

# Today's class

- Randomized quick sort
- Pointers
- Dynamic arrays

# Randomized Quick sort

```
void quicksort(int arr[], int lo, int hi)
{
    if (lo >= hi) {
        return;
    }
    int p = partition(arr, lo, hi);
    quicksort(arr, lo, p - 1);
    quicksort(arr, p + 1, hi);
}
```

```
int partition(int arr[], int lo, int hi) {
    int idx = get_random_idx(lo, hi);
    exchange(arr, lo, idx);
    int pivot = arr[lo];
    int left = lo + 1;
    int right = hi;

    while (left <= right) {
        while (left <= right && arr[left] < pivot) {
            left += 1;
        }
        while (right >= left && arr[right] > pivot) {
            right -= 1;
        }
        if (left <= right) {
            exchange(arr, left, right);
            left++; right--;
        }
    }

    exchange(arr, lo, right);
    return right;
}
```

In the randomize quick sort algorithm, we pick the pivot element randomly. If the pivot is truly random, then the probability of getting the final position of the pivot as 0, 1, 2, ..., N-1 would be  $1/N$ . However, generating a truly random number is hard, and that is a separate discussion outside the scope of this course. This algorithm is suitable for average time complexity.

# Time complexity

```
void quicksort(int arr[], int lo, int hi)
{
    if (lo >= hi) {
        return;
    }
    int p = partition(arr, lo, hi);
    quicksort(arr, lo, p - 1);
    quicksort(arr, p + 1, hi);
}
```

## Average case:

In the randomized algorithm, the probabilities of getting the final position of the pivot as 0, 1, 2, ..., n-1 are equal. So, the average time complexity is the arithmetic mean of the number of operations in all N possible partitions.

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} T(i) + T(n-i-1) + C_1n + C_2$$

```
int partition(int arr[], int lo, int hi) {
    int idx = get_random_idx(lo, hi);
    exchange(arr, lo, idx);
    int pivot = arr[lo];
    int left = lo + 1;
    int right = hi;

    while (left <= right) {
        while (left <= right && arr[left] < pivot) {
            left += 1;
        }
        while (right >= left && arr[right] > pivot) {
            right -= 1;
        }
        if (left <= right) {
            exchange(arr, left, right);
            left++; right--;
        }
    }

    exchange(arr, lo, right);
    return right;
}
```

If the partitioning algorithm returns an index  $i$ , the first recursive call performs  $T(i)$  operations and the second recursive call performs  $T(n-i-1)$  operations. If we take the average of all possible outcomes for  $i$ , we get the recurrence relation shown on this slide.

## Time complexity (average case)

$$T(1) = c$$

$$T(n) = \frac{1}{n} \left( \sum_{i=0}^{n-1} (T(i) + T(n-1-i)) \right) + c_1 n + c_2$$

$$nT(n) = \left( \sum_{i=0}^{n-1} (T(i) + T(n-1-i)) \right) + c_1 n^2 + c_2 n$$

$$nT(n) = \left( \sum_{i=0}^{n-1} T(i) + \sum_{i=n-1}^0 T(i) \right) + c_1 n^2 + c_2 n$$

$$nT(n) = 2 \sum_{i=0}^{n-1} T(i) + c_1 n^2 + c_2 n$$

$$(n-1)T(n-1) = 2 \sum_{i=0}^{n-2} T(i) + c_1 (n-1)^2 + c_2 (n-1)$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + c_1(2n-1) + c_2$$

$$nT(n) = 2T(n-1) + (n-1)T(n-1) + 2c_1 n + c_3$$

To solve the recurrence relations in which the entire history is present, we can compute the value of  $T(n-1)$  and subtract it from  $T(n)$ . This will ensure that most of the history will be canceled out and the corresponding equation is in the form that can be solved using the expansion method.

## Time complexity (average case)

$$nT(n) = (n+1)T(n-1) + 2c_1n + c_3$$

$$T(n) = \frac{n+1}{n} T(n-1) + 2c_1 + \frac{c_3}{n}$$

$$T(n) \leq \frac{n+1}{n} T(n-1) + 2c_1 \quad \text{TYPO: Use } c_4 \text{ instead of } 2c_1 \text{ from here onwards}$$

$$\leq \frac{n+1}{n} \left( \frac{n}{n-1} T(n-2) + 2c_1 \right) + 2c_1$$

$$\leq \frac{n+1}{n-1} T(n-2) + 2c_1 \left( 1 + \frac{n+1}{n} \right)$$

$$\leq \frac{n+1}{n-1} \left( \frac{n-1}{n-2} T(n-3) + 2c_1 \right) + 2c_1 \left( 1 + \frac{n+1}{n} \right)$$

$$\leq \frac{n+1}{n-2} T(n-3) + 2c_1 \left( 1 + \frac{n+1}{n} + \frac{n+1}{n-1} \right)$$

$$\leq \frac{n+1}{n-k+1} T(n-k) + 2c_1 \left( 1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \dots + \frac{n+1}{n-k+2} \right)$$

Here, we are assuming that  $c_3/n$  will always be less than some other constant. Therefore, we can use another constant  $c_4$  instead of  $2c_1 + c_3/n$ .

## Time complexity (average case)

$$T(n) \leq \frac{n+1}{n-k+1} T(n-k) + 2c_1 \left( 1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \dots + \frac{n+1}{n-k+2} \right)$$

Substitute :  $n-k = 1$

$$T(n) \leq \frac{n+1}{2} T(1) + 2c_1 \left( 1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \dots + \frac{n+1}{3} \right)$$

$$\leq \frac{n+1}{2} \times c + 2c_1 (n+1) \left( 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n+1} - \frac{3}{2} \right)$$

$$\leq \frac{n+1}{2} \times c + 2c_1 (n+1) \left( 2 \log(n+1) - \frac{3}{2} \right)$$

$$O(n \log n)$$

# Harmonic series sum

- [https://en.wikipedia.org/wiki/Harmonic\\_series\\_\(mathematics\)](https://en.wikipedia.org/wiki/Harmonic_series_(mathematics))
  - Comparison test
  - Integral test

# Harmonic series sum

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

$$\leq 1 + \underbrace{\left(\frac{1}{2} + \frac{1}{2}\right)} + \underbrace{\left(\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}\right)} + \dots +$$

let's say there are  $k$  phases

$$\leq k$$

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \leq \log_2(n+1)$$

$$1 + 2 + 2^2 + \dots + 2^{k-1} = n$$

$$2^k - 1 = n$$

$$k = \log_2(n+1)$$

To find an upper bound, we can replace  $1/3$  with  $1/2$ ;  $1/5$ ,  $1/6$ ,  $1/7$  with  $1/4$ ;  $1/9, 1/10, \dots, 1/15$  with  $1/8$ , and so on until we have replaced all terms. Let's say the group of all the terms with common denominators is a phase, and there are  $k$  phases. Therefore, the sum of the harmonic series would be  $k$ . Now, let's add the number of terms in each phase to get the total number of terms. The number of terms in phase-1 is 1, the number of terms in phase-2 is 2, the number of elements in phase 3 is  $2^2$ , and so on. Therefore, the number of elements in the  $k$ th phases would be  $2^{(k-1)}$ . The sum of the number of terms in all phases is the sum of the geometric series:  $1, 2, 2^2, 2^3, \dots, 2^{(k-1)}$ , which is equal to the total number of terms  $n$ . Solving these equations gives us the upper bound as  $\log_2(n+1)$ .



# Harmonic series sum

$$\begin{aligned}
 & 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} \\
 & \geq 1 + \frac{1}{2} + \underbrace{\left(\frac{1}{4} + \frac{1}{4}\right)}_{\text{phase 1}} + \underbrace{\left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right)}_{\text{phase 2}} + \dots \\
 & \geq 1 + \frac{k}{2}
 \end{aligned}$$

$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$   
 $\geq 1 + \frac{\log_2 n}{2}$

$$\begin{aligned}
 1 + 2 + 2^2 + \dots + 2^{k-1} &= n-1 \\
 2^k - 1 &= n-1 \\
 2^k &= n \\
 k &= \log_2 n
 \end{aligned}$$

To find a lower bound, we can group them differently. We can replace  $1/3$  with  $1/4$ ;  $1/5, 1/6, 1/7$  with  $1/8$ ;  $1/9, 1/10, \dots, 1/15$  with  $1/16$ , and so on until we have replaced all terms. Let's say the group of all the terms with common denominators except the first term is a phase, and there are  $k$  phases. The sum of the harmonic series would be  $1 + k/2$ . Now, let's add the number of terms in each phase to get the total number of terms. The number of terms in phase-1 is 1, the number of terms in phase-2 is 2, the number of elements in phase 3 is 4, and so on. Therefore, the number of elements in the  $k$ th phases is  $2^{(k-1)}$ . The sum of the number of terms in all phases is the sum of the geometric series:  $1, 2, 2^2, 2^3, \dots, 2^{(k-1)}$ , which is equal to  $n-1$ . Solving these equations gives us the lower bound as  $1 + (\log_2(n)/2)$ .

## Time complexity (average case)

- Time complexity, when the partitioning algorithm returns  $i$

$$T(n) = T(i) + T(n - 1 - i) + c_1n + c_2$$

If each index has the same probability of being selected as a target position for the pivot, then the average complexity is

$$T(n) = \frac{1}{n} \left( \sum_{i=0}^{n-1} (T(i) + T(n - 1 - i)) \right) + c_1n + c_2$$

## Time complexity (average case)

$$T(n) = \frac{1}{n} \left( \sum_{i=0}^{n-1} (T(i) + T(n-1-i)) \right) + (c_1 n + c_2)$$

$$T(n) = \frac{2}{n} \left( \sum_{i=0}^{n-1} T(i) \right) + (c_1 n + c_2) \quad \text{(Full history recurrence relation)}$$

$$nT(n) = 2 \left( \sum_{i=0}^{n-1} T(i) \right) + c_1 n^2 + c_2 n$$

$$(n-1)T(n-1) = 2 \left( \sum_{i=0}^{n-2} T(i) \right) + c_1 (n-1)^2 + c_2 (n-1)$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + c_1(2n-1) + c_2$$

$$T(n) = \frac{n+1}{n} T(n-1) + \frac{c_2 - c_1}{n} \leq \frac{n+1}{n} T(n-1) + c$$

Here, we are assuming that “ $(c_2 - c_1)/n$ ” will always be less than some other constant  $c_3$ , and  $2c_1 + c_3 = c$ .

## Time complexity(average case)

$$\begin{aligned}T(n) &\leq \frac{n+1}{n}T(n-1) + c \\&= \frac{n+1}{n}\left(\frac{n}{n-1}T(n-2) + c\right) + c = \frac{n+1}{n-1}T(n-2) + c\left(1 + \frac{n+1}{n}\right) \\&= \frac{n+1}{n-1}\left(\frac{n-1}{n-2}T(n-3) + c\right) + c\left(1 + \frac{n+1}{n}\right) = \frac{n+1}{n-2}T(n-3) + \\&\quad c\left(1 + \frac{n+1}{n} + \frac{n+1}{n-1}\right) \\&= \dots \\&= \frac{n+1}{n-k+1}T(n-k) + c\left(1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \dots + \frac{n+1}{n-k+2}\right)\end{aligned}$$

## Time complexity (average case)

Substituting  $k = n - 1$

$$\begin{aligned} T(n) &\leq \frac{n+1}{2} T(1) + c \left( 1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \dots + \frac{n+1}{3} \right) \\ &= \frac{n+1}{2} c_3 + c(n+1) \left( 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n+1} - \frac{3}{2} \right) \end{aligned}$$

Because,  $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$  is  $O(\log n)$

$$= \frac{n+1}{2} c_3 + c(n+1) (c_4(\log(n+1)) - \frac{3}{2}) = O(n \log n)$$

## Harmonic series sum

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

$$\leq 1 + \left(\frac{1}{2} + \frac{1}{2}\right) + \left(\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}\right) + \dots$$

$$\leq 1 + 1 + 1 + \dots \text{upto } k \text{ terms} = k$$

$$1 + 2 + 2^2 + \dots + 2^{k-1} = n$$

$$2^k - 1 = n$$

$$k = \log_2(n + 1)$$

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \leq \log_2(n + 1)$$

# Harmonic series sum

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

$$\geq 1 + \frac{1}{2} + \left(\frac{1}{4} + \frac{1}{4}\right) + \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right) + \cdots$$

$$\geq 1 + \frac{k}{2}$$

$$1 + 2 + 2^2 + \cdots + 2^{k-1} = n - 1$$

$$2^k - 1 = n - 1$$

$$k = \log_2 n$$

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \geq 1 + \frac{\log_2 n}{2}$$

# Quick sort vs merge sort

## Quick sort

- The worst-case time complexity is  $O(n^2)$
- Require very little extra space (an in-place algorithm)
- Empirically, quick sort performs better than merge sort

## Merge sort

- The worst-case time complexity is  $O(n \log n)$
- Requires extra space during the merge operation (not in-place)
- Empirically, merge sort performs poorly than quick sort



# Quick sort

- C++ standard library implements a hybrid sorting algorithm, Introsort, to mitigate the worst-case behavior of the quick sort
  - <https://en.wikipedia.org/wiki/Introsort>
  - Quick sort + Heap sort (worst case  $n \log n$ ) + Insertion sort ( $O(n^2)$ )

# Homework

- Modify the randomize quick sort algorithm to find the median of  $n$  numbers
  - No need to sort the entire array
- There is also an algorithm with  $O(n)$  worst-case complexity for finding median of  $n$  numbers
  - You will get a chance to implement that algorithm in the bonus assignment

# Asymptotic analysis

# Shortcomings of asymptotic analysis

- Ignores the highest-order constant factors
  - An algorithm that takes  $100n$  operations is considered the same as an algorithm that takes  $2n$  operations, but the later is  $50x$  faster
    - As asymptotic analysis is done at a very high level, computing the correct constant factor is challenging because of its dependency on the underlying hardware and compiler
- Other constant factors are also ignored
  - $2n^2$  is better than  $1000n + 10000$  for small inputs, but asymptotic analysis ignores it
- Only consider the worst-case performance
  - Even though the average running time is significantly lower than the worst case, the algorithm may be still be considered bad because the average-time analysis is extremely complex for many problems

# Pointers

# Assignment

- We can only do an assignment if the type of the variable in LHS and the type of the value in the RHS are the same
  - Otherwise, it is an error
- When we do an assignment: `a = b`, where the type of `a` and `b` is `Ty`, `sizeof(Ty)` bytes are copied from `b` to `a`

# Assignment

```
int a;
```

```
a = 10;
```

```
// Is this a valid assignment?  Yes
```

```
// How many bytes are copied during the assignment?  4
```

`a = 10` is a valid assignment because both sides are of type `int`. Four bytes will be copied because the size of `int` is 4.

# Assignment

```
struct record {  
    int a;  
    int b;  
    int c[20];  
};
```

```
struct record var1;  
struct record var2;  
var1.a = 10;  
var1.b = 20;  
var2 = var1;
```

// Is this a legal assignment? YES

// How many bytes will be copied? 88

```
printf("%d %d %d %d\n", var2.a, var2.b, var2.c[0], var2.c[1]);
```

// what will be the output of the printf? 10 20

“var2 = var1” is a legal assignment because both are of type struct record. Eighty-eight bytes will be copied because the size of struct record is 88.



# Assignment

```
struct record {  
    int a;  
    int b;  
    int c[20];  
};
```

```
struct record var1;  
int var2;  
var1.a = 10;  
var1.b = 20;  
var2 = var1;
```

// Is this a legal assignment? **No**  
// How many bytes will be copied?

```
printf("%d %d %d %d\n", var2.a, var2.b, var2.c[0], var2.c[1]);  
// what will be the output of the printf?
```

var2 = var1 is not a legal assignment because their types are different.

# Assignment

```
struct record {  
    int a;  
    int b;  
    int c[20];  
};
```

```
struct record var1;  
struct record var2;  
var1.a = 10;  
var1.b = 20;  
var2.a = var1;
```

// Is this a legal assignment? **No**  
// How many bytes will be copied?

```
printf("%d %d %d %d\n", var2.a, var2.b, var2.c[0], var2.c[1]);  
// what will be the output of the printf?
```

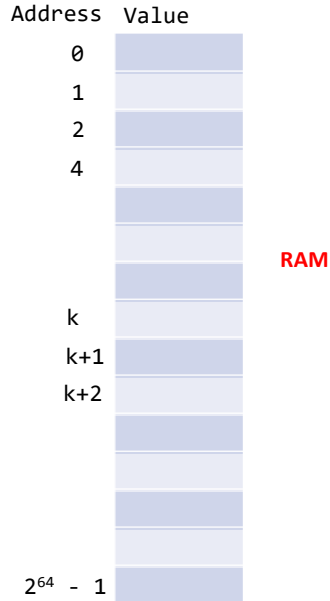
var2.a = var1 is not a legal assignment because their types are different.

# Memory layout

Every byte of the RAM has a unique address.

For every variable in your program, the compiler allocates space of size `sizeof(sizeof(variable))` in the RAM.

The address of (&) operator returns the address of a variable.



# Address of (&) operator

```
struct record {  
    int a;  
    int b;  
    int c[20];  
};  
  
struct record var1;  
var1.a = 10;  
var1.b = 20;  
  
printf("%p\n", &var1);  
// What will be the output of the printf?  
// How can we store the address of var1 in a variable?
```

The & operator returns the address of a variable in the RAM.

# Pointers

```
int a;
```

The **type** of **&a** is **int\***. **int\*** is called **pointer** to an integer.

If a variable “a” is of type “int”, the type of “&a” is “int\*”.

# Pointers

- A pointer variable stores the address of a variable (i.e., a RAM address)

```
int a = 10;  
int *x = &a;    // the type of &a is int*  
int **y = &x;   // the type of &x is int**  
int ***z = &y;  // the type of &z is int***
```

**struct record r;**

What is the type of variable that can store the address of variable r?

~~struct record \*v = &r;~~

Because “a” is of type “int”, the type of “&a” is “int\*”. Therefore, x = &a is a valid assignment because both sides are of the same type. Similarly, the type of &x is int\*\* because the type of x is int\*. In general, the type of the address of a variable of type

# Pointers

```
struct record {  
    int a;  
    int b;  
    struct record *next;  
};
```

```
struct record var1;  
var1.a = 10;  
var1.b = 20;  
var1.next = 30;
```

Is this a legal program? *No*

This is not a legal program because at the line "var1.next = 30", the type of var1.next is "struct record\*", and the type of 30 is int. Therefore, it is not a valid assignment.

# Pointers

```
struct record {  
    int a;  
    int b;  
    struct record *next;  
};
```

```
struct record var1;  
var1.a = 10;  
var1.b = 20;  
var1.next = &var1;
```

Is this a legal program?

yes

This is a legal program because the statement “var1.next = &var1” is valid. The type of var1.next is "struct record\*", and the type of &var1 is also "struct record\*".



# Pointers

```
struct record {  
    int a;  
    int b;  
    struct record *next;  
};
```

```
struct record var1;  
struct record *var2;  
var2 = &var1;  
var1.a = 10;  
var1.b = 20;  
var1.next = &var1;
```

Is this a legal program? *yes*

This is a legal program because the assignment `var2 = &var1` is valid. The type of `var2` is "struct record\*", and the type of `&var1` is also "struct record\*".

# Copying address

- We can copy the RAM address stored in the pointer variable `ptr` in a variable `var` using:

```
var = ptr;
```

```
int a = 10;  
int *x = &a;  
int y;  
int *z;
```

```
y = x;  
// Is this a legal assignment? No  
// How many bytes will be copied?  
// What will be the value of y?
```

```
z = x;  
// Is this a legal assignment? Yes  
// How many bytes will be copied? 8  
// What will be the value of z? &a
```

We can copy the RAM address stored in a pointer variable into another pointer variable of the same type by a simple assignment operation. The assignment `y = x` is not legal because the type of `x` is `int*` and `y` is `int`. The statement `z = x` is legal because the type of `x` and `z` both are `int*`. This statement copies the 8-byte RAM address stored in `x` to `z`. Notice that the size of the type `int*` is eight.

# Loading from RAM

- We can load the value stored in the RAM at an address stored in the pointer variable `ptr` into a variable `var` using:

```
var = *(ptr)
```

```
int a = 10;  
int *x = &a;  
int **y = &x;  
int ***z = &y;
```

```
The type of *(x) is int  
The type of *(y) is int *  
The type of *(z) is int **
```

```
What is the type of **(z)? int  
What is the type of ***(z)? int  
What is the type of **(y)? int  
What is the type of *(x)? error  
What is the type of *(a)? error
```

If the type of `ptr` is `int*`, the type of `*(ptr)` is `int`. If the type of `ptr` is `int**`, the type of `*(ptr)` will be `int*`. In general, if the type of a variable `ptr` is `Ty` followed by `i` stars, then the type of `*(ptr)` would be `Ty` followed by `(i-1)` stars, the type of `** (ptr)` would be `Ty` followed by `(i-2)` stars and so on. Dereferencing a variable `v` using `*v`, where the type of `v` is not a pointer type, is an error.

# Loading from RAM

- We can load the value stored in the RAM at an address stored in the pointer variable `ptr` into a variable `var` using:

```
var = *(ptr)
```

```
struct record {  
    int a;  
    int b;  
    struct record *next;  
};
```

```
struct record var1;  
struct record *var2;  
struct record var3;
```

```
var1.a = 10;  
var1.b = 20;  
var1.next = &var1;  
var2 = &var1;
```

```
// What is the type of &var1? struct record *  
// Is this a legal assignment? yes  
// How many bytes will be copied? 8
```

```
var3 = *(var2);
```

```
// What is the type of *(var2)? struct record  
// Is this a legal assignment? yes  
// How many bytes will be copied? 16 bytes
```

```
printf("%d %d %p\n", var3.a, var3.b, var3.next);
```

```
// What will be the output of the printf? 10, 20, &var1
```

`var2 = &var1` is a legal assignment because both sides are of type `struct record*`. `var3 = *(var2)` is also a legal assignment because both are of type `struct record`. During `var3 = *(var2)`, 16 bytes will be copied because the type of both sides is "struct record". When we do a pointer dereference in this case, it will load the 16-byte value into `var3` from the RAM starting from the address stored in `var2`. Because `var2` contains the address of `var1` at this point, the loading from RAM will eventually copy `var1` to `var3`.

# Storing in RAM

- You can store the value of a variable `var` in RAM at an address stored in the pointer variable `ptr` using:

`*(ptr) = var`

```
struct record {  
    int a;  
    int b;  
    struct record *next;  
};
```

```
struct record var1;  
struct record *var2;  
struct record var3;
```

```
var1.a = 10;  
var1.b = 20;  
var1.next = &var1;  
var2 = &var3;
```

```
// What is the type of &var3? struct record *  
// Is this a legal assignment? yes  
// How many bytes will be copied? 8
```

```
*(var2) = var1;  
// What is the type of *(var2)? struct record  
// Is this a legal assignment? yes  
// How many bytes will be copied? 16
```

```
printf("%d %d %p\n", var3.a, var3.b, var3.next);
```

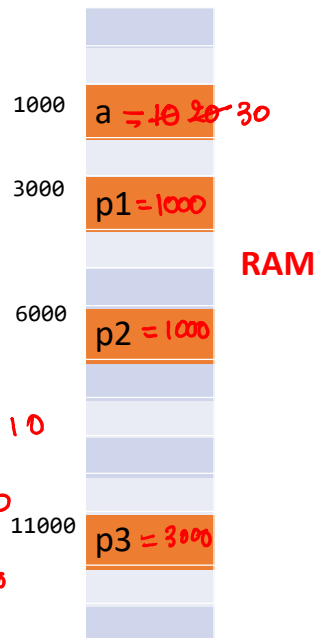
```
// What will be the output of the printf? 10 20 0x4d1
```

`var2 = &var3` is a legal assignment because both sides are of type `"struct record*"`.  
`*(var2) = var` is also a legal assignment because both are of type `"struct record"`.  
During `"*(var2) = var1"`, 16 bytes will be copied because the type of both sides is `"struct record"`. At this statement, a 16-byte value of `var1` will be stored in RAM starting from the address stored in `var2`. Because `var2` contains the address of `var3` at this point, storing in RAM will eventually copy `var1` to `var3`.

# Pointers

```
int a;
int *p1;
int *p2;
int **p3;
```

```
a = 10;
printf("%p %d\n", &a, a);      1000  10
p1 = &a;
printf("%p %p %d\n", &p1, p1, *(p1)); 3000  1000  10
p2 = p1;
printf("%p %p %d\n", &p2, p2, *(p2)); 6000  1000  10
p3 = &p1;
printf("%p %p %p %d\n", &p3, p3, *(p3), **(p3)); 11000  3000  1000  10
*(p1) = 20;
printf("%p %d %p %d\n", p1, *(p1), p2, *(p2)); 1000  20  1000  20
*(p2) = 30;
printf("%p %d %p %d\n", p1, *(p1), p2, *(p2)); 1000  30  1000  30
```



We are assuming that the starting address of a is 1000, p1 is 3000, p2 is 6000, and p3 is 11000. These addresses are assigned at runtime. When you execute this program, you may get different addresses on different machines.

# Pointer arithmetic

- The type of the expression  $(ptr + i)$ , where  $i$  is an integer and  $ptr$  is a pointer variable, is the same as the type of  $ptr$

```
struct record *p;  
struct record **q;  
int i;
```

What is the type of " $p + i$ "?

What is the type of " $p - i$ "?

What is the type of " $q + i$ "?

What is the type of " $q - i$ "?

struct record \*  
struct record \*  
struct record \*\*  
struct record \*\*

# Pointer arithmetic

- The **pointer arithmetic** rules are different from **integer arithmetic**
  - $(p + i)$  is the RAM address of type **typeof(p)** after skipping  $i$  values of type **typeof(\*p)** starting from  $p$
  - $(p - i)$  is the RAM address of type **typeof(p)** after skipping  $i$  values of type **typeof(\*p)** starting from  $p$  in the reverse direction



# Pointer arithmetic

- The rules for pointer arithmetic are different from integer arithmetic

```
int *p;  
int x;  
int *q;  
  
p = &x;  
// Let's assume, &x == 1000  
  
q = p + 1;  
What will be the value of q? 1004
```

Because the type of `p` is `int*`, `p + 1` is the address after skipping one "int" value starting from `p`. If the value of `p` is 1000, `p + 1` will be 1004. The type of `p+1` is the same as `p`, i.e., `int*`. `q = p + 1` is a valid assignment because the type of both `q` and `p+1` is `int*`.

# Pointer arithmetic

- The rules for pointer arithmetic are different from integer arithmetic

```
struct record {  
    int a[20];  
    int b;  
};
```

```
struct record *p;  
struct record x;  
struct record *q;
```

```
p = &x;  
// Let's assume, &x == 1000
```

```
q = p + 5;
```

What will be the value of q?

1000 + (84\*5)

Because the type of p is "struct record \*", the address p + 5 will be computed as the address after skipping five "struct record" values. The size of "struct record" is 84; therefore, the actual RAM address corresponding to p+5 is 1000 + (84\*5).

# Arrays and Pointers

# Array

- The array elements are stored in contiguous memory locations

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
------	------	------	------	------	------	------	------	------	------

```
int a[10];
```

The base (starting) address of the array “a” is &a[0].

If `&a[0] == X`, where  $(0 \leq X \leq 2^{64} - 1)$ .

In this case,

`&a[1] == X + 4`

`&a[2] == X + 8`

`&a[3] == X + 12`

...

`&a[i] == X + ?`

# Array

- The array elements are stored in contiguous memory locations

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
------	------	------	------	------	------	------	------	------	------

```
struct record {  
    int a;  
    struct record *next;  
};
```

```
struct record a[10];
```

If  $\&a[0] == X$ , where  $(0 \leq X \leq 2^{64} - 1)$ .

In this case,

$\&a[1] == X + 12$

$\&a[2] == X + 24$

$\&a[3] == X + 36$

...

$\&a[i] == X + (i \times 12)$