SCC.121: Fundamentals of Computer Science **Memory, Pointers, Records**

Amit Chopra

amit.chopra@lancaster.ac.uk

Byte Addressable Memory

- Computer memory is usually byte addressable.
- That is to say memory can be thought of as an array of bytes, and each memory “cell” has a unique address.
- You can think of this as a character array, where the address is the index of the element (memory cell).

address	content	value
40 30 1c	48	'H'
40 30 1d	65	'e'
40 30 1e	6c	'l'
40 30 1f	6c	'l'
40 30 20	6f	'o'
40 30 21	20	' '
40 30 22	74	't'
40 30 23	68	'h'
40 30 24	65	'e'
40 30 25	72	'r'

Words

- Word size can differ from machine to machine.
- In this course, we are assuming a word is 32 bits (4 bytes) long.
- Integer values are usually word size.

A words-eye view

- Here we are looking at memory in terms of "words" (or 32-bit values, which occupy 4 bytes).

- The addresses go up in steps of 4.

address	memory
40 30 04	0
40 30 08	6c 6c 65 48
40 30 0c	68 74 20 6f
40 30 10	2c 65 72 65
40 30 14	6c 6f 66 20
40 30 18	21 73 6b
40 30 1c	6c 6c 65 48

32 bits wide

A words-eye view (2)

address	memory	Occupies bytes in address range
40 30 08	6c 6c 65 48	08 .. 0b
40 30 0c	68 74 20 6f	0c .. 0f
40 30 10	2c 65 72 65	10 .. 13
40 30 14	6c 6f 66 20	14 .. 17
40 30 18	21 73 6b	18 .. 1b

32 bits wide

Words mapping to bytes

address	memory
40 30 04	00
40 30 08	6c 6c 65 48
40 30 0c	68 74 20 6f
40 30 10	2c 65 72 65
40 30 14	6c 6f 66 20
40 30 18	21 73 6b
40 30 1c	6c 6c 65 48

32 bits wide

address	memory
40 30 13	...
40 30 14	20
40 30 15	66
40 30 16	6f
40 30 17	6c
40 30 18	...

8 bits wide

(This is little-endian format – least significant byte stored at first memory address)

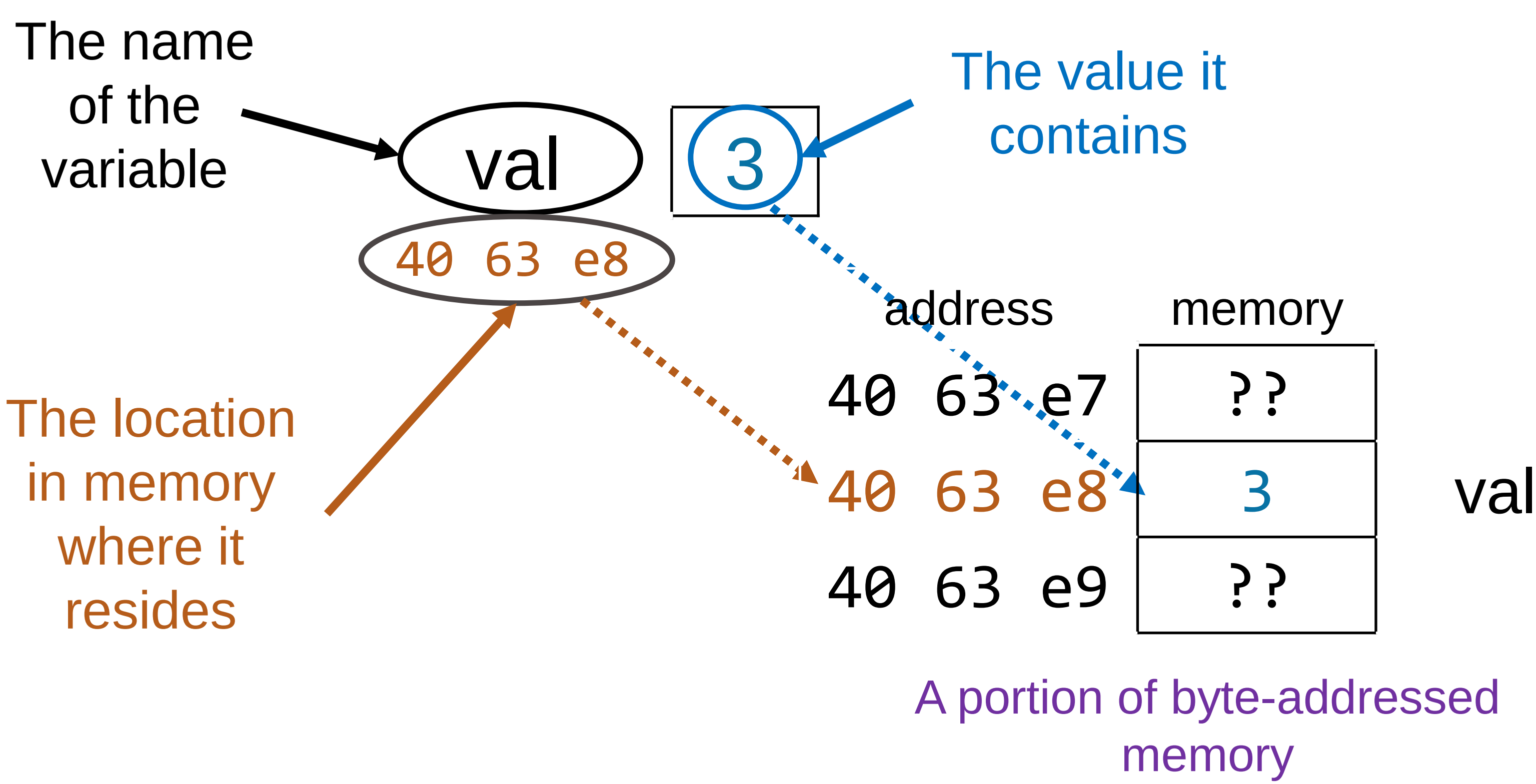
Variables

```
char val;
val = 3;
```

- A variable has three aspects:
  - A symbolic name
  - A value it contains
  - An address where it resides in memory.

Variable in memory - notation

```
char val;
val = 3;
```



Assignment statements

```
val = val + 2;
```

- The way we interpret the variable “val” is different depending on which side of the assignment statement it appears.

Fetch and Store

```
val = val + 2;
```

val $\boxed{3}$

40 63 e8

```
val = 3 + 2;
```

On the RHS we **fetch** the value stored at the address of “val”

```
val = 5;
```

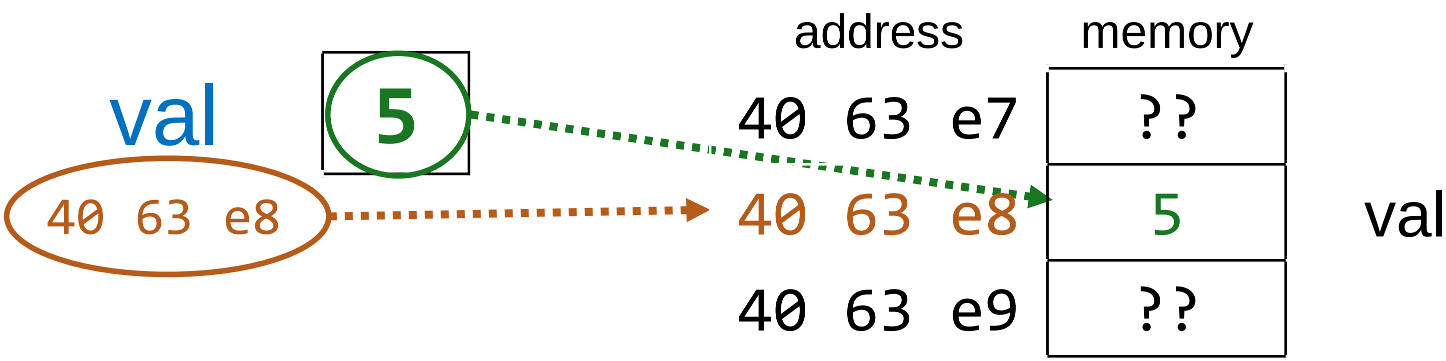
val $\boxed{5}$

40 63 e8

On the LHS we use the address of “val” as the place where we **store** the results of the expression on the RHS

Source and Destination

```
val = 5;
```



source of a value

```
val = val + 2;
```

destination of the value “delivered” by the expression on the RHS

Pointers

All C variables have scope local to the function they appear in. Then how can two or more functions work on the same conceptual object? For example, a function creates a student record and another modifies *that* student record?

Pointers address this problem by letting programmers pass memory addresses to the objects as values.

Some pointer code

```
char val;
char* addr;

val = 3;
addr = &val;
printf("val = %d, addr = %x\n", val, addr);
*addr = *addr + 2;
printf("val = %d, addr = %x\n", val, addr);
```

```
val = 3, addr = 4063e8
val = 5, addr = 4063e8
```

The effect of this code has been to add 2 to the value stored in the `val` variable.

How?

Variable and Pointer

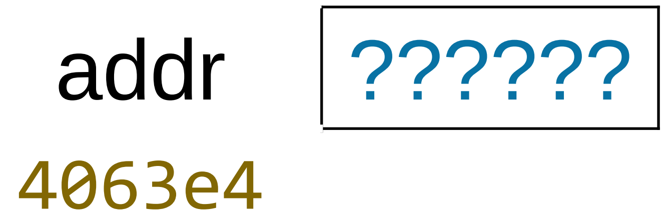
```
char val;
char* addr;
```

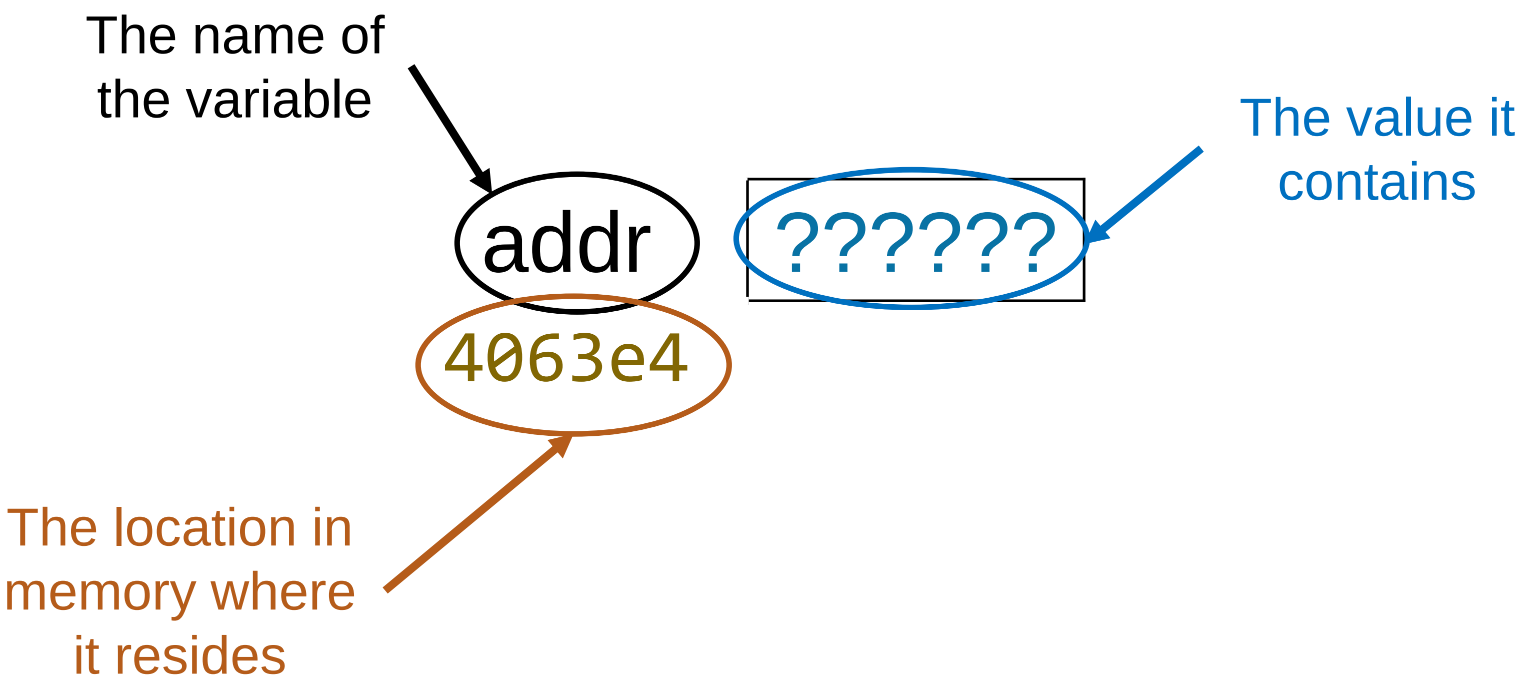
- `val` is a normal char **variable** (occupying a single byte of memory). We will be treating it as an 8-bit integer value.

- `addr` is a **pointer** to a char
  - it contains the address of a location in memory where a char should be stored.
  - it is word sized, as it needs to be able to store a 32-bit address.

char* addr (initialization)

```
char* addr;
```





addr = &val;

```
char val;
char* addr;

val = 3;
addr = &val;
printf("val = %d, addr = %x\n", val, addr);
*addr = *addr + 2;
printf("val = %d, addr = %x\n", val, addr);
```

```
val = 3, addr = 4063e8
val = 5, addr = 4063e8
```

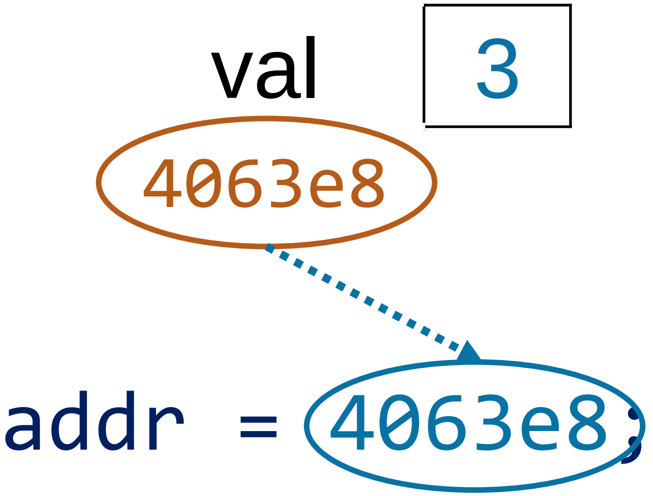
The effect of this code has been to add 2 to the value stored in the `val` variable.

How?

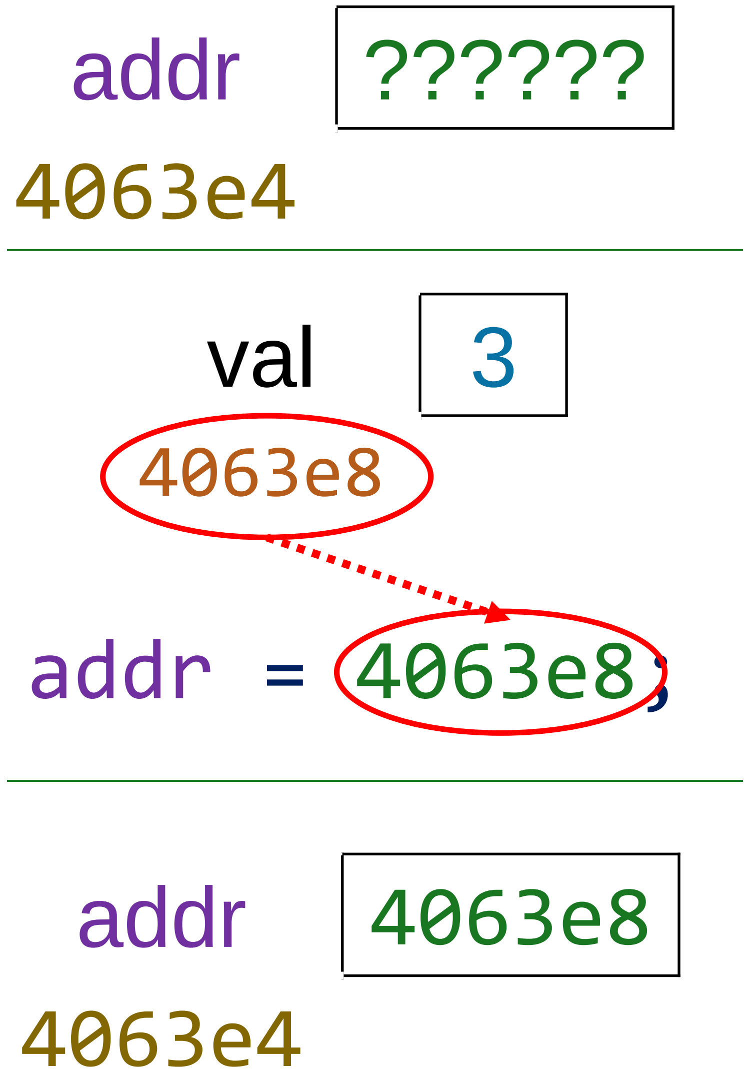
RHS of `addr = &val;`

- The **& unary operator** returns the address of its operand.

- So in this example, the value returned by the expression `&val` is `4063e8`.



LHS of `addr = &val;`



- Just like a simple assignment statement, the value on the RHS is stored in the memory location given on the LHS.

- The variable name is `addr` and the 32-bit value on the RHS will be stored at location `4063e4` (where `addr` lives).

*addr = *addr + 2;

```
char val;
char* addr;

val = 3;
addr = &val;
printf("val = %d, addr = %x\n", val, addr);
*addr = *addr + 2;
printf("val = %d, addr = %x\n", val, addr);
```

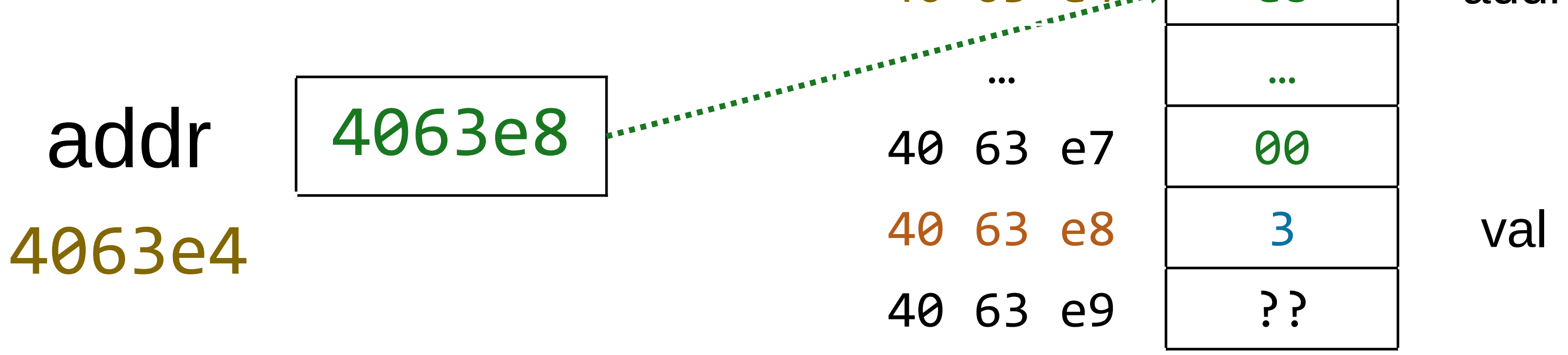
```
val = 3, addr = 4063e8
val = 5, addr = 4063e8
```

The effect of this code has been to add 2 to the value stored in the `val` variable.

How?

The * operator

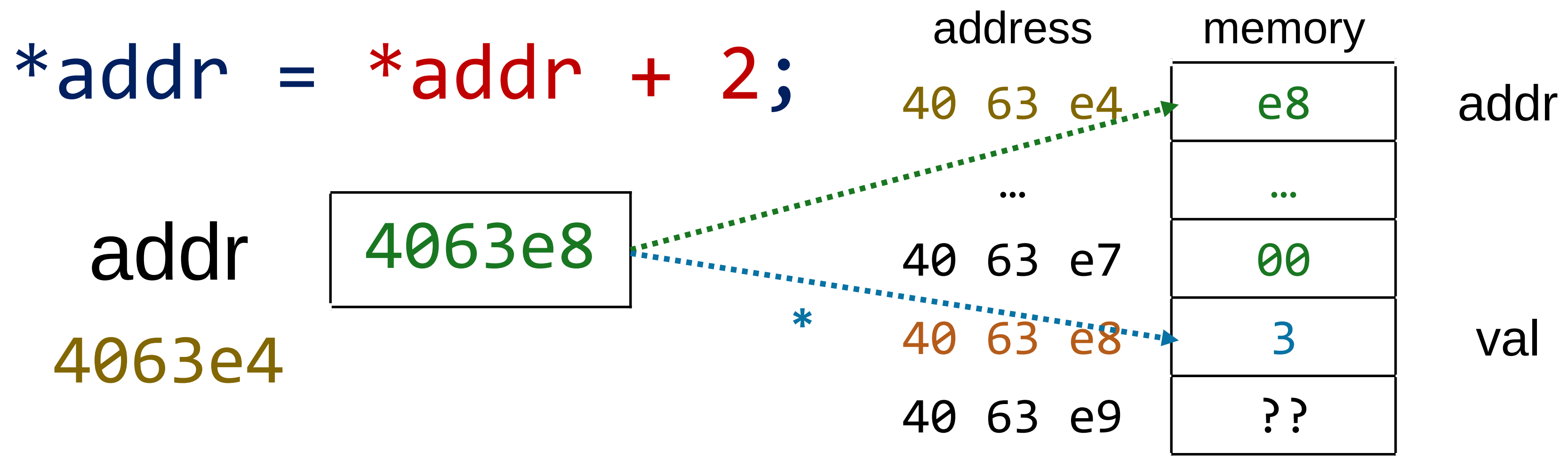
```
*addr = *addr + 2;
```



- On the RHS of this statement, the *** unary operator** returns the value stored at the address contained within its operand.

- So in this example, the value returned by the expression **`*addr`** is 3.

The * operator: two fetches



- We **fetch** the value (`4063e8`) stored at the address (`4063e4`) where “addr” lives.
- Then we treat that value as an address (* operator).

```
*addr = 4063e8 [ 3 ] + 2
```

- Now, we **fetch** the value (`3`) stored at that address (`4063e8`).

```
*addr = 3 + 2
*addr = 5
```

*addr LHS

```
*addr = 5
```

addr `4063e8`

`4063e4`

- We’ve evaluated the RHS of the assignment statement and reduced it to a single value. Now we have to store that value somewhere.

- We **fetch** the value (`4063e8`) stored at the address (`4063e4`) where `addr` lives.

- Then we treat that value as an address, and store the value there.

`4063e8` `3` `= 5`

`4063e8` `5`

*addr = 5: Before and After

before

addr: 4063e8

4063e4

val: 3

4063e8

address	memory	
40 63 e4	e8	addr
40 63 e5	63	
40 63 e6	40	
40 63 e7	00	
40 63 e8	3	val
40 63 e9	??	

after

addr: 4063e8

4063e4

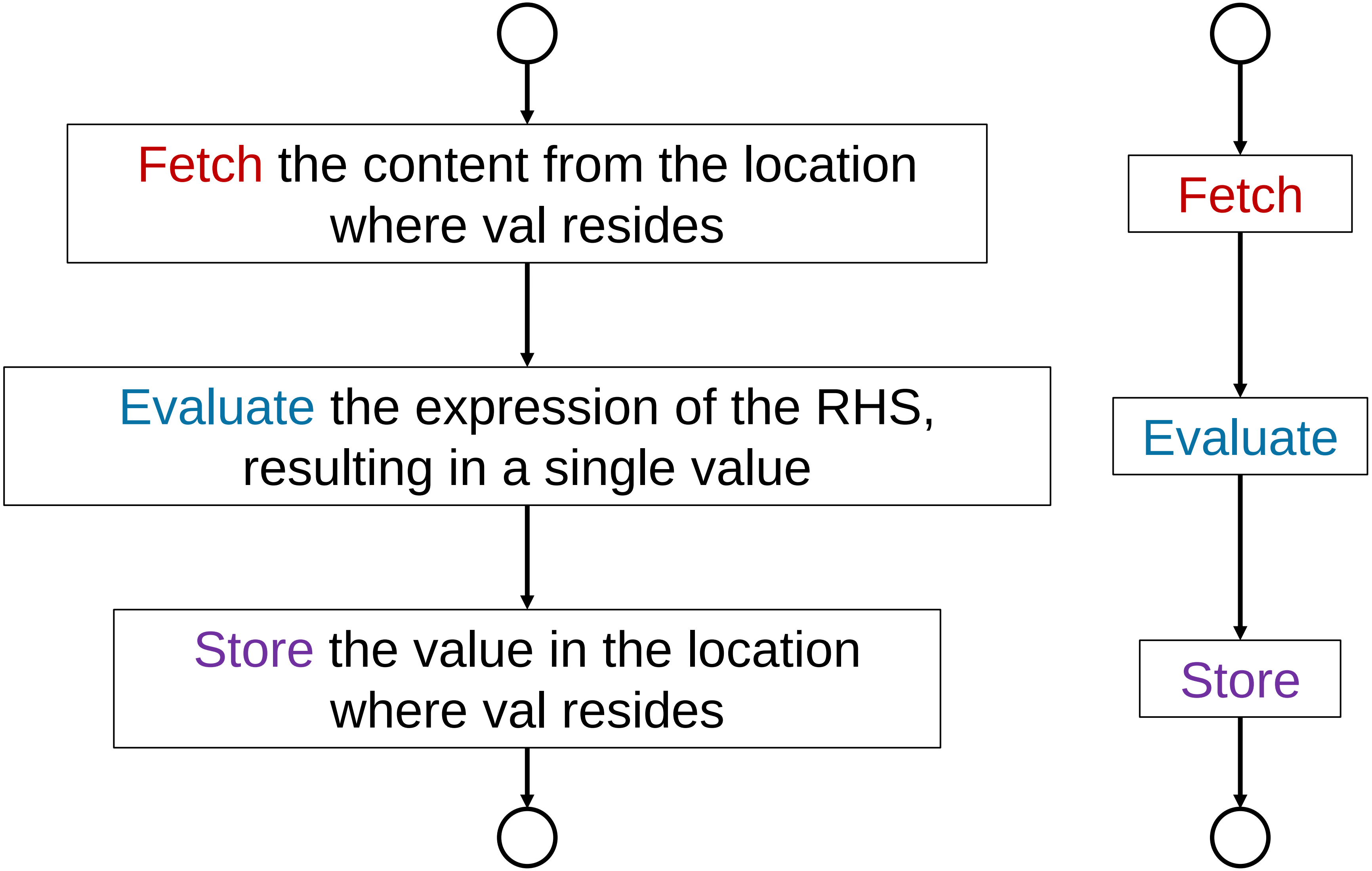
val: 5

4063e8

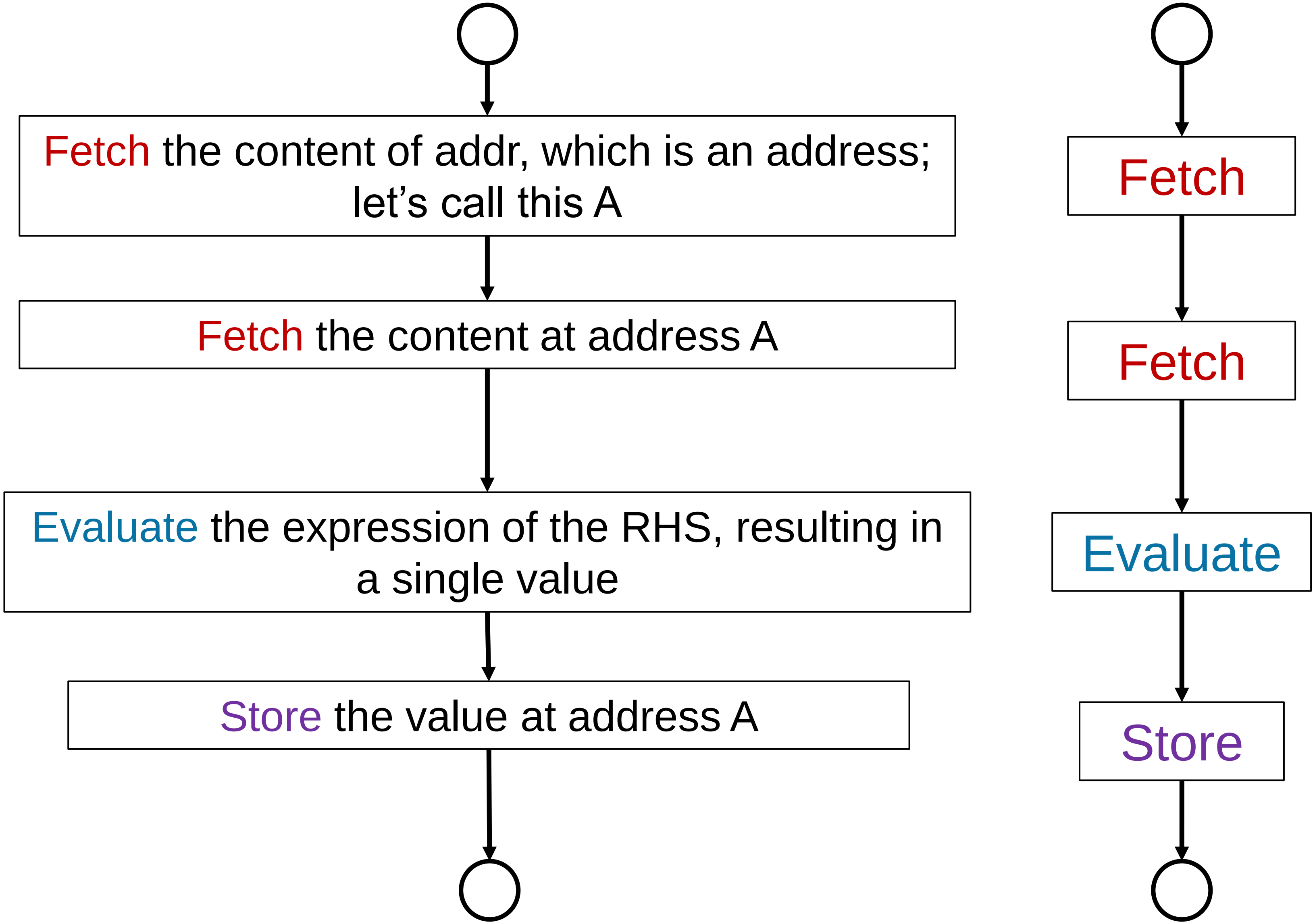
address	memory	
40 63 e4	e8	addr
40 63 e5	63	
40 63 e6	40	
40 63 e7	00	
40 63 e8	5	val
40 63 e9	??	

Levels of Indirection

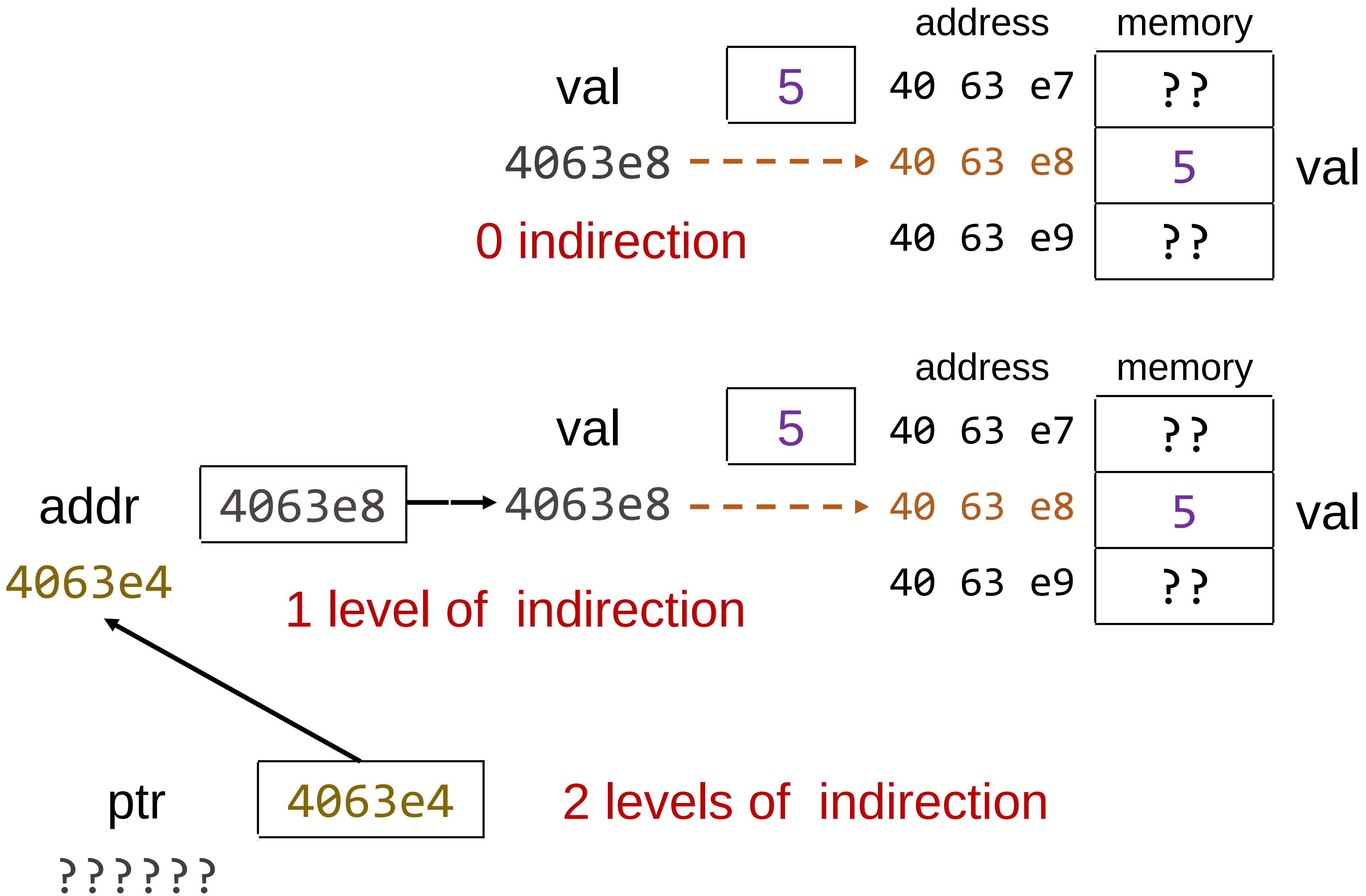
val = val + 2


Fetch the content from the location where val resides
Evaluate the expression of the RHS, resulting in a single value
Store the value in the location where val resides
Fetch
Evaluate
Store

```
*addr = *addr + 2
```


Fetch the content of addr, which is an address; let's call this A
Fetch the content at address A
Evaluate the expression of the RHS, resulting in a single value
Store the value at address A
Fetch
Fetch
Evaluate
Store

Levels of indirection



Appendix: Notation

- We indicate that a variable is a pointer with the ***type modifier ****

```
int* x;
int *x;
int * x;
```

- All of the above are equivalent (C ignores the whitespace)
- Can be read as “x is a pointer to an int”

- The *unary operator* ***** is known as the **dereference operator** (aka indirection operator)

`*ptr` fetches the value stored at `ptr`

- The *unary operator* **&** is known as the **address-of operator**.

`&val` returns the address of `val`

Entities

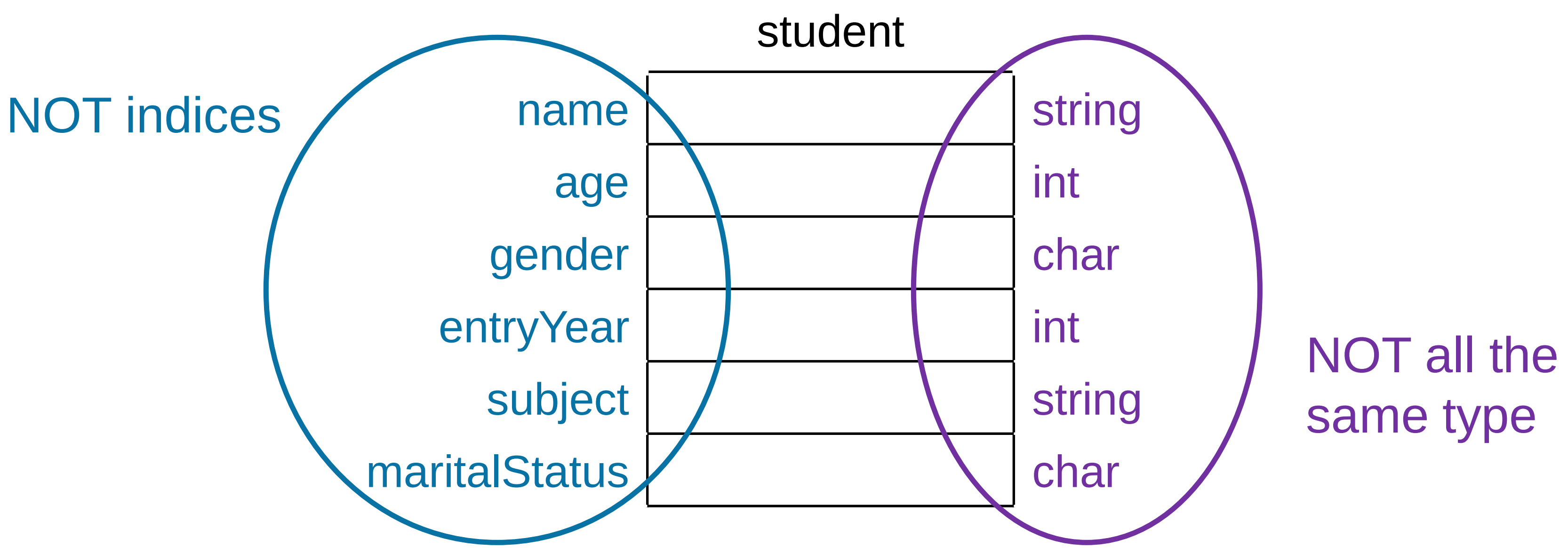
- In looking at compound data structures, so far we have examined arrays.
- Arrays allow us to model a collection of values, all of which have the same type (an array of integers, for example).
- Sometimes we will want to model an entity which consists of values of different types.
- Traditionally this is referred to as a record.

Records

- A record is a compound item
- Unlike an array (which is homogeneous), it is heterogeneous – i.e., components of various types:
  - Its components are called *fields*
  - Each *field* is identified using a *name* (**not** an index)
- A record holds various different properties of a single entity

Example

- Here is a 'student' record, which contains 6 fields.
- The names of the fields are : name, age, gender, entryYear, subject and maritalStatus
- The fields types are: string, int, char, int, string, char



Records in C

- We can define a new “record” type in C as follows:

typedef: type definition

struct: a structure

```
#define MAXSIZE 20

typedef struct student {
    char name[MAXSIZE];
    int age;
    char gender;
    int entryYear;
    char subject[MAXSIZE];
    char maritalStatus;
} Student;
```

student - name of the struct

Student - name of the type

Allocating space for a record

- Here is a function which creates the space for a new Student record, and returns a pointer to that space.

```
Student* newStudent()
/**
   allocates space for a new Student record
   returns pointer to allocated space
*/
{
   Student* pt = malloc(sizeof(Student));
   return pt;
};
```

The "malloc" and "sizeof" functions

- C has a function called "malloc" (for Memory ALLOCation) which dynamically allocates memory from an area within your program's space called the heap.

```
Student* pt = malloc(sizeof(Student));
```

- We use the `sizeof` function to tell us how many bytes a Student record occupies.

- And then use `malloc` to actually allocate the space we require, returning the base address of the area of memory allocated.

The Arrow (->) Operator

- Here are some lines of code that create a new Student record (by calling the “newStudent” function on the previous slide) and then allocates values to the individual fields.

- Note the use of the arrow (->) operator. This selects an attribute or field of the record.

```
Student* stu = newStudent();
strcpy(stu->name, "James T. Kirk");
stu->age = 19; //(*stu).age = 19
stu->gender = 'M';
stu->entryYear = 2252;
strcpy(stu->subject, "Space Command");
stu->maritalStatus = 'X'; //(intentional mistake)
```

The Dot (.) Operator

- You can declare a new Student variable directly.
- But then to access the fields, we use the `.` (dot) notation rather than the `->` (arrow) notation.

```
Student stu2;
strcpy(stu2.name, "Nyota Uhura");
stu2.age = 18;
stu2.gender = 'F';
stu2.entryYear = 2257;
strcpy(stu2.subject, "Communications");
stu2.maritalStatus = 's';
```

Arrays of Records

- Because Student is a (user defined) type, we can use it anywhere we could use a “base” (provided) type.
- So just as we can have an array of integers, we can have an array of Students.
- What we have here is an array of pointers to Student records.

```
Student* arrayOfStudents[2];
arrayOfStudents[0] = stu;
arrayOfStudents[1] = &stu2;
```

“set” (values) functions

- The four marital statuses we are using are: s, d, m, w.
- We have modelled these as a single character.

```
stu->maritalStatus = 'X’;
```

- The programmer can assign any character value they choose.
- We can still provide the programmer with a function that sets the value of “marital status” after checking that a valid status has been supplied.

setMaritalStatus function

- This function sets the value of a Student's marital status but only after ensuring the value provided is one of the four valid options.
- Returns a bool to indicate whether the marital status has been updated or not.

```
bool setMaritalStatus(Student* s, char x)
{
   bool ok = false;
   switch (x)
   {
      case 'm': case 'w':
      case 's': case 'd': ok = true; break;
   };
   if (ok) s->maritalStatus = x;
   return ok;
}
```

could expand this to accept upper case as well

or detect upper case has been used and convert it to lower case