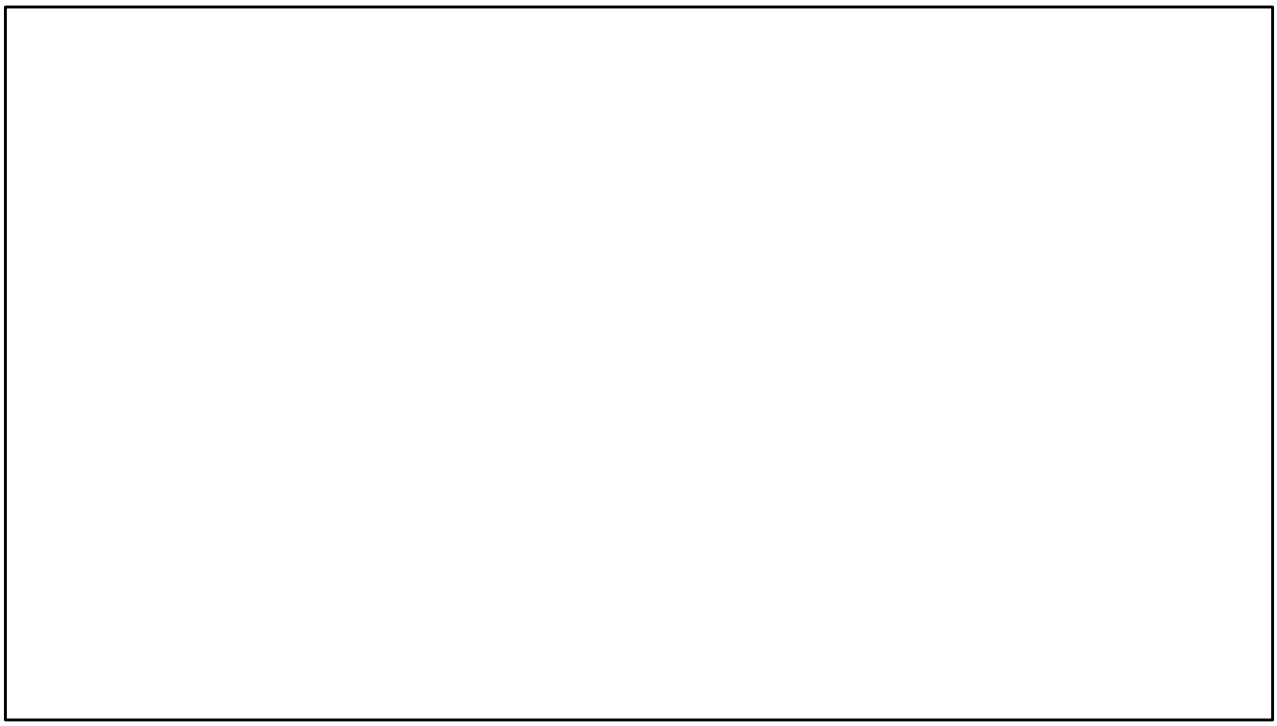


Today's class

- Pointers
- Manual memory management
- Dynamic arrays

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]

1000 1004 1008 . ..

```
int a[10];

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

If &a[0] == X, where (0 ≤ X ≤ 2^64 − 1).

In this case,
&a[1] == X + 4
&a[2] == X + 8
&a[3] == X + 12
…
&a[i]  == X + ?
```

The array elements are stored at consecutive addresses. If the address of the first element is X, the address of the second element will be X+4, the address of the third element will be X+12, and so on. The address of the ith element will be X + i*4. Here, 4 is the number of bytes to store an integer element.

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 ≤ X ≤ 2^64 − 1).

In this case,
&a[1] == X + 12
&a[2] == X + 24
&a[3] == X + 36
…
&a[i]  == X + (i*12)
```

If we have an array of type "struct record" and the address of the first element is X, the address of the second element will be X+12, the third will be X+24, and so on. The address of the ith element will be X + i*12. Here, 12 is the number of bytes to store an element of type "struct record".

Array

- Storing address of array elements in pointers

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];
struct record *p;
struct record *q;

If &a[0] == X, where (0 ≤ X ≤ 2^64 − 1).
p = &a[0];    // is this assignment valid?
q = p + 20;   // what would be the value of q?   X + (20*12)
q = p + 3;    // what would be the value of q?   X + (3*12)
q = p + i;    // what would be the value of q?   X + (i*12)
```

The assignment, p = &a[0] is valid because the type of a[0] is “struct record”, and therefore, the type of &a[0] is “struct record*”. Because the type of p is “struct record*” this assignment is valid. If &a[0] is X, then the value of p will be X after the assignment. Using the pointer arithmetic rule discussed before, the value of p+20 will be X+(20*12), p+3 will be X+(3*12), and p+i would be X+(i*12), where 12 is the size of an element of type struct record.

Array

- Accessing array elements using pointers

10	20	30	40	50	60	70	80	90	100

```
int a[10];
int *p;
int b;

p = &a[0];
b = *(p);     // what would be the value of b?
b = *(p + 5);  // what would be the value of b?
b = *(p + 20); // what would be the value of b?
*(p) = 100;
*(p+1) = 101;
*(p+1) = 102;
// what would be the contents of the array at this point?
```

When we load from p using *(p), we are actually loading from &a[0]. Therefore, the loaded value would be the first element of the array. *(p+5) loads the value of the sixth element of the array, and *(p+20) tries to load the value of the 20th element of the array (which may cause an exception as well). Similarly, storing using *(p) updates the first element of the array, and storing using *(p+1) updates the second element. Therefore, we can read/write array elements using two ways. One is directly using array “a” and secondly, storing the address of “&a[0]” in some pointer variable and accessing array elements using the pointer dereference operator (i.e., *).

Array

- Accessing array elements using pointers

10	20	30	40	50	60	70	80	90	100

```
int a[10];
int *p;
int b;

p = &a[0];
b = *(p);       ⇔    b = a[0];
b = *(p + 5);   ⇔    b = a[5];
b = *(p + 20);  ⇔    b = a[20];
*(p) = 100;     ⇔    a[0] = 100;
*(p+1) = 101;   ⇔    a[1] = 101;
*(p+1) = 102;   ⇔    a[1] = 102;
// what would be the contents of the array at this point?
```

In this case, *(p) is equivalent to a[0], *(p+20) is equivalent to a[20], and so on.

Syntactic sugar for load/store

Let's p is a variable of pointer type, i is an integer, and x is a variable of type *(p), then

*(p + i) = x; can also be written as p[i] = x;
x = *(p + i); can also be written as x = p[i];

Please don't confuse it with array indexing. It's just syntactic sugar.

Syntactic sugar makes things easier to express.

There is an alternative syntax (or syntactic sugar syntax) for writing *(p+i) that is p[i]. This syntax makes it easier to write programs that use pointer dereference.

Manual memory management

Static memory allocation

- So far, we have used static memory allocation that is done by the compiler automatically

```
void foo() {
  int a[100];
  int b;
  int c;
  …
}
```

- The memory for the variables a, b, and c are automatically allocated on entry to foo and deallocated before exiting from foo by the compiler

Manual memory management

```
void foo() {
  int a[100];
  int b;
  int c;
  …
}
```

- Using manual memory management, the programmers can control when to allocate and deallocate memory
  - e.g., programmers can allocate memory in foo and keep it valid even after foo returns. The programmers can control when to delete the memory allocated in foo.

malloc and free

- At runtime, a program can allocate space from RAM using malloc
  - malloc is a manual memory allocation API
  - manual allocation is also called dynamic memory allocation
- malloc takes the size of the memory area as an argument (say size), reserves size consecutive bytes in the RAM, and returns the starting address of the reserved area
- malloc ensures that nobody else can allocate the reserved area until the program invokes free to release the reserved area of memory

malloc

- By default, malloc returns an address of type void *
  - If the callers want to store a value of type int or struct record in the allocated area, they need to typecast it with int* or struct record*
- Typecasting is an unsafe operation, and it should be avoided
  - Unfortunately, in some cases, such as malloc, not typecasting is not an option because we certainly don't want to store a void value in the memory buffer
  - What is the other alternative if we want to avoid typecasting for malloc?

malloc

- By default, malloc returns an address of type void *
  - If the callers want to store a value of type int or struct record in the allocated area, they need to typecast it with int* or struct record*
- Typecasting is an unsafe operation, and it should be avoided
  - Unfortunately, in some cases, such as malloc, not typecasting is not an option because we certainly don't want to store a void value in the memory buffer
  - What is the other alternative if we want to avoid typecasting for malloc?
    - We need to define malloc routines corresponding to all possible return types
      - Not a feasible solution because we don't know all possible types in advance
      - Programmers can always create a new type using struct
  - C++ provides an alternative syntax for manual memory management that doesn't require a typecast operation

malloc

- We can use the following code to allocate memory for 10 integers

```
int *p = (int*)malloc(10 * sizeof(int));
```

Notice that we can't store a value of type void* in p, so we need to explicitly cast it to int* before assigning it to p

We can allocate a memory buffer using malloc. For example, the malloc call on this slide allocates a 40 bytes buffer from the RAM that can be used to store ten integers. We can store the starting address in a pointer and then access array elements using a combination of pointer arithmetic and dereference operations (as discussed earlier). One problem we encounter here is the return type of malloc is void*. If we want to store integers, we need to store the address in a variable of type int*. We can't directly store a void* value in a variable of type int* because of the type mismatch. To store the return value of malloc in a variable p of type int*, we can use an explicit typecast operation to convert the type to int*, before assigning it to p. In that case, the compiler will allow the assignment; however, these explicit typecast operations are unsafe, and we should avoid them as much as possible. In the case of malloc, we don't have any other option than the explicit typecast.

malloc

```
int *p = (int*)malloc(40);
```

We can store integers, e.g., 10, 20, …, 100, in the buffer using:

```
*(p) = 10;
*(p+1) = 20;
*(p+2) = 30;
…
*(p+9) = 100;
```

After storing the address in p, we can use the syntax listed on this slide to store values in the array.

malloc

```
int *p = (int*)malloc(40);
```

We can store integers, e.g., 10, 20, …, 100, in the buffer using the syntactic sugar syntax:

```
p[0] = 10;
p[1] = 20;
p[2] = 30;
…
p[9] = 100;
```

We can also use the syntactic sugar syntax discussed earlier to store values in the array.

malloc

```
int *p = (int*)malloc(40);
int q;
```

We can load integers from the buffer using the following:

```
q = *(p);
q = *(p+1);
q = *(p+2);
…
q = *(p+9);
```

After storing the address in p, we can use the syntax listed on this slide to load values from the array.

malloc

```
int *p = (int*)malloc(40);
int q;
```

We can also load integers from the buffer using the syntactic sugar syntax:

```
q = p[0];
q = p[1];
q = p[2];
…
q = p[9];
```

We can also use the syntactic sugar syntax discussed earlier to load values from the array.

malloc

We can use other types as well, e.g.,

```
struct record *p = (struct record*)malloc(sizeof(struct record) * 10);
struct record q;

q = p[0];
q = p[1];
p[2] = q;
p[9] = q;
…
```

if we want to allocate an array of “struct record” we can typecast the return value of malloc to “struct record*”, and store in a variable of type “struct record*”.

free

- Once we are done with the memory buffer, we can use free to release the memory allocated using malloc

```
struct record *p = (struct record*)malloc(sizeof(struct record) * 10);
struct record q;

q = p[0];
q = p[1];
p[2] = q;
p[9] = q;
…
free(p);
```

free

- What if an application never frees a malloced address?
  - Will the program behave properly?

free

- What if an application never frees a malloced address?
  - Will the program behave properly?
    - Yes, if we have enough memory, but the application will unnecessarily waste resources

Implicit typecast

- Sometimes, a C compiler performs implicit typecast when the types are not the same

```
char c = -128;
int a = c;    // implicit conversion from char to int
              // extend a 1-byte value to 4-byte

printf("%d %d\n", c, a);  // output: -128 -128

a = 129;
c = a;      // implicit conversion from int to char
            // truncate a 4-byte value to 1-byte
            // We may lose some information

printf("%d %d\n", c, a);  // output: -127 129
```

Sometimes, the compiler automatically does type conversion without giving any warning or error. For example, we you assign a character value to an integer variable, the 1-byte value automatically gets extended to 4-byte. Similarly, we you assign an int value to a char the 4-byte value get truncated to a 1-byte value. The later operation is unsafe because we may lose some information. You can generate warnings for unsafe implicit typecast using the "-Wconversion" flag during compilation.

Implicit typecast

- Implicit typecast could be a problem because some information may get lost due to the type mismatch
  - Use the gcc flag "-Wconversion" to enable warnings for unsafe implicit conversions

Parameters passing

Passing parameters

```
1. void swap(int a, int b) {
2.     int tmp = a;
3.     a = b;
4.     b = tmp;
5. }

6. int main() {
7.     int i = 10, j = 20;
8.     swap(i, j);
9.     printf("i=%d j=%d\n", i, j);
10.    return 0;
11.}
```

During the function call at line-8, the compiler creates space for variables (arguments) a and b in swap and copies the value of the parameters i and j in variables a and b.

int a = i;
int b = j;

How can we change the value of variables i and j in the swap routine?

During a function call, the compiler creates new variables corresponding to the arguments and assigns the parameter passed to the function to these variables. For example, in this case, when we call the swap routine, the compiler creates two new variables, "a" and "b", corresponding to the function's arguments, and stores the value of i in a and the value of j in b. The output of this program is i=10 j=20.

Passing parameters

```
1. void swap(int *a, int *b) {
2.     int tmp = *(a);
3.     *(a) = *(b);
4.     *(b) = tmp;
5. }

6. int main() {
7.     int i = 10, j = 20;
8.     swap(&i, &j);
9.     printf("i=%d j=%d\n", i, j);
10.    return 0;
11.}
```

During the function call at line-9, the compiler creates space for variables (arguments) a and b in swap and copies the value of the parameters &i and &j in variables a and b.

int *a = &i;
int *b = &j;

In this case, because a contains the &i, and b contains &j
tmp = *(a); // loads the value from &i in tmp
*(a) = *(b); // loads the value from &j and store it into &i
*(b) = tmp; // stores the value of tmp in &j

Therefore, when swap returns, the values of i and j have already been swapped.

In this case, when we call the swap routine, the compiler creates two new variables, “a” and “b”, corresponding to the arguments of the function and store the value of “&i” in “a” and the value of “&j” in “b”. The output of this program is i=20 j=10.

Array in function arguments

```
void foo(int arr[10]);

void foo(int arr[]);

void foo(int *arr);
```

All three declarations are the same.

The C compiler ignores the number of elements in the first dimension in the array arguments.

```
1. void foo(int arr[10]) {
2.   …
3. }

4. int main() {
5.   int a[10];
6.   foo(&a[0]);
7.   return 0;
8. }
```

During the function call at line-6, the compiler creates space for the variable arr in foo and copies the value of &a[0] in arr.

int *arr = &a[0];

The compiler converts types of arguments "int arr[10]" or "int arr[]" in the function declaration to "int *arr". It doesn't allocate an array of size 10 when the type of a function argument is declared as "int arr[10]".

Array in function arguments

```
void foo(int arr[10]);

void foo(int arr[]);

void foo(int *arr);
```

All three declarations are the same.

The C compiler ignores the number of elements in the first dimension in the array arguments.

```
1. void foo(int arr[10]) {
2.  arr[-1] = 1;
3.  arr[-2] = 2;
4.  …
5. }

6. int main() {
7.   int a[10];
8.   foo(&a[2]); // interior address
9.   return 0;
10.}
```

We can also pass an interior address of an array. During the function call at line-8, the compiler creates space for the variable arr in foo and copies the value of &a[2] in arr.

int *arr = &a[2];

You can also store the address of an internal element of an array in a pointer variable or pass it to a function.

Structures

Structure

- Structures are used to create new data types using existing data types
  - Structure is a user-defined type
- Structure can be used to manage a set of closely related variables efficiently in large programs
  - Structure helps manage closely related variables as a unit rather than as separate entities

Distance between 2D points

```
// returns the distance between two two-dimensional points
double distance2d(int x1, int y1, int x2, int y2) {
   double xpow2 = pow(x2 - x1, 2.0);
   double ypow2 = pow(y2 – y1, 2.0);
   return pow(xpow2 + ypow2, 0.5);
}

int main() {
   int x1 = 2, y1 = 2;  // point-1 coordinates
   int x2 = 3, y2 = 5;  // point-2 coordinates
   double r = distance2d(x1, y1, x2, y2);
   printf(“%lf\n”, r);
   return 0;
}
```

If we want to write a program that deals with 2D points, we often need to keep track of two variables corresponding to the x and y coordinates whenever we want to operate on points.

Distance between 3D points

```
// returns the distance between two three-dimensional points
double distance3d(int x1, int y1, int z1, int x2, int y2, int z2) {
   double xpow2 = pow(x2 - x1, 2.0);
   double ypow2 = pow(y2 – y1, 2.0);
   double zpow2 = pow(z2 – z1, 2.0);
   return pow(xpow2 + ypow2 + zpow2, 0.5);
}

int main() {
   int x1 = 2, y1 = 2, z1 = 2; // point-1 coordinates
   int x2 = 3, y2 = 5, z2 = 8; // point-2 coordinates
   double r = distance3d(x1, y1, z1, x2, y2, z2);
   printf(“%lf\n”, r);
   return 0;
}
```

If we want to write a program that deals with 3D points, we often need to keep track of three variables whenever we want to operate on points. This makes the program clumsy and hard to understand.

Database example

```
#define MAX_ENTRIES 10000
int main() {
  char *name_database[MAX_ENTRIES] = {NULL};
  int year_database[MAX_ENTRIES] = {0};
  int sem_database[MAX_ENTRIES] = {0};
  char* addr_database[MAX_ENTRIES] = {NULL};
  int age_database[MAX_ENTRIES] = {0};
  int entry_no = 0;
  int year = 0, sem = 0, age = 0;

  while (entry_no < MAX_ENTRIES) {
    char *name = malloc(20);
    assert(name != NULL);
    char *addr = malloc(128);
    assert(addr != NULL);

    int ret = get_record(name, addr, &year, &sem, &age);
    if (ret == 0)
      break;
    add_record(name, entry_no, year, sem, addr, age,
         name_database, year_database, sem_database,
         addr_database, age_database);
    entry_no += 1;
  }
  // some more code
}
```

```
// store a record in a database
void add_record(char *name, int entry_no,
                int year, int sem,
                char *addr, int age,
                char **name_database,
                int *year_database,
                int *sem_database,
                char **addr_database,
                int *age_database)
{
   name_database[entry_no] = name;
   year_database[entry_no] = year;
   sem_database[entry_no] = sem;
   addr_database[entry_no] = addr;
   age_database[entry_no] = age;
}
```

In this example, we want to manage a database of records corresponding to students using an array. Notice that in the absence of the structure, we need to create multiple arrays corresponding to different attributes (e.g., name, year, addr, etc.) and pass around these arrays in different parts of the program. Using structure, we can have a single array of structure elements where the structure type has fields corresponding to name, year, addr, etc.

Structures

- A large program would be challenging to manage if we just use the primitive types

Distance between 3D points

```
// returns the distance between two three-dimensional points
double distance3d(int x1, int y1, int z1, int x2, int y2, int z2) {
   double xpow2 = pow(x2 - x1, 2.0);
   double ypow2 = pow(y2 – y1, 2.0);
   double zpow2 = pow(z2 – z1, 2.0);
   return pow(xpow2 + ypow2 + zpow2, 0.5);
}

int main() {
   int x1 = 2, y1 = 2, z1 = 2; // point-1 coordinates
   int x2 = 3, y2 = 5, z2 = 8; // point-2 coordinates
   double r = distance3d(x1, y1, z1, x2, y2, z2);
   printf(“%lf\n”, r);
   return 0;
}
```

The next slide shows an implementation using structures.

3D points using structure

```
// returns the distance between two 3D points
double distance3d(struct point3d pt1,
                  struct point3d pt2)
{
   double xpow2 = pow(pt2.x – pt1.x, 2.0);
   double ypow2 = pow(pt2.y – pt1.y, 2.0);
   double zpow2 = pow(pt2.z – pt1.z, 2.0);
   return pow(xpow2 + ypow2 + zpow2, 0.5);
}

int main() {
   int x1 = 2, y1 = 2, z1 = 2; // point-1
   int x2 = 3, y2 = 5, z2 = 8; // point-2
   struct point3d pt1 = make_point3d(x1, y1, z1);
   struct point3d pt2 = make_point3d(x2, y2, z2);

   double r = distance3d(pt1, pt2);
   printf("%lf\n", r);
   return 0;
}
```

```
struct point3d {
    int x;
    int y;
    int z;
};

struct point3d make_point3d(int x,
                            int y,
                            int z)
{
   struct point3d t;
   t.x = x;
   t.y = y;
   t.z = z;
   return t;
}
```

Notice that this program is much easier to read and understand.

Database example

```
#define MAX_ENTRIES 10000
int main() {
  char *name_database[MAX_ENTRIES] = {NULL};
  int year_database[MAX_ENTRIES] = {0};
  int sem_database[MAX_ENTRIES] = {0};
  char* addr_database[MAX_ENTRIES] = {NULL};
  int age_database[MAX_ENTRIES] = {0};
  int entry_no = 0;
  int year = 0, sem = 0, age = 0;

  while (entry_no < MAX_ENTRIES) {
    char *name = malloc(20);
    assert(name != NULL);
    char *addr = malloc(128);
    assert(addr != NULL);

    int ret = get_record(name, addr, &year, &sem, &age);
    if (ret == 0)
      break;
    add_record(name, entry_no, year, sem, addr, age,
         name_database, year_database, sem_database,
         addr_database, age_database);
    entry_no += 1;
  }
  // some more code
}
```

```
// store a record in a database
void add_record(char *name, int entry_no,
                int year, int sem,
                char *addr, int age,
                char **name_database,
                int *year_database,
                int *sem_database,
                char **addr_database,
                int *age_database)
{
   name_database[entry_no] = name;
   year_database[entry_no] = year;
   sem_database[entry_no] = sem;
   addr_database[entry_no] = addr;
   age_database[entry_no] = age;
}
```

The next slide shows an implementation using structures.

```
#define MAX_ENTRIES 10000
int main() {
  struct record database[MAX_ENTRIES] = {0};
  int entry_no = 0;
  struct record r;

  while (entry_no < MAX_ENTRIES) {
    char *name = malloc(20);
    assert(name != NULL);
    char *addr = malloc(128);
    assert(addr != NULL);
    r.name = name;
    r.addr = addr;

    int ret = get_record(&r);
    if (ret == 0)
      break;
    add_record(r, database, entry_no);
    entry_no += 1;
  }
  // some more code
}
```

```
struct record {
  char *name;
  int year, sem, age;
  char *addr;
};

// store a record in a database
void add_record(struct record r,
                struct record *database,
                int entry_no)
{
   database[entry_no].name = r.name;
   database[entry_no].year = r.year;
   database[entry_no].sem = r.sem;
   database[entry_no].addr = r.addr;
   database[entry_no].age = r.age;
}
```

Database example using structure

Notice that this program is much easier to read and understand.

Dynamic arrays

https://en.wikipedia.org/wiki/Dynamic_array

Dynamic arrays

- Let's look at the problem of storing the records of items sold in a day
  - a record can be represented using a structure with different fields
    - item-name, price, customer phone no., etc.

- What would be the problem if we store the records in an array?

Dynamic arrays

- Let's look at the problem of storing the records of items sold in a day
  - a record can be represented using a structure with different fields
    - item-name, price, customer phone no., etc.

- What would be the problem if we store the records in an array?
  - We need to over-approximate the total number of records, say N
  - Allocate a big array that can store N records
  - Most of the time, sold items would be less than N
    - resulting in the waste of a lot of space
  - The program may fail if the over-approximation fails

Insert

- Let's say the maximum size of the array is N, and right now array has m elements, where m < N. Inserting a new element x just after the m^{th} element takes one operation

  arr[m] = x

- What if m == N?
  - How can we insert a new element?

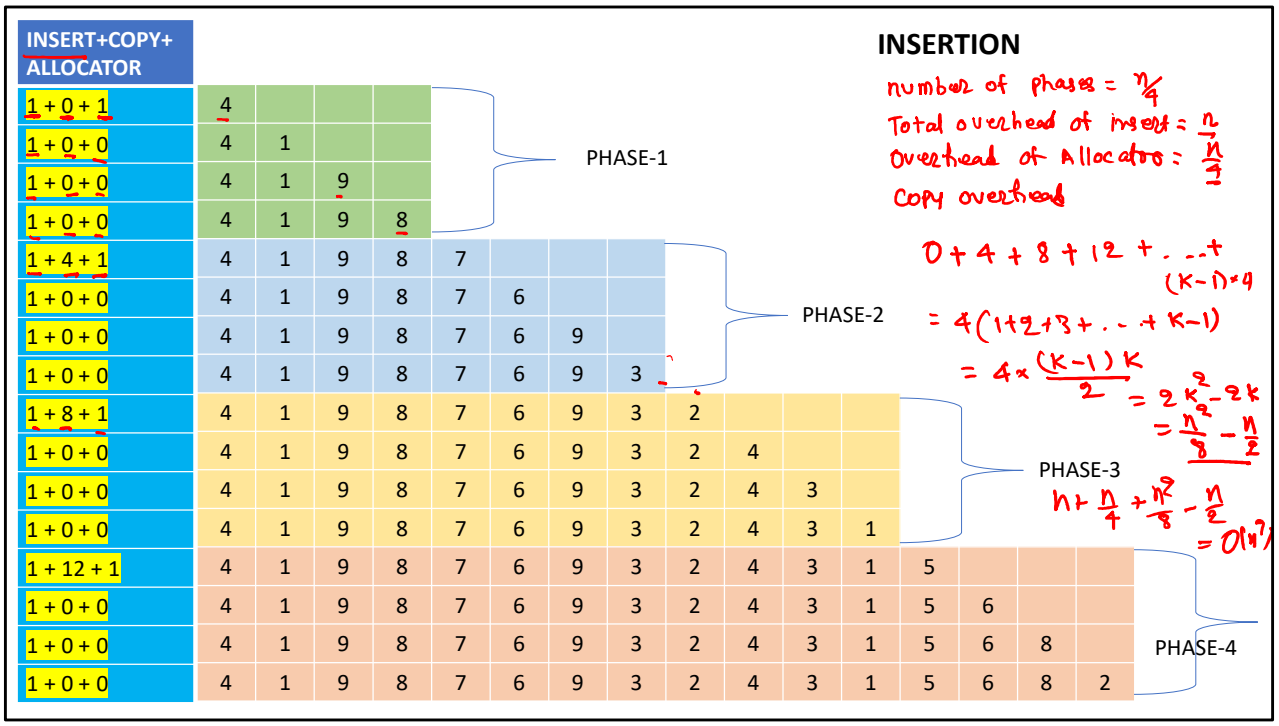
Dynamic array

- Initially, allocate an array of small size
- When the array is full, allocate a new array of bigger size
- Copy the elements from the old array to the new array
- Delete the old array
- Insert the element in the new array

Constant expansion

Constant expansion

- If the array of size "s" is full, create a new array of size "s + c", where c is a constant



In this case, initially, the size of the array is zero. We expand the array when no additional space is available during an insertion. During the expansion, we create a new array of size s + 4, where s is the size of the old array, copy elements from the old array to the new array, free the old array, and insert the element in the new array. A phase consists of a sequence of insertions between two consecutive expansions. In this analysis, we are assuming that the cost of allocating a new array and freeing the old array is one and calling it the allocator overhead. The cost of copying one element from the old array to the new array is also one. The cost of storing an element at a given index is also one; we call it insertion overhead. We need to allocate an array of size four during the first insertion because there is no space. In this step, we don't need to copy anything because the old array size is zero. Therefore, (INSERTION + COPY + ALLOCATOR) overhead is (1 + 0 + 1). There is already some space in the array during the second insertion, so the cost is (1 + 0 + 0). During the fifth insertion, we need to allocate an array of size 8, copy four elements from the old array to the new array, free the old array, and insert the 5th element. So, (INSERT + COPY + ALLOCATOR) overhead is (1+4+1). Similarly, during the 9th insertion, the (INSERT+COPY+ALLOCATOR) overhead is (1+8+1), and so on. If we count the total number of operations after inserting n elements (where n is the number of elements in the last phase), we get $n + (n^2/8) - (n/4)$, i.e., $O(n^2)$ operations. Therefore, the

average cost of a single insert operation is O(n), which is very expensive.

Amortized cost (constant expansion)

Amortized cost (constant expansion)

Total number of phases: $k = n/4$
Total number of operations for insertion: n
Total number of operations by allocator: $k = n/4$
Copy overhead:
$0 + 4 + 8 + 12 + 16 + \ldots + (k - 1) * 4$
$= 4(1 + 2 + 3 + \ldots + k-1)$
$= 4 *(k-1) * k / 2 = 2k^2 - 2k = n^2/8 - n/2$

Total Overhead: $n + n/4 + n^2/8 - n/2 = O(n^2)$

Amortized cost (constant expansion)

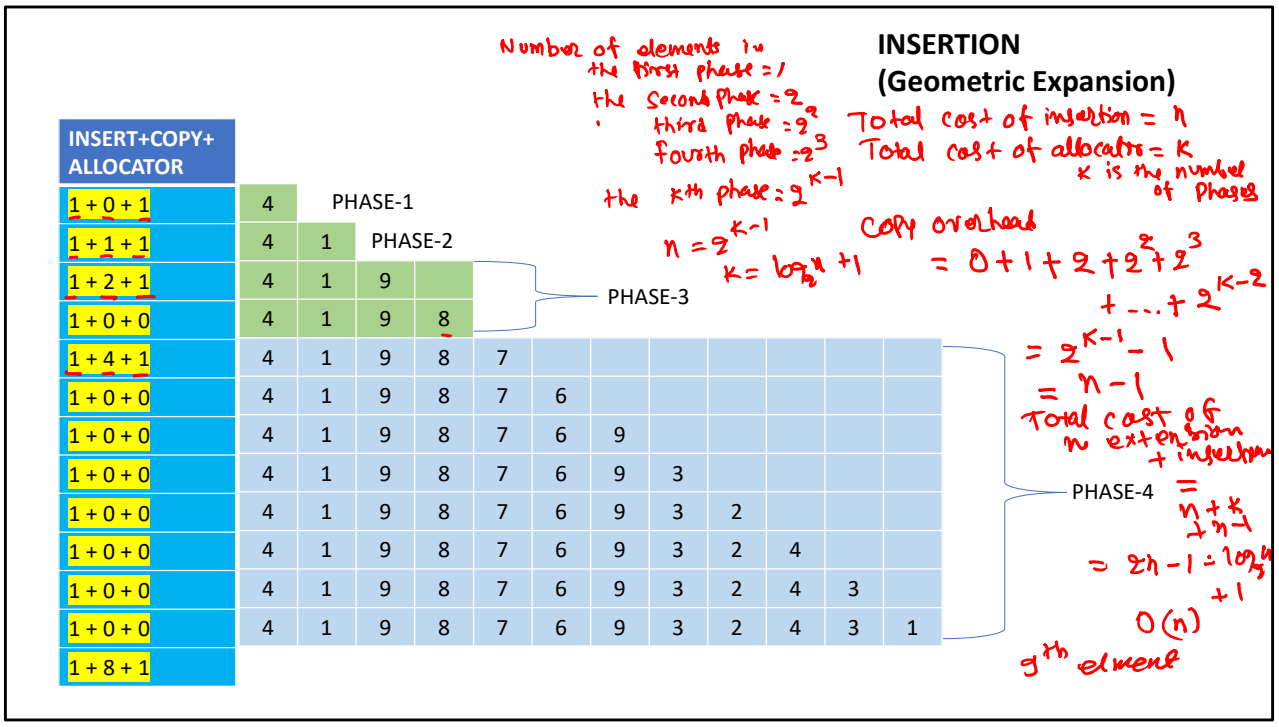
- What would be the amortized cost of n insert operations if we increase the size by a constant c instead of 4?

  $O(n^2)$

Amortized cost (constant expansion)

- What would be the amortized cost of n insert operations if we increase the size by a constant c instead of 4?
  - $O(n^2)$

Geometric expansion



In this case, initially, the size of the array is zero. We expand the array when no additional space is available during an insertion. During the expansion, we create a new array of size one if the old size is zero; otherwise, we allocate an array of size s * 2, where s is the size of the old array. A phase consists of a sequence of insertions between two consecutive expansions. In this analysis, we assume that the cost of allocating a new array and freeing the old one is one and calling it allocator overhead. The cost of copying one element from the old array to the new array is also one. The cost of storing an element at a given index is also one; we call it insertion overhead. We need to allocate an array of size one during the first insertion because the size of the old array is zero. We don't need to copy anything from the old array in this step. Therefore, (INSERTION + COPY + ALLOCATOR) overhead is (1 + 0 + 1). During the second insertion, no space is left, so we allocate an array of double size, i.e., two, copy one element from the old array to the new array, delete the old array, and insert the new element. The (INSERT + COPY + ALLOCATOR) cost is (1 + 1 +1). During the third insertion, no space is left, so we allocate an array of size four, copy two elements from the old array to the new array, delete the old array, and insert the new element. The (INSERT + COPY + ALLOCATOR) cost is (1 + 2 +1). During the fourth insertion, however, we don't need to allocate a new array because there is already space left to store one additional element, so the overhead is (1 + 0 + 0). During the

fifth insertion, the overhead is (1+4+1); 9th insertion, the overhead is (1+8+1), and so on. If we compute the total number of operations for inserting n elements where n is the number of elements in the last phase, we get (2 * n) - $\log_2(n)$, which is O(n). Therefore, the average cost of a single insertion is O(1).

Amortized cost (geometric expansion)

Amortized cost (geometric expansion)

Total number of phases: k

Number of elements in the kth phase =>$2^{k-1} = n$

$k = (\log_2 n) + 1$

Total number of operations for insertion: n

Total number of operations by the allocator: k

Copy overhead:

$0 + 1 + 2 + 2^2 + \cdots + 2^{k-2} = 2^{k-1} - 1 = n - 1$

Total overhead:

n + k + n – 1 = 2n -1 + ($\log_2$n) + 1 = O(n)

Amortized cost

- The geometric expansion of array takes around O(n) operations for n insertions
  - Therefore, the amortized cost of an insert operation is O(1)

- What is the maximum amount of memory wastage in this scheme?

Amortized cost

- The geometric expansion of array takes around O(n) operations for n insertions
  - Therefore, the amortized cost of an insert operation is O(1)

- What is the maximum amount of memory wastage in this scheme?
  - N/2

The maximum memory wastage is N/2, where N is the size of the array. This is because when the array is full, we are doubling the size of the array, and at that point, nearly half of the array is empty.

Delete

- The memory wastage can be large if we delete a lot of elements from an array after a large number of insertions
- With dynamic arrays, we also need to handle the case when most of the array is unused due to deletes after expansion

Delete (geometric expansion)

- After a deletion, only N/2 elements are present in the array, whereas the array has space to store N elements
  - What can we do in this case?

Delete (geometric expansion)

- After a deletion, only N/2 elements are present in the array, whereas the array has space to store N elements
  - What can we do in this case?
    - Deleting half of the array doesn't make sense because if the next operation is an insertion, then we need to double the size of the array
- Possible strategy:
  - If after a deletion, only N/4 elements are present in the array
    - create a new array of size N/2
    - copy N/4 elements to the new array
    - delete the old array

Delete (geometric expansion)

Initially, the size of the array is 64, and the number of elements is also 64

Number of operations for first 47 deletes = 47 (one operation each)
Deleting 48th element = 1 operations
number of elements after delete = 16 = 64/4
Allocate an array of 32 elements
Copy 16 elements to the new array (16 operations)
Delete the old array of 64 elements
Allocator cost = 1 operations
Total number of operations during the deletion of the 48th element: 1 + 16 + 1
Deleting the next 7 elements take 7 operations (one operation for each delete)
During the 8th delete number of operations = 1 (delete) + 8 copy + 1 allocator

Delete

- What is the maximum memory wastage during the geometric expansion solution?

$\frac{3n}{4}$

If we do shrinking during deletion, the maximum memory wastage is 3n/4, where n is the total size of the array.

Delete

- What is the maximum memory wastage during the geometric expansion solution?
  - 3n/4

Homework

- Amortized cost of a delete operation after n consecutive deletes in the geometric expansion solution

Amortized analysis

- Amortize analysis gives the average performance of each operation in the worst case
  - It looks at a sequence of operations rather than a single operation
- Amortized analysis is different from average case analysis because it doesn't consider the probability of a particular distribution of the input data
- Amortized analysis is generally used when most operations are cheaper, except for a few expensive operations
- Amortized complexity may not be applicable when the cost of individual operations also matters
  - e.g., video streaming platforms

The amortized complexity is not suitable for real-time applications. For example, if the amortized streaming speed for the whole duration of a cricket match is good, it doesn't capture the fact that there could be glitches due to low streaming speed during small time frames, resulting in a bad user experience.