Multilevel Pointers

Advanced C

Pointers – Multilevel

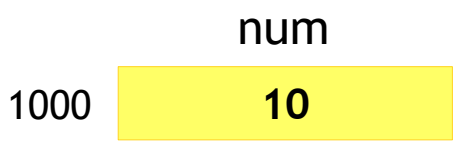
- A pointer, pointing to another pointer which can be pointing to others pointers and so on is know as multilevel pointers.
- We can have any level of pointers.
- As the depth of the level increase we have to bit careful while dealing with it.

Advanced C

Pointers - Multilevel

001_example.c

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



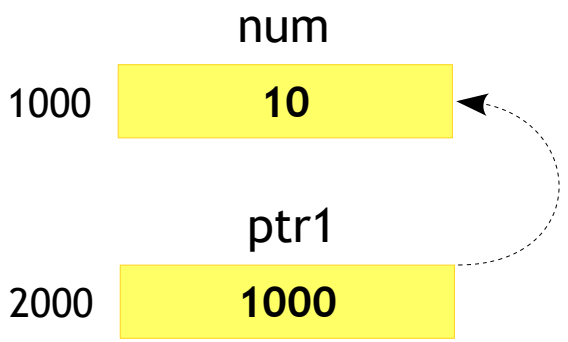
EMERTXE

Advanced C

Pointers - Multilevel

001_example.c

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



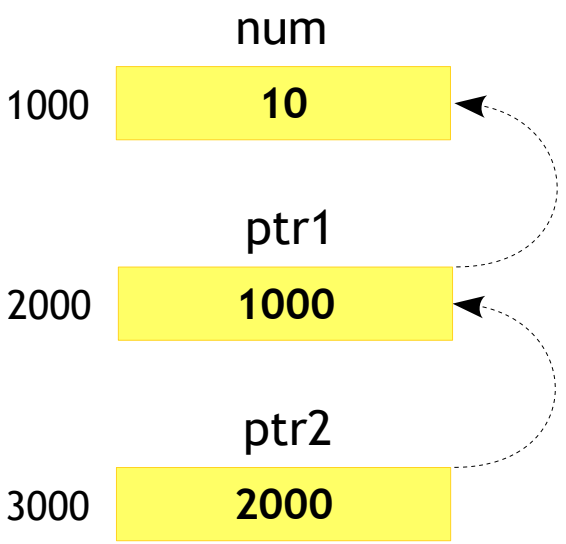
EMERTXE

Advanced C

Pointers - Multilevel

001_example.c

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



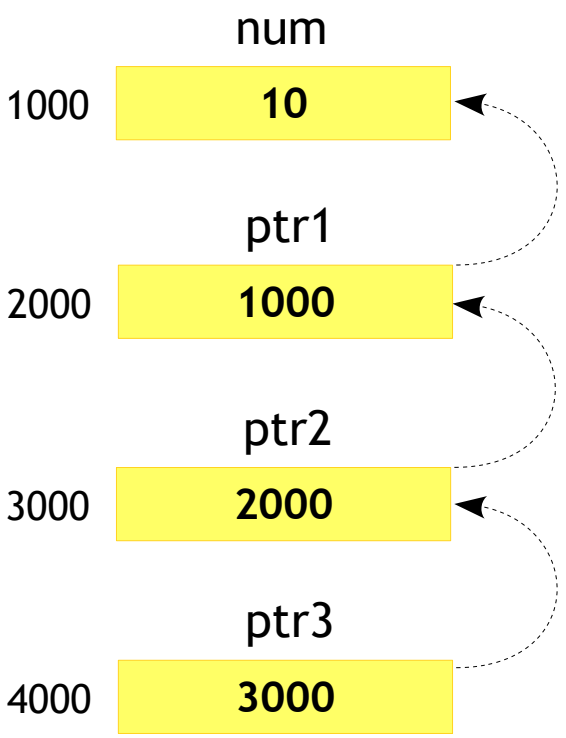
EMERTXE

Advanced C

Pointers - Multilevel

001_example.c

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



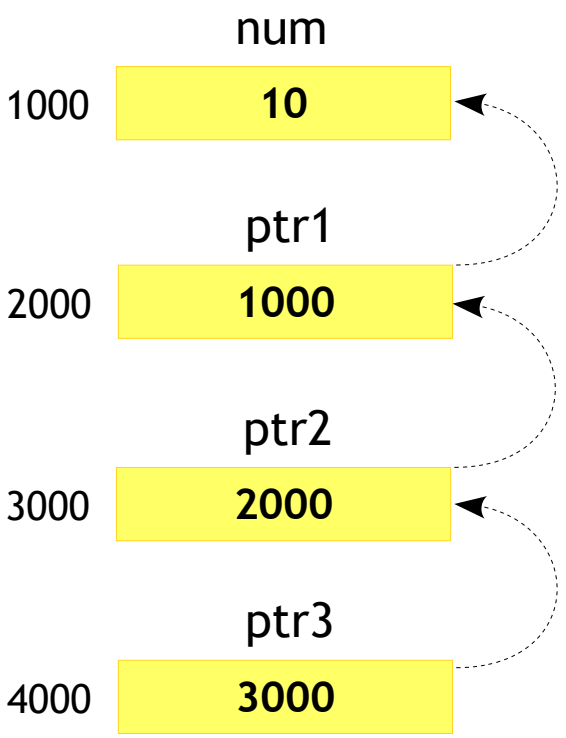
EMERTXE

Advanced C

Pointers - Multilevel

001_example.c

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



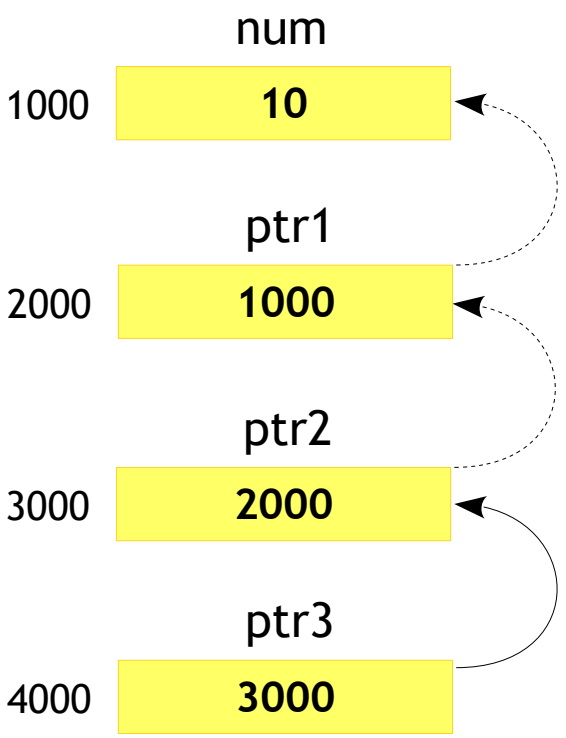
Output → 3000

Advanced C

Pointers - Multilevel

001_example.c

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



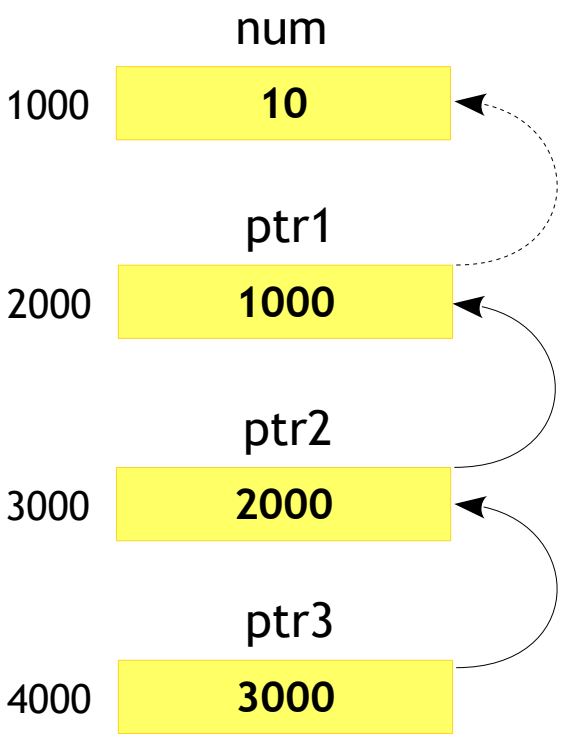
Output → 2000

Advanced C

Pointers - Multilevel

001_example.c

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



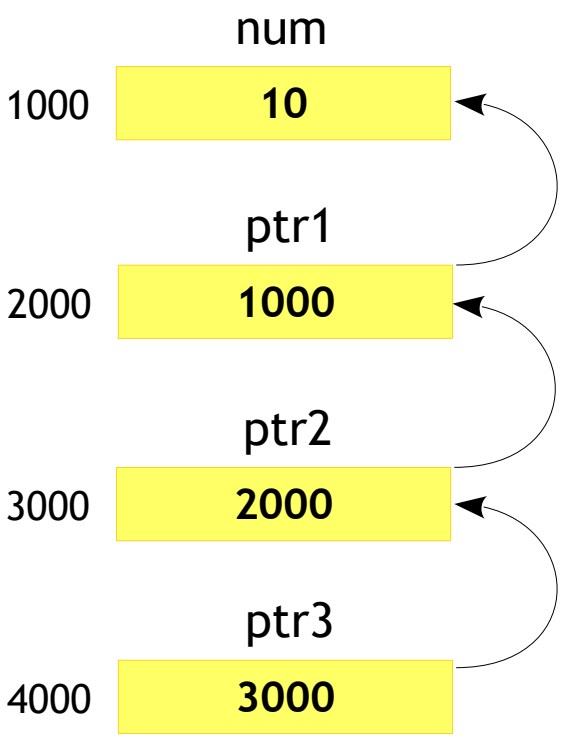
Output → 1000

Advanced C

Pointers - Multilevel

001_example.c

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



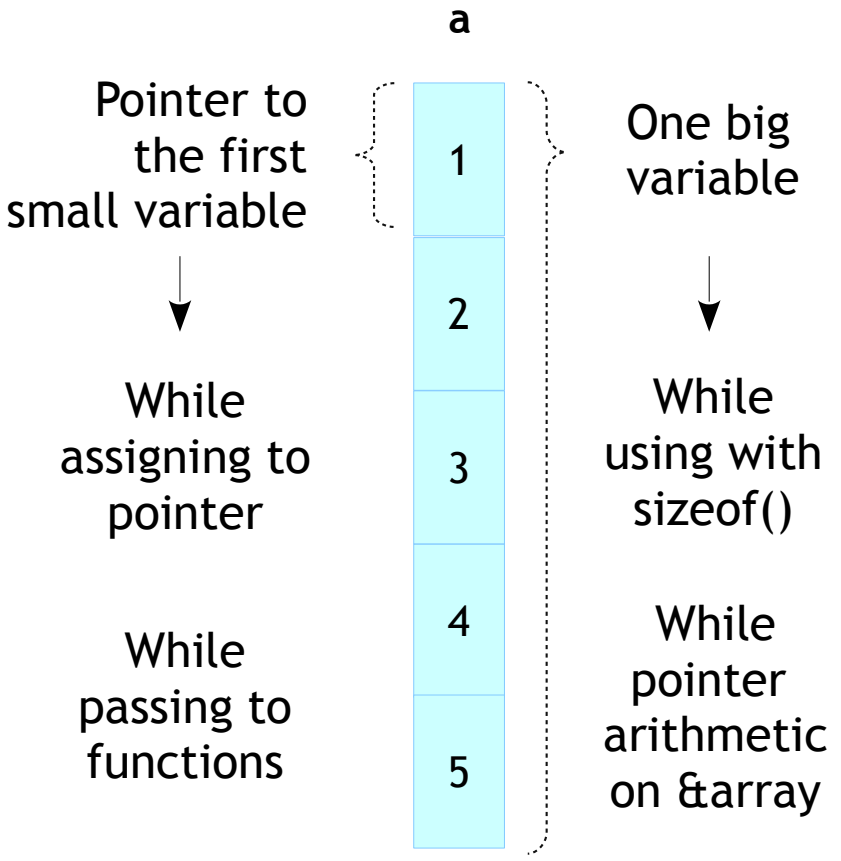
Output → 10

Advanced C

Arrays - Interpretations

Example

```c
#include <stdio.h>

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

    return 0;
}
```

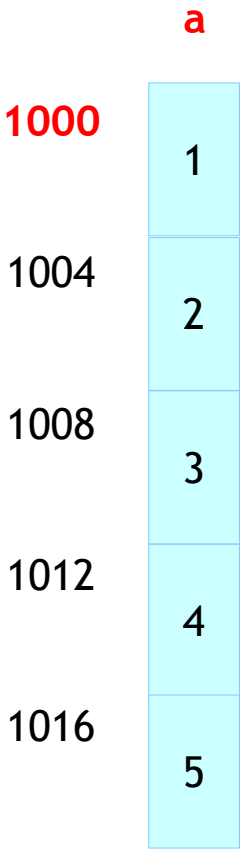


Advanced C

Arrays - Interpretations

002_example.c

```c
#include <stdio.h>

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

    printf("%p\n", a);
    printf("%p\n", &a[0]);
    printf("%p\n", &a);

    return 0;
}
```



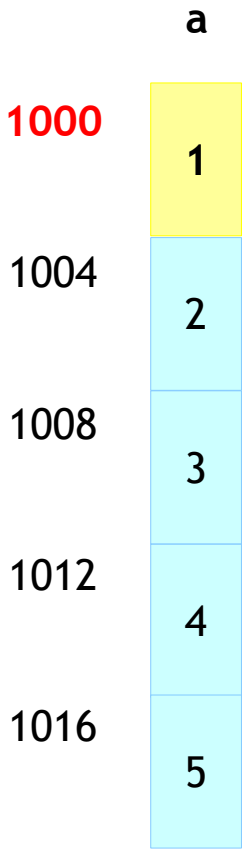
EMERTXE

Advanced C

Arrays - Interpretations

002_example.c

```c
#include <stdio.h>

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

    printf("%p\n", a);
    printf("%p\n", &a[0]);
    printf("%p\n", &a);

    return 0;
}
```

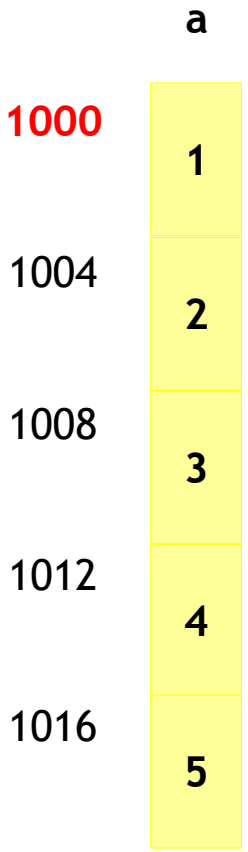


Advanced C

Arrays - Interpretations

002_example.c

```c
#include <stdio.h>

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

    printf("%p\n", a);
    printf("%p\n", &a[0]);
    printf("%p\n", &a);

    return 0;
}
```

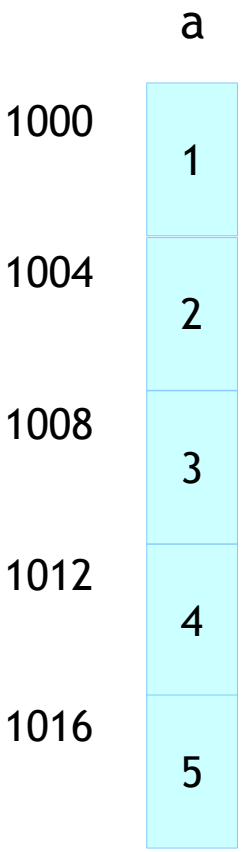


Advanced C

Arrays - Interpretations

003_example.c

```c
#include <stdio.h>

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

    printf("%p\n", a + 1);
    printf("%p\n", &a[0] + 1);
    printf("%p\n", &a + 1);

    return 0;
}
```



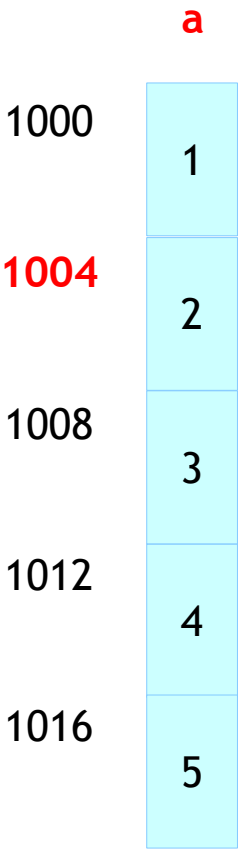
EMERTXE

Advanced C

Arrays - Interpretations

003_example.c

```c
#include <stdio.h>

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

    printf("%p\n", a + 1);
    printf("%p\n", &a[0] + 1);
    printf("%p\n", &a + 1);

    return 0;
}
```



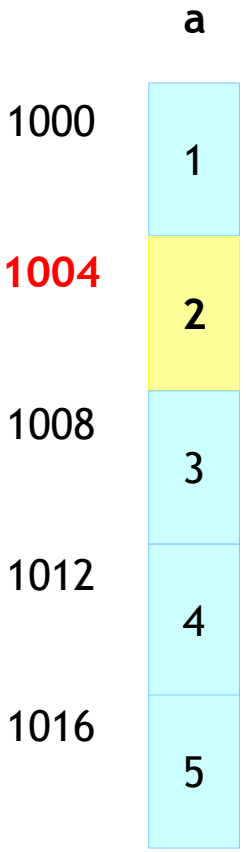
EMERTXE

Advanced C

Arrays - Interpretations

003_example.c

```c
#include <stdio.h>

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

    printf("%p\n", a + 1);
    printf("%p\n", &a[0] + 1);
    printf("%p\n", &a + 1);

    return 0;
}
```



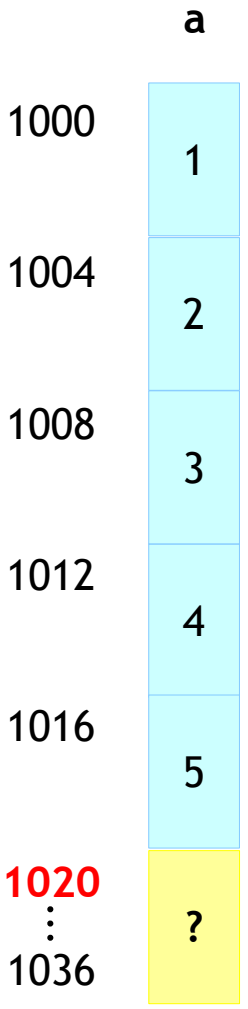
EMERTXE

Advanced C

Arrays - Interpretations

003_example.c

```c
#include <stdio.h>

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

    printf("%p\n", a + 1);
    printf("%p\n", &a[0] + 1);
    printf("%p\n", &a + 1);

    return 0;
}
```



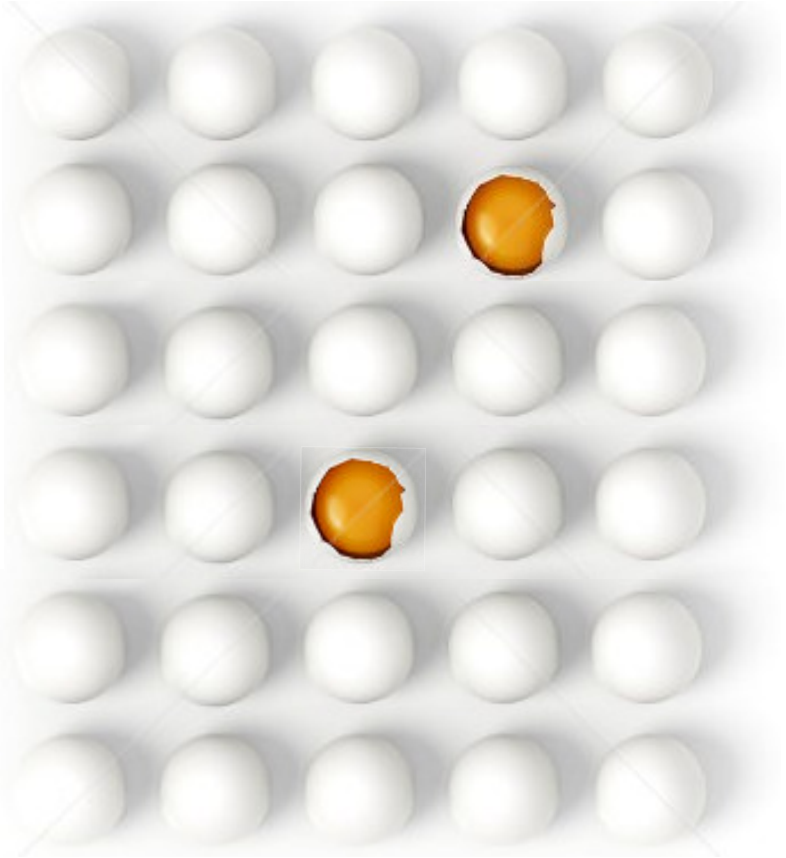
Advanced C

Arrays - Interpretations

- So in summary, if we try to print the address of a[]
  - a - prints the value of the constant pointer
  - &a[0] - prints the address of the first element pointed by a
  - &a - prints the address of the whole array which pointed by a
- Hence all the lines will print **1000** as output
- These concepts plays a very important role in multi dimension arrays

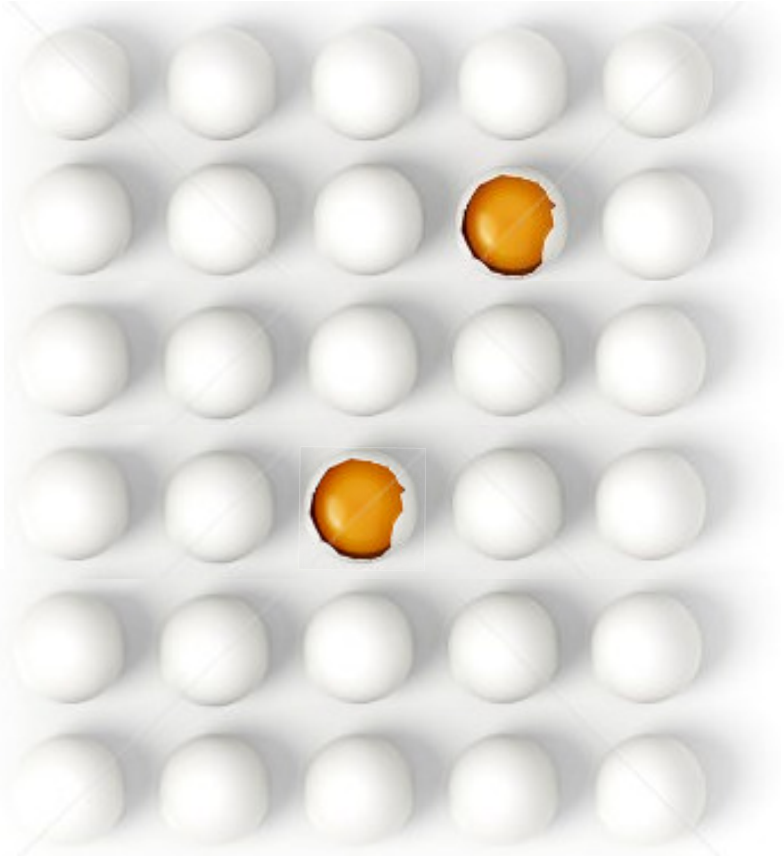
Advanced C

Arrays



Advanced C

Arrays – 2D

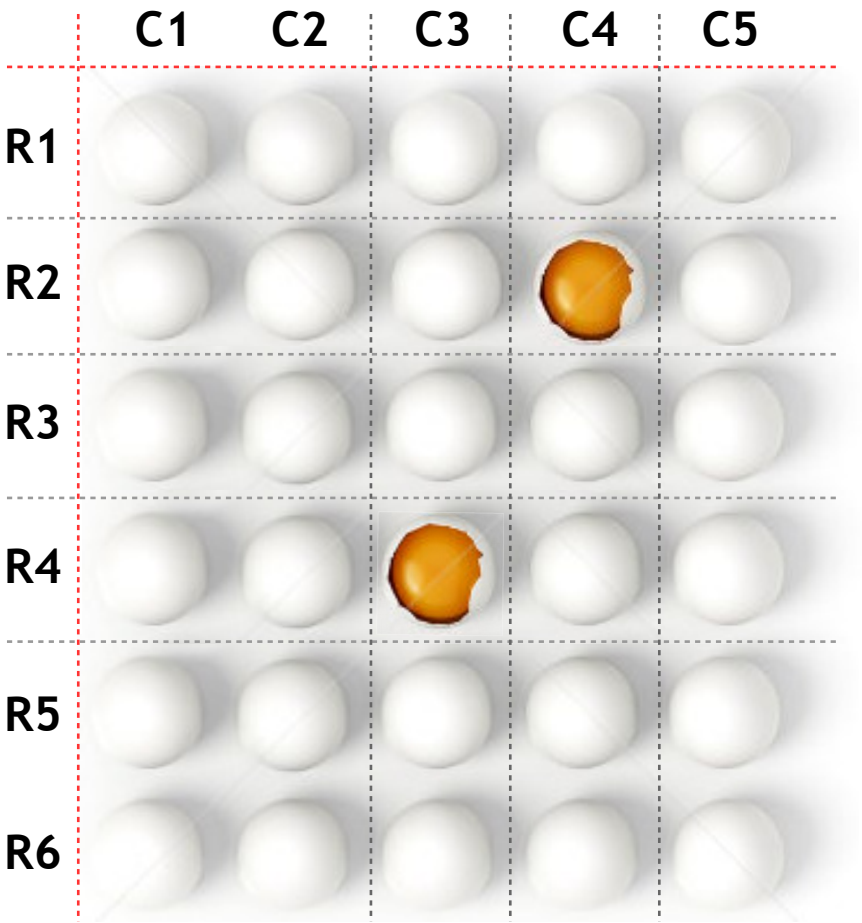


- Find the broken eggs!

- Hmm, how should I proceed with count??

Advanced C

Arrays - 2D

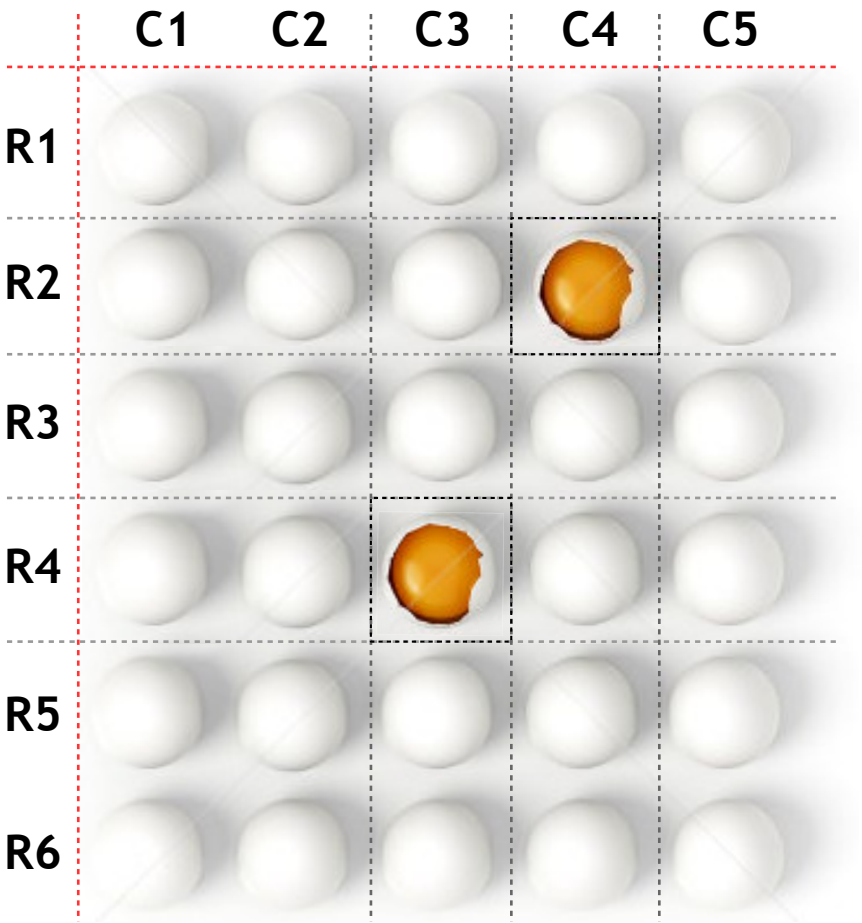


- Now is it better to tell which one broken??

EMERTXE

Advanced C

Arrays - 2D



- So in matrix method it becomes bit easy to locate items
- In other terms we can reference the location with easy indexing
- In this case we can say the broken eggs are at

  R2-C4 and R4-C3

  or

  C4-R2 and C3-R4

Advanced C

Arrays – 2D

- The matrix in computer memory is a bit tricky!!
- Why?. Since its a sequence of memory
- So pragmatically, it is a concept of dimensions is generally referred
- The next slide illustrates the expectation and the reality of the memory layout of the data in a system

Advanced C

Arrays - 2D

Concept Illustration

	C0	C1	C2	C3
R0	123	9	234	39
R1	23	155	33	2
R2	100	88	8	111
R3	201	101	187	22

System Memory

1001	123
1002	9
1003	234
1004	39
1005	23
1006	155
1007	33
1008	2
1009	100
1010	88
1011	8
1012	111
1013	201
1014	101
1015	187
1016	22

Advanced C

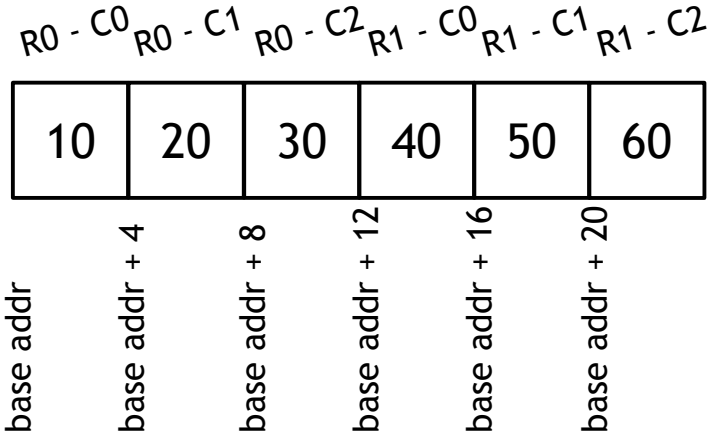
Arrays - 2D

Syntax

```
data_type name[ROW][COL];

Where ROW * COL represents number of elements
Memory occupied by array = (number of elements * size of an element)
                         = (ROW * COL * <size of data_type>)
```

Example

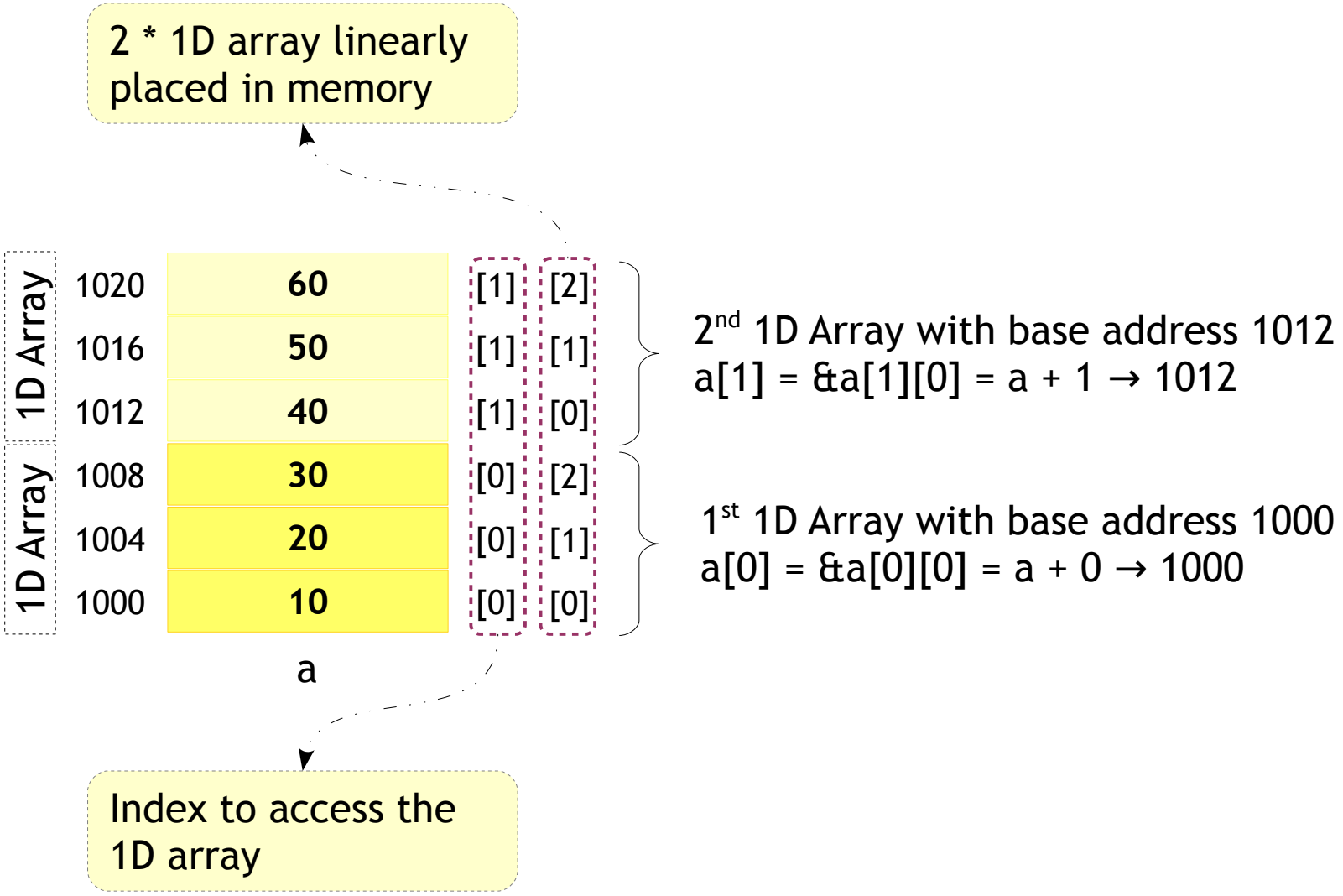
```
int a[2][3] = {{10, 20, 30}, {40, 50, 60}};
```



EMERTXE

Advanced C

Arrays – 2D - Referencing



2nd 1D Array with base address 1012
a[1] = &a[1][0] = a + 1 → 1012

1st 1D Array with base address 1000
a[0] = &a[0][0] = a + 0 → 1000

Advanced C

Arrays – 2D - Dereferencing

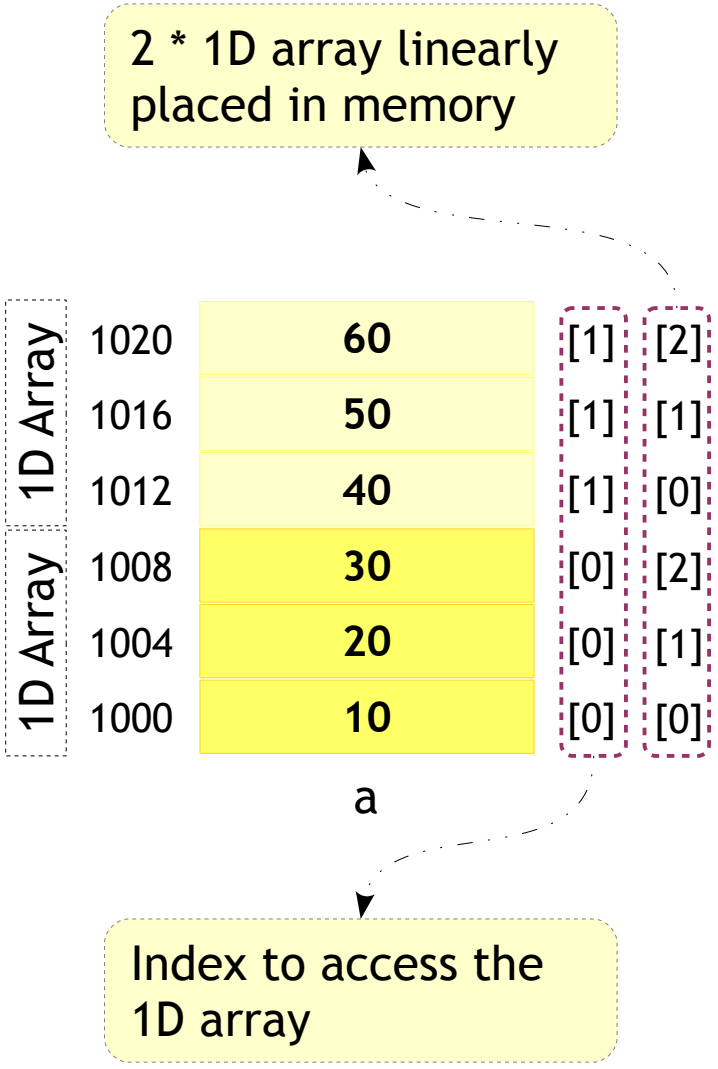
Core Principle

- Dereferencing n^{th} - dimensional array will return $(n - 1)^{th}$ -dimensional array
  - Example : dereferencing 2D array will return 1D array
- Dereferencing 1D array will return 'data element'
  - Example : Dereferencing 1D integer array will return integer

Array	Dimension
&a	n + 1
a	n
*a	n - 1

Advanced C

Arrays – 2D - Dereferencing



Example 1: Say **a[0][1]** is to be accessed, then decomposition happens like,

a[0][1] =

= *(a[0] + (1 * sizeof(type)))

= *(*(a + (0 * sizeof(1D array))) + (1 * sizeof(type)))

= *(*(a + (0 * 12)) + (1 * 4))

= *(*(a + 0) + 4)

= *(*a + 0 + 4)

= *(*a + 4)

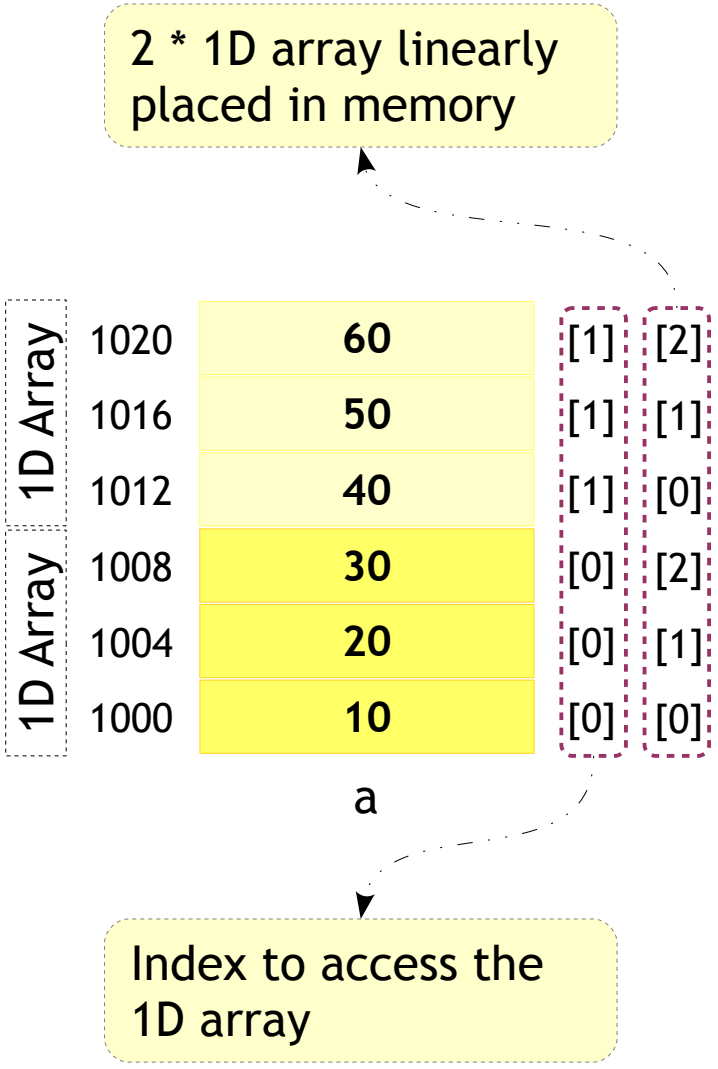
= *(1000 + 4)

= *(1004))

= 20

Advanced C

Arrays - 2D - Dereferencing



Example 1: Say **a[1][1]** is to be accessed, then decomposition happens like,

a[1][1] =

= *(a[1] + (1 * sizeof(type)))

= *(*(a + (1 * sizeof(1D array))) + (1 * sizeof(type)))

= *(*(a + (1 * 12)) + (1 * 4))

= *(*(a + 12) + 4)

= *(*a + 12 + 4)

= *(*a + 16)

= *(1000 + 16)

= *(1016)

= 50

Address of a[r][c] = value(a) + r * sizeof(1D array) + c * sizeof(type)

ΣMERTXE

Advanced C

Arrays – 2D - DIY

- WAP to find the MIN and MAX of a 2D array

Advanced C

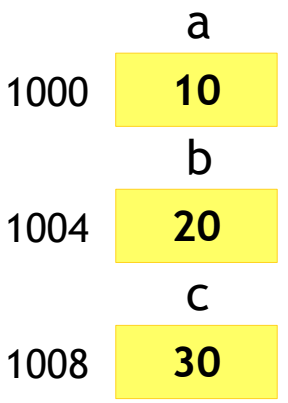
Pointers - Array of pointers

Synatx

```
datatype *ptr_name[SIZE]
```

004_example.c

```c
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;

    int *ptr[3];

    ptr[0] = &a;
    ptr[1] = &b;
    ptr[2] = &c;

    return 0;
}
```



EMERTXE

Advanced C

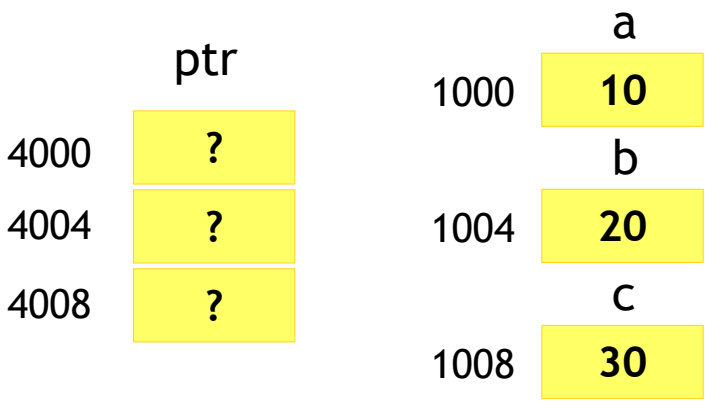
Pointers - Array of pointers

Synatx

```
datatype *ptr_name[SIZE]
```

004_example.c

```
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;

    int *ptr[3];

    ptr[0] = &a;
    ptr[1] = &b;
    ptr[2] = &c;

    return 0;
}
```



Advanced C

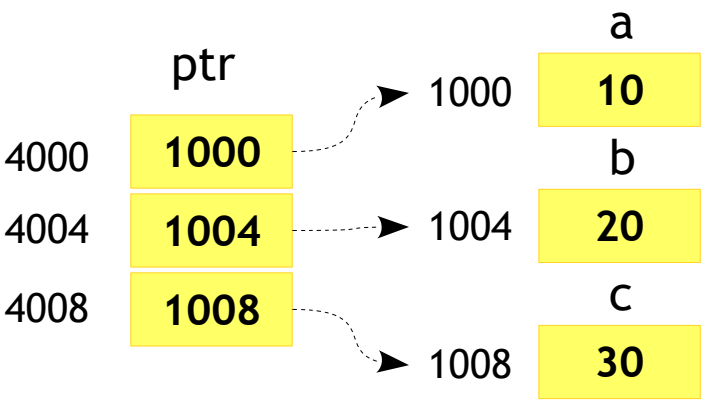
Pointers - Array of pointers

Synatx

```
datatype *ptr_name[SIZE]
```

004_example.c

```c
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;

    int *ptr[3];

    ptr[0] = &a;
    ptr[1] = &b;
    ptr[2] = &c;

    return 0;
}
```



Advanced C

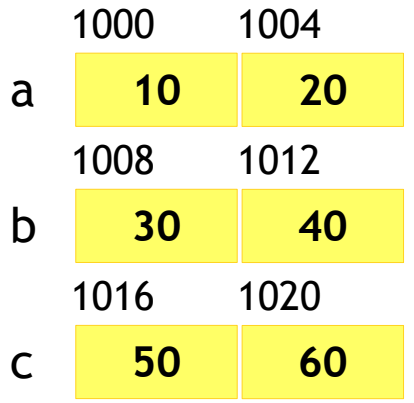
Pointers - Array of pointers

Synatx

```
datatype *ptr_name[SIZE]
```

005_example.c

```c
#include <stdio.h>

int main()
{
    int a[2] = {10, 20};
    int b[2] = {30, 40};
    int c[2] = {50, 60};

    int *ptr[3];

    ptr[0] = a;
    ptr[1] = b;
    ptr[2] = c;

    return 0;
}
```



EMERTXE

Advanced C

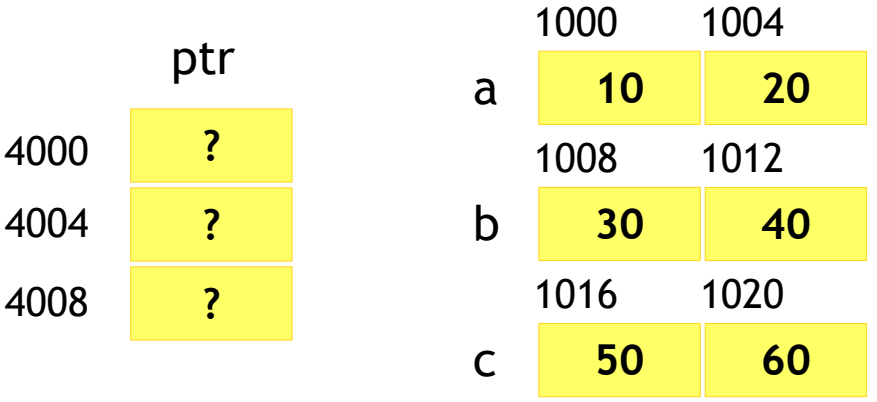
Pointers - Array of pointers

Synatx

```
datatype *ptr_name[SIZE]
```

005_example.c

```
#include <stdio.h>

int main()
{
    int a[2] = {10, 20};
    int b[2] = {30, 40};
    int c[2] = {50, 60};

    int *ptr[3];

    ptr[0] = a;
    ptr[1] = b;
    ptr[2] = c;

    return 0;
}
```



Advanced C

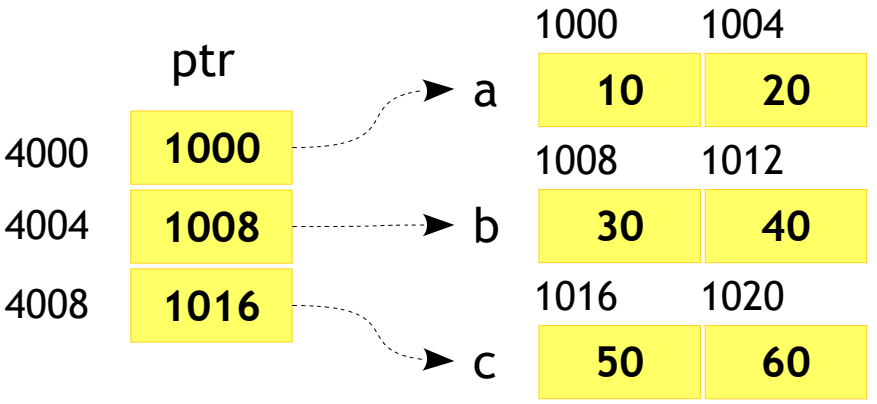
Pointers - Array of pointers

Synatx

```
datatype *ptr_name[SIZE]
```

005_example.c

```c
#include <stdio.h>

int main()
{
    int a[2] = {10, 20};
    int b[2] = {30, 40};
    int c[2] = {50, 60};

    int *ptr[3];

    ptr[0] = a;
    ptr[1] = b;
    ptr[2] = c;

    return 0;
}
```

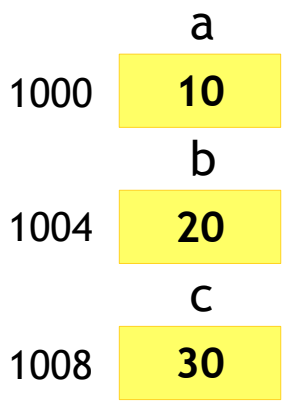


Advanced C

Pointers - Array of pointers

006_example.c

```c
#include <stdio.h>

void print_array(int **p)
{
    int i;

    for (i = 0; i < 3; i++)
    {
        printf("%d ", *p[i]);
        printf("at %p\n", p[i]);
    }
}

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;
    int *ptr[3] = {&a, &b, &c};

    print_array(ptr);

    return 0;
}
```



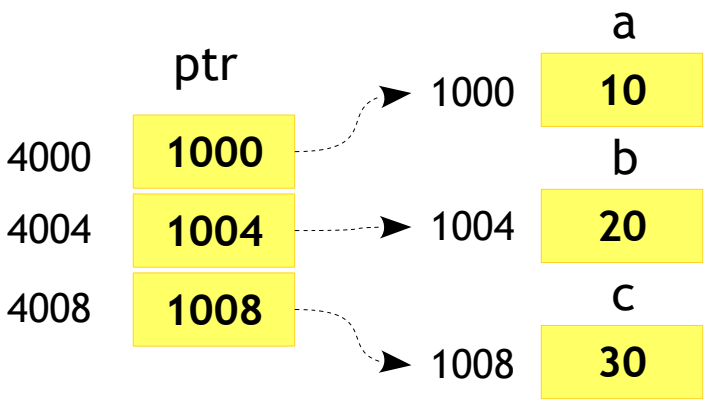
EMERTXE

Advanced C

Pointers - Array of pointers

006_example.c

```c
#include <stdio.h>

void print_array(int **p)
{
     int i;

     for (i = 0; i < 3; i++)
     {
          printf("%d ", *p[i]);
          printf("at %p\n", p[i]);
     }
}

int main()
{
     int a = 10;
     int b = 20;
     int c = 30;
     int *ptr[3] = {&a, &b, &c};

     print_array(ptr);

     return 0;
}
```



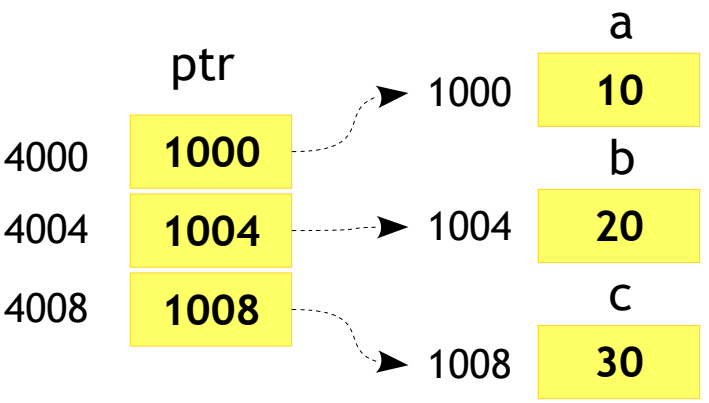
EMERTXE

Advanced C

Pointers - Array of pointers

006_example.c

```c
#include <stdio.h>

void print_array(int **p)
{
    int i;

    for (i = 0; i < 3; i++)
    {
        printf("%d ", *p[i]);
        printf("at %p\n", p[i]);
    }
}

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;
    int *ptr[3] = {&a, &b, &c};

    print_array(ptr);

    return 0;
}
```



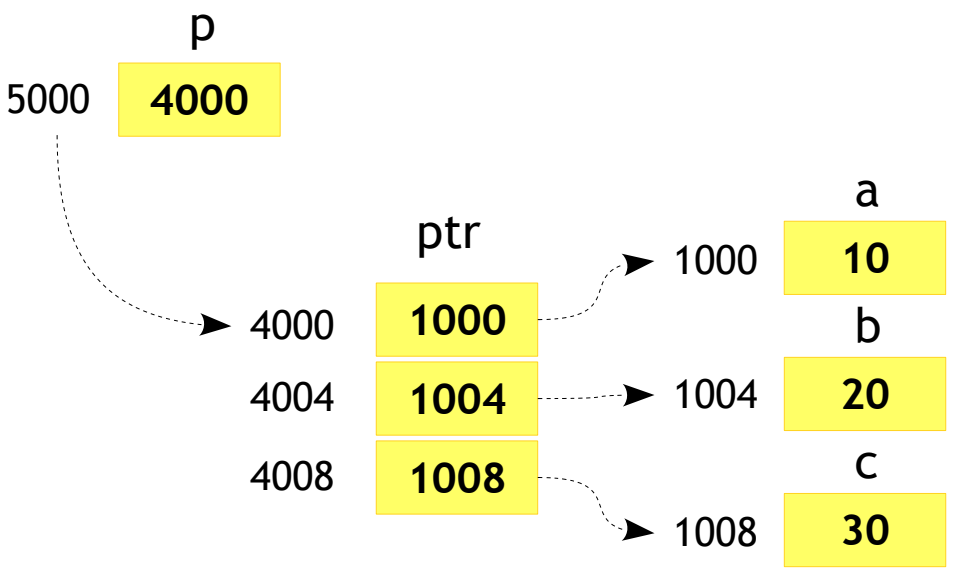
EMERTXE

Advanced C

Pointers - Array of pointers

006_example.c

```c
#include <stdio.h>

void print_array(int **p)
{
    int i;

    for (i = 0; i < 3; i++)
    {
        printf("%d ", *p[i]);
        printf("at %p\n", p[i]);
    }
}

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;
    int *ptr[3] = {&a, &b, &c};

    print_array(ptr);

    return 0;
}
```

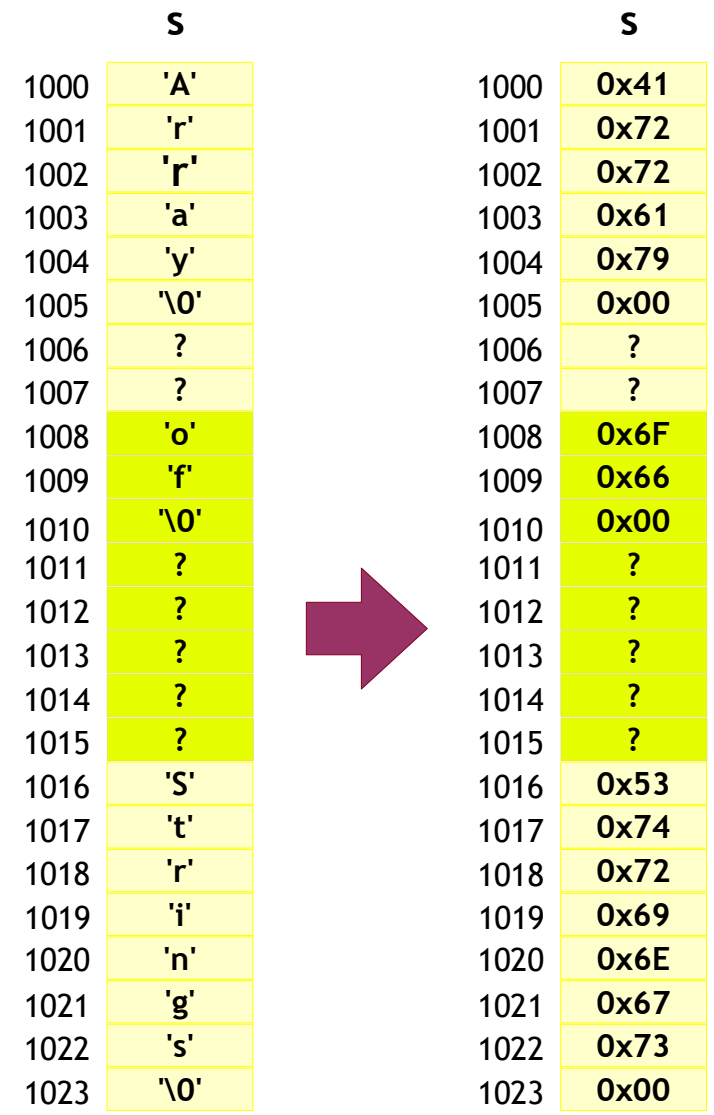


Advanced C

Pointers - Array of strings

007_example.c

```c
#include <stdio.h>

int main()
{
    char s[3][8] = {
                    "Array",
                    "of",
                    "Strings"
                };

    printf("%s %s %s\n", s[0], s[1], s[2]);

    return 0;
}
```

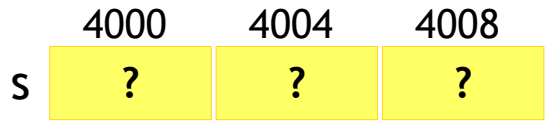


Advanced C

Pointers - Array of strings

008_example.c

```c
#include <stdio.h>

int main()
{
    char *s[3];

    s[0] = "Array";
    s[1] = "of";
    s[2] = "Strings";

    printf("%s %s %s\n", s[0], s[1], s[2]);

    return 0;
}
```



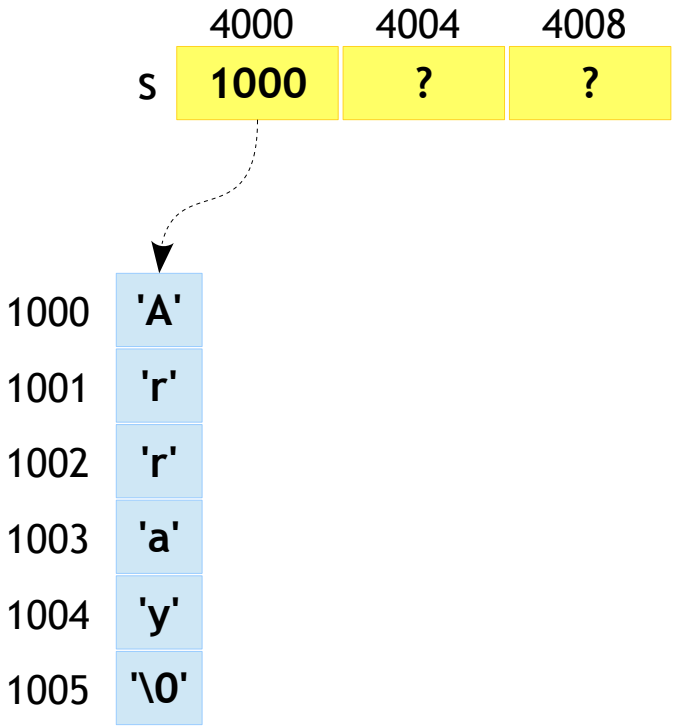
EMERTXE

Advanced C

Pointers - Array of strings

008_example.c

```c
#include <stdio.h>

int main()
{
    char *s[3];

    s[0] = "Array";
    s[1] = "of";
    s[2] = "Strings";

    printf("%s %s %s\n", s[0], s[1], s[2]);

    return 0;
}
```

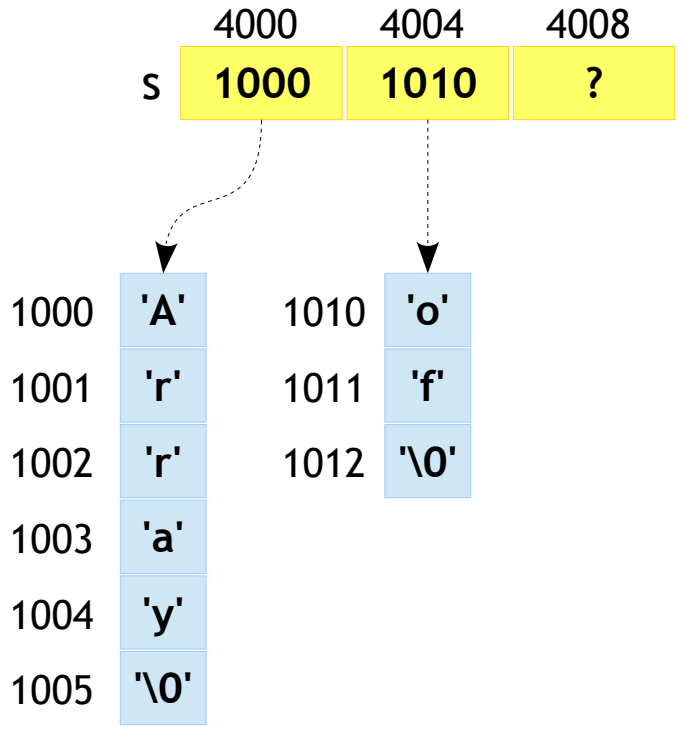


Advanced C

Pointers - Array of strings

008_example.c

```c
#include <stdio.h>

int main()
{
    char *s[3];

    s[0] = "Array";
    s[1] = "of";
    s[2] = "Strings";

    printf("%s %s %s\n", s[0], s[1], s[2]);

    return 0;
}
```

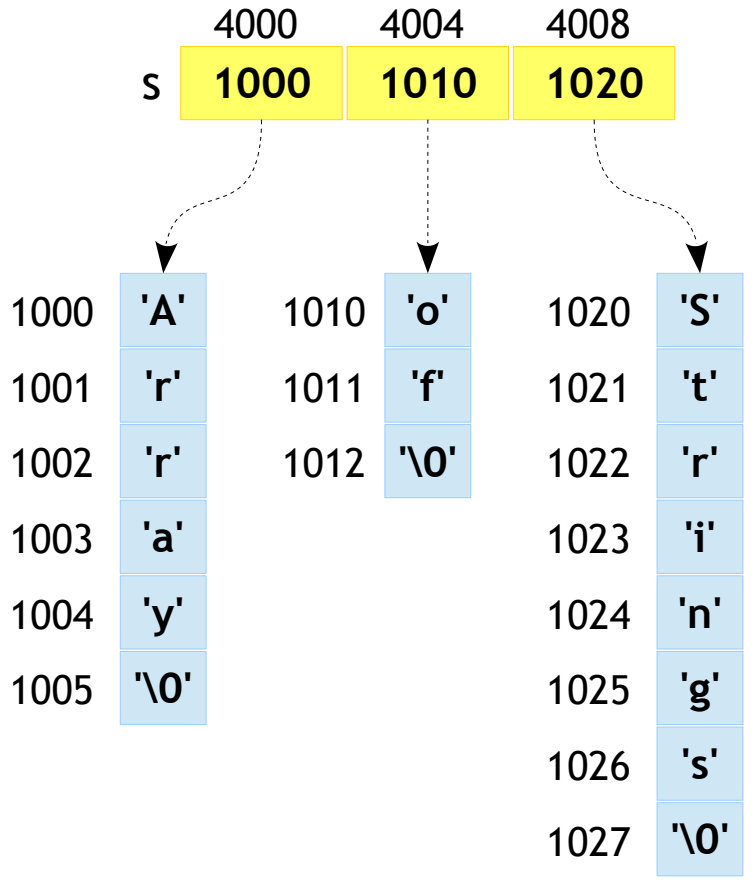


Advanced C

Pointers - Array of strings

008_example.c

```c
#include <stdio.h>

int main()
{
    char *s[3];

    s[0] = "Array";
    s[1] = "of";
    s[2] = "Strings";

    printf("%s %s %s\n", s[0], s[1], s[2]);

    return 0;
}
```



EMERTXE

Advanced C

Pointers - Array of strings

- W.A.P to print menu and select an option
  - Menu options { File, Edit, View, Insert, Help }
- The prototype of print_menu function
  - void print_menu (char **menu);

Screen Shot

```
user@user:~]
user@user:~]./a.out
1. File
2. Edit
3. View
4. Insert
5. Help
Select your option: 2
You have selected Edit Menu
user@user:~]
```

Advanced C

Pointers – Array of strings

- Command line arguments
  - Refer to PPT “11_functions_part2”

Advanced C

Pointers - Pointer to an Array

Syntax

```
datatype (*ptr_name)[SIZE];
```

009_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int *ptr;

    ptr = array;

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

    return 0;
}
```

- Pointer to an array!!, why is the syntax so weird??
- Isn't the code shown left is an example for pointer to an array?
- Should the code print as 1 in output?
- Yes, everything is fine here except the dimension of the array.
- This is perfect code for 1D array

EMERTXE

Advanced C

Pointers - Pointer to an Array

Syntax

```
datatype (*ptr_name)[SIZE];
```

010_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int (*ptr)[3];

    ptr = array;

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

    return 0;
}
```

- So in order to point to 2D array we would prefer the given syntax
- Ookay, Isn't a 2D array linearly arranged in the memory?

  So can I write the code as shown?
- Hmm!, Yes but the compiler would warn you on the assignment statement
- Then how should I write?

EMERTXE

Advanced C

Pointers - Pointer to an Array

Syntax

```
datatype (*ptr_name)[SIZE];
```

011_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int (*ptr)[3];

    ptr = &array;

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

    return 0;
}
```

- Hhoho, isn't **array** is equal to **&array**?? what is the difference?
- Well the difference lies in the compiler interpretation while pointer arithmetic and hence
- Please see the difference in the next slides

EMERTXE

Advanced C

Pointers - Pointer to an Array

012_example.c

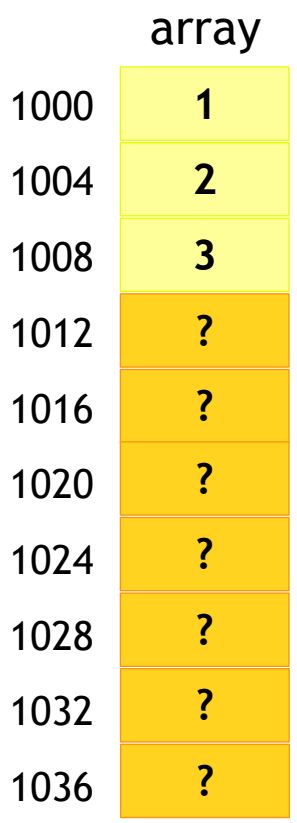
```
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```

Advanced C

Pointers - Pointer to an Array

012_example.c

```c
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



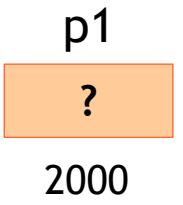
EMERTXE

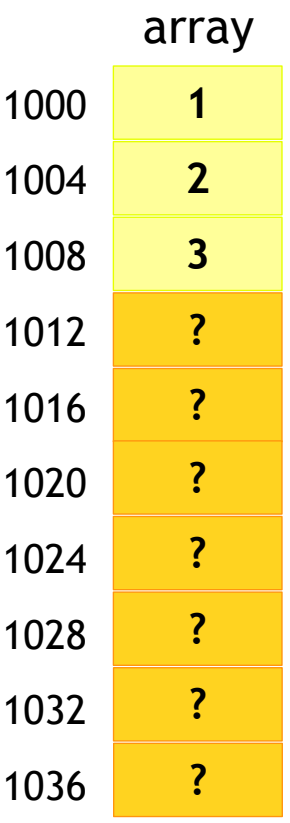
Advanced C

Pointers - Pointer to an Array

012_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```





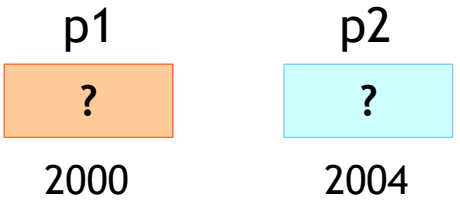
EMERTXE

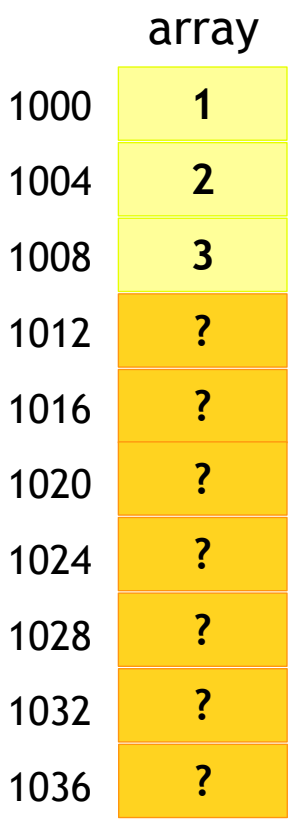
Advanced C

Pointers - Pointer to an Array

012_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



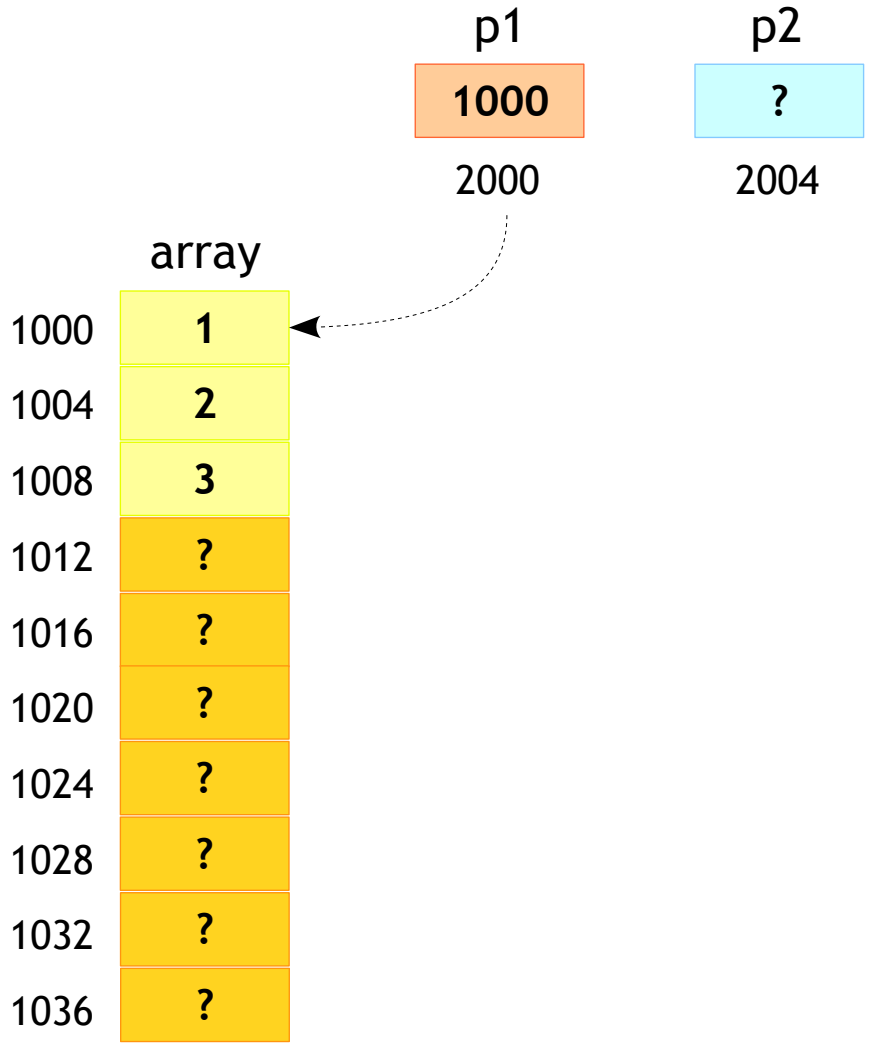


Advanced C

Pointers - Pointer to an Array

012_example.c

```c
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



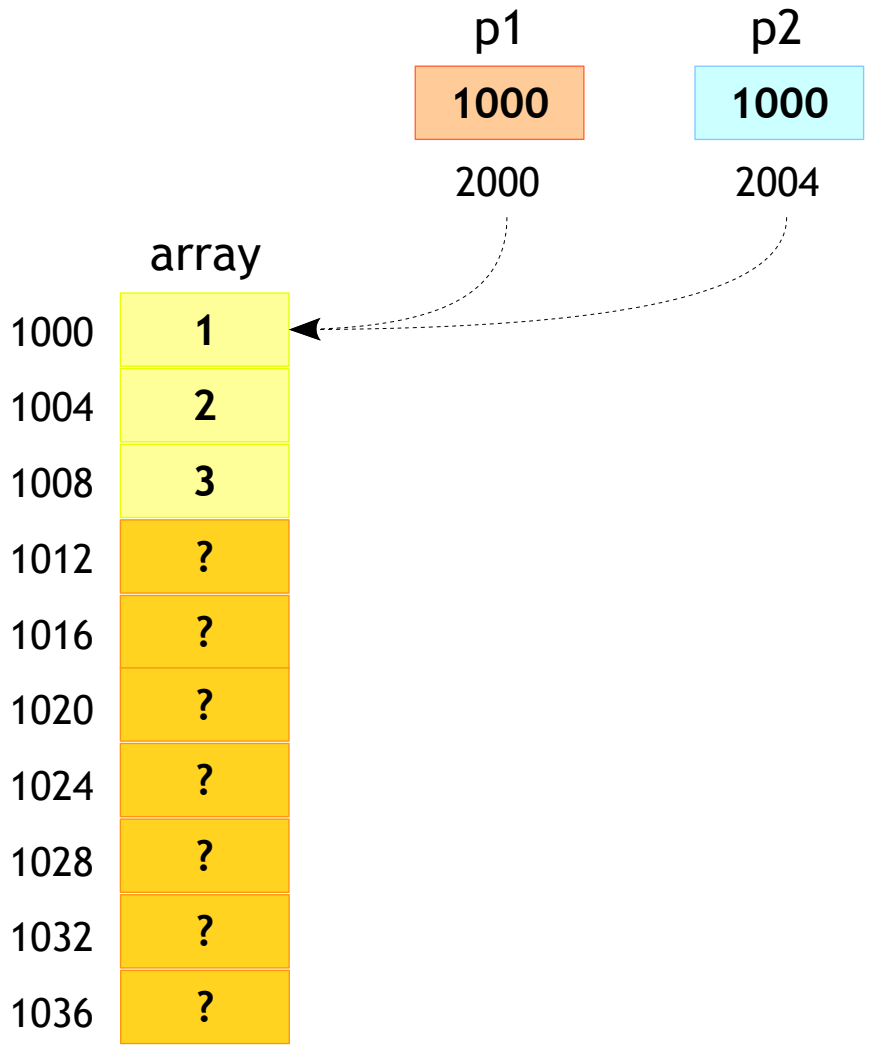
EMERTXE

Advanced C

Pointers - Pointer to an Array

012_example.c

```c
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



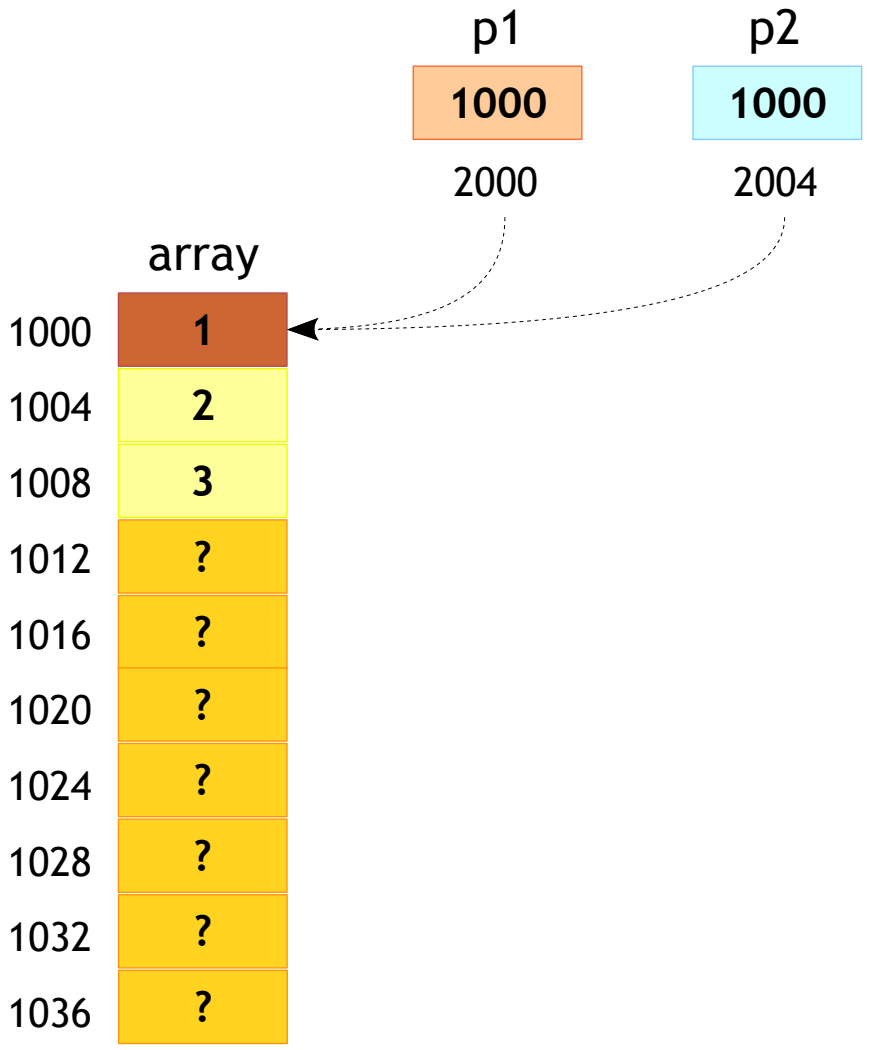
EMERTXE

Advanced C

Pointers - Pointer to an Array

012_example.c

```c
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```

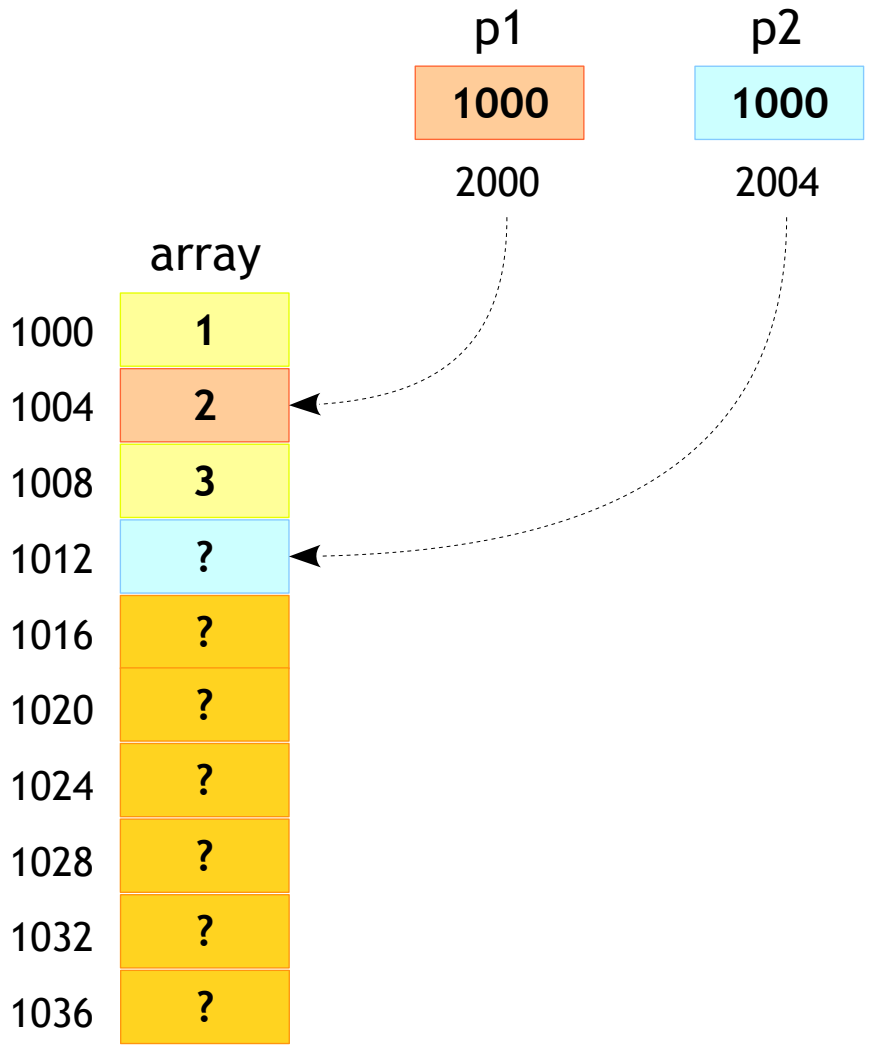


Advanced C

Pointers - Pointer to an Array

012_example.c

```c
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```

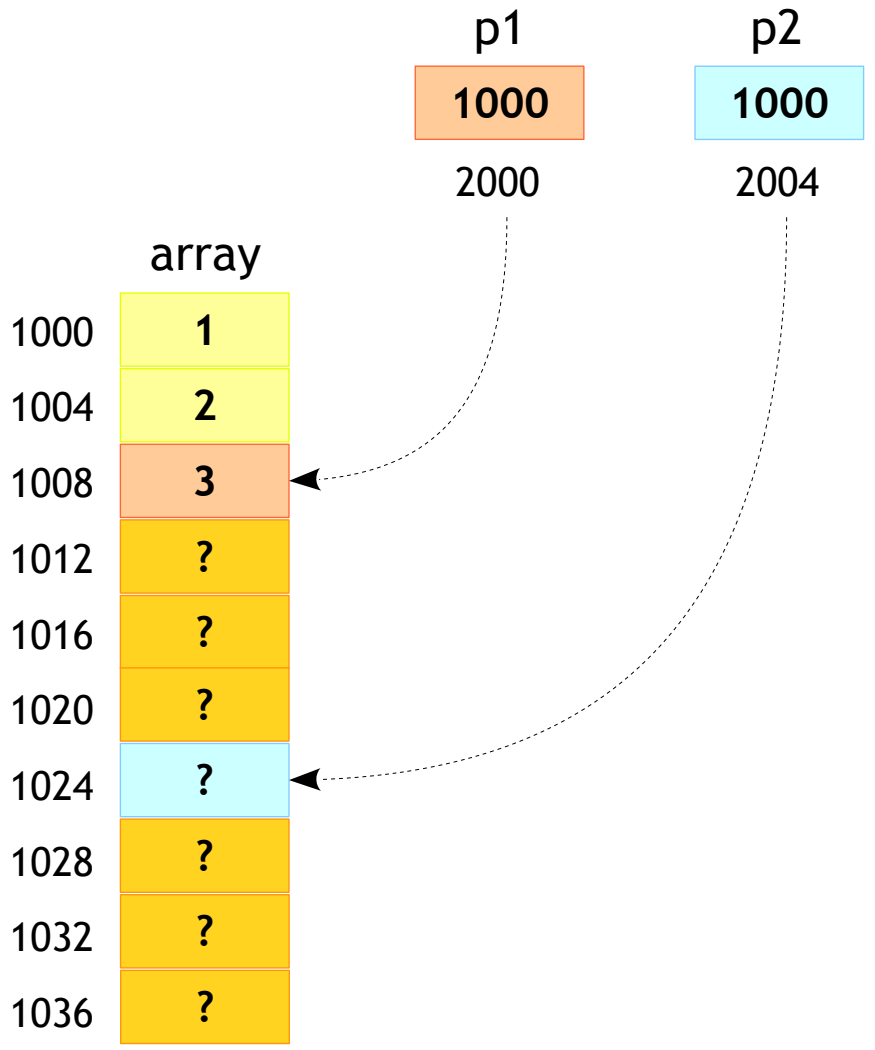


Advanced C

Pointers - Pointer to an Array

012_example.c

```c
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



Advanced C

Pointers – Pointer to an Array

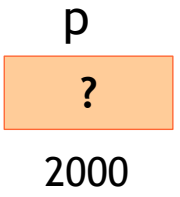
- So as a conclution we can say the
  - Pointer arithmetic on 1D array is based on the size of datatype
  - Pointer arithmetic on 2D array is based on the size of datatype and size of 1D array
- Still one question remains is what is real use of this syntax if can do p[i][j]?
  - In case of dynamic memory allocation as shown in next slide

Advanced C

Pointers - Pointer to an Array

013_example.c

```
int main()
{
    int (*p)[3];

    p = malloc(sizeof(*p) * 3);

    (*(p + 0))[0] = 1;
    (*(p + 1))[1] = 2;
    (*(p + 2))[2] = 3;

    printf("%d\n", p[0][0]);
    printf("%d\n", p[1][1]);
    printf("%d\n", p[2][2]);

    return 0;
}
```

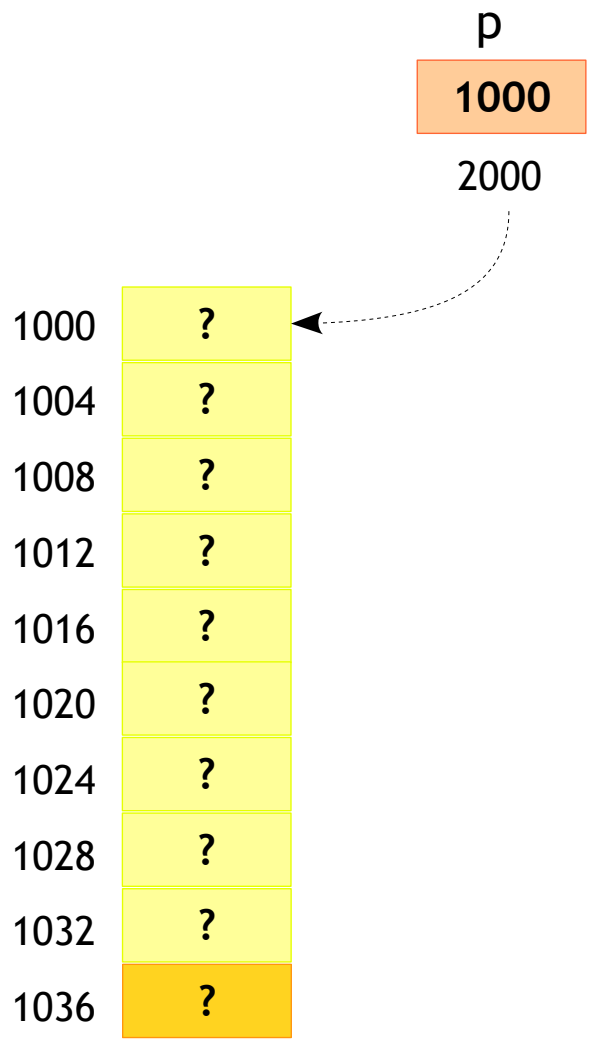


Advanced C

Pointers - Pointer to an Array

013_example.c

```
int main()
{
    int (*p)[3];

    p = malloc(sizeof(*p) * 3);

    (*(p + 0))[0] = 1;
    (*(p + 1))[1] = 2;
    (*(p + 2))[2] = 3;

    printf("%d\n", p[0][0]);
    printf("%d\n", p[1][1]);
    printf("%d\n", p[2][2]);

    return 0;
}
```



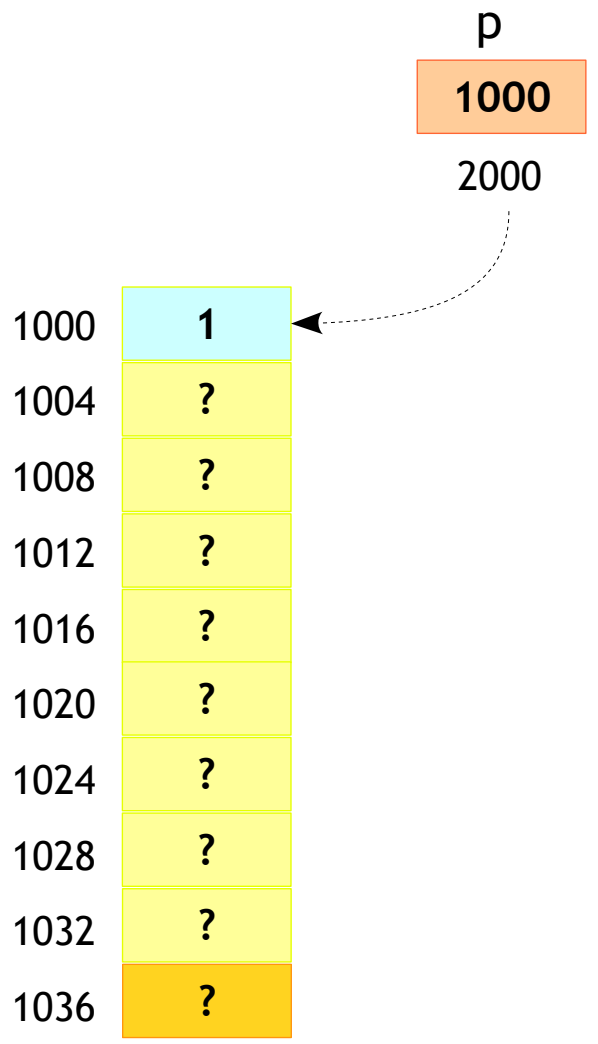
EMERTXE

Advanced C

Pointers - Pointer to an Array

013_example.c

```
int main()
{
    int (*p)[3];

    p = malloc(sizeof(*p) * 3);

    (*(p + 0))[0] = 1;
    (*(p + 1))[1] = 2;
    (*(p + 2))[2] = 3;

    printf("%d\n", p[0][0]);
    printf("%d\n", p[1][1]);
    printf("%d\n", p[2][2]);

    return 0;
}
```



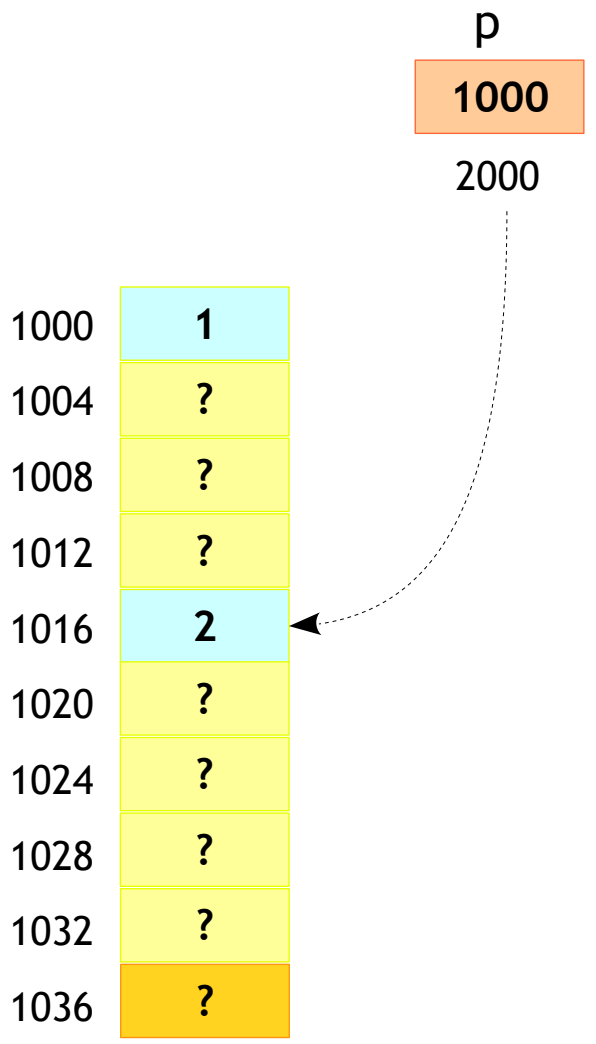
EMERTXE

Advanced C

Pointers - Pointer to an Array

013_example.c

```
int main()
{
    int (*p)[3];

    p = malloc(sizeof(*p) * 3);

    (*(p + 0))[0] = 1;
    (*(p + 1))[1] = 2;
    (*(p + 2))[2] = 3;

    printf("%d\n", p[0][0]);
    printf("%d\n", p[1][1]);
    printf("%d\n", p[2][2]);

    return 0;
}
```



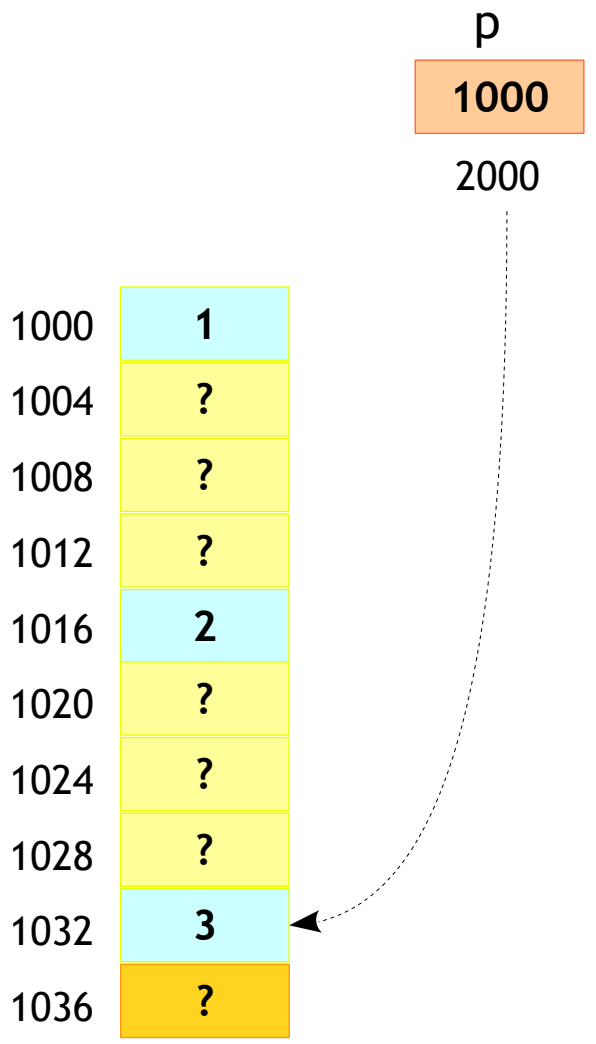
EMERTXE

Advanced C

Pointers - Pointer to an Array

013_example.c

```
int main()
{
    int (*p)[3];

    p = malloc(sizeof(*p) * 3);

    (*(p + 0))[0] = 1;
    (*(p + 1))[1] = 2;
    (*(p + 2))[2] = 3;

    printf("%d\n", p[0][0]);
    printf("%d\n", p[1][1]);
    printf("%d\n", p[2][2]);

    return 0;
}
```

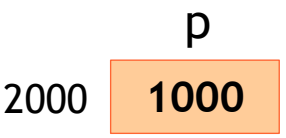


Advanced C

Pointers - Pointer to an 2D Array

014_example.c

```c
int main()
{
    int (*p)[3];
    int a[2][3] = {{1, 2, 3}, {4, 5, 6}};

    p = a;

    return 0;
}
```

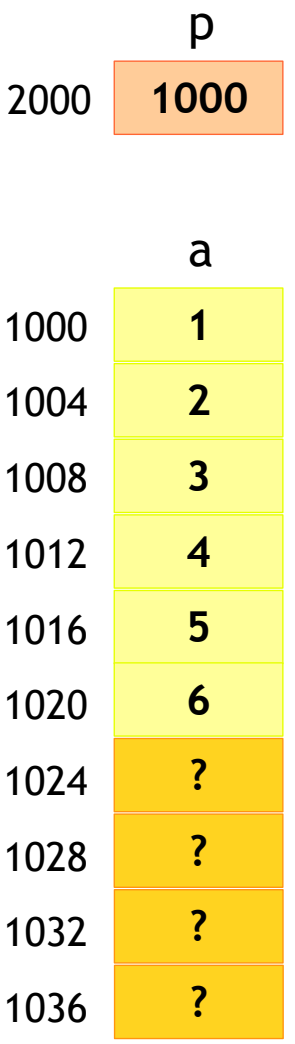


Advanced C

Pointers - Pointer to an 2D Array

014_example.c

```c
int main()
{
    int (*p)[3];
    int a[2][3] = {{1, 2, 3}, {4, 5, 6}};

    p = a;

    return 0;
}
```

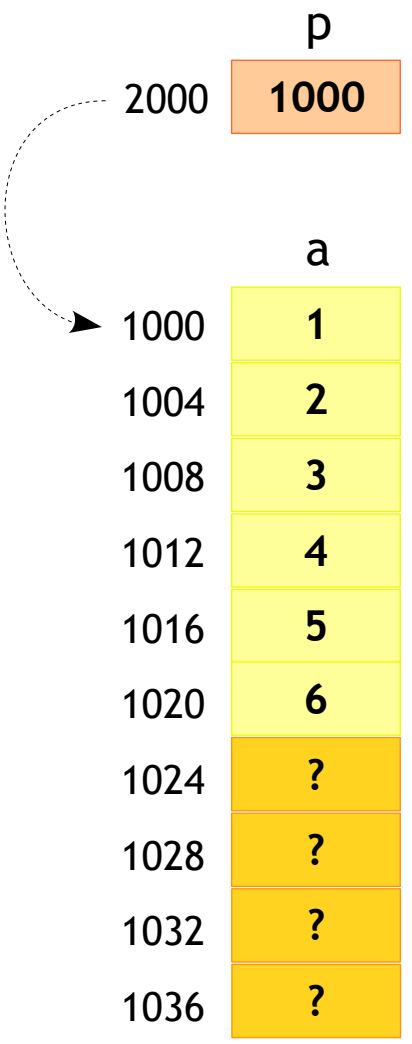


Advanced C

Pointers - Pointer to an 2D Array

014_example.c

```c
int main()
{
    int (*p)[3];
    int a[2][3] = {{1, 2, 3}, {4, 5, 6}};

    p = a;

    return 0;
}
```



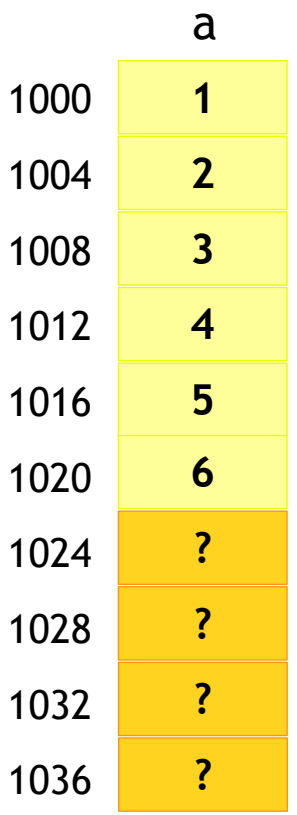
EMERTXE

Advanced C

Pointers - Passing 2D array to function

015_example.c

```c
#include <stdio.h>

void print_array(int p[2][3])
{
    int i, j;

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 3; j++)
        {
            printf("%d\n", p[i][j]);
        }
    }
}

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

    print_array(a);

    return 0;
}
```



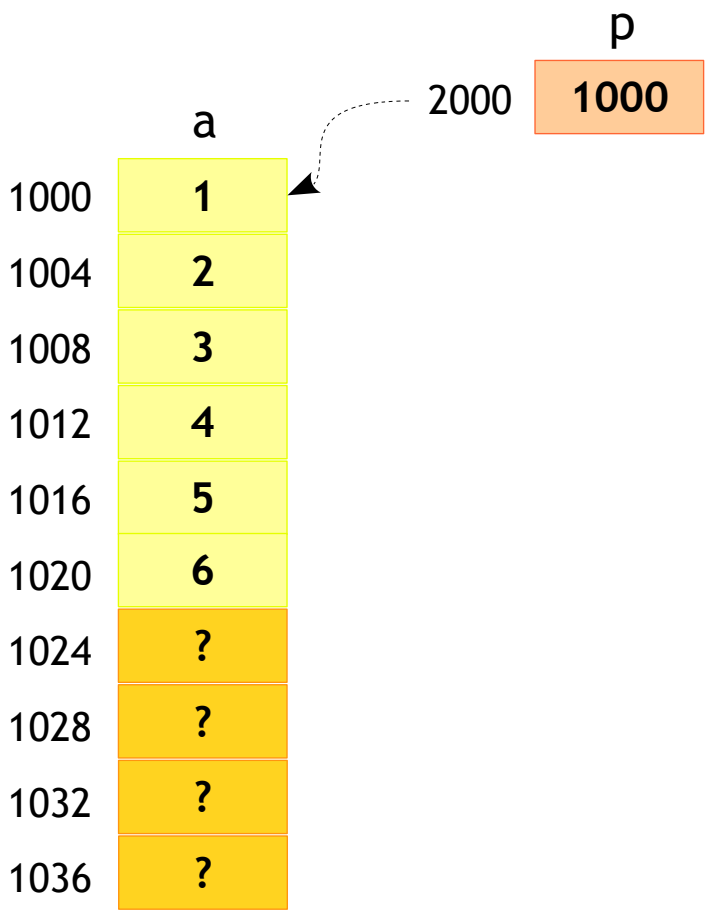
EMERTXE

Advanced C

Pointers - Passing 2D array to function

015_example.c

```c
#include <stdio.h>

void print_array(int p[2][3])
{
    int i, j;

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 3; j++)
        {
            printf("%d\n", p[i][j]);
        }
    }
}

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

    print_array(a);

    return 0;
}
```

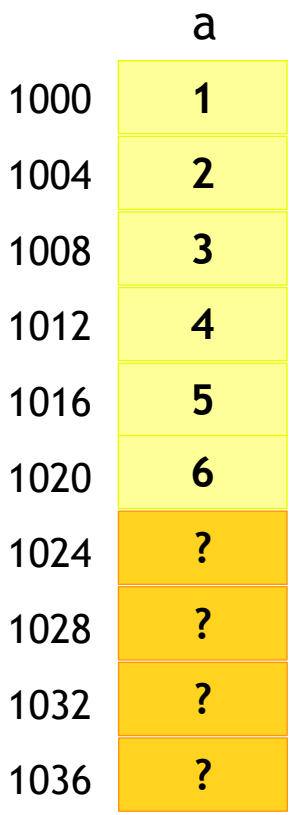


Advanced C

Pointers - Passing 2D array to function

016_example.c

```c
#include <stdio.h>

void print_array(int (*p)[3])
{
    int i, j;

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 3; j++)
        {
            printf("%d\n", p[i][j]);
        }
    }
}

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

    print_array(a);

    return 0;
}
```



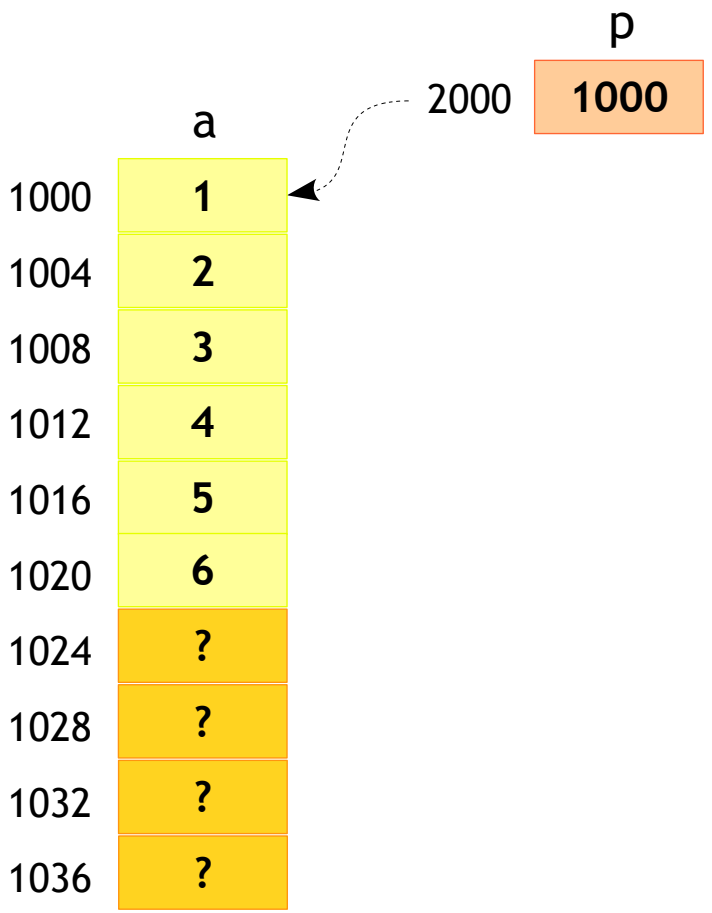
EMERTXE

Advanced C

Pointers - Passing 2D array to function

016_example.c

```c
#include <stdio.h>

void print_array(int (*p)[3])
{
    int i, j;

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 3; j++)
        {
            printf("%d\n", p[i][j]);
        }
    }
}

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

    print_array(a);

    return 0;
}
```



EMERTXE

Advanced C

Pointers - Passing 2D array to function

017_example.c

```c
#include <stdio.h>

void print_array(int row, int col, int (*p)[col])
{
    int i, j;

    for (i = 0; i < row; i++)
    {
        for (j = 0; j < col; j++)
        {
            printf("%d\n", p[i][j]);
        }
    }
}

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

    print_array(2, 3, a);

    return 0;
}
```

Advanced C

Pointers - Passing 2D array to function

018_example.c

```c
#include <stdio.h>

void print_array(int row, int col, int *p)
{
    int i, j;

    for (i = 0; i < row; i++)
    {
        for (j = 0; j < col; j++)
        {
            printf("%d\n", *((p + i * col) + j));
        }
    }
}

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

    print_array(2, 3, (int *) a);

    return 0;
}
```

EMERTXE

Advanced C

Pointers – 2D Array Creations

- Each Dimension could be Static or Dynamic
- Possible combination of creation could be
  - BS: Both Static (Rectangular)
  - FSSD: First Static, Second Dynamic
  - FDSS: First Dynamic, Second Static
  - BD: Both Dynamic

Advanced C

Pointers - 2D Array Creations - BS

018_example.c

```c
#include <stdio.h>

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

    return 0;
}
```

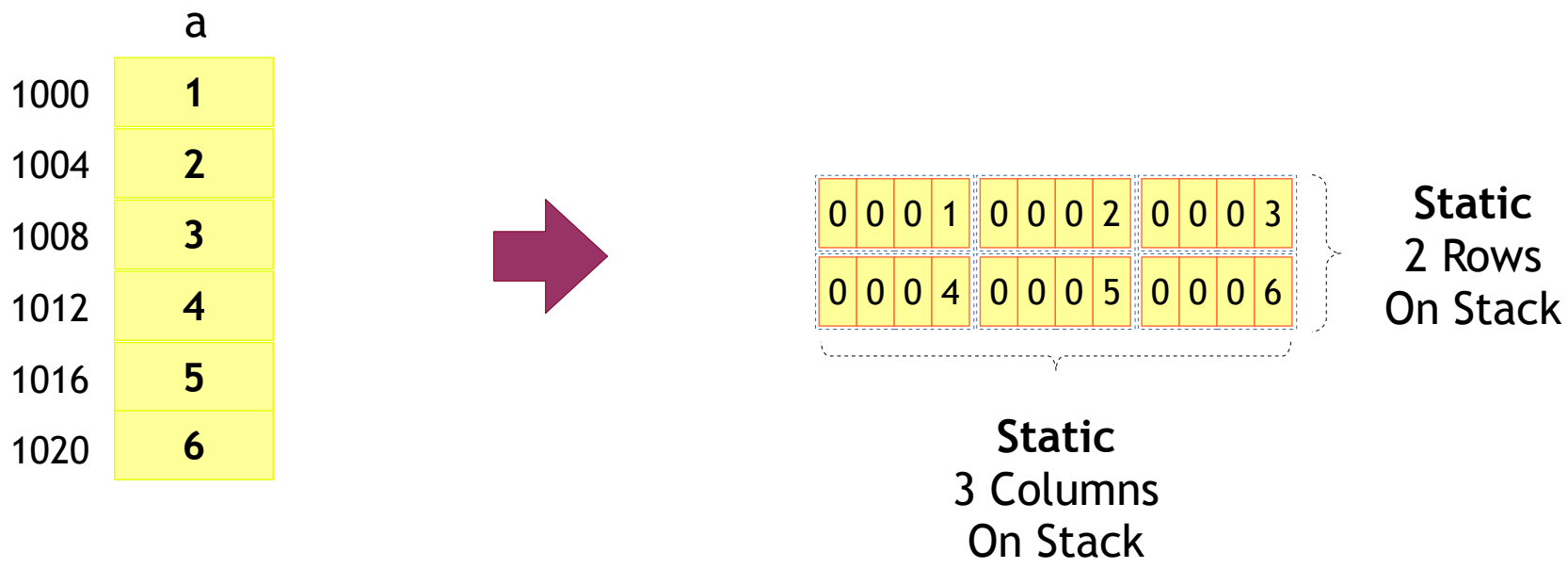
- Both Static (BS)
- Called as an rectangular array
- Total size is

  2 * 3 * sizeof(datatype)

  2 * 3 * 4 = 24 Bytes
- The memory representation can be as shown in next slide

Advanced C

Pointers – 2D Array Creations - BS



Advanced C

Pointers - 2D Array Creations - FSSD

019_example.c

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *a[2];

    for ( i = 0; i < 2; i++)
    {
        a[i] = malloc(3 * sizeof(int));
    }

    return 0;
}
```

- First Static and Second Dynamic (FSSD)
- Mix of Rectangular & Ragged
- Total size is

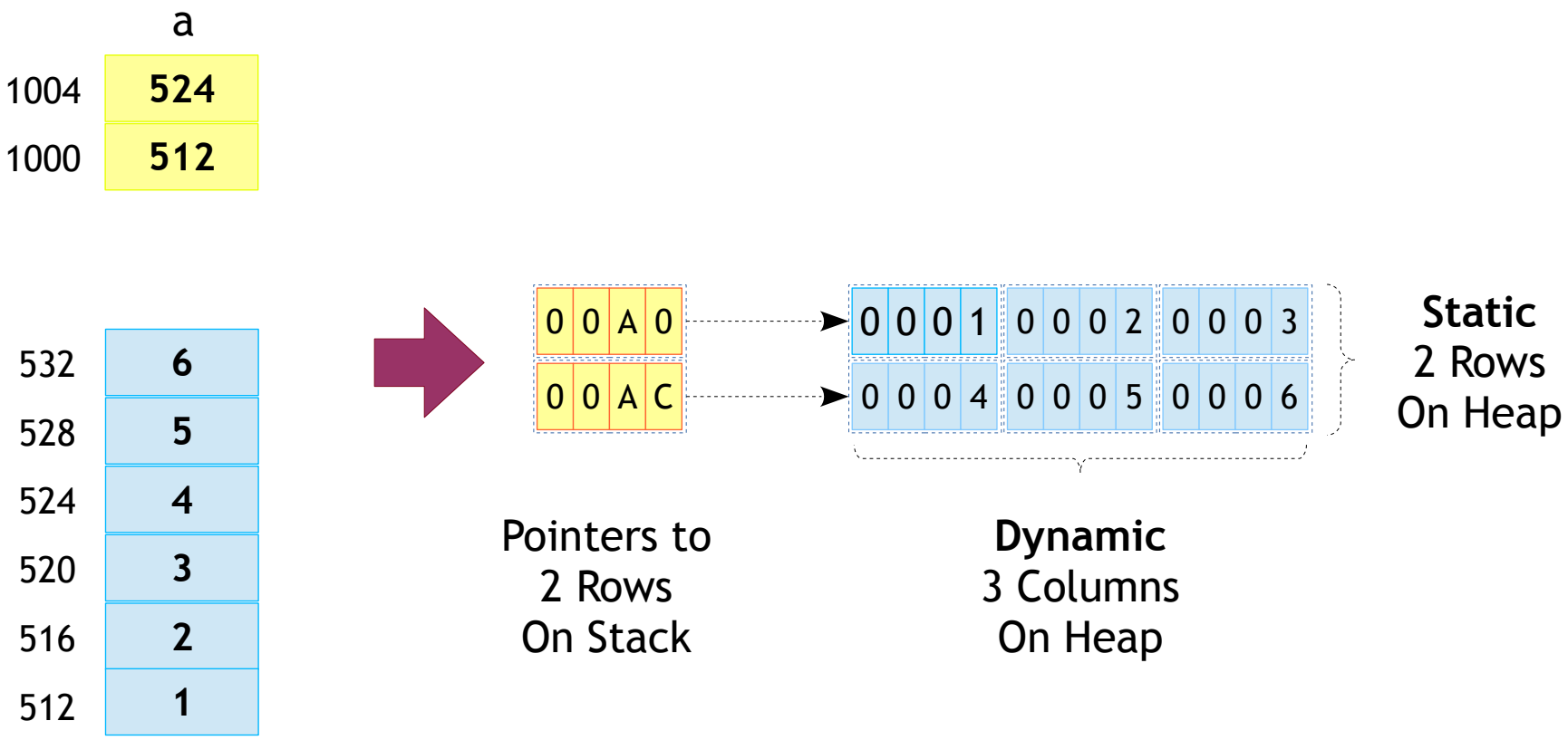
  2 * sizeof(datatype *) +

  2 * 3 * sizeof(datatype)

  2 * 4 + 2 * 3 * 4 = 32 Bytes
- The memory representation can be as shown in next slide

Advanced C

Pointers – 2D Array Creations - FSSD



Advanced C

Pointers - 2D Array Creations - FDSS

020_example.c

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int (*a)[3];

    a = malloc(2 * sizeof(int [3]));

    return 0;
}
```

- First Dynamic and Second Static (FDSS)
- Total size is

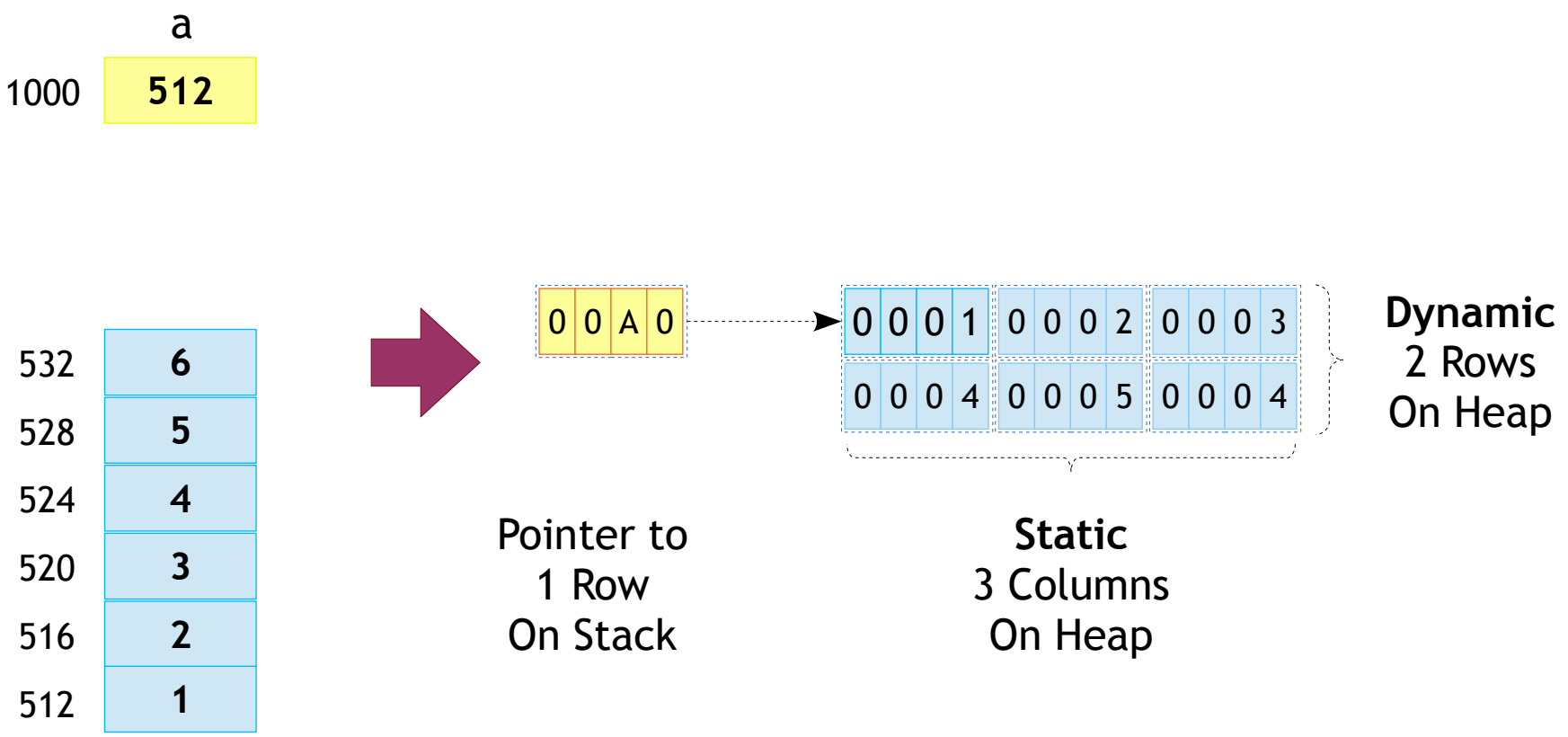
  sizeof(datatype *) +

  2 * 3 * sizeof(datatype)

  4 + 2 * 3 * 4 = 28 Bytes
- The memory representation can be as shown in next slide

Advanced C

Pointers – 2D Array Creations - FDSS



Advanced C

Pointers - 2D Array Creations - BD

021_example.c

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int **a;
    int i;

    a = malloc(2 * sizeof(int *));

    for (i = 0; i < 2; i++)
    {
        a[i] = malloc(3 * sizeof(int));
    }

    return 0;
}
```

- Both Dynamic (BD)
- Total size is

  sizeof(datatype **) +

  2 * sizeof(datatype *) +

  2 * 3 * sizeof(datatype)

  4 + 2 * 4 + 2 * 3 * 4 = 36 Bytes
- The memory representation can be as shown in next slide

Advanced C

Pointers – 2D Array Creations - BD