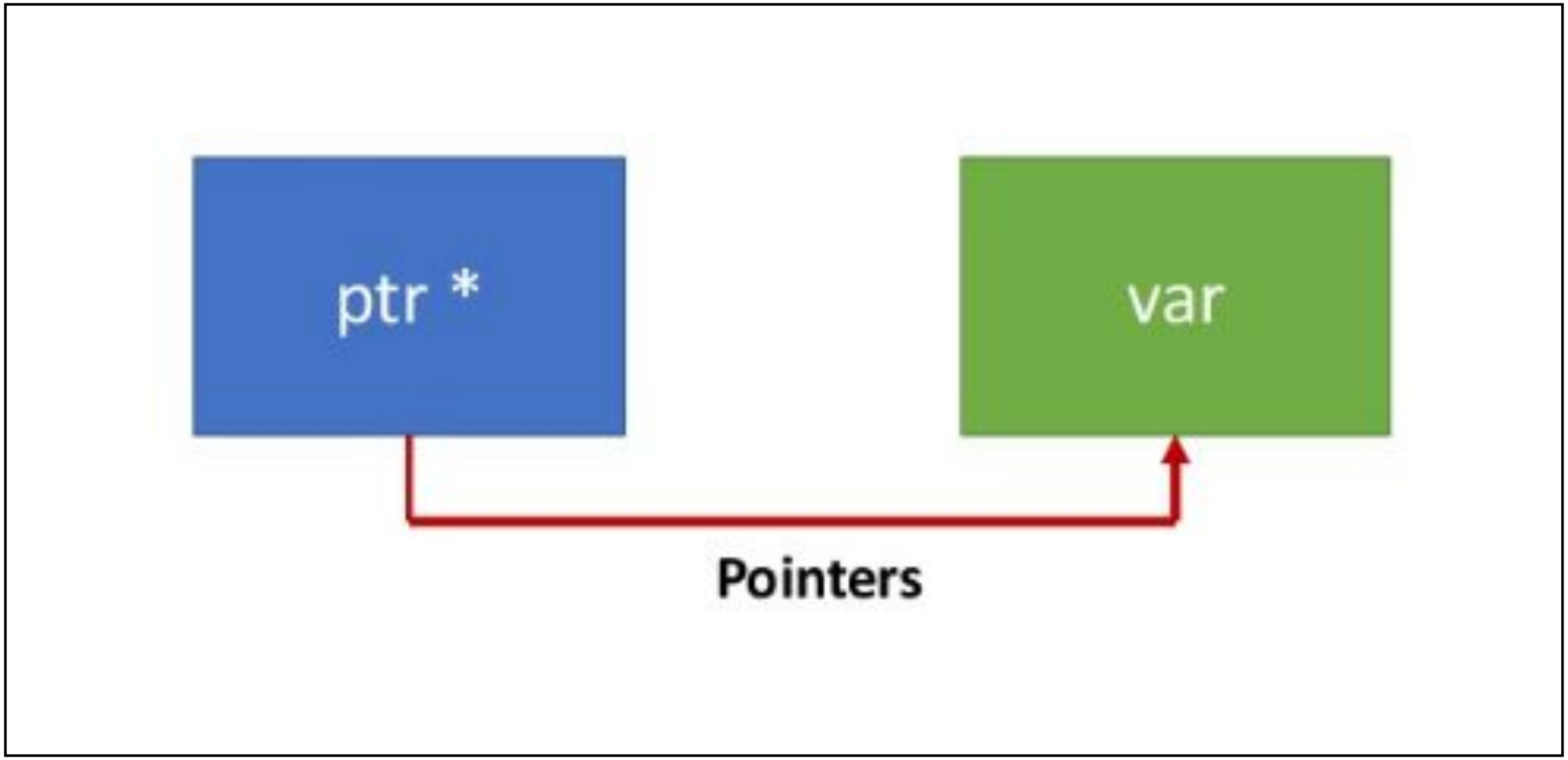
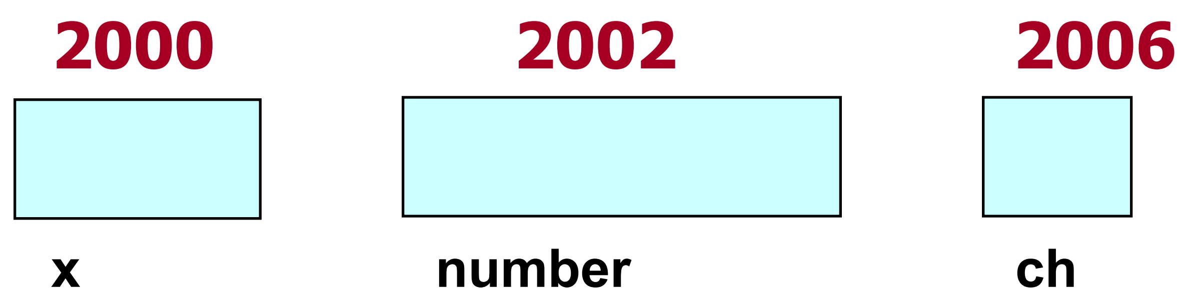
Pointers



Addresses in Memory

- When a variable is declared, enough memory to hold a value of that type is allocated for it at an unused memory location. This is the address of the variable

```
int     x;
float   number;
char    ch;
```



What is a pointer variable?

- A pointer variable is a **variable whose value is the address of a location in memory.**
- To declare a pointer variable, you must specify the type of value that the pointer will point to, for example,

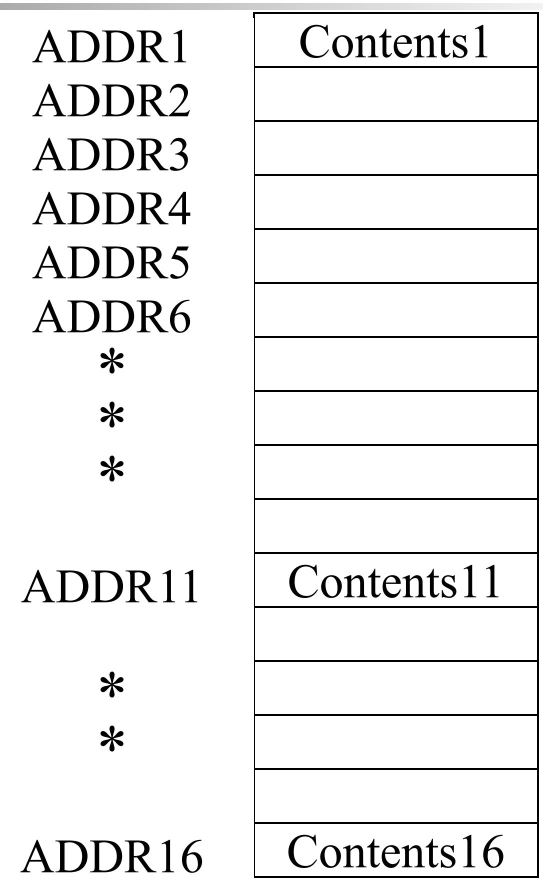
```
int*   ptr; // ptr will hold the address of an int
char*  q;   // q will hold the address of a char
```

POINTERS

- Pointers are variables that contain *memory addresses* as their values.
- A variable name *directly* references a value.
- A pointer *indirectly* references a value. Referencing a value through a pointer is called *indirection*.
- A pointer variable must be declared before it can be used.

Concept of Address and Pointers

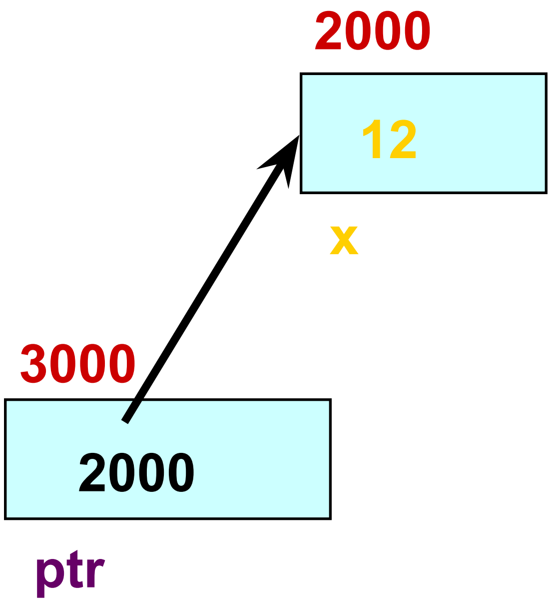
- Memory can be conceptualized as a linear set of data locations.
- Variables reference the contents of a locations
- Pointers have a value of the address of a given location



Using a Pointer Variable

```
  int  x;
 x = 12;


 int*  ptr;
 ptr = &x;
```



NOTE: **Because ptr holds the address of x, we say that ptr "points to" x**

POINTERS

- Examples of pointer declarations:

  FILE ***fptr**;

  int ***a**;

  float ***b**;

  char ***c**;

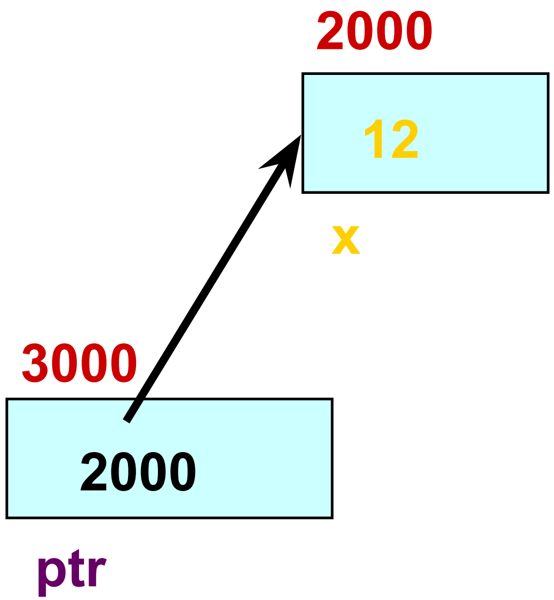
- The asterisk, when used as above in the declaration, tells the compiler that the variable is to be a pointer, and the type of data that the pointer points to, but NOT the name of the variable pointed to.

*: dereference operator

```
int  x;
x = 12;


int*  ptr;
ptr = &x;

cout  <<  *ptr;
```



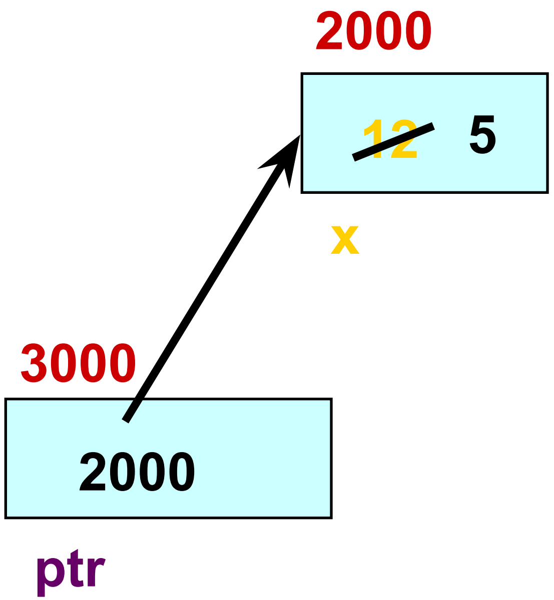
NOTE: **The value pointed to by ptr is denoted by *ptr**

Using the Dereference Operator

```
int  x;
x = 12;


int*  ptr;
ptr = &x;

*ptr = 5;
```



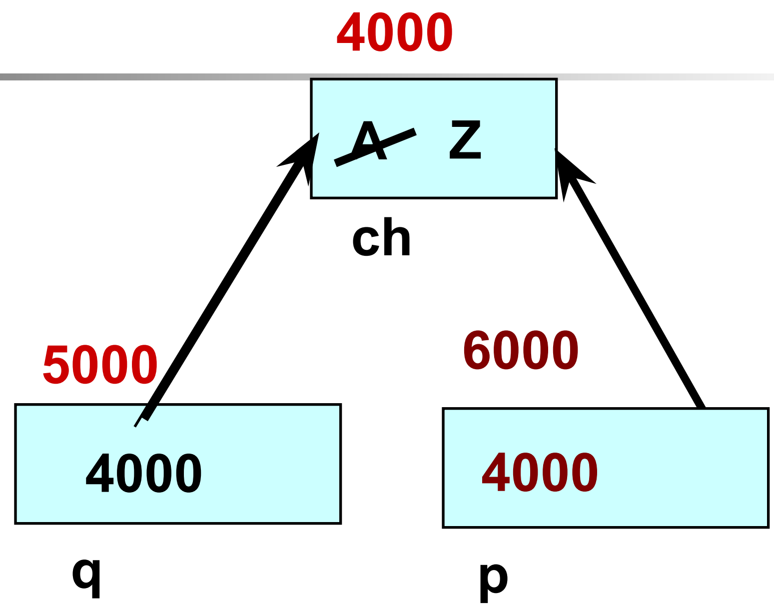

```
// changes the value at the
   address ptr points to 5
```

Self –Test on Pointers

```
char  ch;
ch =  'A';

char*  q;
q  = &ch;

*q = 'Z';
char*  p;
p = q;
```

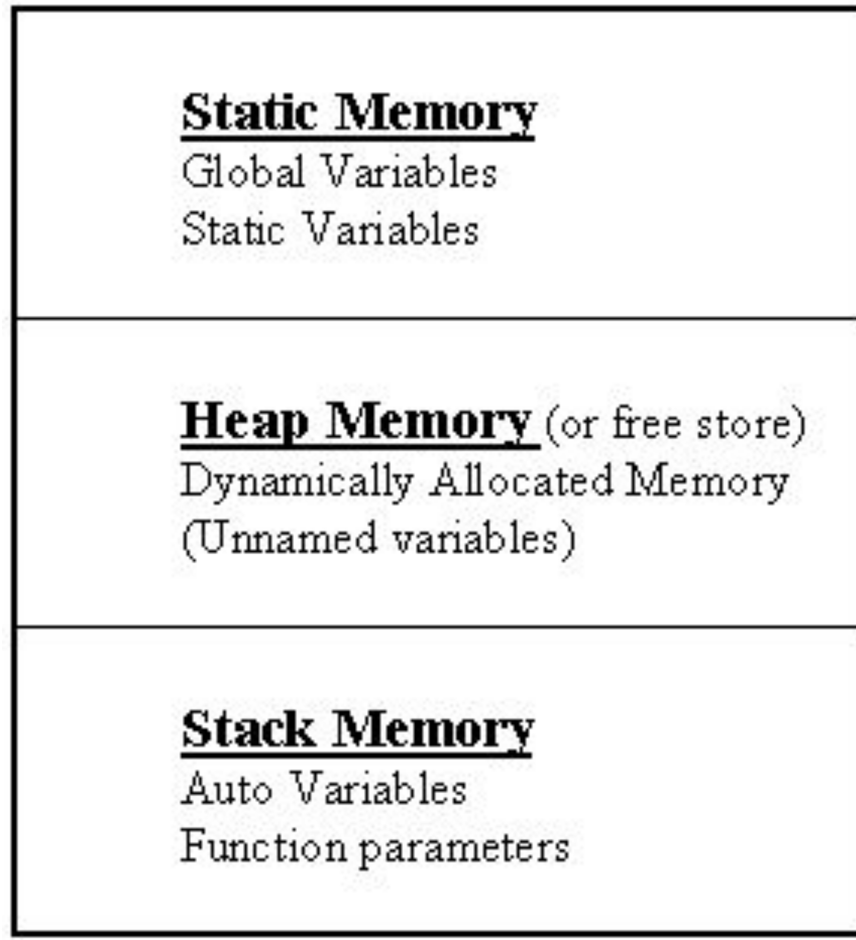



```
// the rhs has value 4000
// now p and q both point to ch
```

Dynamic Memory Allocation

In C and C++, three types of memory are used by programs:

- **Static memory** - where global and static variables live
- **Heap memory** - dynamically allocated at execution time
  - "managed" memory accessed using pointers
- **Stack memory** - used by automatic variables



The `NULL` Pointer

- There is a pointer constant called the "null pointer" denoted by NULL
- But NULL is not memory address 0.
- NOTE: It is an error to dereference a pointer whose value is NULL. Such an error may cause your program to crash, or behave erratically. It is the programmer's job to check for this.

```
while (ptr != NULL) {
   . . .    // ok to use *ptr here
}
```

Example

```
char  *ptr ;

ptr  =  new  char[ 5 ];

strcpy( ptr, “Bye” );

ptr[ 0 ] = ‘u’;



ptr = NULL;
```

ptr

3000

NULL

// deallocates the array pointed to by ptr
// ptr itself is not deallocated
// the value of ptr becomes undefined

Array Basics

char str [8];

- **str** is the base address of the array.
- We say **str** is a pointer because its value is an address.
- It is a <u>pointer constant</u> because the value of **str** itself cannot be changed by assignment. It "points" to the memory location of a `char`.

6000

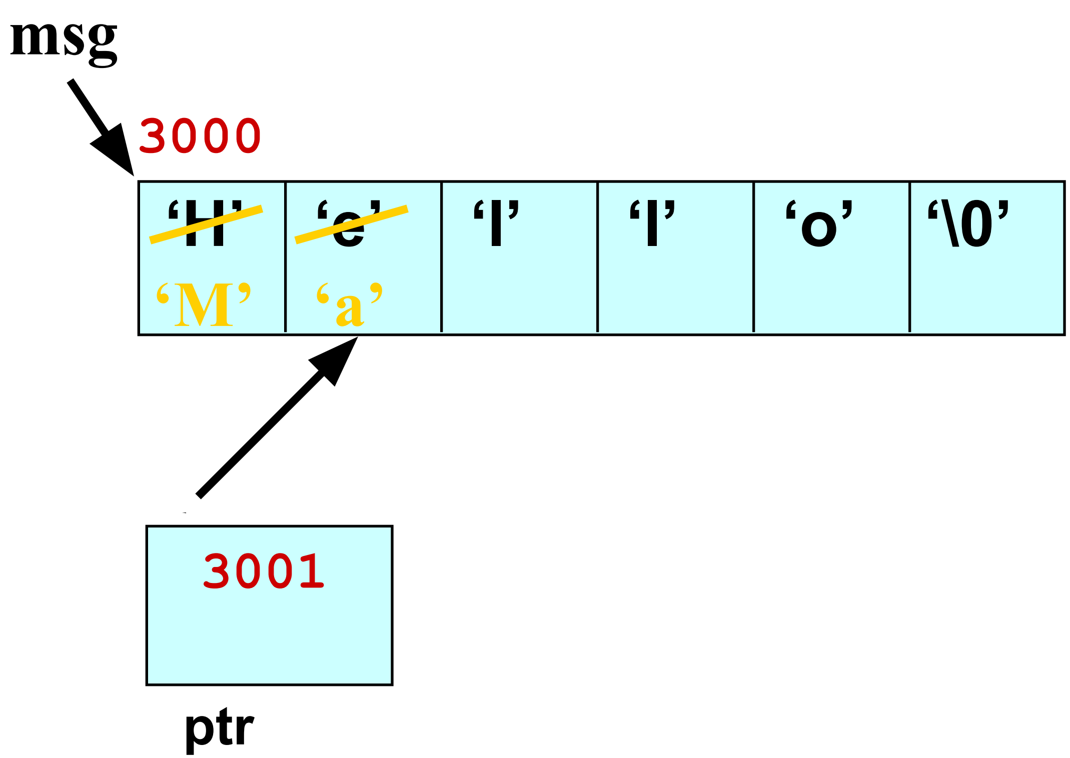
'H'	**'e'**	**'l'**	**'l'**	**'o'**	**'\0'**		
`str` **[0]**	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**

Pointers into Arrays

```
char   msg[ ] ="Hello";
char*  ptr;
ptr  =  msg;
*ptr  = 'M' ;
ptr++;

*ptr = 'a';


ptr = &msg[4];
// *ptr = o
```



An Array Variable Is a Pointer

In C, when we declare an array statically

```
float static_element[100];
```

we are setting up a block in memory, but we're doing it at compile time instead of at runtime.

Otherwise, an array is identical to a pointer. Specifically, it's a **pointer** to the block of memory that holds the array.

String Literals

```
char* p = “Hello”;
char[] q = “Hello”;
char* r = “Hello”;
```

- p[4] = ‘O’; // error: assignment to constant
- q[4] = ‘O’; // ok, q is an array of 5 characters
- p == r; // false; implementation dependent

Pointers and Constants

- char s[] = “Hello”;
- const char* pc = s;   // pointers to constant
- pc[3] = ‘g’;   // error
- pc = p;   // ok

- char *const cpc = s; // constant pointer
- cpc[3] = ‘a’;   // ok
- cp = p;   // error

Pointers and Functions

- Pointers can be used to pass addresses of variables to called functions, thus allowing the called function to alter the values stored there.

- We looked earlier at a swap function that did not change the values stored in the main program because only the values were passed to the function swap.

- This is known as "call by value".

Pointers and Functions

- If instead of passing the values of the variables to the called function, we pass their addresses, so that the called function can change the values stored in the calling routine. This is known as "call by reference" since we are *referencing* the variables.

- The following shows the swap function modified from a "call by value" to a "call by reference". Note that the values are now actually swapped when the control is returned to main function.

Pointers with Functions (example)

```
#include <stdio.h>
void swap ( int *a, int *b ) ;
int main ( )
{
   int a = 5, b = 6;
   printf("a=%d b=%d\n",a,b)
    ;
   swap (&a, &b) ;
   printf("a=%d b=%d\n",a,b)
    ;
   return 0 ;
}
```

```
void swap( int *a, int *b )
{
   int temp;
   temp= *a;  *a= *b;  *b =
    temp ;
   printf ("a=%d  b=%d\n", *a,
    *b);
}
```

Results:

a=5 b=6

a=6 b=5

a=6 b=5

Arithmetic and Logical Operations on Pointers

- A pointer may be incremented or decremented
- An integer may be added to or subtracted from a pointer.
- Pointer variables may be subtracted from one another.
- Pointer variables can be used in comparisons, but usually only in a comparison to NULL.

Using the C Language Special Keyword

sizeof

- This keyword can be used to determine the number of bytes in a data type, a variable, or an array
- Example:

  double **array [10]**;

  sizeof (double);   **/* Returns the value 8 */**

  sizeof (**array**);   **/* Returns the value 80 */**

  sizeof(**array**)**/**sizeof(double);   **/* Returns 10 */**

Using the C Language Special Keyword

sizeof

- This keyword can be used to determine the number of bytes in a data type, a variable, or an array
- Example:

  double **array [10]**;

  sizeof (double);   **/* Returns the value 8 */**

  sizeof (**array**);   **/* Returns the value 80 */**

  sizeof(**array**)**/**sizeof(double);   **/* Returns 10 */**

C Primitive data types

Implicit specifier(s)	Explicit specifier	Number of bits
`signed char`	*same*	8
`unsigned char`	*same*	8
`char`	*one of the above*	8
`short`	`signed short int`	16
`unsigned short`	`unsigned short int`	16
`int`	`signed int`	16 or 32
`unsigned`	`unsigned int`	16 or 32
`long`	`signed long int`	32 or 64
`unsigned long`	`unsigned long int`	32 or 64
`long long`	`signed long long int`	64
`unsigned long long`	`unsigned long long int`	64

Data Structures

The Need for Data Structures

Data structures organize data
$\Rightarrow$ more efficient programs.

More powerful computers $\Rightarrow$ more complex applications.

More complex applications demand more calculations.

Complex computing tasks are unlike our everyday experience.

What is a data structure?

- In a general sense, any data representation is a data structure. Example: An integer
- More typically, a *data structure* is meant to be an <u>*organization for a collection of data items*</u>.

Organizing Data

Any organization for a collection of records can be searched, processed in any order, or modified.

The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.

Efficiency

A solution is said to be efficient if it solves the problem within its resource constraints.

- Space
- Time

- The cost of a solution is the amount of resources that the solution consumes.

Costs and Benefits

- A data structure requires a certain amount of:
- space for each data item it stores
- time to perform a single basic operation
- programming effort.

Example: Banking Application

- Operations are (typically):
  - Open accounts (far less often than access)
  - Close accounts (far less often than access)
  - Access account to Add money
  - Access account to Withdraw money

Example: Banking Application

- Teller and ATM transactions are expected to take little time.
- Opening or closing an account can take much longer (perhaps up to an hour).

Example: Banking Application

- When considering the choice of data structure to use in the database system that manages the accounts, we are looking for a data structure that:
  - Is inefficient for deletion
  - Highly efficient for search
  - Moderately efficient for insertion

Example: City Database

- The database must answer queries quickly enough to satisfy the patience of a typical user.
- For an exact-match query, a few seconds is satisfactory
- For a range queries, the entire operation may be allowed to take longer, perhaps on the order of a minute.

Some Questions to Ask

- Are all data inserted into the data structure at the beginning, or are insertions interspersed with other operations?
- Can data be deleted?
- Are all data processed in some well-defined order, or is random access allowed?

Data Structure Philosophy

Each data structure has costs and benefits.

Rarely is one data structure better than another in all situations.

A data structure requires:

- space for each data item it stores,
- time to perform each basic operation,
- programming effort.

Data Structure Philosophy

Each problem has constraints on available space and time.

Only after a careful analysis of problem characteristics can we know the best data structure for the task.

Bank example:

- Start account: a few minutes
- Transactions: a few seconds
- Close account: overnight