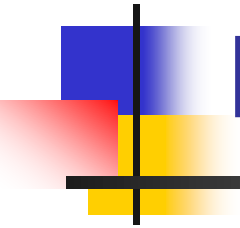
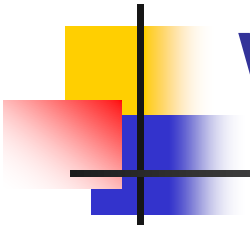
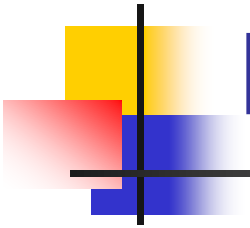


Playing with Pointers



What is a pointer?

- Answer 1
  - I don’t know why are you asking me?
- Answer 2
  - A Variable
  - That stores address
  - Of another variable

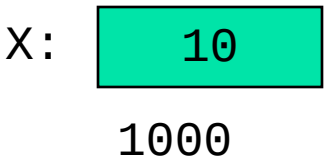


Basics

- & - address of operator
  - &x → address of x
- * - dereferencing operator or indirection operator
  - *x → value at x

Basics

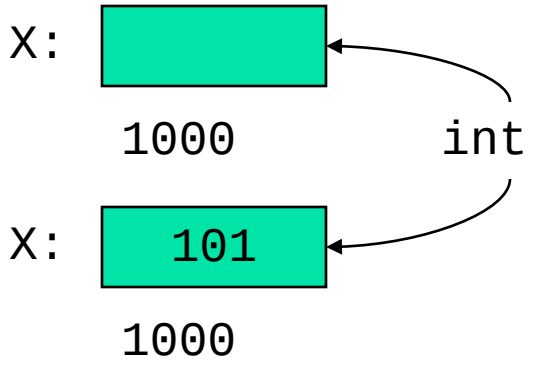
```
int X = 10;
```



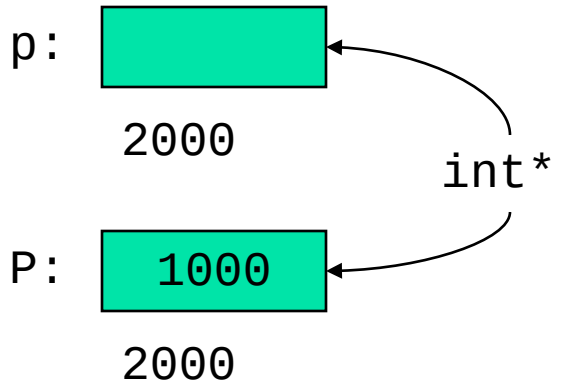

```
printf("Value of X = %u", X);                    → 10
printf("Address of X = %u", &X);                 → 1000
printf("Value at address of X = %u",*(&X));      → 10
```

Basics

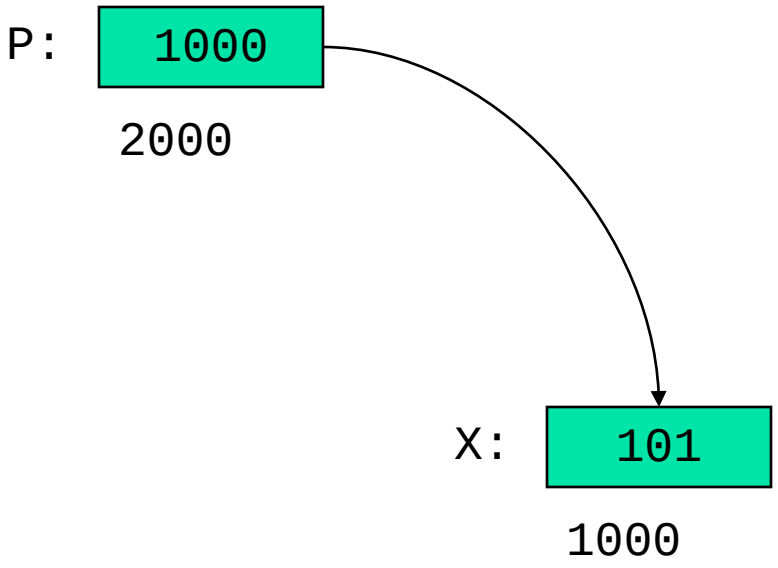
```
int X;

X = 101;

Here X is an int
```




```
int *P;

P = &X;

Here P is a pointer to int
```



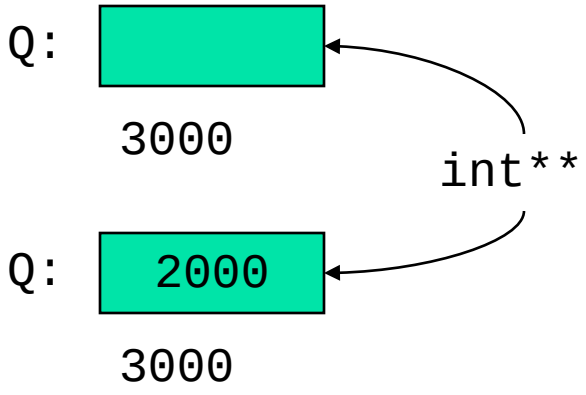
Basics



Hence we say that
P points to X

Basics

```
int **Q;
```




```
Q = &P;
```

Here Q is a pointer to pointer to int

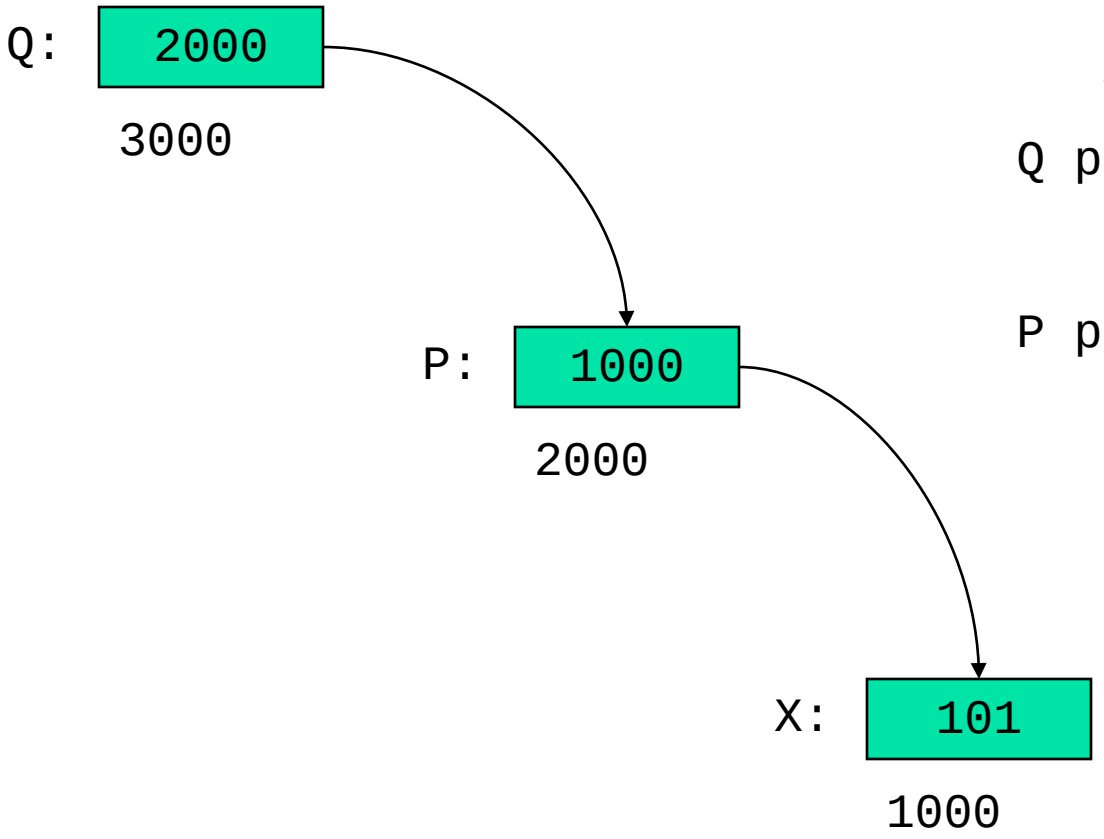
OR

Q is a pointer to int pointer

OR

Q is a pointer to int*

Basics

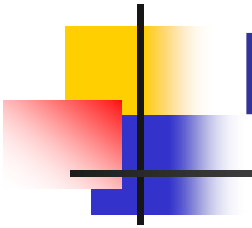


SO now,

Q points to P

&

P points to X



Basics

```
printf("Value of X = %u", X);                          → 101
printf("Value of P = %u", P);                          → 1000
printf("Value at value of P = %u", *P);                → 101
printf("Value of Q = %u", Q);                          → 2000
printf("Value at value of Q = %u", *Q);                → 1000
printf("Value at value at value of Q = %u", **Q);      → 101

Therefore,
X == *P       AND    P == *Q       AND    X == **Q
```

Pointer declarations

- `int  X1;`
  - X1 is an int
- `int* X2;`
  - X2 is an int pointer
- `int *X2;`
  - *X2 is an int
  - hence X2 is an int pointer

Quiz time

- `int*  X1, X2, *X3;`

- What are X1, X2, X3?
  a) X1, X2 are int* & X3 is an int**
  b) X1, X3 are int* & X2 is an int
  c) *X3 cause compiler error, otherwise X1, X2 are int*

- B is correct

Pointer declarations

- `*` is associated with the variable name & not the data type

- Thus the declaration: `int*  X1, X2, *X3;` is equivalent to: `int  *X1, X2, *X3;`

- or

```
int *X1;
int X2;
int *X3;
```

Pointer declarations

- We can define our own type as `INTPTR`
- So we can write declarations such as
  - `INTPTR x, y, z;`
- Such that `x, y, z` are all of type `int*`
- We have two options

A.
```
#define  INTPTR   int*
```

B.
```
typedef  int*  INTPTR;
```

- B is correct

Pointer declarations

- `#define  INTPTR  int*`
- `INTPTR x, y, z;`
- After preprocessing the above statement becomes
  - `int* x, y, z;`
- i.e. x is an int* & y, z are int
- Not exactly what we had in mind

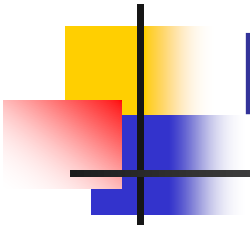
Pointer declarations

- `typedef int* INTPTR;`

- `INTPTR x, y, z;`
- Since typedef defines a type, the above declaration is equivalent to:

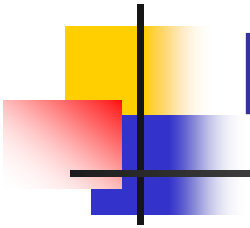
```
INTPTR x;
INTPTR y;
INTPTR z;
```

- Thus x, y, z all are of type pointer to int



Pointer arguments

- When do we need pointer type arguments?
  - A function can return only ONE value per call. Pointers are used when we need to get more than one values to be returned after a call.
  - When we want a function, to be able to modify the contents of variables local to the calling function.



Pointer arguments

- Consider the function:

```
int sumAB(int A, int B)
{
   int s = A+B;
   return s;
}
```

- We want a function sumAndDiffAB which returns both sum and difference

Pointer arguments

- The solution is:

```
int sumAndDiffAB(int A, int B)
{
  int s = A + B;
  int d = A – B;
  return s;
  return d;
}
```

Pointer arguments

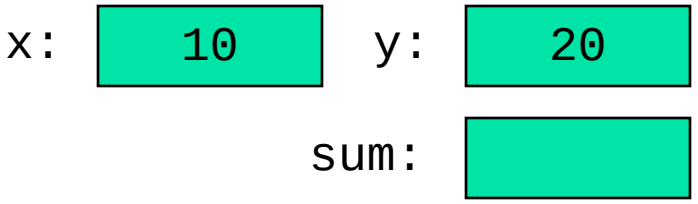
- The CORRECT solution is:

```
void sumAndDiffAB(int A, int B,
                  int *sum, int *diff)
{
   *sum  = A + B;
   *diff = A – B;
}
```

Pointer arguments

- Consider the following sequence of execution:

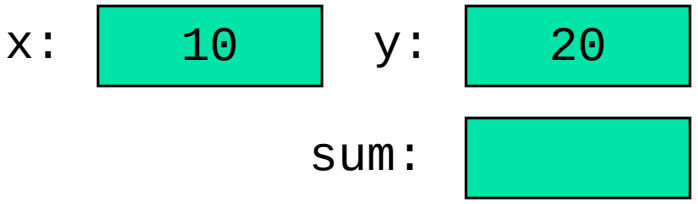
```
1.  int main()
2.  {
3.    int x=10, y=20, sum;
4.    sum = sumAB(x, y);
5.  }

6.  int sumAB(int A, int B)
7.  {
8.    int s= A + B;
9.    return s;
10. }
```



Pointer arguments

- Consider the following sequence of execution:

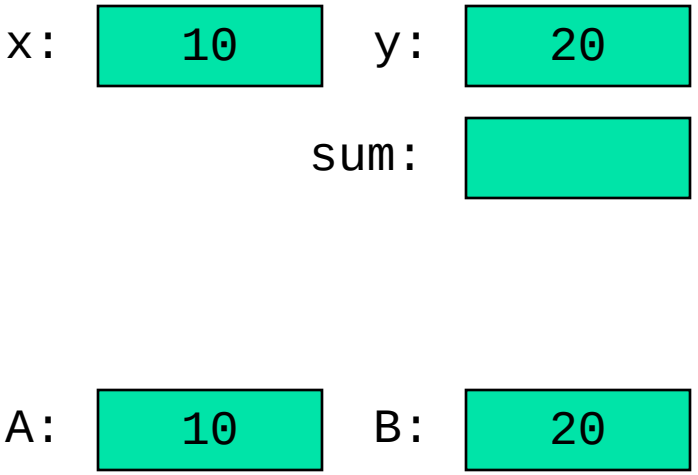
```
1.  int main()
2.  {
3.    int x=10, y=20, sum;
4.    sum = sumAB(x, y);
5.  }

6.  int sumAB(int A, int B)
7.  {
8.    int s= A + B;
9.    return s;
10. }
```



Pointer arguments

- Consider the following sequence of execution:

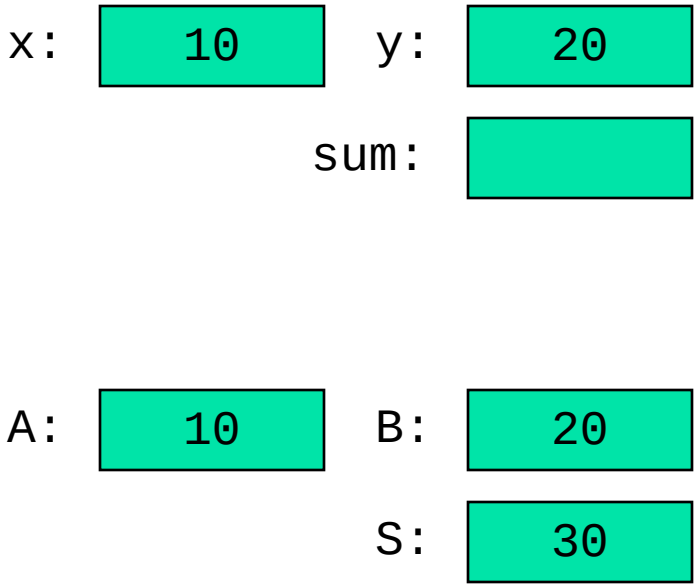
```
1.  int main()
2.  {
3.    int x=10, y=20, sum;
4.    sum = sumAB(x, y);
5.  }

6.  int sumAB(int A, int B)
7.  {
8.    int s= A + B;
9.    return s;
10. }
```



Pointer arguments

- Consider the following sequence of execution:

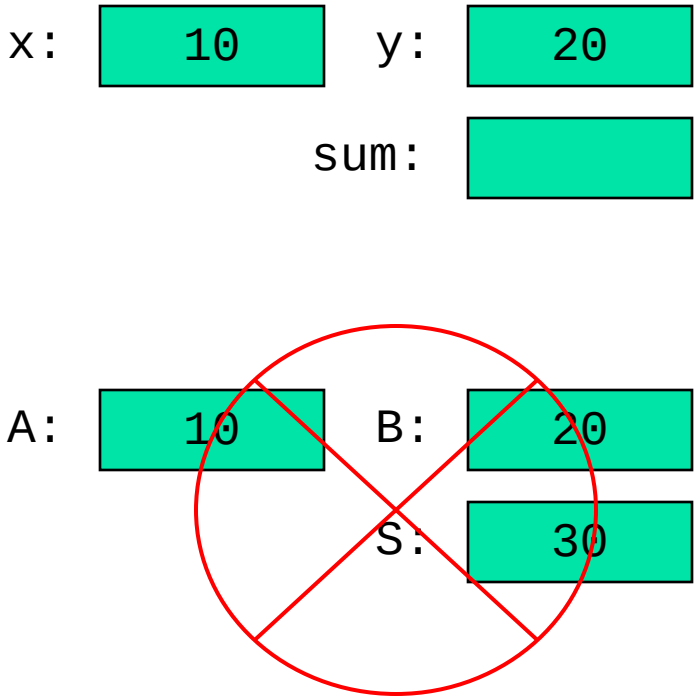
```
1.   int main()
2.   {
3.     int x=10, y=20, sum;
4.     sum = sumAB(x, y);
5.   }

6.   int sumAB(int A, int B)
7.   {
8.     int s= A + B;
9.     return s;
10.  }
```



Pointer arguments

- Consider the following sequence of execution:

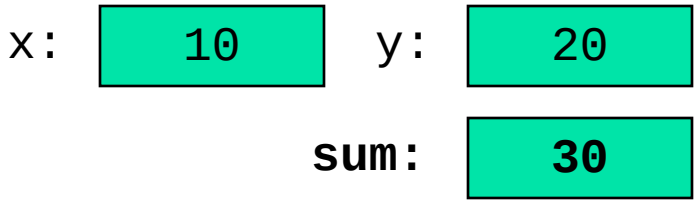
```
1.   int main()
2.   {
3.     int x=10, y=20, sum;
4.     sum = sumAB(x, y);
5.   }

6.   int sumAB(int A, int B)
7.   {
8.     int s= A + B;
9.     return s;
10.  }
```



Pointer arguments

- Consider the following sequence of execution:

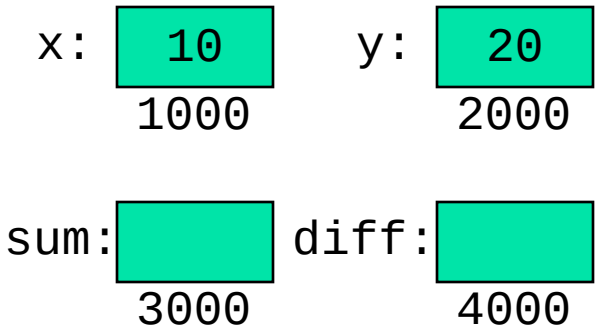
```
1.   int main()
2.   {
3.     int x=10, y=20, sum;
4.     sum = sumAB(x, y);
5.   }

6.   int sumAB(int A, int B)
7.   {
8.     int s= A + B;
9.     return s;
10.  }
```



Pointer arguments

- Consider the following sequence of execution:

```
1.  int main()
2.  {
3.    int x=10, y=20, sum, diff;
4.    sumAndDiffAB(x, y, &sum, &diff);
5.  }


6.  void sumAndDiffAB(int A, int B, int
    *s, int *d)
7.  {
8.    *s = A + B;
9.    *d = A - B;
10. }
```

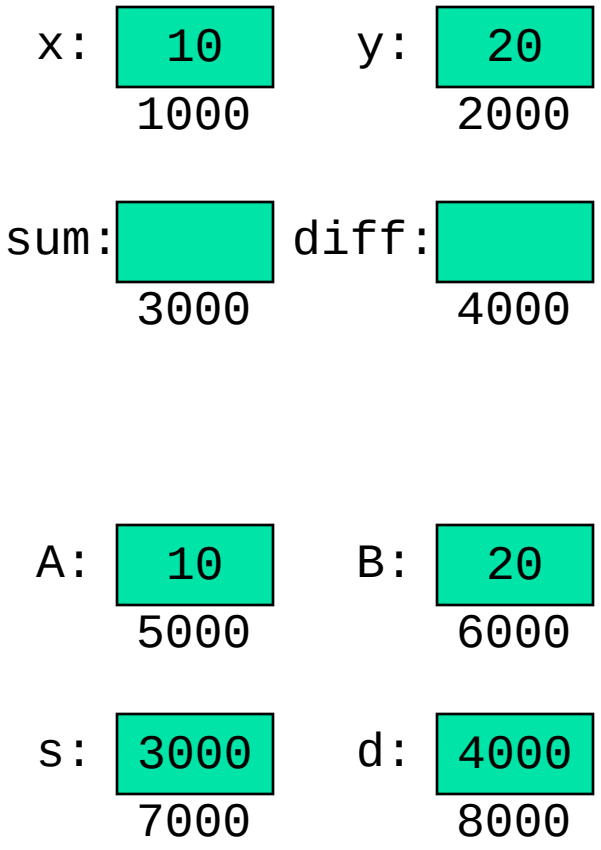


Pointer arguments

- Consider the following sequence of execution:

```
1.  int main()
2.  {
3.    int x=10, y=20, sum, diff;
4.    sumAndDiffAB(x, y, &sum, &diff);
5.  }
```

```
6.  void sumAndDiffAB(int A, int B, int
    *s, int *d)
7.  {
8.    *s = A + B;
9.    *d = A - B;
10. }
```

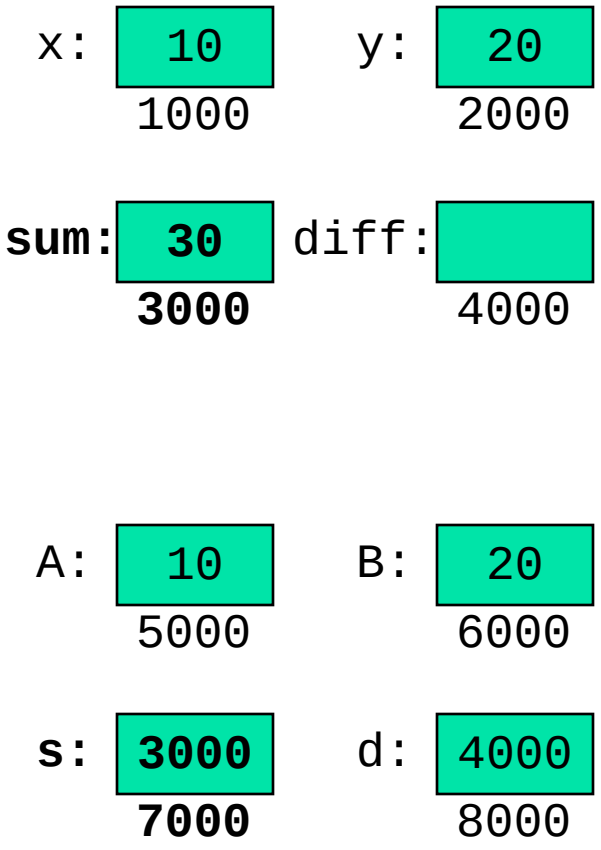


Pointer arguments

- Consider the following sequence of execution:

```
1.  int main()
2.  {
3.    int x=10, y=20, sum, diff;
4.    sumAndDiffAB(x, y, &sum, &diff);
5.  }
```

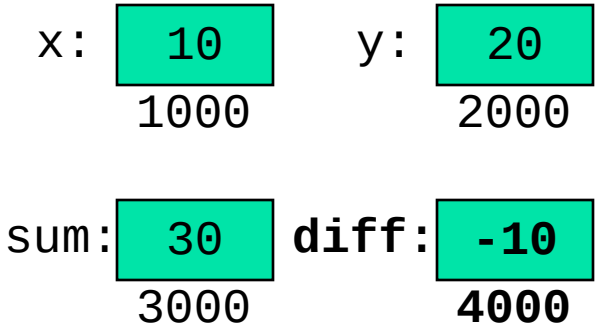
```
6.  void sumAndDiffAB(int A, int B, int
    *s, int *d)
7.  {
8.    *s = A + B;
9.    *d = A - B;
10. }
```



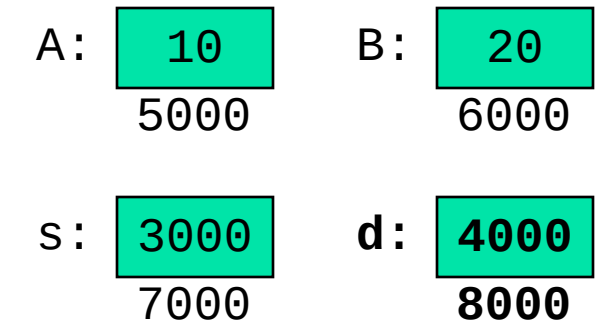
Pointer arguments

- Consider the following sequence of execution:

```
1.  int main()
2.  {
3.    int x=10, y=20, sum, diff;
4.    sumAndDiffAB(x, y, &sum, &diff);
5.  }
```



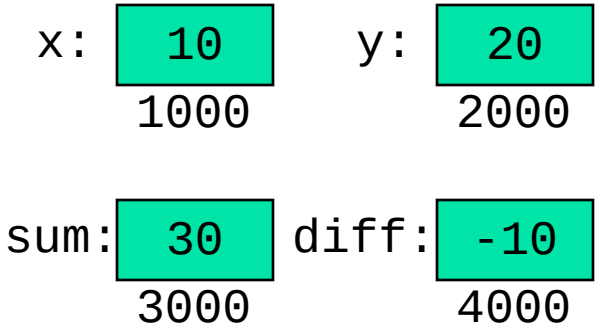

```
6.  void sumAndDiffAB(int A, int B, int
    *s, int *d)
7.  {
8.    *s = A + B;
9.    *d = A - B;
10. }
```



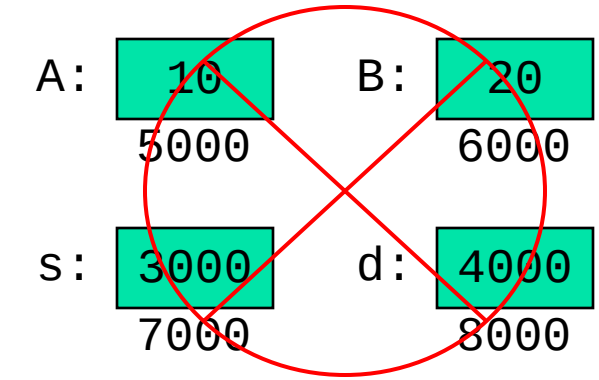
Pointer arguments

- Consider the following sequence of execution:

```
1.  int main()
2.  {
3.    int x=10, y=20, sum, diff;
4.    sumAndDiffAB(x, y, &sum, &diff);
5.  }
```




```
6.  void sumAndDiffAB(int A, int B, int
    *s, int *d)
7.  {
8.    *s = A + B;
9.    *d = A - B;
10. }
```



Pointer arguments

- The swap function:

```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap(x, y);
5.   }

6.   void swap(int A, int B)
7.   {
8.     int T = A;
9.     A = B;
10.    B = T;
11.  }
```

Pointer arguments

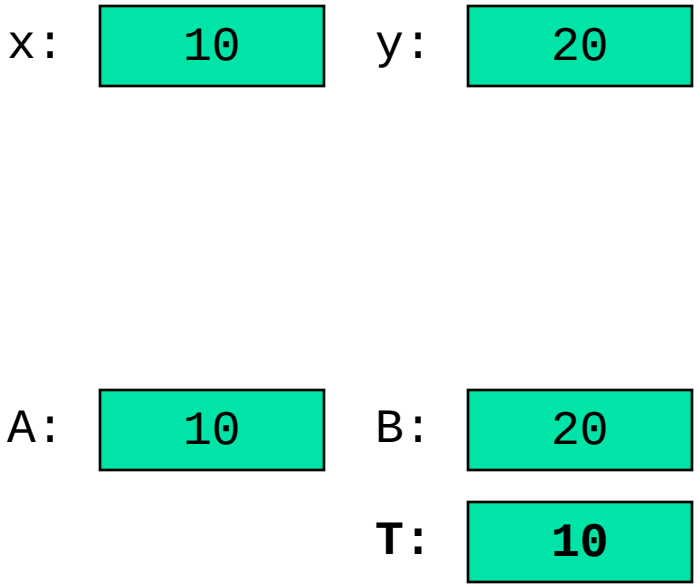
- The swap function:

```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap(x, y);
5.   }

6.   void swap(int A, int B)
7.   {
8.     int T = A;
9.     A = B;
10.    B = T;
11.  }
```

Pointer arguments

- The swap function:

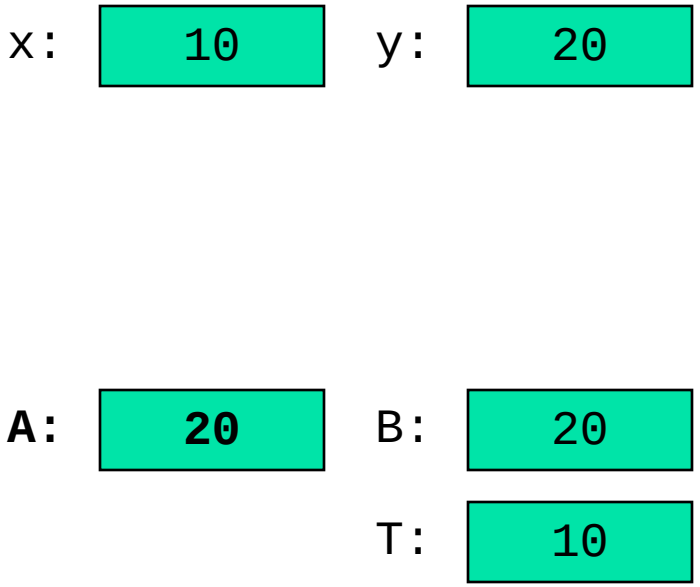
```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap(x, y);
5.   }

6.   void swap(int A, int B)
7.   {
8.     int T = A;
9.     A = B;
10.    B = T;
11.  }
```



Pointer arguments

- The swap function:

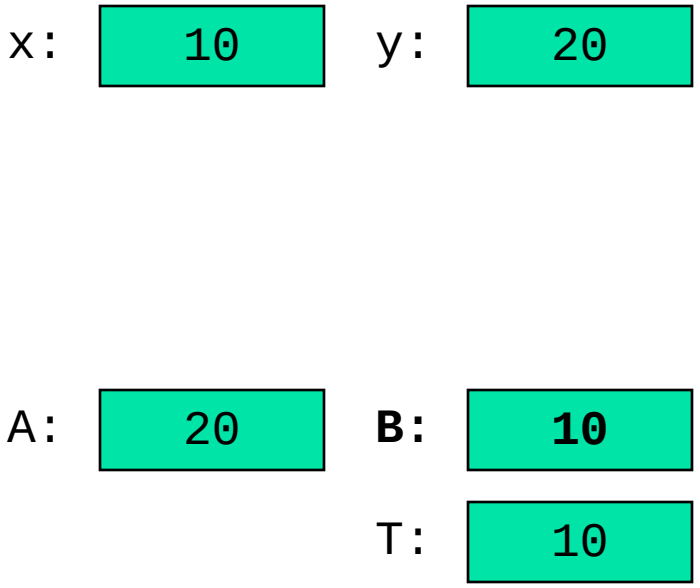
```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap(x, y);
5.   }

6.   void swap(int A, int B)
7.   {
8.     int T = A;
9.     A = B;
10.    B = T;
11.  }
```



Pointer arguments

- The swap function:

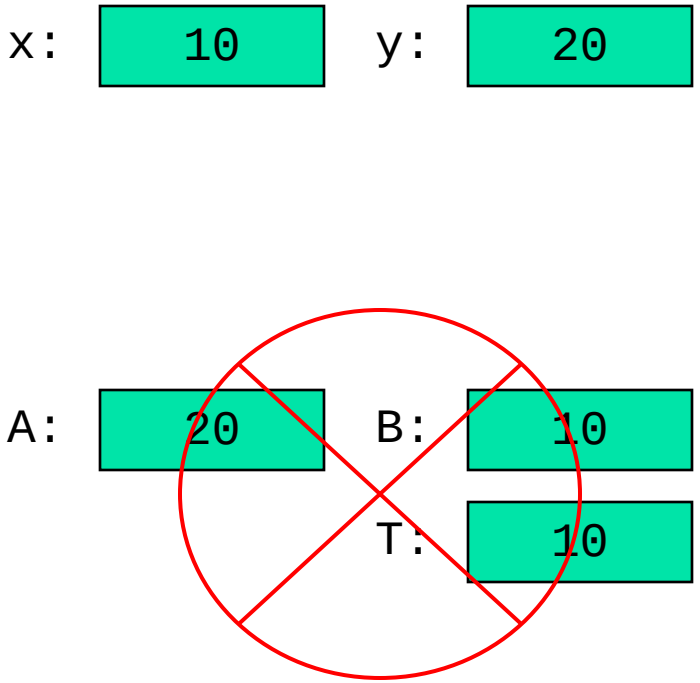
```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap(x, y);
5.   }

6.   void swap(int A, int B)
7.   {
8.     int T = A;
9.     A = B;
10.    B = T;
11.  }
```



Pointer arguments

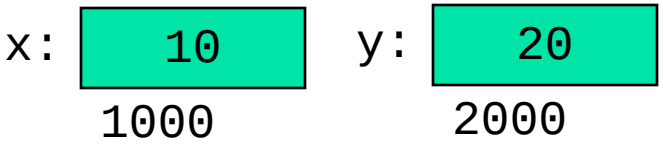
- The swap function:

```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap(x, y);
5.   }

6.   void swap(int A, int B)
7.   {
8.     int T = A;
9.     A = B;
10.    B = T;
11.  }
```



Pointer arguments

- The correct swap function:

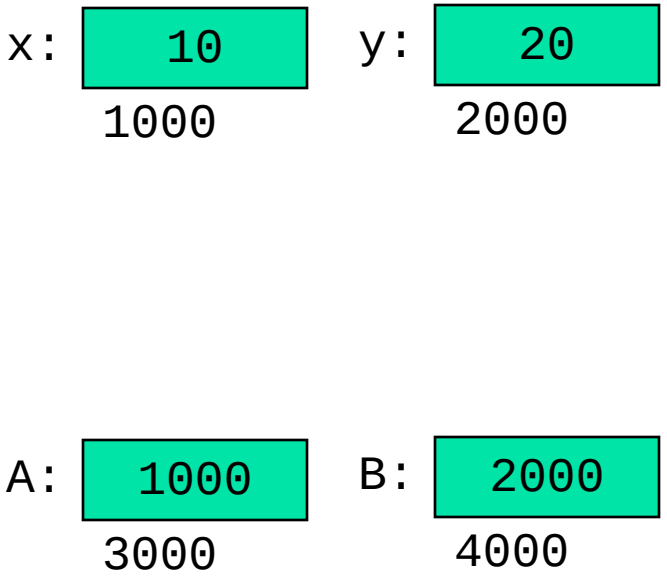



```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap (&x, &y);
5.   }

6.   void swap (int *A, int *B)
7.   {
8.     int T = *A;
9.     *A = *B;
10.    *B = T;
11.  }
```

Pointer arguments

- The correct swap function:

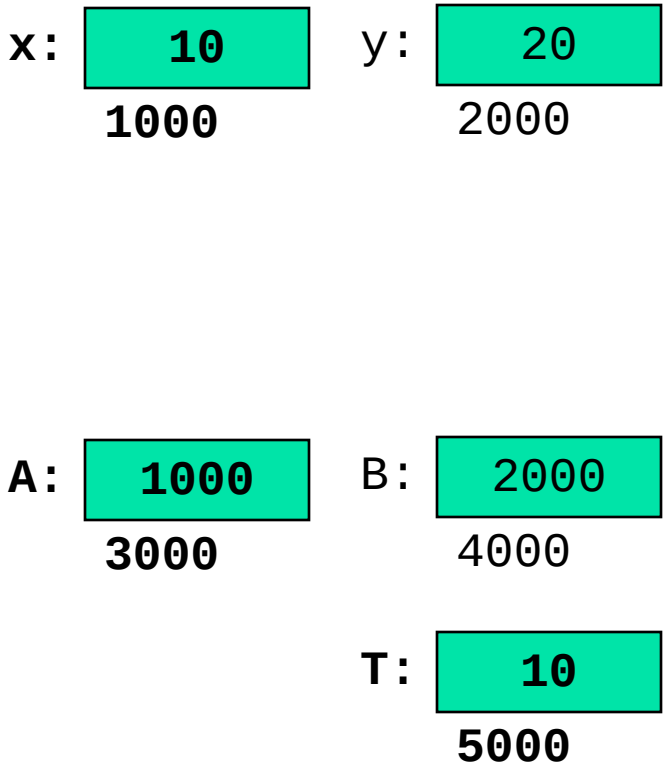
```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap (&x, &y);
5.   }

6.   void swap (int *A, int *B)
7.   {
8.     int T = *A;
9.     *A = *B;
10.    *B = T;
11.  }
```



Pointer arguments

- The correct swap function:

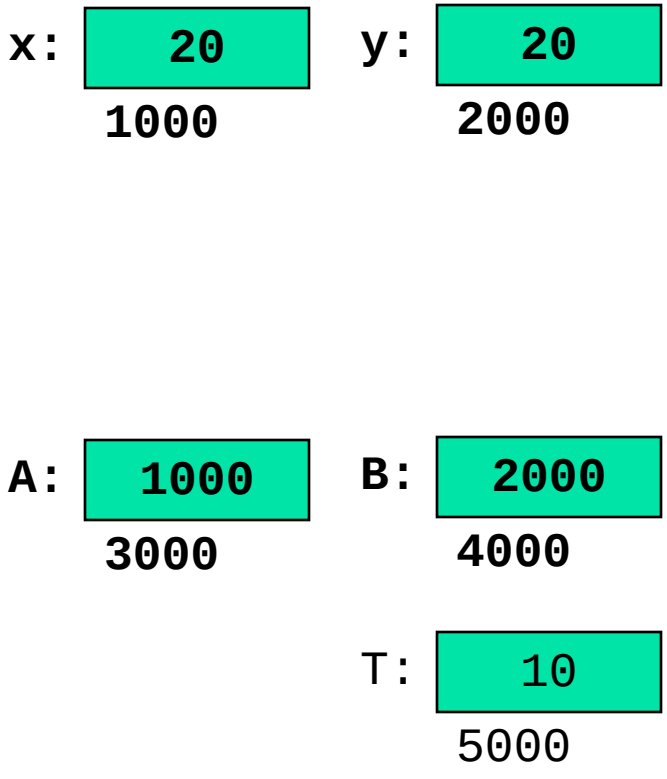
```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap (&x, &y);
5.   }

6.   void swap (int *A, int *B)
7.   {
8.     int T = *A;
9.     *A = *B;
10.    *B = T;
11.  }
```



Pointer arguments

- The correct swap function:

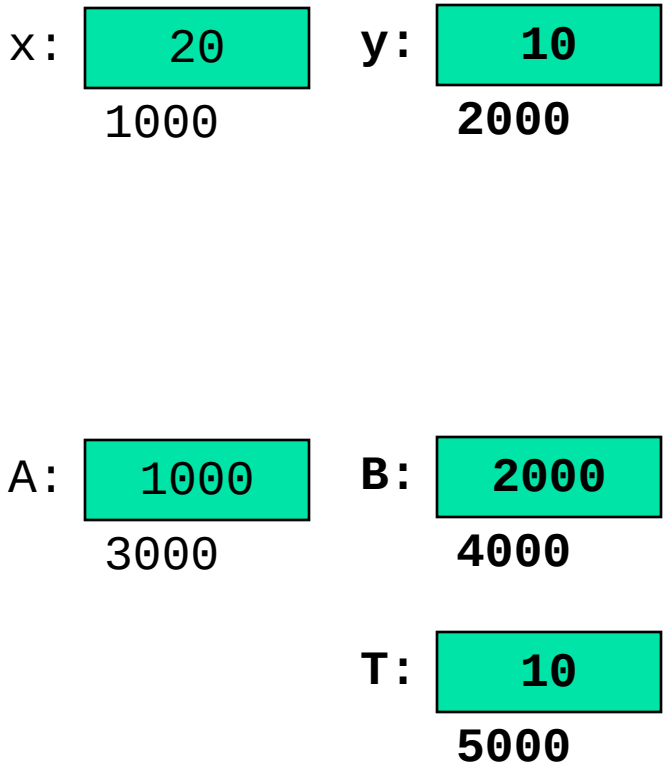
```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap (&x, &y);
5.   }

6.   void swap (int *A, int *B)
7.   {
8.     int T = *A;
9.     *A = *B;
10.    *B = T;
11.  }
```



Pointer arguments

- The correct swap function:

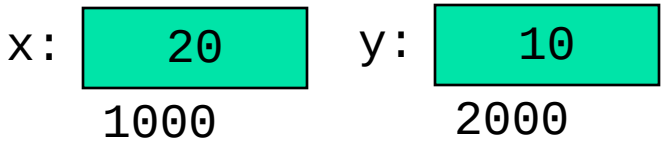
```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap (&x, &y);
5.   }

6.   void swap (int *A, int *B)
7.   {
8.     int T = *A;
9.     *A = *B;
10.    *B = T;
11.  }
```

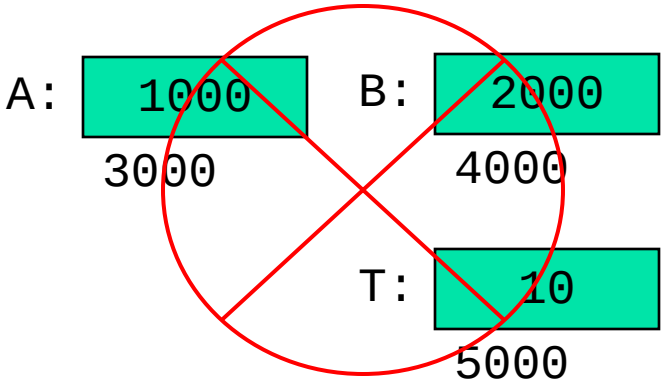


Pointer arguments

- The correct swap function:

```
1.   int main()
2.   {
3.     int x=10, y=20;
4.     swap (&x, &y);
5.   }

6.   void swap (int *A, int *B)
7.   {
8.     int T = *A;
9.     *A = *B;
10.    *B = T;
11.  }
```

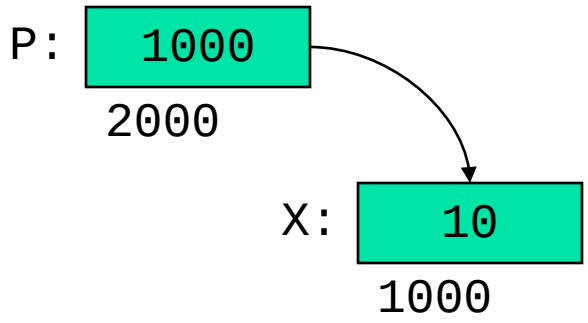




Pointers & const

- Consider these two variables
  - `int X = 10;`
  - `int *P = &X;`

- There are 3 cases
  1. `X is constant`
  2. `P is constant`
  3. `Value pointed by P is constant`



Pointers & const

- Consider following:
  - `int X1 = 10, Y1 = 20;`
    - `X1 = 15; Y1 = 25;` `// Ok`
  - `const int X2;` `// Error`
  - `const int X2 = 10, Y2 = 20;`
    - `X2 = 20; Y2 = 30;` `// Error`
  - `int *Pi;`
    - `Pi = &X1; Pi = &Y1; *Pi = 50;` `// OK`
    - `Pi = &X2; Pi = &Y2;` `// Error`

Pointers & const

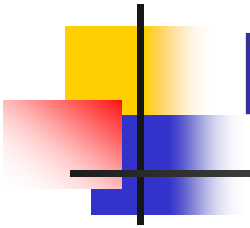
- Continued...
  - `const int *Pci;`
    - `Pci = &X1; Pci = &Y1;` `// OK`
    - `Pci = &X2; Pci = &Y2;` `// OK`
    - `*Pci = 50;` `// Error`
  - `int * const Cpi;` `// Error`
  - `int * const Cpi = &X1;`
  - `int * const Cpi = &X2;` `// Error`
    - `Cpi = &Y1;` `//Error`
    - `*Cpi = 20;` `//OK`

Pointers & const

- Continued...
  - `const int * const Cpci;` Error
  - `const int * const Cpci = &X1;` OK
  - `const int * const Cpci = &X2;` OK
  - `const int * const Cpci = &Y1;` OK
  - `const int * const Cpci = &Y2;` OK
  - `Cpci = &X2;` Error
  - `*Cpci = 50;` Error

Pointer arithmetic

- Addition
  - We can add integers to a pointer.
  - We cannot add two pointers
  - E.g. P + 2, P + I, P++
- Subtraction
  - We can subtract integers from pointer
  - We can subtract one pointer from another provided they are of same type
  - P – 1, P – I, P1 – P2, P--



Pointer arithmetic

- Multiplication & Division operations are not allowed on pointers.
- Increment or Decrement operations on a pointer, make it point to the immediate next or previous element respectively.
- When we subtract one pointer from another we get the number of elements in between.

Break: sizeof operator

- **Consider Declarations**
  - `int i, *pi, **ppi;`
  - `char ch;`
  - `char *s1 = "Hello";`
  - `char s2[50] = "Hello";`
  - `char s3[] = "Hello";`
  - `char s4[] = {'H', 'E', 'L', 'L', 'O'};`
  - `int ai1[10];`
  - `int ai2[] = {1, 2, 3};`

  - **Note:** *s1, s2, s3 are strings but s4 is just an array*

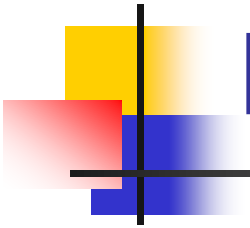
sizeof(10)	4
sizeof(int)	4
sizeof(i)	4
sizeof(pi)	4
sizeof(ppi)	4
sizeof(*pi)	4
sizeof(ch)	1
sizeof('A')	4
sizeof(s1)	4
sizeof(*s1)	1

Break: sizeof operator

- **Consider Declarations**
  - `int i, *pi, **ppi;`
  - `char ch;`
  - `char *s1 = "Hello";`
  - `char s2[50] = "Hello";`
  - `char s3[] = "Hello";`
  - `char s4[] = {‘H’, ‘E’, ‘L’, ‘L’, ‘O’};`
  - `int ai1[10];`
  - `int ai2[] = {1, 2, 3};`

  - **Note:** *s1, s2, s3 are strings but s4 is just an array*

sizeof(s2)	50
sizeof(s2[0])	1
sizeof(*s2)	1
sizeof(s3)	6
sizeof(s4)	5
sizeof(“Ram”)	4
sizeof(“OK”)	3
sizeof(ai1)	40
sizeof(ai2)	12



Break: sizeof operator

- The `sizeof` operator does not evaluate the expression passed as argument
- It only evaluates the type of the expression
- i.e. `sizeof(I++)` will return 4 but it wont increment `I`;

Pointers & Arrays

- Some common array declarations
  - `int A[5] = {10, 20, 30, 40, 50};`
  - `char B[4] = “Ram”;`
  - `char C[]={‘R’, ‘A’, ‘M’};`

A:

10	20	30	40	50
1000	1004	1008	1012	1016
A[0]	A[1]	A[2]	A[3]	A[4]

B:

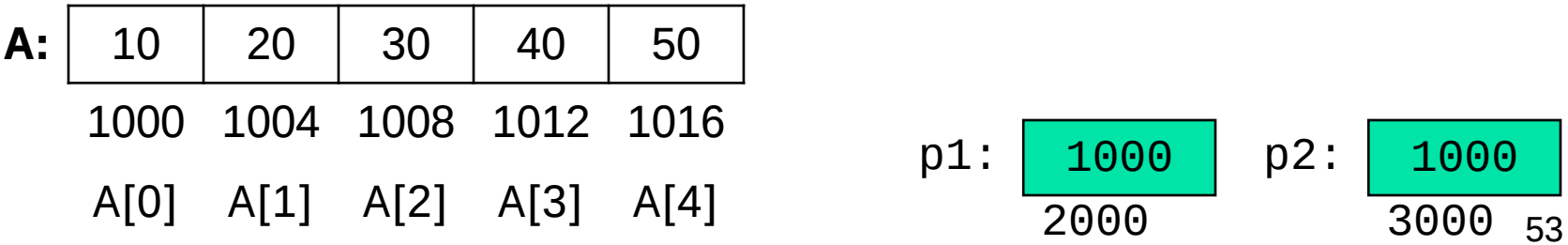
R	a	m	\0
2000	2001	2002	2003
B[0]	B[1]	B[2]	B[3]

C:

R	A	M
3000	3001	3002
C[0]	C[1]	C[2]

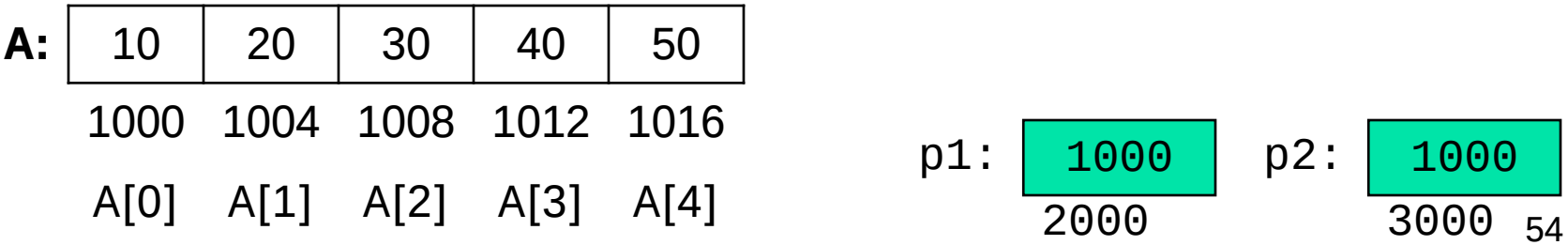
Pointers & Arrays

- Accessing array elements
  - `int A[5] = {10, 20, 30, 40, 50};`
  - `int *p1 = A;` `// May give warning`
  - `int *p2 = &A[0];`
  - We generally access elements as A[i], P1[i]
  - **`A[i] == i[A] == *(A + i) == *(i + A)`**
  - **Infact all such expression are converted to *(A + i) form, even *P1 becomes *(P1 + 0)**



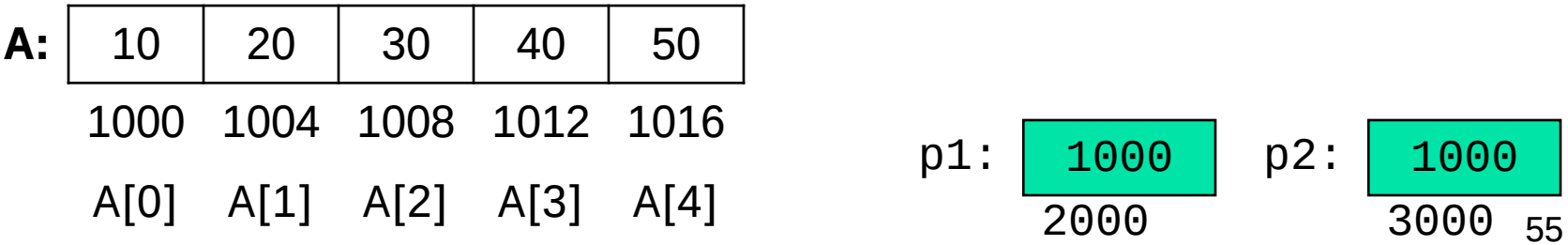
Pointers & Arrays

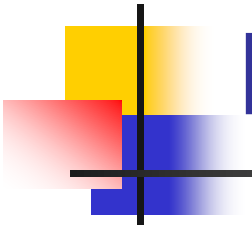
- What does ‘A’ represent?
  - ‘A’ is an array
  - If we print ‘A’ we get the address of the first element of the array. i.e. `printf(“%u”,a);` → `1000`
  - ‘A’ is also called a constant pointer in the sense that the address associated with an array is always constant.
  - So, `A++` or `++A` gives compiler error unlike `p1++` or `p2++`



Pointers & Arrays

- How will A[2] be evaluated?
  - `Since A is an array of int so the scale factor is sizeof(int)`
  - `A[2] == *(A + 2) = *(1000 + 2*sizeof(int))`
  - `*(1008) == 30`
- Similarly p1[2] or p2[2] will result in 30





Pointers & Arrays

- Array elements as arguments
  - Rule 1: *There is no way we can pass all elements of the array in one go. However we can pass individual elements one-by-one.*
  - Rule 2: *Using the the array itself as an argument actually passes the address of its first element.*
  - Note: *Array elements are always stored in contiguous memory locations*

Pointers & Arrays

- Passing 1-D array as argument

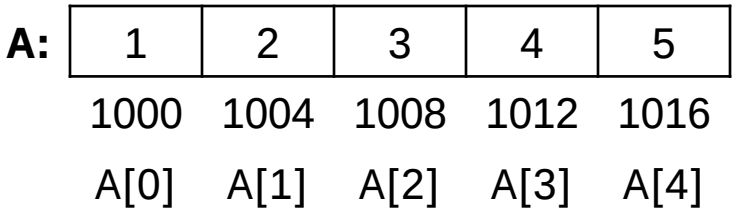
```
1.  int main()
2.  {
3.    int A[5] = {1, 2, 3, 4, 5};
4.    display(&A[0]);
5.  }
6.  void display(int *a, int count)
7.  {
8.    int i;
9.    for(i=0; i<count; i++)
10.   {
11.    printf("%d",a[i]);
12.   }
13. }
```

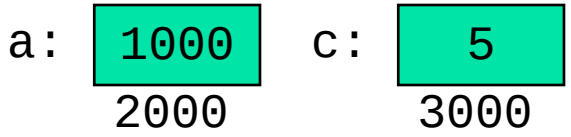
A:	1	2	3	4	5
	1000	1004	1008	1012	1016
	A[0]	A[1]	A[2]	A[3]	A[4]

Pointers & Arrays

- Passing 1-D array as argument

```
1.  int main()
2.  {
3.    int A[5] = {1, 2, 3, 4, 5};
4.    display(&A[0], 5);
5.  }
6.  void display(int *a, int c)
7.  {
8.    int i;
9.    for(i=0; i<c; i++)
10.   {
11.    printf("%d",a[i]);
12.   }
13. }
```

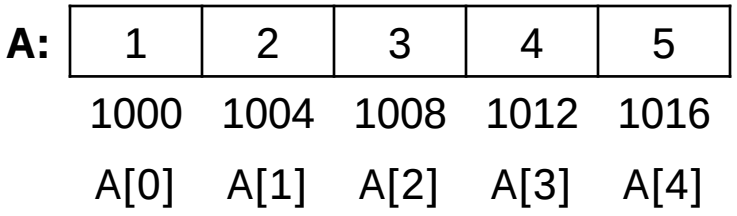


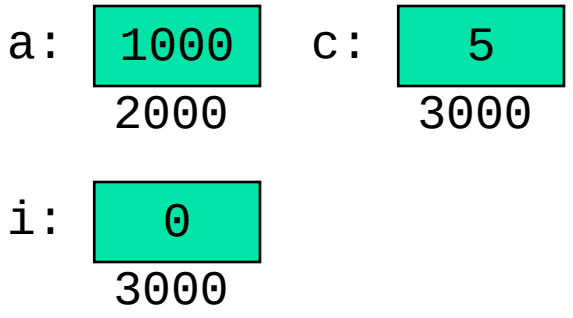


Pointers & Arrays

- Passing 1-D array as argument

```
1.  int main()
2.  {
3.    int A[5] = {1, 2, 3, 4, 5};
4.    display(&A[0], 5);
5.  }
6.  void display(int *a, int c)
7.  {
8.    int i;
9.    for(i=0; i<c; i++)
10.   {
11.    printf("%d",a[i]);
12.   }
13. }
```

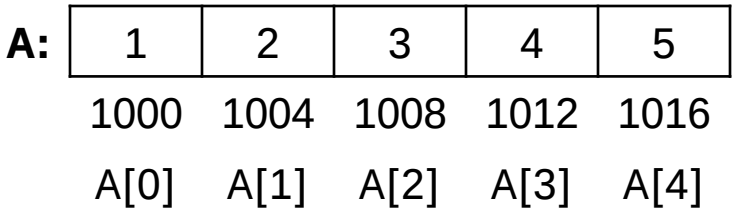


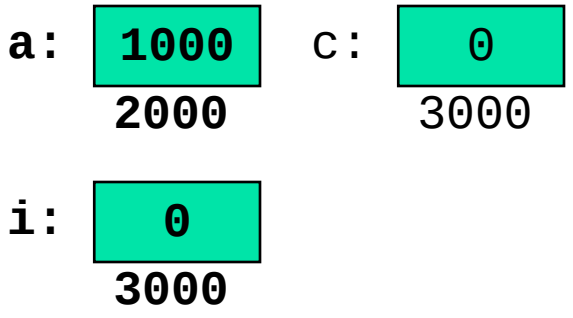


Pointers & Arrays

- Passing 1-D array as argument

```
1.  int main()
2.  {
3.    int A[5] = {1, 2, 3, 4, 5};
4.    display(&A[0], 5);
5.  }
6.  void display(int *a, int c)
7.  {
8.    int i;
9.    for(i=0; i<c; i++)
10.   {
11.    printf("%d",a[i]);
12.   }
13. }
```

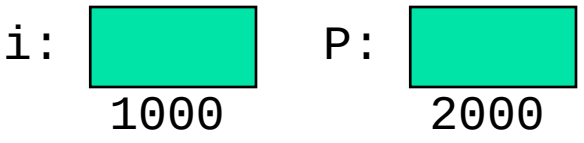




Pointers & Arrays

- Dynamic allocation of 1-D array

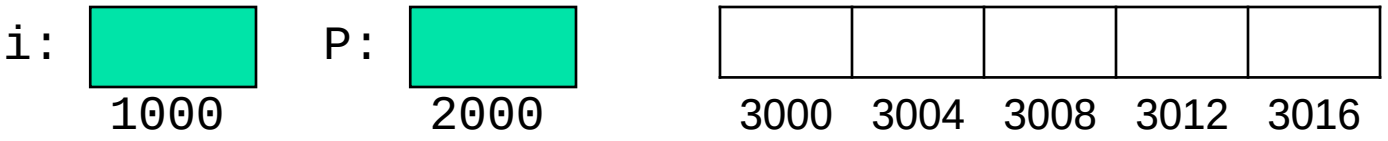
```
1.  int i,*P;
2.  P = (int*)malloc(sizeof(int)*5);
3.  for(i=0; i<5; i++)
4.    P[i] = i+1;
5.  for(i=0; i<5; i++)
6.    printf("%d",P[i]);
7.  free(P);
```



Pointers & Arrays

- Dynamic allocation of 1-D array

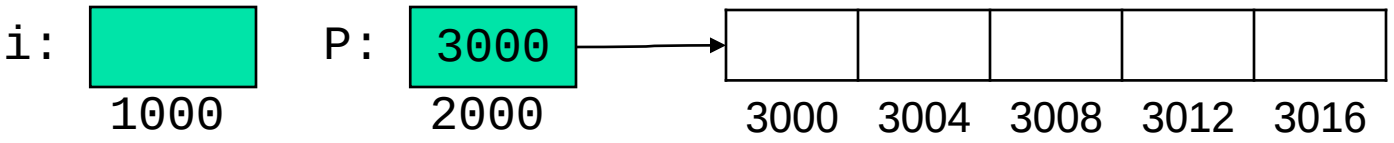
```
1.  int i,*P;
2.  P = (int*)malloc(sizeof(int)*5);
3.  for(i=0; i<5; i++)
4.    P[i] = i+1;
5.  for(i=0; i<5; i++)
6.    printf("%d",P[i]);
7.  free(P);
```



Pointers & Arrays

- Dynamic allocation of 1-D array

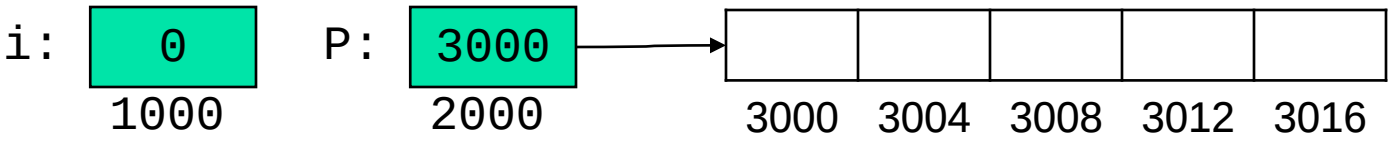
```
1.  int i,*P;
2.  P = (int*)malloc(sizeof(int)*5);
3.  for(i=0; i<5; i++)
4.    P[i] = i+1;
5.  for(i=0; i<5; i++)
6.    printf("%d",P[i]);
7.  free(P);
```



Pointers & Arrays

- Dynamic allocation of 1-D array

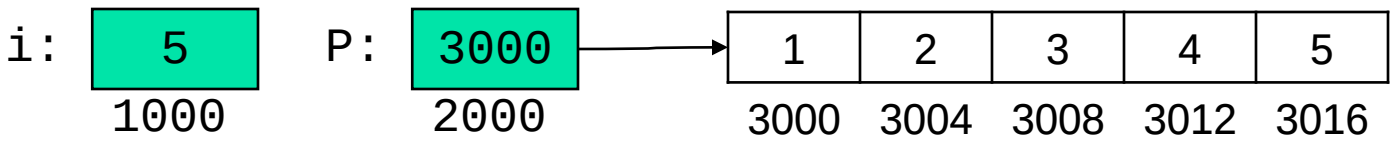
```
1.  int i,*P;
2.  P = (int*)malloc(sizeof(int)*5);
3.  for(i=0; i<5; i++)
4.    P[i] = i+1;
5.  for(i=0; i<5; i++)
6.    printf("%d",P[i]);
7.  free(P);
```



Pointers & Arrays

- Dynamic allocation of 1-D array

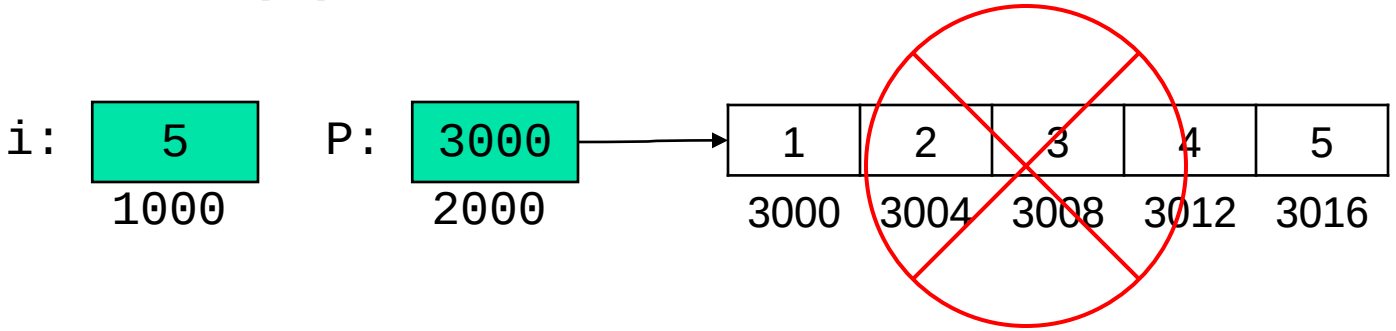
```
1.  int i,*P;
2.  P = (int*)malloc(sizeof(int)*5);
3.  for(i=0; i<5; i++)
4.    P[i] = i+1;
5.  for(i=0; i<5; i++)
6.    printf("%d",P[i]);
7.  free(P);
```

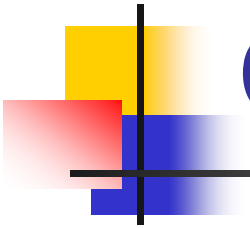


Pointers & Arrays

- Dynamic allocation of 1-D array

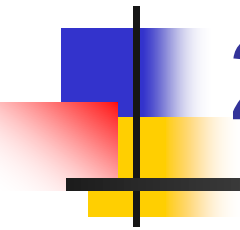
```
1.  int i,*P;
2.  P = (int*)malloc(sizeof(int)*5);
3.  for(i=0; i<5; i++)
4.    P[i] = i+1;
5.  for(i=0; i<5; i++)
6.    printf("%d",P[i]);
7.  free(P);
```





Quiz time

- If
```
int a[5] = {1, 2, 3, 4, 5};
  int *p = &a[0];
```
- Then what will be the value of the following expressions and p after evaluation of each expression? Assuming a == 1000.
  - `++*p`
  - `*++p`
  - `*p++`
  - `(*p)++`



2-D Arrays

Double trouble

2-D Arrays

- Also called “Array of Arrays”
- E.g.

```
int A[2][3]={1, 2, 3,
             4, 5, 6};
```

- What is A?
  - A two element array of type int[3]
- What is A[0] or A[1]?
  - A three element array of type int

	0	1	2	
A:	1	2	3	0
	4	5	6	1

The actual storage:

A:	1	2	3	4	5	6
	3000	3004	3008	3012	3016	3020

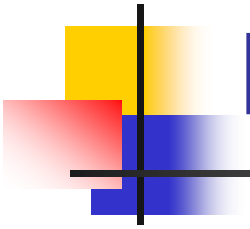
2-D Arrays

- Hence,
  - `A == 3000`
  - `A[0] == 3000`
  - `A[1] == 3012`
  - `sizeof(A)` `== 24`
  - `sizeof(A[0])` `== 12`
  - `sizeof(A[0][0])` `== 4`

	0	1	2	
A:	1	2	3	0
	4	5	6	1

The actual storage:

A:	1	2	3	4	5	6
	3000	3004	3008	3012	3016	3020



Break: Special cases

- When we create an array:
  - `int A[5] = {1, 2, 3, 4, 5};`
  - Whenever we use A it is replaced by the **address of the first element** or the so called base address
- However when,
  - A is used as argument in sizeof operator it returns **the total number of bytes allocated**
  - Address of operator (&) is applied on A we get the **address of the entire array** & not the first element. Thou the two numerical values may be same

Break: Special cases

- For e.g.
  - `int A[5] = {1, 2, 3, 4, 5};`
  - `int B[2][3] = { {1, 2, 3}, {4, 5, 6} };`
    - *Assume A begins at 1000 & B at 2000*
  - `sizeof(A) = 20 and sizeof(B) = 24`
  - `sizeof(B[0]) = 12 and sizeof(B[1]) = 12`
  - `A + 1, &A + 1`
    - `A + 1 = 1004, &A + 1 = 1020`
  - `B + 1, &B + 1, &B[0] + 1`
    - `B + 1 = 2012, &B + 1 = 2024, &B[0][0] + 1 = 2004`

2-D Arrays

- What happens when we write A[1][2]?
  - It is converted to pointer notation.
  - `(*(A + 1))[2]`
  - `*(*(A + 1) + 2)`
  - `*(*(3000 + 1*sizeof(int[3])) + 2)`
  - `*( *(3000 + 1*12) + 2)`
  - `*(3012 + 2*sizeof(int))`
  - **`*(3012 + 2*4)`**
  - `*(3020) == 6`
  - *Similar conversion is performed for higher dimensions*

The actual storage:

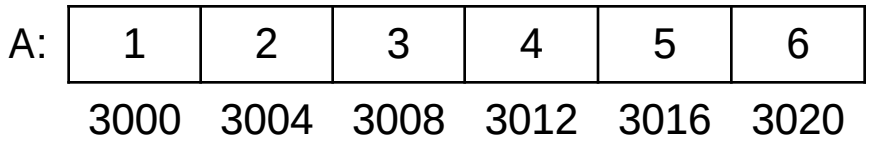
A:	1	2	3	4	5	6
	3000	3004	3008	3012	3016	3020

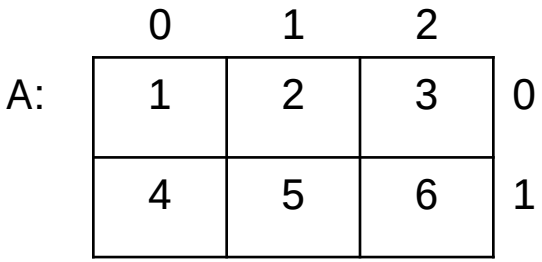
A:	0	1	2	
	1	2	3	0
	4	5	6	1

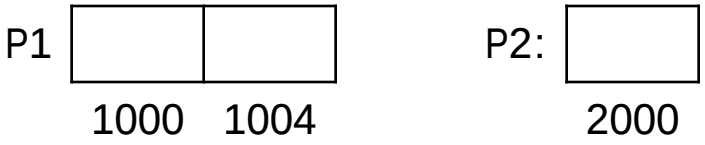
2-D Arrays

- Pointers for 2-D array
  - **`int *P1[2];`**
    - P1 is an array of 2 elements of type pointer to int
  - **`int (*P2)[3];`**
    - P2 is a pointer to an array of 3 elements of type int
  - What is sizeof(p1) & sizeof(p2)?
    - 8 & 4 respectively

The actual storage:



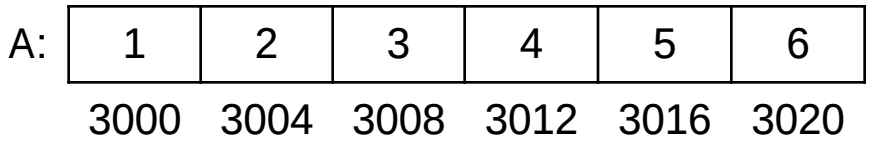


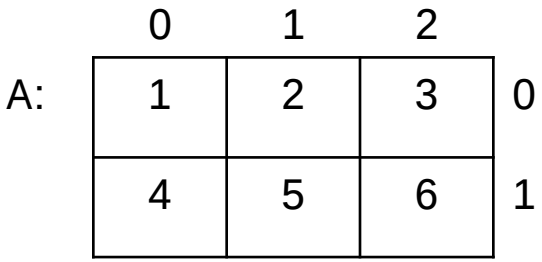


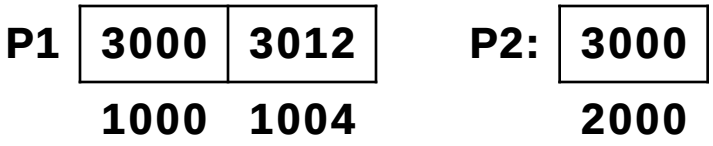
2-D Arrays

- Pointers for 2-D array
  - **int *P1[2];**
    - P1 is an array of 2 elements of type pointer to int
  - **int (*P2)[3];**
    - P2 is a pointer to an array of 3 elements of type int
  - What is sizeof(p1) & sizeof(p2)?
    - 8 & 4 respectively
  - **P1[0] = &A[0][0];**
  - **P1[1] = &A[1][0];**
  - **P2 = &A[0];**

The actual storage:

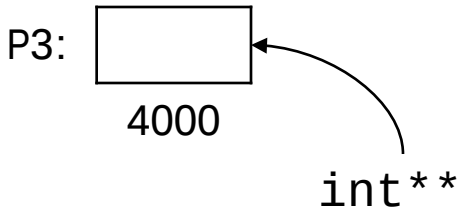






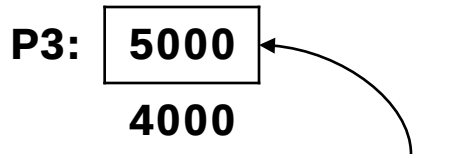
2-D Arrays

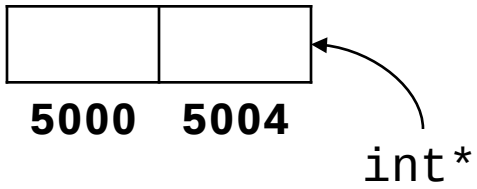
- Pointers for 2-D array
  - **`int **P3;`**
    - `P3 is a pointer to a pointer to int`
    - `P3 is a pointer to an int pointer`
  - `What is sizeof(p3)?`
    - `4`
  - `P3 = (int**)malloc(2*sizeof(int*));`
  - `P3[0] = &A[0][0];`
  - `P3[1] = &A[1][0];`



2-D Arrays

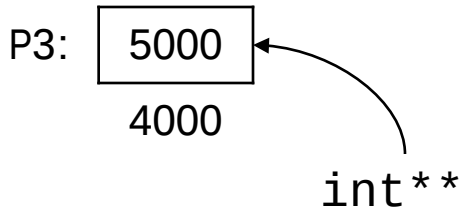
- Pointers for 2-D array
  - `int **P3;`
    - P3 is a pointer to a pointer to int
    - P3 is a pointer to an int pointer
  - What is sizeof(p3)?
    - 4
  - **`P3 = (int**)malloc(2*sizeof(int*));`**
  - `P3[0] = &A[0][0];`
  - `P3[1] = &A[1][0];`

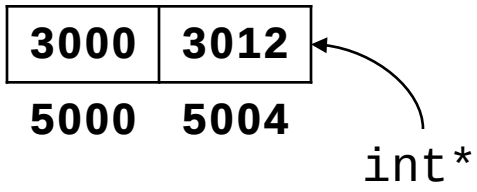




2-D Arrays

- Pointers for 2-D array
  - `int **P3;`
    - P3 is a pointer to a pointer to int
    - P3 is a pointer to an int pointer
  - What is sizeof(p3)?
    - 4
  - `P3 = (int**)malloc(2*sizeof(int*));`
  - **`P3[0] = &A[0][0];`**
  - **`P3[1] = &A[1][0];`**





2-D Arrays

- Evaluate P1[1][2]
  - `*( *(P1 + 1) + 2)`
  - `*( *(1000 + 1*sizeof(int*)) + 2)`
  - `*( *(1000 + 1*4) + 2)`
  - `*( *(1004) + 2)`
  - `*( 3012 + 2*sizeof(int) )`
  - `*( 3012 + 2*4 )`
  - `*(3020)`
  - 6

2-D Arrays

- Evaluate P2[1][2]
  - `*( *(P2 + 1) + 2)`
  - `*( *(3000 + 1*sizeof(int[3]) ) + 2)`
  - `*( *(3000 + 1*12) + 2)`
  - `*( *(3012) + 2)`
  - `*( 3012 + 2*sizeof(int))`
  - `*( 3012 + 2*4)`
  - `*(3020)`
  - 6

2-D Arrays

- Evaluate P3[1][2]
  - `*( *(P3 + 1) + 2)`
  - `*( *(5000 + 1*sizeof(int*) ) + 2)`
  - `*( *(5000 + 1*4) + 2)`
  - `*( *(5004) + 2)`
  - `*( 3012 + 2*sizeof(int))`
  - `*( 3012 + 2*4)`
  - `*(3020)`
  - 6

Dynamic allocation

- Assume a data type ALPHA
- To create an array of type ALPHA having N elements we write
  - `ALPHA *P;`
  - `P = (ALPHA*)malloc(N*sizeof(ALPHA));`
- This is the general syntax to allocate memory dynamically

Dynamic allocation

- Let us declare an 4x5 array dynamically
- Step 1: It's a 2-D structure so we take a double pointer

```
int **P;
```

- Step 2: We want 4 rows of 5 integers each

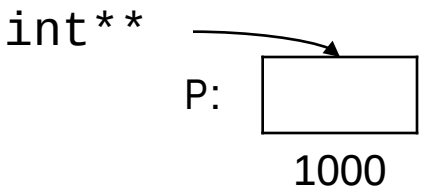
```
P = (int**) malloc(4*sizeof(int*));
```

- Step 3: Each row is to be associated with an array of 5 integers

```
for(i=0; i<4; i++)
  P[i] = (int*) malloc(5*sizeof(int));
```

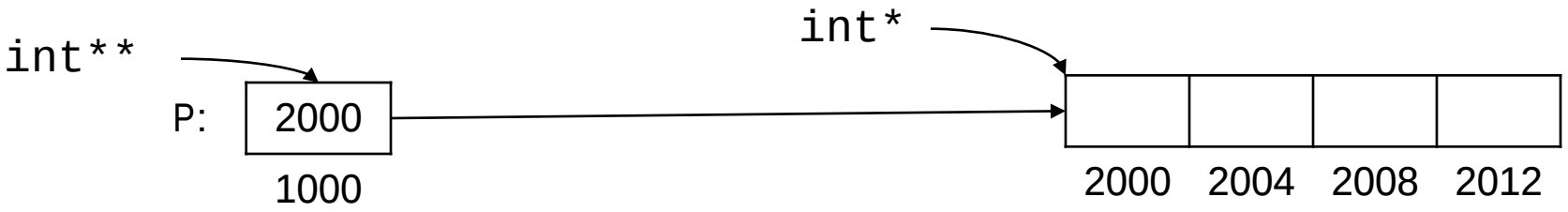
Dynamic allocation

```
1.   int **P;
2.   P = (int**) malloc(4*sizeof(int*));
3.   for(i=0; i<4; i++)
4.     P[i] = (int*) malloc(5*sizeof(int));
```



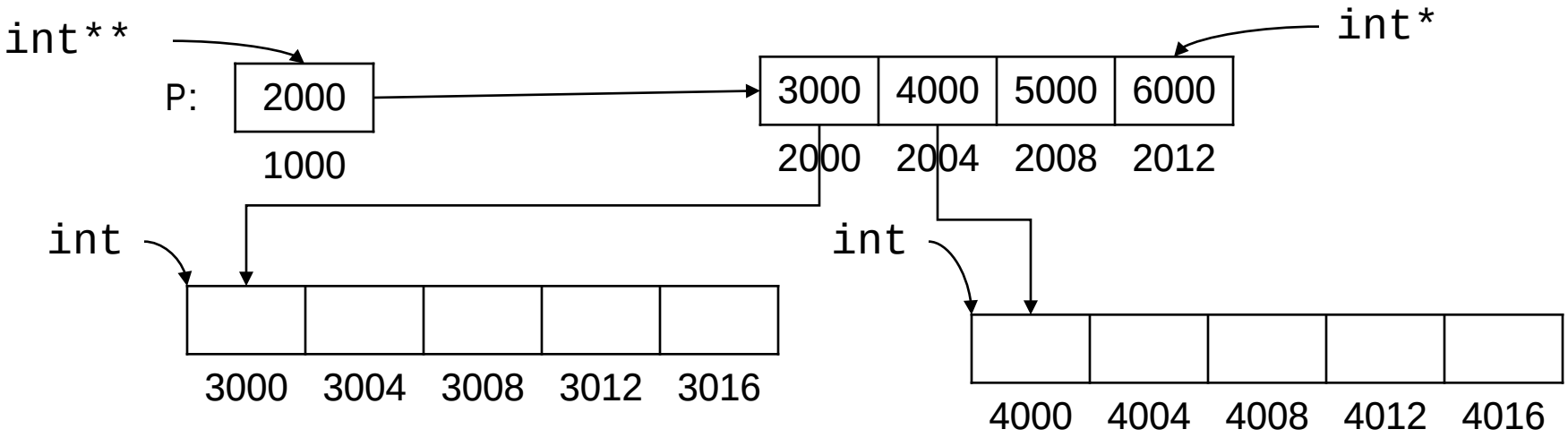
Dynamic allocation

```
1.   int **P;
2.   P = (int**) malloc(4*sizeof(int*));
3.   for(i=0; i<4; i++)
4.     P[i] = (int*) malloc(5*sizeof(int));
```



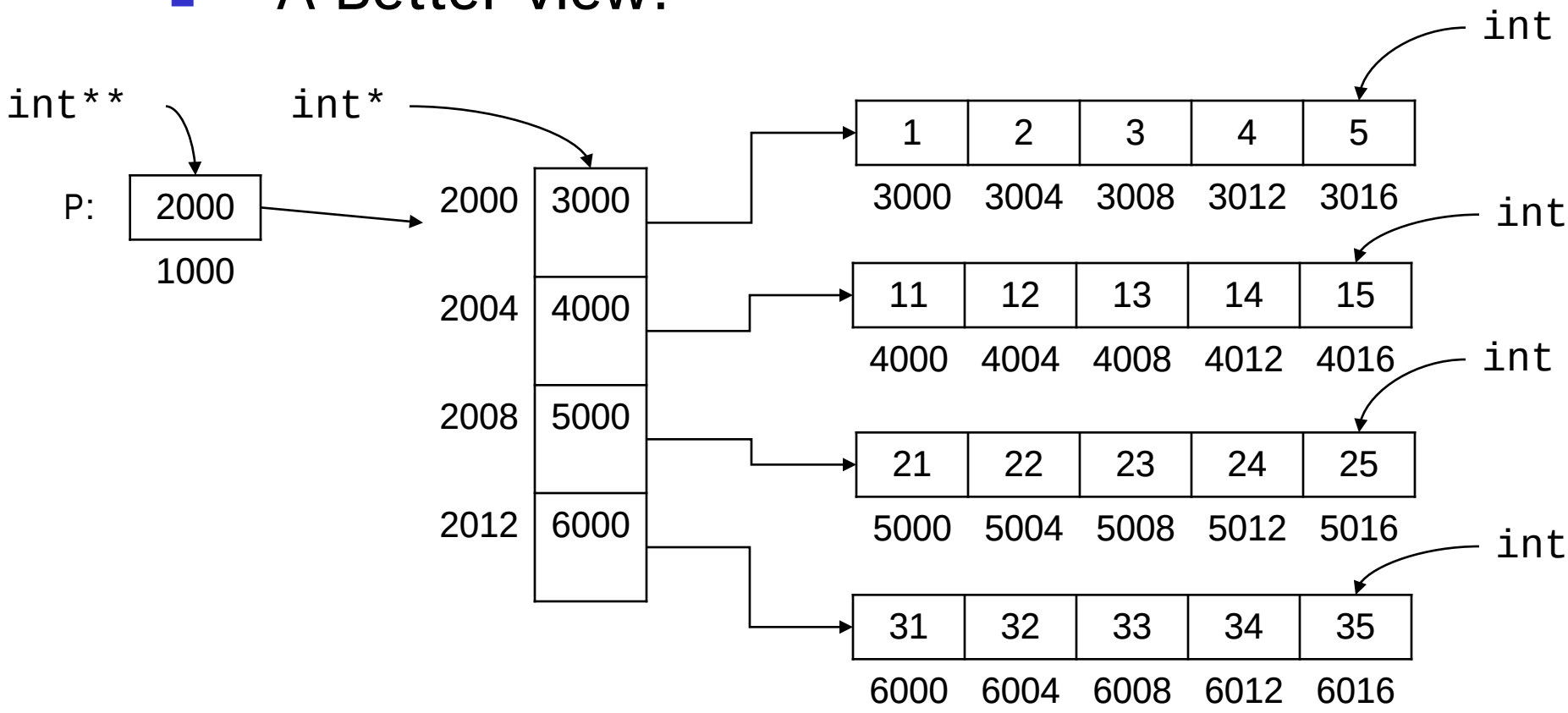
Dynamic allocation

```
1.   int **P;
2.   P = (int**) malloc(4*sizeof(int*));
3.   for(i=0; i<4; i++)
4.     P[i] = (int*) malloc(5*sizeof(int));
```



Dynamic allocation

- A Better view:

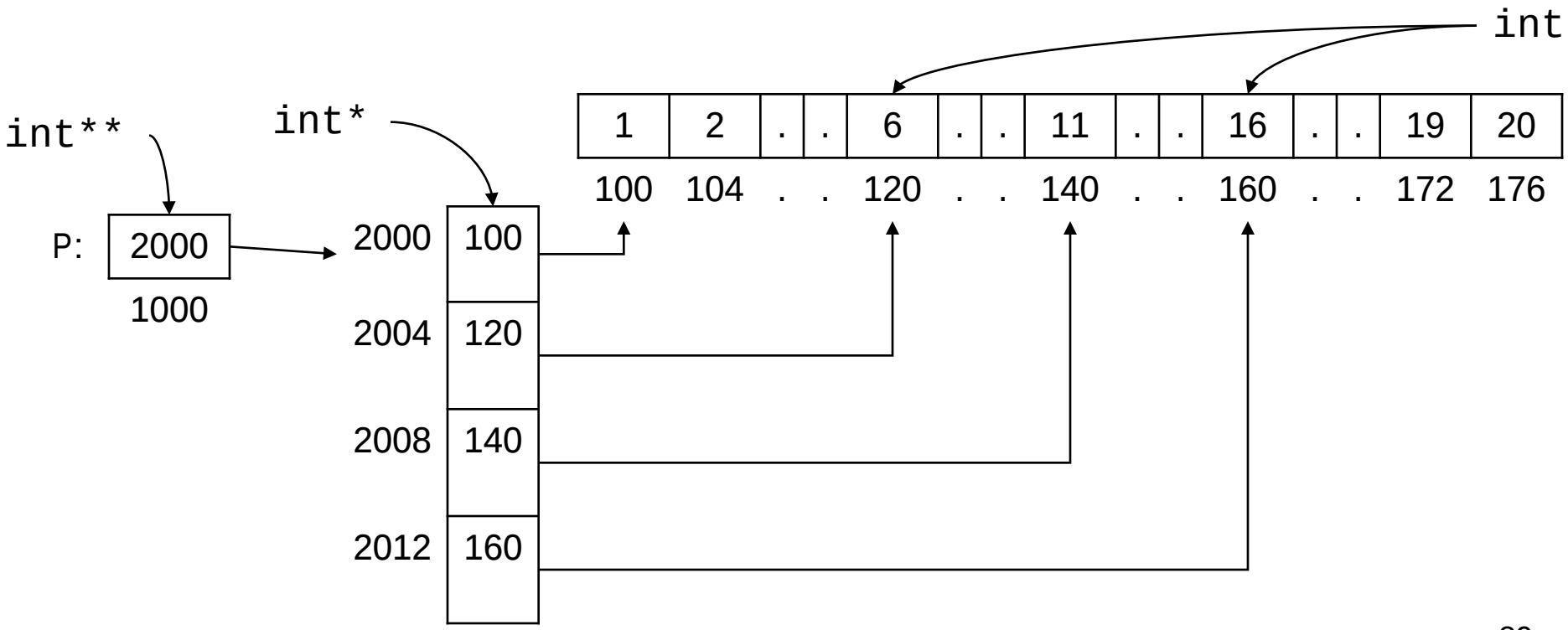


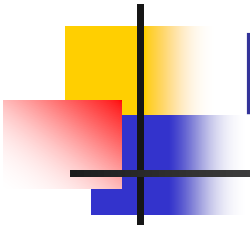
Dynamic allocation

- Evaluate P[2][3]
  - `*( *(P + 2) + 3)`
  - `*( *(2000 + 2*sizeof(int*) ) + 3)`
  - `*( *(2000 + 2*4) + 3)`
  - `*( *(2008) + 3)`
  - `*( 5000 + 3*sizeof(int))`
  - `*( 5000 + 3*4)`
  - `*(5012)`
  - `24`

Quiz time

- Write code to allocate 2-D array such that ALL elements are stored contiguously





Dynamic allocation

- Dangling pointer
  - When a pointer variable continues to point to some de-allocated memory.
  - Dangling pointer may cause segment related run-time errors. The famous segmentation fault
- Memory leakage
  - When there exists no pointer or reference to a dynamically allocated memory area.
  - May limit the programs capabilities
- *Both are e.g. of bad programming practices*

Break: Declarations

- `int ***P1;`
- `int *P2[5];`
- `int (*P3)[5];`

- P1 is a pointer to pointer to pointer to int
- P2 is an array of 5 elements of type pointer to int
- P3 is a pointer to an array of 5 elements of type int

Break: Declarations

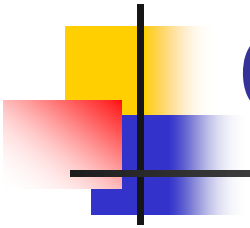
- `int (*P4)[4][5];`
- `int *P5[2][3];`
- `int *(*P6)[2][3];`

- P4 is a pointer to an array of 4 elements of type array of 5 elements of type int
- P5 is an array of 2 elements of type array of 3 elements of type int pointer
- P6 is a pointer to an array of 2 elements of type array of 3 elements of type int*

Break: Declarations

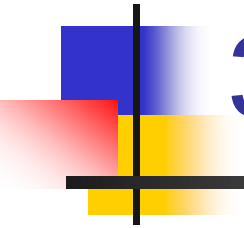
- `int *F1();`
- `int (*F2())[4];`
- `int (*F3())[3][4];`
- `int (*F4)(char);`

- F1 is function returning int pointer
- F2 is a function returning pointer to array of 4 elements of type int
- F3 is a function returning pointer to a 3x4 array of type int
- F4 is a pointer to function taking char and returning int



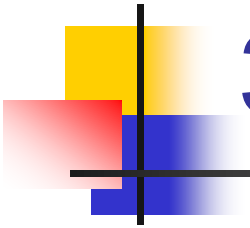
Quiz time

- Dynamically allocate memory for storing triangular arrays.
- Note: Using the regular 2-D array wastes half of the memory.



3-D Arrays

I don’t know how to teach these.



3-D Arrays

- A small revision:
- For 1-D array
  - `int A[n]`
  - `int *A`
- For 2-D array
  - `int A[m][n]`
  - `int *A[m]`
  - `int (*A)[n]`
  - `int **A`
- For 3-D Arrays
  - `int A[x][y][z]`
  - `int *A[x][y]`
  - `int **A[x]`
  - `int ***A`
  - `int (*A)[y][z]`
  - `int* (*A)[y]`

Pointer return types

- Write a function that returns an array of size 5 with its elements initialized to 0.
- Answer 1

```
int* func1()
{
  int A[5] = {0};
  return &A[0];
}
```

Pointer return types

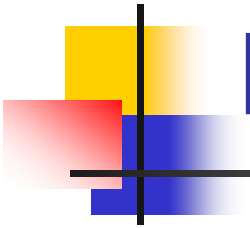
- Answer 2

```
int* func2()
{
  static int A[5] = {0};
  return &A[0];
}
```

Pointer return types

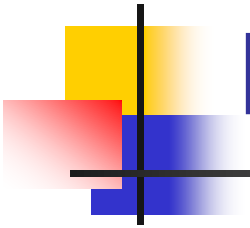
- Answer 3

```
int* func3()
{
  int i, *A = malloc(5*sizeof(int));
  for(i=0; i<5; i++)
    A[i] = 0;
  return A;
}
```



Pointer return types

- Analyzing the answers
- Answer 1
  - Segmentation fault. Why?
  - The address returned is address of deallocated memory.
- Answer 2
  - You cannot use this function more than once. Why?
  - Static variables are allocated only once and remain there forever. This may cause data corruption.



Pointer return types

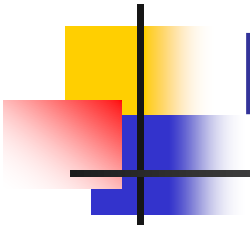
- Analyzing the answers
- Answer 3
  - Correct answer. Why?
  - Each time this function is called, a fresh block of 20 bytes is allocated. Unlike Answer 1 this memory is not de-allocated because it uses dynamic allocation. And dynamically allocated memory can only be EXPLICITLY de-allocated using free.
  - Thus,
    - No Segmentation fault.
    - No Data corruption.

Some more e.g.

- `char S1[] = “Hello world”;`
- `char *S2 = “Hello world”;`
- `char *S3;`
- `S1 = “Ram”;  // Error`
  - S1 is an array and thus its address is constant
- `S2 = “Ram”;  // OK`
- `S1[0] = ‘X’;  // OK`

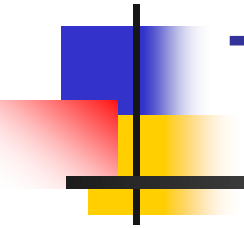
Some more e.g.

- `S2[0] = 'X';` `// Error`
  - S2 is a pointer to constant char
- `scanf("%s", S1);` `// OK`
- `scanf("%s", S2);` `// Error`
  - Reason is same as above
- `scanf("%s", S3);` `// Error`
  - Causes Segmentation fault because we never allocated memory for S3. So it may point to an inaccessible location.



References

- Pointers in C
  - – Kanetkar
- C Programming Language
  - – Ritchie



Thank You