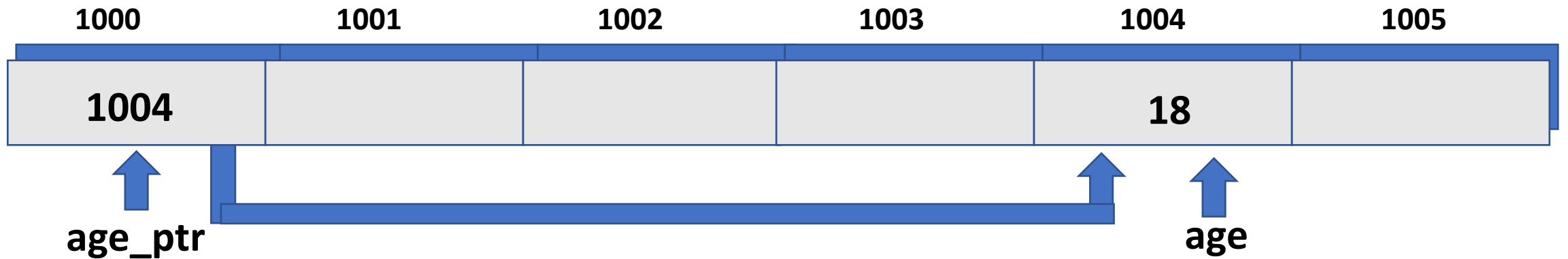
← The * indicates age_ptr will hold a pointer to an integer variable.



- An address is just a number and can be treated like any other number.
- To create a pointer, you need to declare a second variable to hold the address of the first variable.
- Declaring age_ptr above allocates space to hold an address to any variable defined as an integer. age_ptr has not yet been initialized, is not defined to hold an integer data type, but is defined to point to an integer type.

Pointers

```
int    age = 18;  
int    *age_ptr;
```



- To initialize `age_ptr`, we set `age_ptr` to the memory address of where `age` is stored, which in this case is the address of 1004, by using:

```
age_ptr = &age;
```

- `age_ptr` now points to `age` or is a pointer to `age`, because `age_ptr` holds the address of where `age` is stored in memory.

Pointers

- Pointers can point to different data types, like int, float, and typedef struct, using the following format:

```
data_type *name_of_ptr;
```

e.g.

```
int *age_ptr;
```

- The * is the indirection operator.
- The * indicates the variable name_of_ptr is a pointer to a data_type variable, e.g., age_ptr can be a pointer to any variable defined as an int.
- But name_of_ptr is not a variable of type data_type.

Pointer Declaration Examples

```
int    trip, *trip_ptr;
```

← Pointer to an integer value.

```
double percent, *percent_ptr;
```

← Pointer to a double value.

```
typedef struct person_struct {  
    char    first_name[15];  
    char    last_name[25];  
    float   age;  
} PERSON;
```

```
PERSON    person, *person_ptr;
```

← Pointer to a PERSON struct.

Pointers – You Must Initialize Pointers!

- Pointers are not initialized. To initialize a pointer to point to a variable, you can use:

```
pointer = &variable;
```

e.g.,

```
trip_ptr      = &trip;  
percent_ptr   = &percent;  
person_ptr    = &person;
```

- The & copies the memory address where variable is stored into the pointer variable.
- Note how the & looks like the letter A and ampersand starts with the letter A, which I like to think represents the Address Of.

Pointers – How To Use

- Once a pointer has been defined and initialized, you can use them.

To print out the value of age, you can use:

`printf("age = %d\n", age);` ← Direct Access

or

`printf("age = %d\n", *age_ptr);` ← Indirect Access (or Indirection)

The address of age can be printed using:

`printf("age_ptr = %p or &age = %p\n", age_ptr, &age);`

Pointers – Program to Print Value & Address

```
john@oho:~$ cat pointer.c
#include <stdio.h>
int main()
{
    int    age = 18;
    int    *age_ptr;

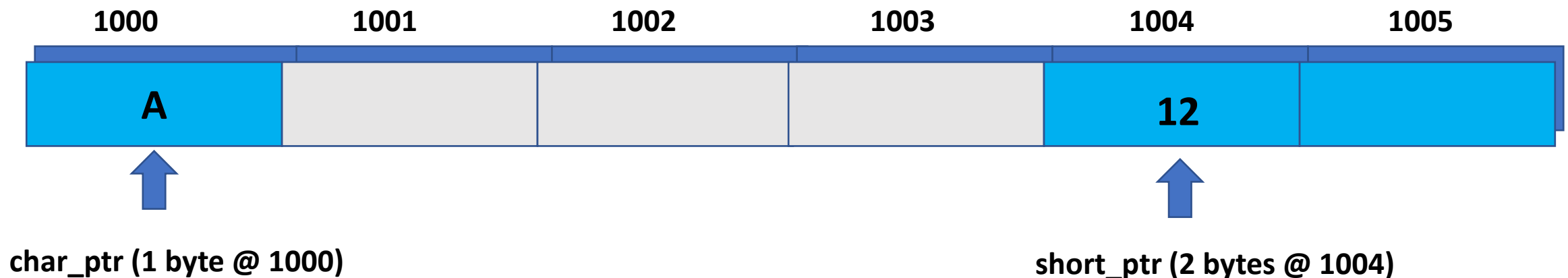
    age_ptr = &age;
    printf("age = %d or *age = %d\n", age, *age_ptr);           // Direct Access
    printf("The address of age_ptr = %p or &age = %p\n", age_ptr, &age); // Indirect Access
}
```

```
john@oho:~$ gcc pointer.c; a.out
age = 18 or *age = 18
The address of age_ptr = 0x7fffd5dc914c or &age = 0x7fffd5dc914c
john@oho:~$ a.out
age = 18 or *age = 18
The address of age_ptr = 0x7fffcdae50c or &age = 0x7fffcdae50c    ← Note: The address of age can change each time.
john@oho:~$ a.out
age = 18 or *age = 18
The address of age_ptr = 0x7ffdcf7ef4c or &age = 0x7ffdcf7ef4c
```

Pointers

Different data types define data of different sizes, e.g., a char is 1 byte, a short int is 2 bytes, an int is 4 bytes, and a struct is a fixed size.

But pointers are smart! A pointer will (1) point to the first byte of the data and (2) the compiler remembers the type of data being pointed to, and as such, knows its size.



Pointers

When using arrays, you're actually using pointers.

To initialize `numbers_ptr` to the first element of the `numbers` array, you can:

```
int    numbers[10], *numbers_ptr;
```

```
numbers_ptr = numbers;
```

or

```
number_ptr = &numbers[0];
```

Pointers – Memory Addresses For Different Types

```
john@oho:~$ cat pointer2.c
#include <stdio.h>
int main()
{
    typedef struct person_struct {
        char first_name[10];
        char last_name[20];
        int age;
    } PERSON;

    int i;
    int integer_array[10];
    PERSON person_array[10];

    for (i=0; i<10; i++) {
        printf("Address of integer_array[%d]=%p,
person_array[%d]=%p\n",
            i, &integer_array[i], i, &person_array[i]);
    }
}
```

```
john@oho:~$ gcc pointer2.c; a.out
```

```
Address of integer_array[0]=0x7fffccf41c40, person_array[0]=0x7fffccf41c70
Address of integer_array[1]=0x7fffccf41c44, person_array[1]=0x7fffccf41c94
Address of integer_array[2]=0x7fffccf41c48, person_array[2]=0x7fffccf41cb8
Address of integer_array[3]=0x7fffccf41c4c, person_array[3]=0x7fffccf41cdc
Address of integer_array[4]=0x7fffccf41c50, person_array[4]=0x7fffccf41d00
Address of integer_array[5]=0x7fffccf41c54, person_array[5]=0x7fffccf41d24
Address of integer_array[6]=0x7fffccf41c58, person_array[6]=0x7fffccf41d48
Address of integer_array[7]=0x7fffccf41c5c, person_array[7]=0x7fffccf41d6c
Address of integer_array[8]=0x7fffccf41c60, person_array[8]=0x7fffccf41d90
Address of integer_array[9]=0x7fffccf41c64, person_array[9]=0x7fffccf41db4
```

The integer_array increments by 4 bytes because an int is 4 bytes.

The PERSON array is 36 bytes, but increments by 24 bytes. Why?

The PERSON struct is only 22 hex bytes in size? Why does it increment by 24 bytes?

Pointers – Addresses By Incrementing Pointers

```
#include <stdio.h>
int main()
{
    typedef struct person_struct {
        char  first_name[10];
        char  last_name[20];
        int   age;
    } PERSON;

    int  i;
    int  integer_array[10];
    int  *integer_array_ptr;
    PERSON person_array[10];
    PERSON *person_array_ptr;

    integer_array_ptr = &integer_array[0];
    person_array_ptr = person_array;

    for (i=0; i<10; i++) {
        printf("Address of integer_array[%d]=%p,
person_array[%d]=%p\n",
            i, integer_array_ptr, i, person_array_ptr);
        integer_array_ptr++;
        person_array_ptr++;
    }
}
```

```
john@oho:~$ gcc pointer3.c; a.out
```

```
Address of integer_array[0]=0x7ffef2b7c10, person_array[0]=0x7ffef2b7c40
Address of integer_array[1]=0x7ffef2b7c14, person_array[1]=0x7ffef2b7c64
Address of integer_array[2]=0x7ffef2b7c18, person_array[2]=0x7ffef2b7c88
Address of integer_array[3]=0x7ffef2b7c1c, person_array[3]=0x7ffef2b7cac
Address of integer_array[4]=0x7ffef2b7c20, person_array[4]=0x7ffef2b7cd0
Address of integer_array[5]=0x7ffef2b7c24, person_array[5]=0x7ffef2b7cf4
Address of integer_array[6]=0x7ffef2b7c28, person_array[6]=0x7ffef2b7d18
Address of integer_array[7]=0x7ffef2b7c2c, person_array[7]=0x7ffef2b7d3c
Address of integer_array[8]=0x7ffef2b7c30, person_array[8]=0x7ffef2b7d60
Address of integer_array[9]=0x7ffef2b7c34, person_array[9]=0x7ffef2b7d84
```

integer_array_ptr increments by 4 bytes.

person_array_ptr increments by 24 bytes.

Pointers

- An array name without brackets points to the array's first value.
- As such ...

`*(array) == array[0]`

← Both point to the value held in array[0]

`*(array+1) == array[1]`

← Both point to the value held in array[1]

`*(array+2) == array[2]`

← Both point to the value held in array[2]

...

`*(array+n) == array[n]`

← Both point to the value held in array[n]

Pointers – Function Calls

- There are two ways to pass arguments to a function:
 1. By Value
 2. By Reference
- To pass an array, you can simply provide the name of the array to the function, which is call by reference.
- Strings are passed from one function to another using the address of the first character, not as the whole array.

Pointers – Passing An Array To Function

```
#include <stdio.h>
#include <string.h>

typedef struct person_struct {
    char first_name[10];
    char last_name[20];
    int age;
} PERSON;

void print_integer_array(int my_int_array[], int count)
{
    int i;
    for (i=0; i<count; i++)
        printf("print_integer_array: my_int_array[%d] = %d\n",
            i, my_int_array[i]);
}

void print_person_array(PERSON my_person_array[], int count)
{
    int i;
    for (i=0; i<count; i++)
        printf("print_person_array: %d. first_name=%s, last_name=%s, age=%d\n",
            i,
            my_person_array[i].first_name,
            my_person_array[i].last_name,
            my_person_array[i].age);
}
```

```
int main()
{
    int i;
    int integer_array[10];
    PERSON person_array[10];
    PERSON *person_array_ptr;

    person_array_ptr = person_array;

    for (i=0; i<10; i++) { // Initialize integer and person arrays.
        integer_array[i] = i;
        sprintf(person_array_ptr->first_name, "John%d", i);
        sprintf(person_array_ptr->last_name, "Smith%d", i);
        person_array_ptr->age = i+10;
        *person_array_ptr++;
    }

    print_integer_array(integer_array, i);

    print_person_array(person_array, i);
}
```

Pointers

```
john@oho:~$ gcc pointer_function.c; a.out
print_integer_array: my_int_array[0] = 0
print_integer_array: my_int_array[1] = 1
print_integer_array: my_int_array[2] = 2
print_integer_array: my_int_array[3] = 3
print_integer_array: my_int_array[4] = 4
print_integer_array: my_int_array[5] = 5
print_integer_array: my_int_array[6] = 6
print_integer_array: my_int_array[7] = 7
print_integer_array: my_int_array[8] = 8
print_integer_array: my_int_array[9] = 9
print_person_array: 0. first_name=John0, last_name=Smith0, age=10
print_person_array: 1. first_name=John1, last_name=Smith1, age=11
print_person_array: 2. first_name=John2, last_name=Smith2, age=12
print_person_array: 3. first_name=John3, last_name=Smith3, age=13
print_person_array: 4. first_name=John4, last_name=Smith4, age=14
print_person_array: 5. first_name=John5, last_name=Smith5, age=15
print_person_array: 6. first_name=John6, last_name=Smith6, age=16
print_person_array: 7. first_name=John7, last_name=Smith7, age=17
print_person_array: 8. first_name=John8, last_name=Smith8, age=18
print_person_array: 9. first_name=John9, last_name=Smith9, age=19
```

Calling Function Using Pointers

```
john@oho:~/LAB4/CALC$ more c.c
#include <stdio.h>

void myProc(int);
void myProc2(int);

void myCaller(void (*)(int), int);

int main(void) {
    myProc(1);
    myProc2(2);

    myCaller(myProc, 3);
    myCaller(myProc2, 4);

    return 0;
}
```

```
void myCaller(void (*f)(int), int param) {
    (*f)(param);    // call function *f with param
}

void myProc(int d) {
    printf("In myProc().\tParameter = %d\n", d);
}

void myProc2(int d) {
    printf("In myProc2().\tParameter = %d\n", d);
}

john@oho:~/LAB4/CALC$ gcc c.c;a.out
In myProc().  Parameter = 1
In myProc2(). Parameter = 2
In myProc().  Parameter = 3
In myProc2(). Parameter = 4
```