RMIT UNIVERSITY

Computer Science and Information Technology

Advanced Programming Techniques

(a.k.a. Programming in ANSI / ISO C)

Pointers and Arrays in C

“Act in haste and repent at leisure; Code too soon and debug forever.”

-- Raymond Kennington

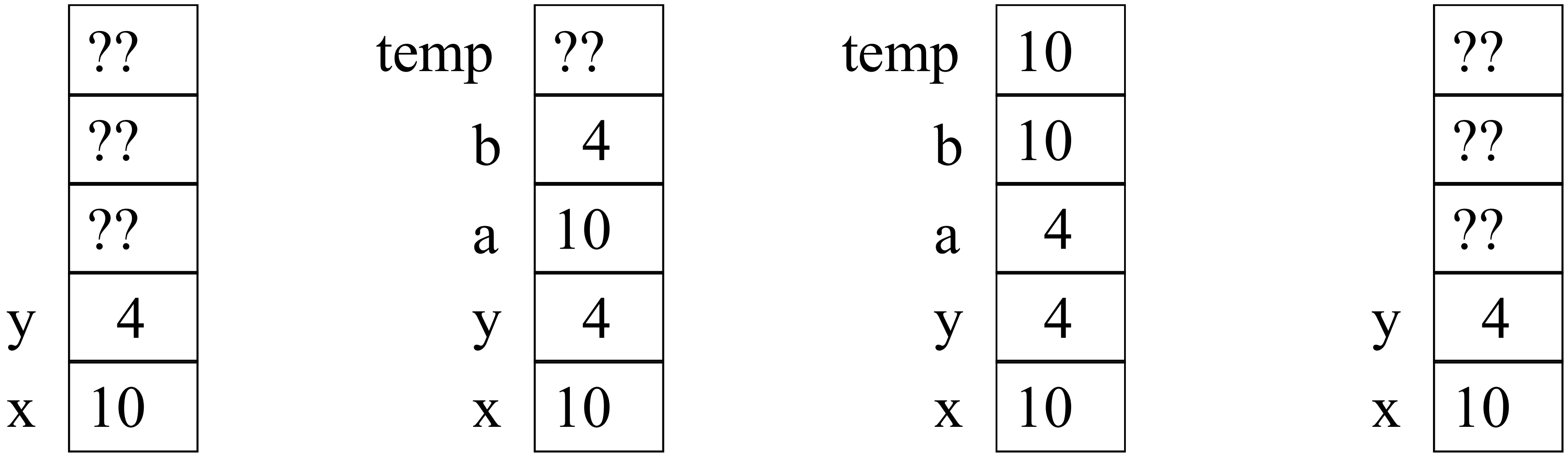
Revision: Parameters – call by value

```
int main(void)
{
    int x, y;
    x = 10;
    y = 4;
    swap(x, y);
    printf(“x:%d y:%d\n”,x,y);
    return EXIT_SUCCESS;
}
```

```
void swap(int a, int b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

What is wrong with this?

Parameters – call by reference (cont'd)



	??
	??
	??
y	4
x	10

before `swap()` is called

temp	??
b	4
a	10
y	4
x	10

initial values of `a` and `b`

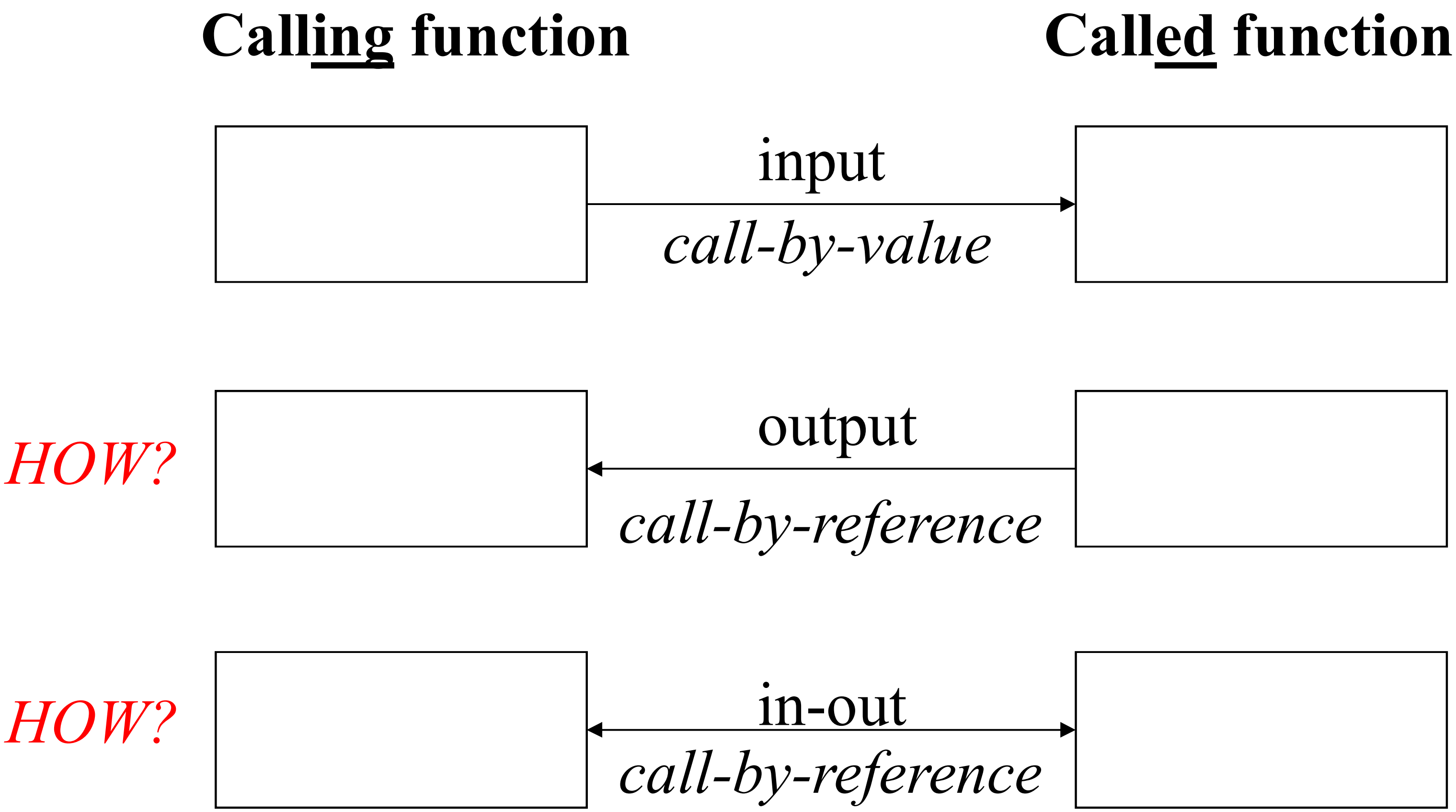
temp	10
b	10
a	4
y	4
x	10

after swapping `a` and `b`

	??
	??
	??
y	4
x	10

after exiting `swap()`

Revision: Types of parameter passing



Calling function **Called function**

input

call-by-value

HOW?

output

call-by-reference

HOW?

in-out

call-by-reference

Revision: I/O: printf and scanf

- Function *scanf* is used for input
- The control string specifies how to interpret the input
- The function takes the address of a variable (specified by **&**) to read the result into

Huh??

```
#include <stdio.h>
int main(void)
{
        int age;
        printf(“How old are you?:”);
        scanf(“%d”,&age);
        printf(“You said you were %d years old\n”,age);
        return 0;
}
```

Arrays as parameters

- Arrays can only be *call-by-reference* parameters, never *call-by-value*

Why??

The *pointer* data type

- The *pointer* data type solves these problems
- A pointer variable is a variable that can store a memory address.
- What is a memory address?
  - binary that represents the bus address of a memory location
- How do we get a memory address?
- If we have an existing variable (e.g. `age`), we can get the address of the variable by using **&** (e.g. `&age`)

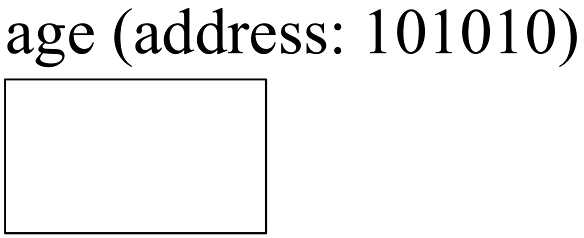
The *pointer* data type

- There are two components to a pointer declaration
  - that the variable is a pointer (indicated by a *)
  - the type for the data at the address the pointer points to
    - we’ll see later why we (usually) need to know what type of thing the pointer is pointing to

The *pointer* data type

- An int variable:

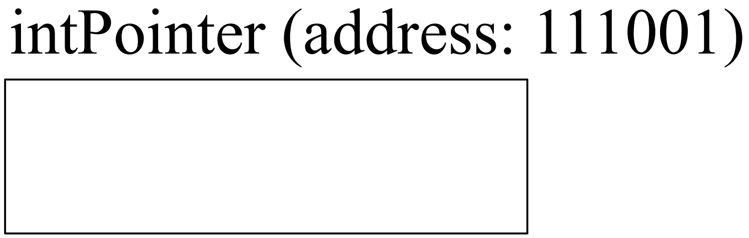
```
    int age;
```



age (address: 101010)

- A pointer variable that can hold the address of an int variable:

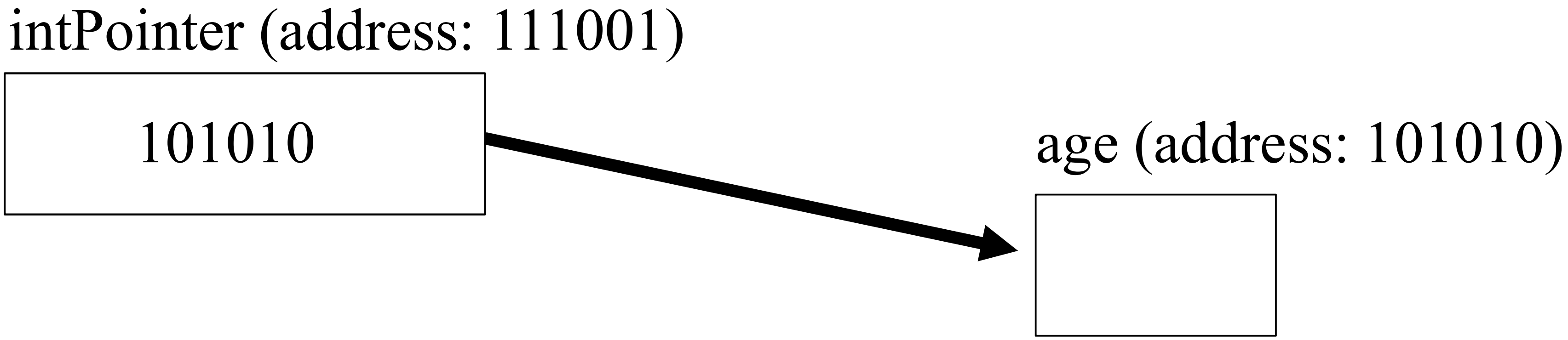
```
    int *intPointer;
```



intPointer (address: 111001)

The *pointer* data type

- The pointer could be set to ‘point’ to the address of `age`



intPointer (address: 111001)

101010

age (address: 101010)

The *pointer* data type

- An int variable on server jupiter/titan is 4 bytes long:

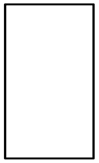
```
int age;
```

age (size 4 bytes address: 101010)

- A char variable on server jupiter/titan is 1 byte long

```
char grade;
```

grade (size 1 byte address: 100111)



The *pointer* data type

- A pointer variable on server jupiter/titan is 8 bytes long

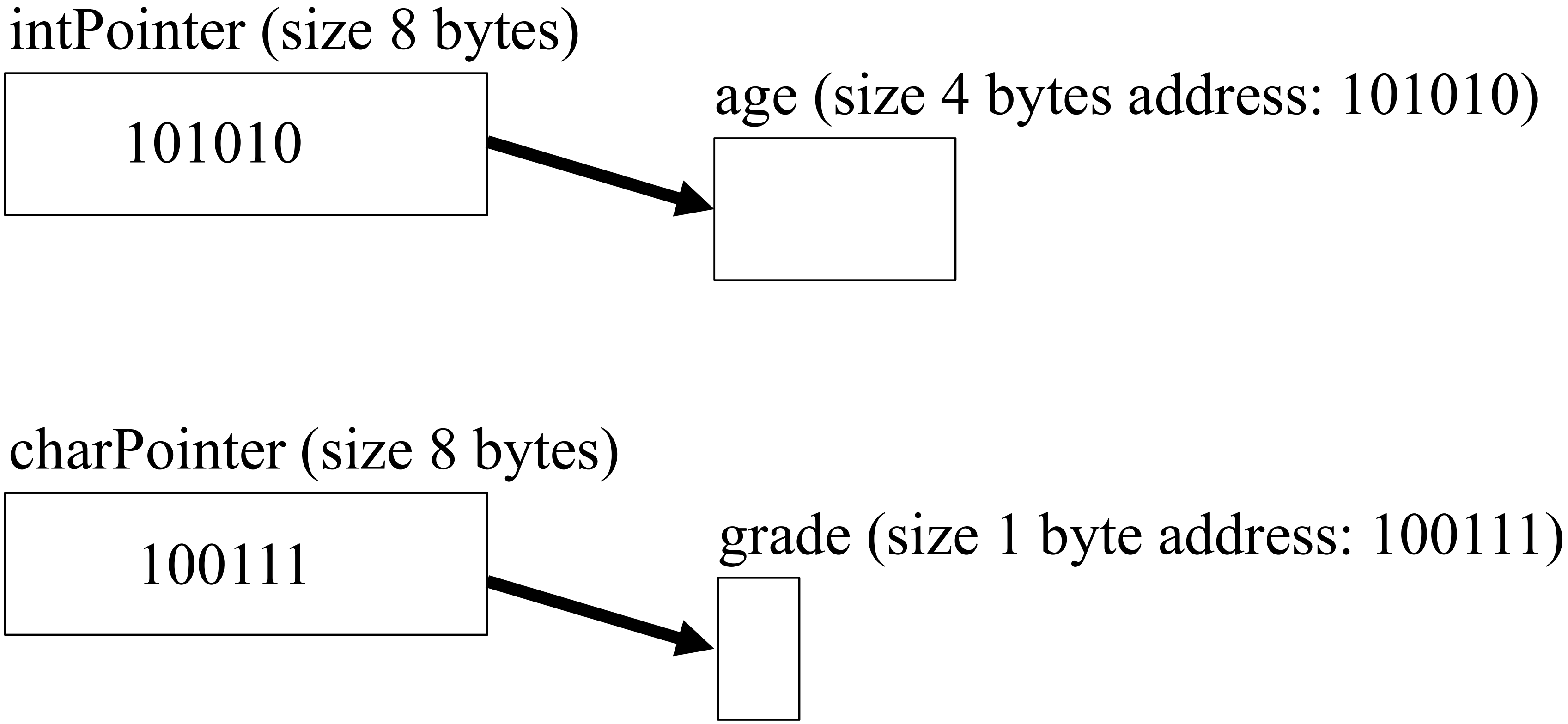
```
int *intPointer;
```

intPointer (size 8 bytes)

```
char *charPointer;
```

charPointer (size 8 bytes)

The *pointer* data type



intPointer (size 8 bytes)

101010

age (size 4 bytes address: 101010)

charPointer (size 8 bytes)

100111

grade (size 1 byte address: 100111)

The *pointer* data type

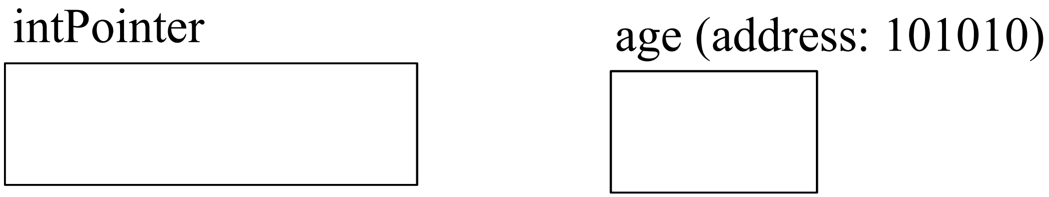
```
int age;
int *intPointer;
```

- The pointer can be set to ‘point’ to the variable `age` by assigning the number of the memory address of `age` to `intPointer`

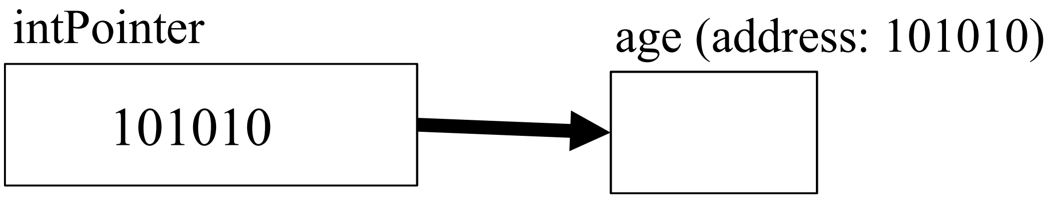
```
intPointer = &age;
```

The *pointer* data type

```
int age;
int *intPointer;
```



```
intPointer = &age;
```



The *pointer* data type

```
int age;
int *intPointer;
intPointer = &age;
```

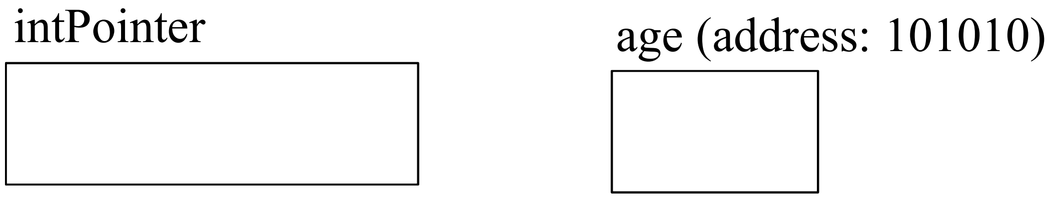
- While * when declaring a variable indicates that it is a pointer variable, * in a statement ‘dereferences’ the pointer
  - i.e. accesses the memory address pointed to by the pointer

```
*intPointer = 21;
```

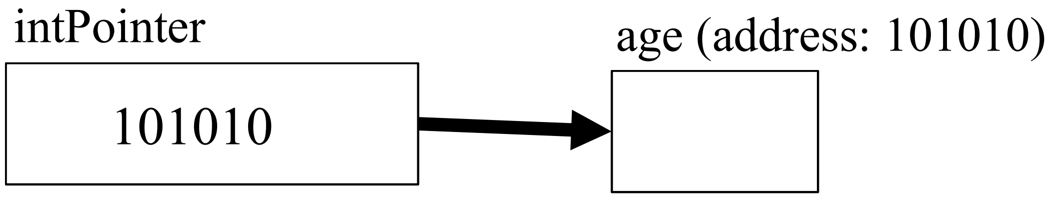
will put 21 into memory at the address of variable `age`

The *pointer* data type

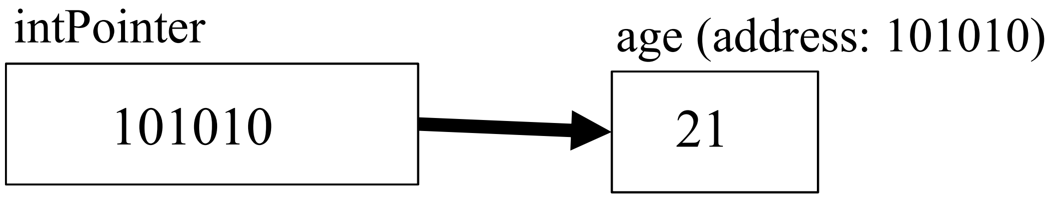
```
int age;
int *intPointer;
```



```
intPointer = &age;
```



```
*intPointer = 21;
```



The pointer data type (cont'd)

- The legal operators with pointers in C are:
  - assigning a memory address to a pointer variable
  - dereferencing the pointer - i.e. accessing the value that is stored at the address that the pointer points to
  - pointer arithmetic - addition and subtraction **only**
  - comparison operators - e.g. `==` `!=` `>` `<` etc.
- No other operations are legal - **nor make sense!**
- The pre-defined constant value of `NULL` can be assigned to a pointer to indicate that the pointer doesn't currently contain a legal address value.

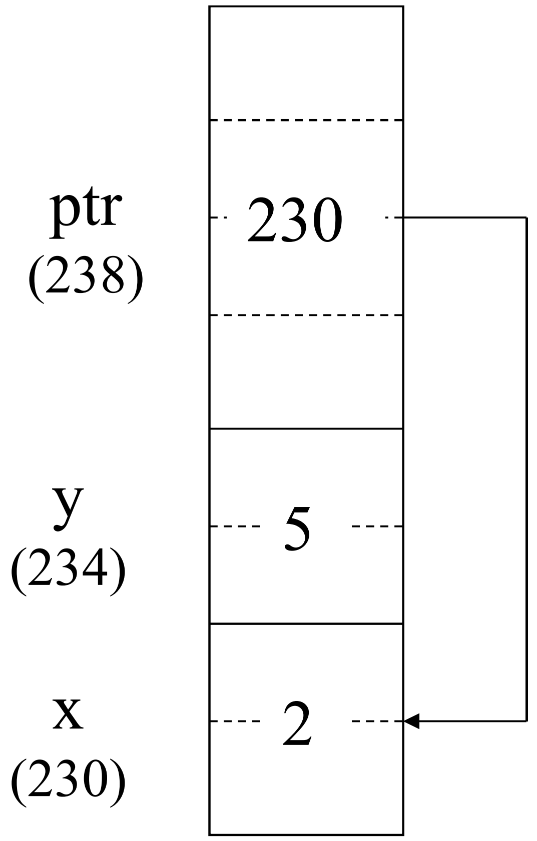
The pointer data type (cont’d)

```
int   x, y;
int   *ptr;
x = 2;
ptr = &x;
y = *ptr + 3;
```

pointer declaration

assigning an address

dereferencing



NB: this example assumes that an integer is 32bits (4 bytes) and we have 32bit (4 byte) addresses

Revision: I/O: scanf

- Function *scanf* is used for input
- The control string specifies how to interpret the input
- The function takes the address of a variable (specified by **&**) to read the result into

```
#include <stdio.h>
int main(void)
{
        int age;
        printf(“How old are you?:”);
        scanf(“%d”,&age);
        printf(“You said you were %d years old\n”,age);
        return 0;
}
```

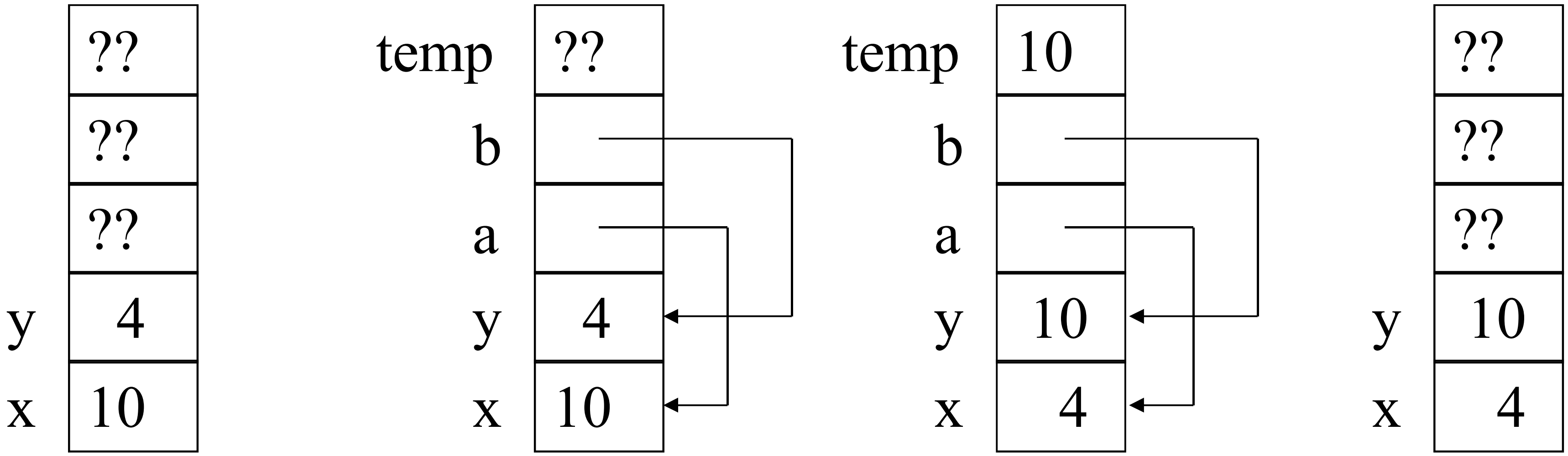
Parameters – call by reference

C uses pointers to implement *call-by-reference*:

```
int main(void)
{
  int x = 10, y = 4;

  swap(&x, &y);
  printf(“%d  %d\n”, x, y);
  ...
}
```

```
void swap(int *a, int *b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```

Parameters – call by reference (cont’d)



before `swap()` is called

initial values of `a` and `b`

after swapping `*a` and `*b`

after exiting `swap()`

Parameters – call by reference (cont’d)

- Using pointers in this way is used to achieve both *output* and *in-out* parameters
  - i.e. C does not explicitly distinguish between them.
- It is important to ensure:
  - actual and formal parameters are of same type
  - addresses are passed, not values (i.e. use & operator)
  - called function dereferences parameters correctly

Structs as parameters (call by reference)

- consider function calculateGrade() which uses values of some members of student to calculate values for other members
  - i.e. student is an *in-out* parameter:

```
int main(void)
{
    StudentType student;
    :
    calculateGrade(&student);
    :
}
```

Structs as parameters (cont’d)

```
void calculateGrade(StudentType *student)
{
  ...
  (*student).total=(*student).ass1 * 0.2
                 + (*student).ass2 * 0.3
                 + (*student).exam * 0.5;
  ...

}
```

- Even the inventors of C saw this as an “ugly” notation and so gave us an alternative ...

Structs as parameters (cont’d)

```
void calculateGrade(StudentType *student)
{
    :
    student->total = student->ass1 * 0.2 +
                     student->ass2 * 0.3 +
                     student->exam * 0.5;
    :
}
```

- The so-called arrow-operator is just an alternative (though more pleasing) syntax for de-referencing members of structs via pointers to a struct

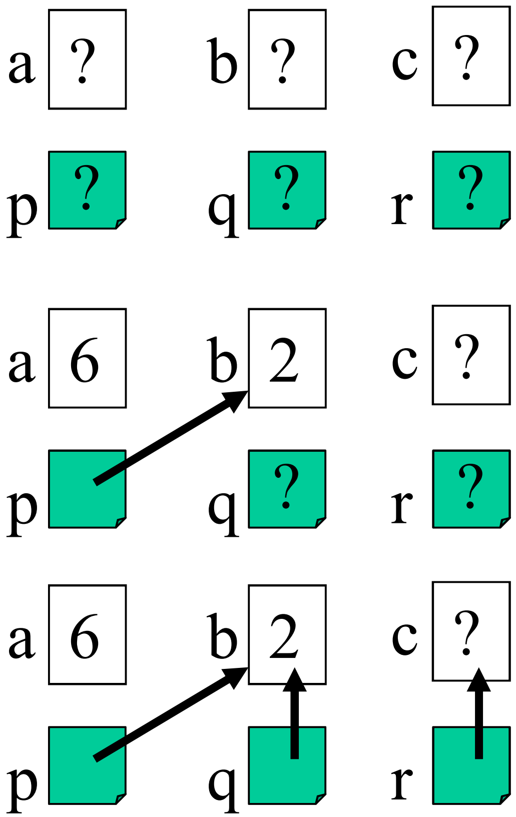
Arrays as parameters

- Arrays can only be *call-by-reference* parameters, never *call-by-value*
- … because of the unique relationship between array names and pointers in C …

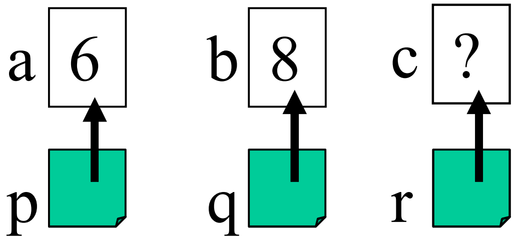
Arrays and Pointers in C

- Array name is the address of 0th element
- Pointer arithmetic
- x[i] is equivalent to *(x + i)
- Array name is effectively a constant pointer
- Multi-dimension arrays
- Arrays as parameters

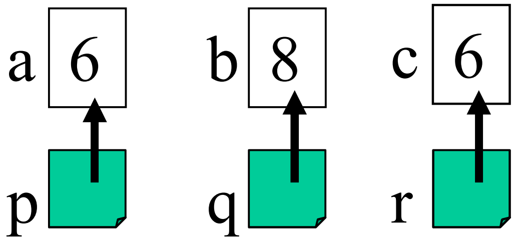
```
#include <stdio.h>
int main(void)
{
    int a;
    int b;
    int c;
    int *p;
    int *q;
    int *r;

    a = 6;
    b = 2;
    p = &b;

    q = p;
    r = &c;
```



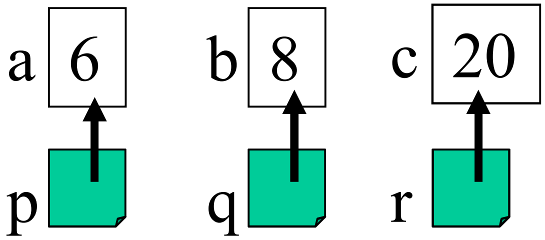
```
p = &a;
*q = 8;
```



```
*r = *p;
```



```
*r = a + *q + *&c;
```



```
    printf("%d %d %d \n", a, b, c);
    printf("%d %d %d \n", *p, *q, *r);

    return 0;
} /* main */
```

Result:

```
6 8 20
6 8 20
```

Array name is the address of 0th element

- An array name in C corresponds to the base address of the array
- Another way of saying this is that for the array:

  ```
  int x[10];
  ```

  the expression (`x`) is equivalent to (`&x[0]`)
- An array name is therefore effectively a (constant) pointer:
  - the type that x points to is the type of the elements of the array
    - i.e. x is effectively of type int *
  - array names however are constants and can **not** be modified:
    - x = &y; is illegal!

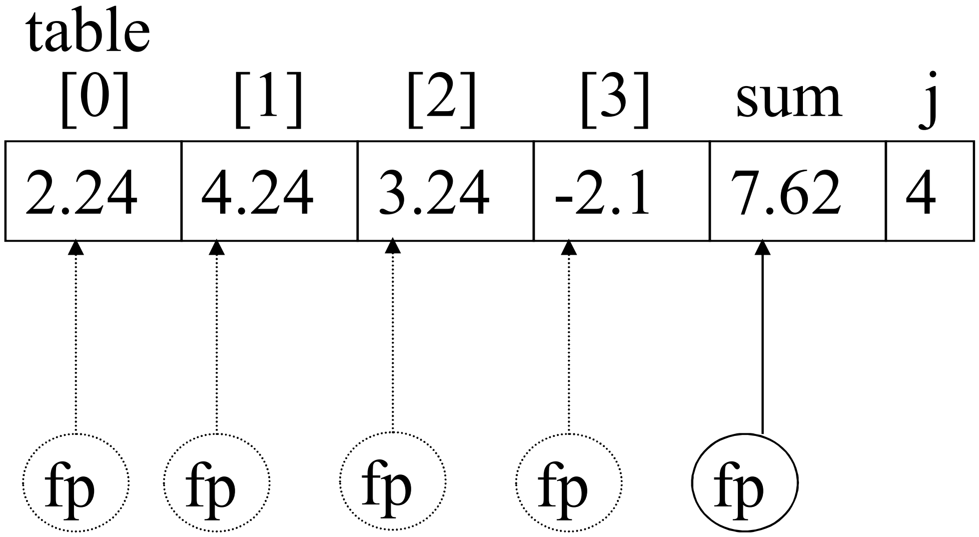
Pointer arithmetic

- Addition and subtraction (only) are allowed with pointers
- Pointer arithmetic is “aware” of the size of the type pointed to so that, for example, adding 2 to a pointer value actually adds:

  (2 * sizeof <type-pointed-thing>)
- This means we can use pointer arithmetic to “step through” an array, regardless of the type of the array element ...

Pointer arithmetic (cont'd)

```
float table[4] = {2.24, 4.24, 3.24, -2.1};
float *fp;
float sum = 0.0;
int  j;

fp = table;
for (j=0; j < 4; j++, fp++)
     sum += *fp;

/*  normally we might
 *  just do:
 *   sum += table[j];
 */
```



Pointer arithmetic (cont’d)

- Another way ...

```
float table[4] = {2.24, 4.24, 3.24, -2.1};
float *fp;
float sum = 0.0;

for (fp=table; fp < table + 4; fp++)
     sum += *fp;
```

Pointer arithmetic (cont'd)

- And yet another way ...

```
float table[4] = {2.24, 4.24, 3.24, -2.1};
float *start = table, *end = table + 4;
float sum = 0.0;

for ( ; start < end; start++)
     sum += *start;
```

Pointer arithmetic (cont’d)

- And yet another way … using our pointer as though it was an array ...

```
float table[4] = {2.24, 4.24, 3.24, -2.1};
float *fp = table;
float sum = 0.0;
int j;

for (j=0; j < 4; j++)           |    for (j=0; j < 4; j++)
   sum += fp[j];                |       sum += *(fp + j);
```

Pointer arithmetic (cont'd)

- … one more time … who needs pointers anyway?

```
float table[4] = {2.24, 4.24, 3.24, -2.1};
float sum = 0.0;
int j;

for (j=0; j < 4; j++)          |  for (j=0; j < 4; j++)
   sum += table[j];            |     sum += *(table + j);
```

- None of these ways is generally superior.

x[i] is equivalent to *(x + i)

- The previous example shows that:

  `table[j]` is equivalent to `*(table + j)`
- This is true for <u>any</u> pointer, including array names
- This special relationship between arrays and pointers in C says that for any array of any type:

  x[i] <u>is equivalent to</u> *(x + i)

  (x + i) is the address of the i^{th} element of x

  x[i] is the (dereferenced) value at location (x + i)

Array name is a constant pointer

- And just a reminder … that one important difference between an array name and a pointer variable is:
  - an array name is a constant pointer
  - i.e. it can not be altered … e.g.

```
int x[10], *ptr;

ptr = x;  /*  is legal (and useful)             */
x = ptr;  /*  is illegal - syntax error!        */
```

Arrays as parameters

- We can see now why *call-by-value* (where a copy of the data is made) is not possible with array names:

```
int numList[SIZE] = { 1, 2, 3, 4, 5 };
int sum;
sum = add(numList);   /* equivalent to ... */

sum = add(&numList[0]);
```

- i.e. numList is an array name which evaluates to:

```
    &numList[0]
```

- … therefore ...

Arrays as parameters (cont'd)

- Function add would look like either :

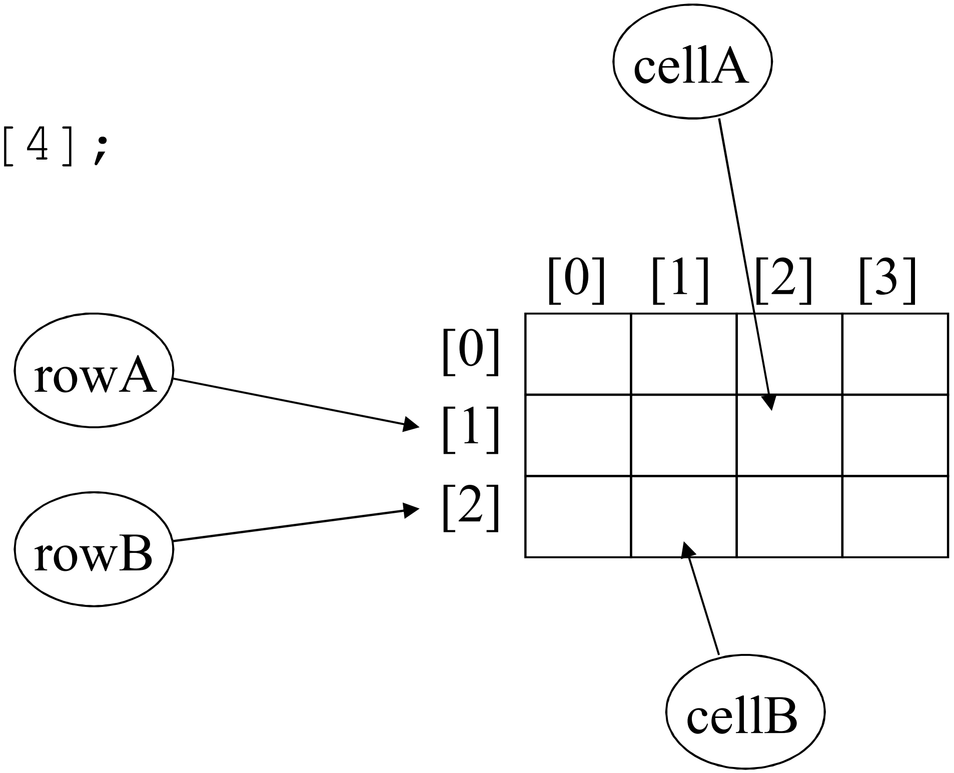
```
int add(int *nums)
{
   int total = 0, j;
   for (j=0; j<SIZE; j++)
   {
      total += *(nums + j);
   }
   return total;
}
```

```
int add(int nums[])
{
   int total = 0, j;
   for (j=0; j<SIZE; j++)
   {
      total += nums[j];
   }
   return total;
}
```

Multi-dimension arrays

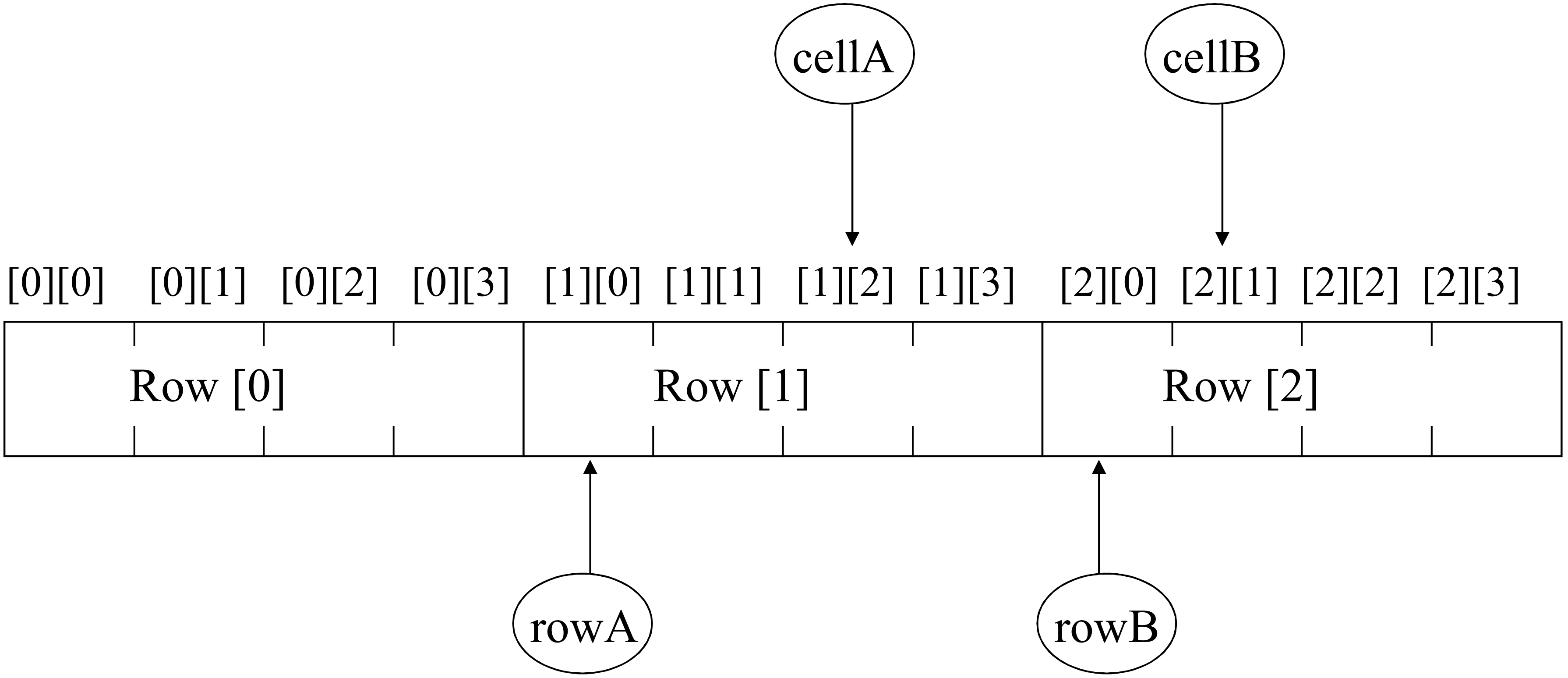
- These properties extend to multi-dimension arrays

```
int matrix[3][4];
int *cellA, *cellB;
int (*rowA)[4], (*rowB)[4];

cellA = &matrix[1][2];
cellB = cellA + 3;

rowA = &matrix[1];
rowB = rowA + 1;

(*rowB)[2] = *cellA;
```



Multi-dimension arrays (cont'd)

- or … viewing the 2D array as an array of arrays ...



Multi-dimension arrays (cont’d)

- The equivalence between array indexing and pointer dereferencing still applies ...

```
int matrix[ROWS][COLS];
int i, j, sum = 0;
for (i=0; i < ROWS; i++)
  for (j=0; j < COLS; j++)
    sum += matrix[i][j];
```

Alternatives:

```
sum += *(matrix[i] + j);

sum += *(*(matrix + i) + j);

sum += (*(matrix + i))[j];
```

Multi-dimension arrays as parameters

Calling function:

```
int matrix[3][4];
int sum;

sum = matrixSum(matrix);
```

Equivalent function definitions:

```
int matrixSum(int mat[][4])
{   ...
    total += mat[i][j];
    ...
}
```

```
int matrixSum(int (*mat)[4])
{   ...
    total += mat[i][j];
    ...
}
```

Multi-dimension arrays as parameters (cont’d)

Calling function:

```
int matrix[3][4];
int sum;

for (i=0; i<ROWS; i++)
    sum = vectorSum(matrix[i]);
```

Equivalent function definitions:

```
int vectorSum(int vector[])
{   …
    total += vector[j];
    …
}
```

```
int vectorSum(int *vector)
{   …
    total += vector[j];
    …
}
```