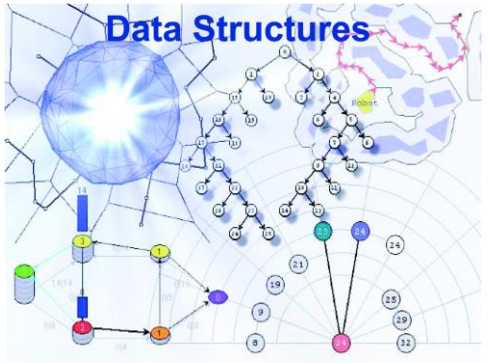
BBM 201
DATA STRUCTURES

Lecture 4:
Records/Structs and Lists

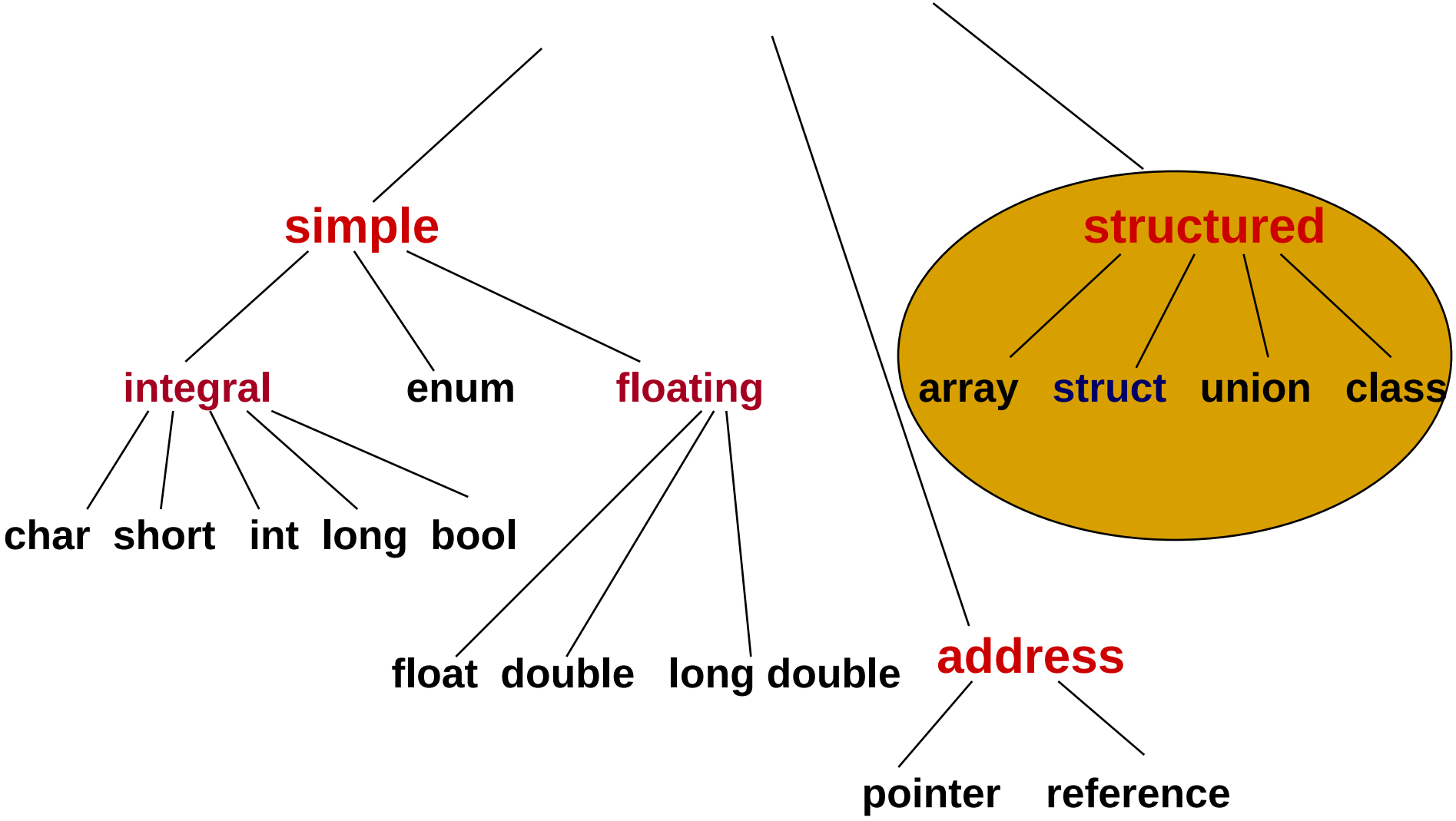


Objectives

- Learn about records (structs)
- Examine various operations on a struct
- Explore ways to manipulate data using a struct
- Learn about the relationship between a struct and functions
- Discover how arrays are used in a struct
- Learn how to create an array of struct items

- Learn about Lists ADT
- A simple array implementation

C++ Data Types


simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
float
double
long double
address
pointer
reference

C++ Data Types

- There are simple data types that hold only one value
- There are structured data types that hold multiple values
- The array was the first example of a structured data type that can hold multiple values
- The structure is the second example

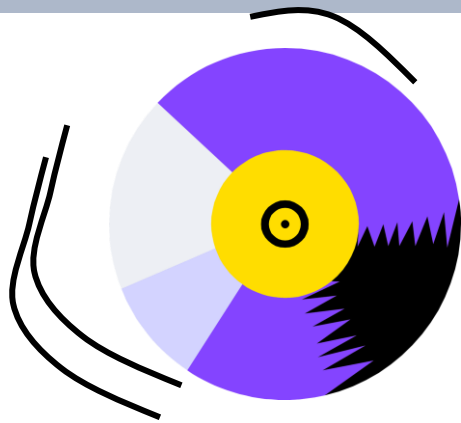
Structured Data Type

A structured data type is a type in which each value is a collection of component items.

- the entire collection has a single name
- each component can be accessed individually

Records
(C++ Structs)

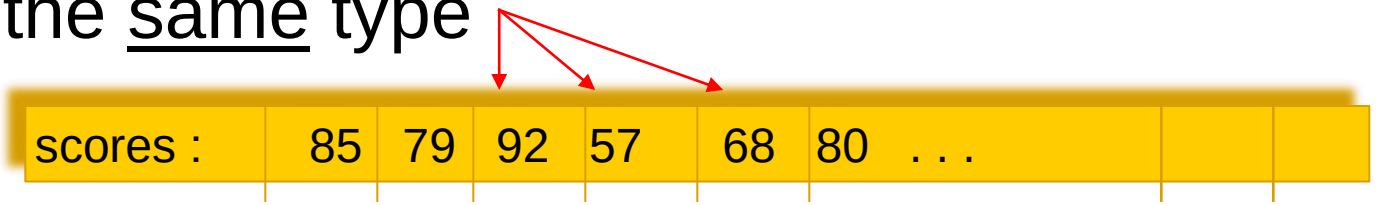
What to do with records?



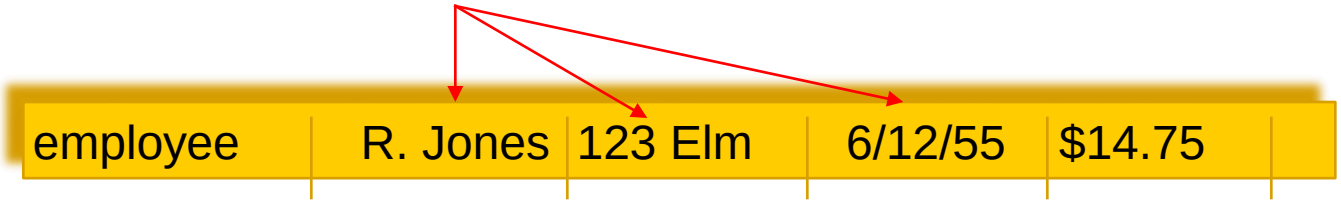
- Declaring records
- Accessing records
- Accessing the field of a record
- Can records be in arrays?

Records

- Recall that elements of arrays must all be of the same type

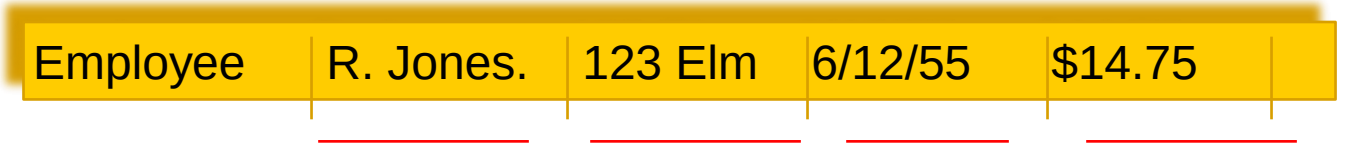


- In some situations, we wish to group elements of different types



Records

- RECORDS are used to group related components of different types
- Components of the record are called fields



- In C++
  - record -> struct (structure)
  - fields -> members

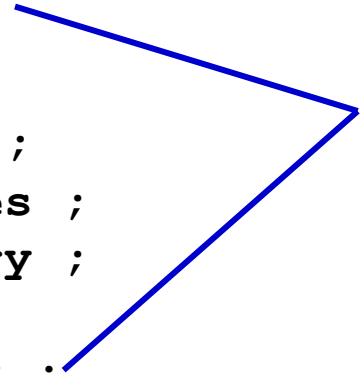
Records

- C++ **struct**
  - structured data type
  - fixed number of components
  - elements accessed by name, not by index
  - components may be of different types

```
struct part_struct {
  char descrip [31], part_num [11];
  float unit_price;
  int qty; };
```

struct AnimalType

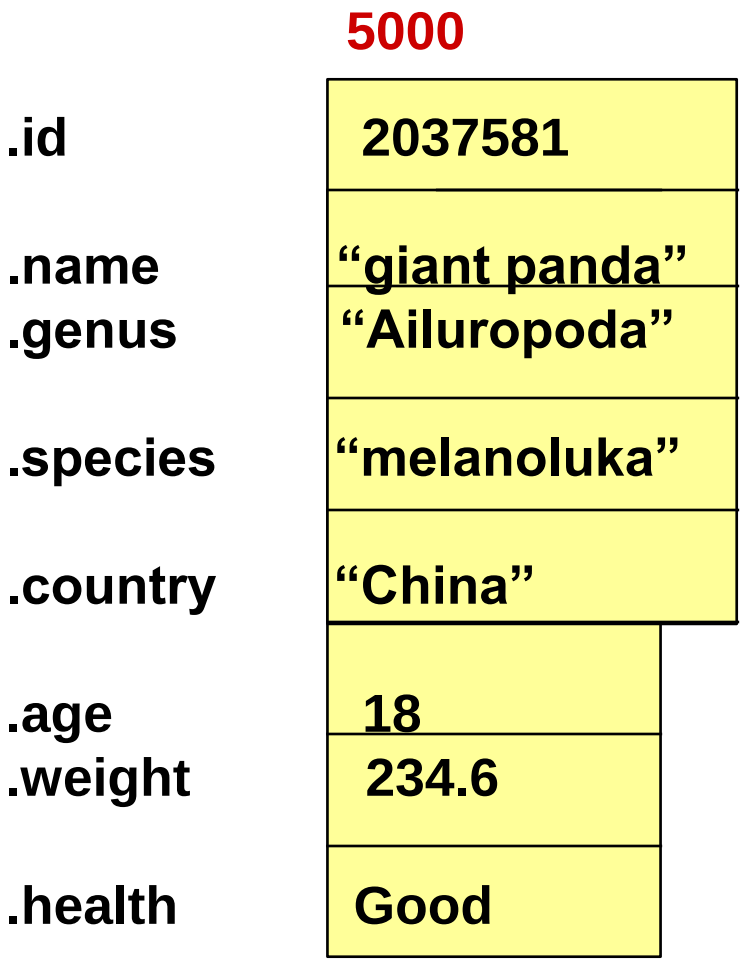
```
struct  AnimalType              // declares a  struct data type
{                               //  does not allocate memory
   long                 id ;
   string               name ;
   string               genus ;
   string               species ;
   string               country ;
   int                  age ;
   float                weight ;
   string               health ;
} ;   // NOTE THE SEMICOLON

AnimalType     thisAnimal ;   // declare  variables of AnimalType
AnimalType     anotherAnimal ;
```



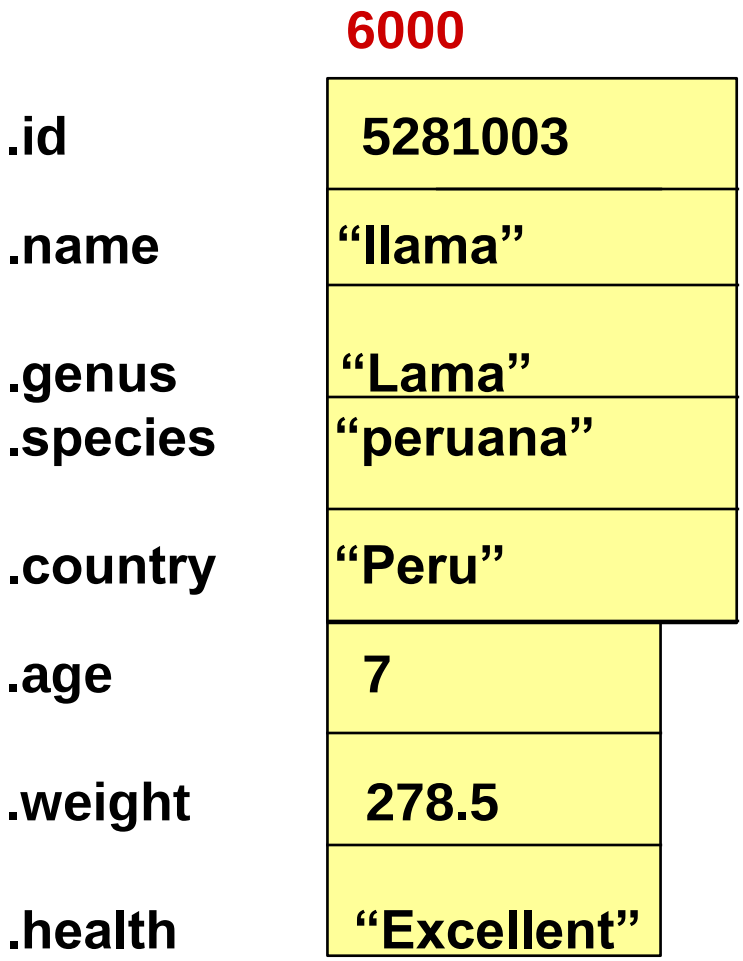
struct members

11

thisAnimal


5000
.id
2037581
.name
“giant panda”
.genus
“Ailuropoda”
.species
“melanoluka”
.country
“China”
.age
18
.weight
234.6
.health
Good

anotherAnimal



`struct` type Declaration

The struct declaration names a type and names the members of the struct.

It does not allocate memory for any variables of that type!

You still need to declare your struct variables.

`struct` type declarations

If the struct type declaration precedes all functions it will be visible throughout the rest of the file. If it is placed within a function, only that function can use it.

It is common to place struct type declarations with TypeNames in a (.h) header file and #include that file (more on this later).

It is possible for members of different struct types to have the same identifiers. Also a non-struct variable may have the same identifier as a structure member.

Accessing `struct` Members

Dot (period) is the member selection operator.

After the struct type declaration, the various members can be used in your program only when they are preceded by a struct variable name and a dot.

EXAMPLES

```
thisAnimal.weight
anotherAnimal.country
```

Valid operations on a struct member depend only on its type

```
thisAnimal.age  =  18;

thisAnimal.id     =  2037581;

cin  >>  thisAnimal.weight;

getline ( cin, thisAnimal.species );

thisAnimal.name = "giant panda";

thisAnimal.genus[ 0 ] = toupper (thisAnimal.genus[ 0 ] ) ;

thisAnimal.age++;
```

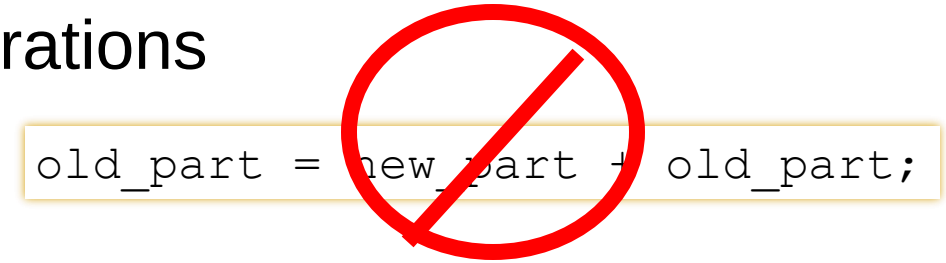
Aggregate Operation

- is an operation on a data structure as a whole, as opposed to an operation on an individual component of the data structure

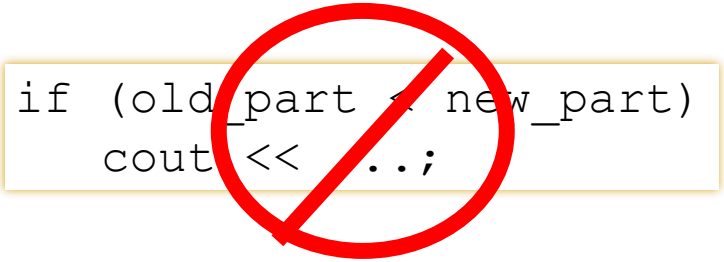
Aggregate Operations with Structures

- Limitations on aggregate operations
  - no I/O

  - no arithmetic operations



  - no comparisons



Aggregate `struct` Operations

I/O, arithmetic, and comparisons of entire struct variables are NOT ALLOWED!

Operations valid on an entire struct type variable:

- assignment to another struct variable of same type,
- pass to a function as argument (by value or by reference),
- return as value of a function

Examples of aggregate `struct` operations

```
anotherAnimal   =   thisAnimal ;      // assignment

WriteOut(thisAnimal);                 // value parameter

ChangeWeightAndAge(thisAnimal); // reference parameter

thisAnimal = GetAnimalData( );    // return value of function


NOW  WE'LL WRITE   FUNCTIONS  USED  HERE . . .
```

```
void  WriteOut(  /*  in  */  AnimalType   thisAnimal)

// Prints out values of all members of thisAnimal
// Precondition:      all members of thisAnimal are assigned
// Postcondition:      all members have been written out
{
   cout   <<"ID # "<<thisAnimal.id<<thisAnimal.name<< endl ;
   cout   <<  thisAnimal.genus   <<  thisAnimal.species << endl ;
   cout   <<  thisAnimal.country << endl ;
   cout   <<  thisAnimal.age   << " years "  << endl ;
   cout   <<  thisAnimal.weight << " lbs. "  << endl ;
   cout   << "General health : " ;

   WriteWord ( thisAnimal.health ) ;
}
```

Passing a `struct` Type by Reference

```
void   ChangeAge ( /* inout */   AnimalType&   thisAnimal)
// Adds 1 to age
// Precondition:       thisAnimal.age is assigned
// Postcondition:    thisAnimal.age ==
   thisAnimal.age@entry + 1
{

   thisAnimal.age++ ;

}
```

```
AnimalType   GetAnimalData ( void )
// Obtains all information about an animal from keyboard
// Postcondition:
//   Function value == AnimalType members entered at  kbd
{
   AnimalType  thisAnimal ;
   char                response ;
   do  {//  have user enter all members until they are correct
                .


                .


                .
   }  while (response != 'Y' ) ;
   return  thisAnimal ;
}
```

Hierarchical Structures

The type of a struct member can be another struct type. This is called nested or hierarchical structures.

Hierarchical structures are very useful when there is much detailed information in each record.

FOR EXAMPLE . . .

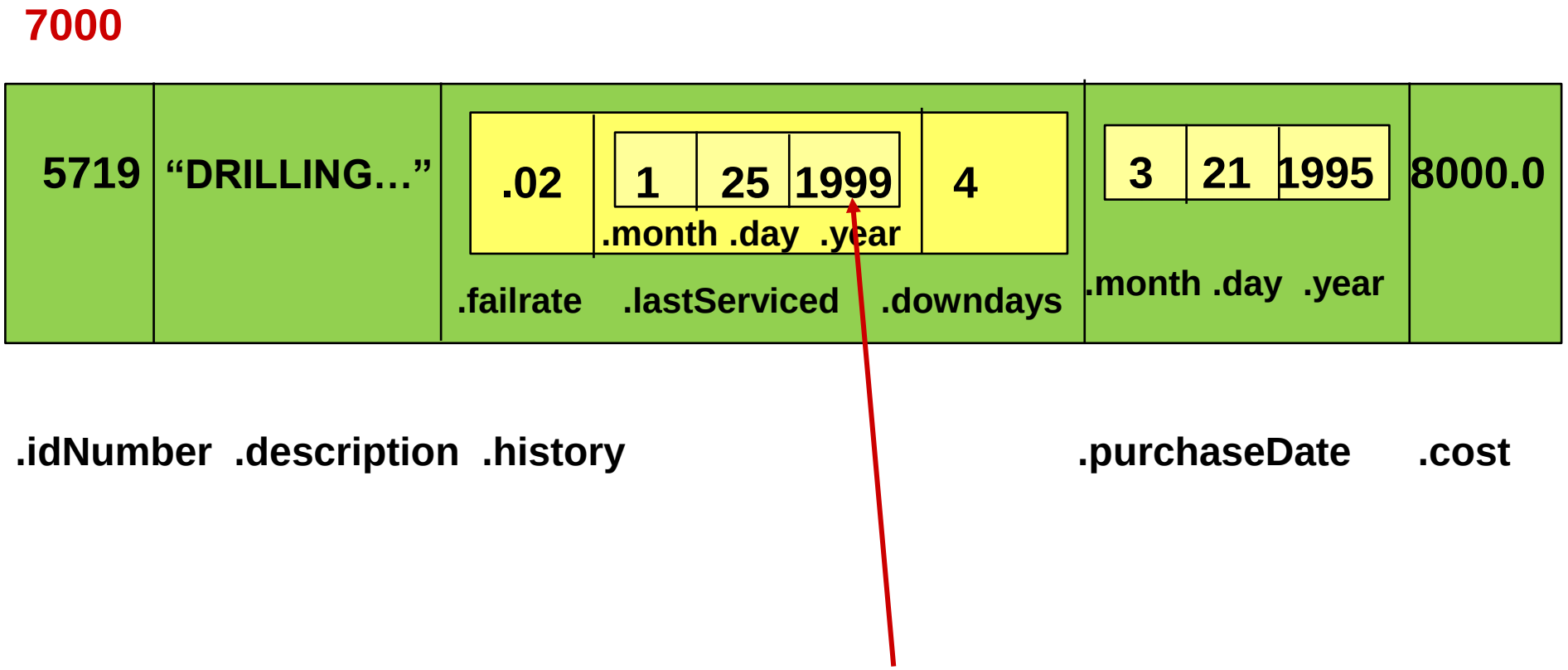
`struct MachineRec`

Information about each machine in a shop contains:

an idNumber,
a written description,
the purchase date,
the cost,
and a history (including failure rate, number of days down, and date of last service).

```
struct  DateType
{       int     month ;          // Assume 1 . . 12
    int     day ;                          // Assume  1 . . 31
                int     year ;             // Assume 1900 . . 2050
};

struct  StatisticsType
{  float                failRate ;
   DateType     lastServiced ;          // DateType is a struct type
   int          downDays ;
} ;

struct MachineRec
{  int              idNumber ;
   string           description ;
   StatisticsType    history ;          // StatisticsType is a struct type
   DateType         purchaseDate ;
   float                    cost ;
} ;

MachineRec      machine ;
```

struct type variable `machine`



machine.history.lastServiced.year has value 1999

Another Struct Example

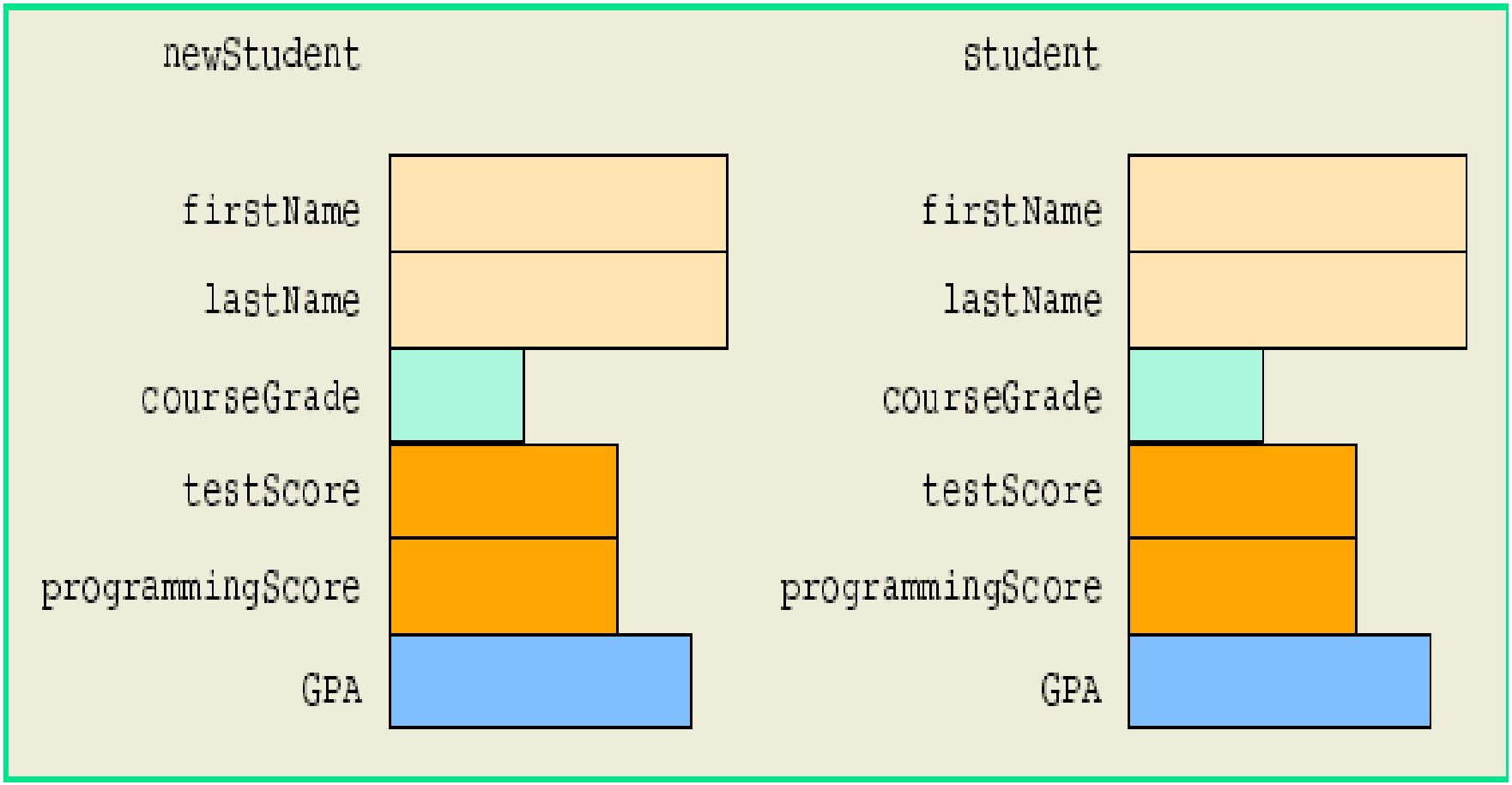
- An example of a studentData struct:

```
struct studentData
{
   string     firstName;
   string     lastName;
   char       courseGrade;
   float      testScore;
   float      programmingScore;
   float      GPA;
};            // NOTE THE SEMICOLON
```

Declaring a struct

- After you have defined a struct, you can declare variables in your program to be of that struct type:

```
studentData   student;
studentData   newStudent;
```



struct newStudent and student

Assignment

- You can copy one structure to another if they have the same type

```
student = newStudent;
```

- You can copy individual members:

```
newStudent.lastName = student.lastName;
```

- Or into a variable of the correct type:

```
thisStudentName = student.lastName;
```

Comparison (Relational Operators)

- Compare struct variables member-wise (NOT THE WHOLE STRUCTURE)
- To compare the values of student and newStudent:

```
if(student.firstName == newStudent.firstName &&
  student.lastName == newStudent.lastName)
    .
    .
    .
```

Input/Output

- No aggregate input/output operations on a struct variable
- Data in a struct variable must be read one member at a time
- The contents of a struct variable must be written one member at a time

struct Variables and Functions

- A struct variable can be passed as a parameter by value or by reference
- A function can return a value of type struct

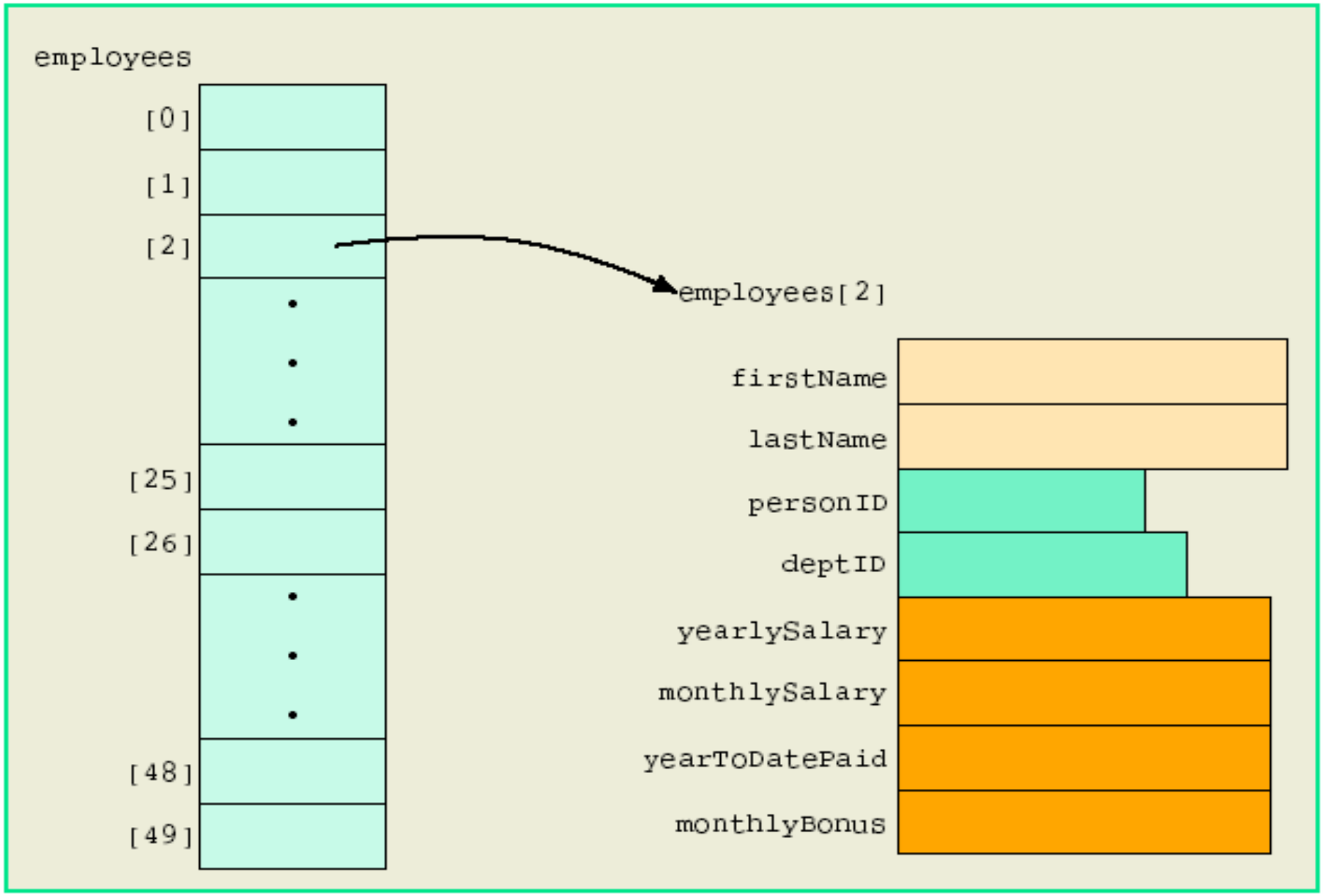
Arrays vs. Structs

	Aggregate Operation	Array	`Struct`
1	Arithmetic	No	No
2	Assignment	No	Yes
3	Input/output	No (except strings)	No
4	Comparison	No	No
5	Parameter passing	By reference only	By value or by reference
6	Function returning a value	No	Yes

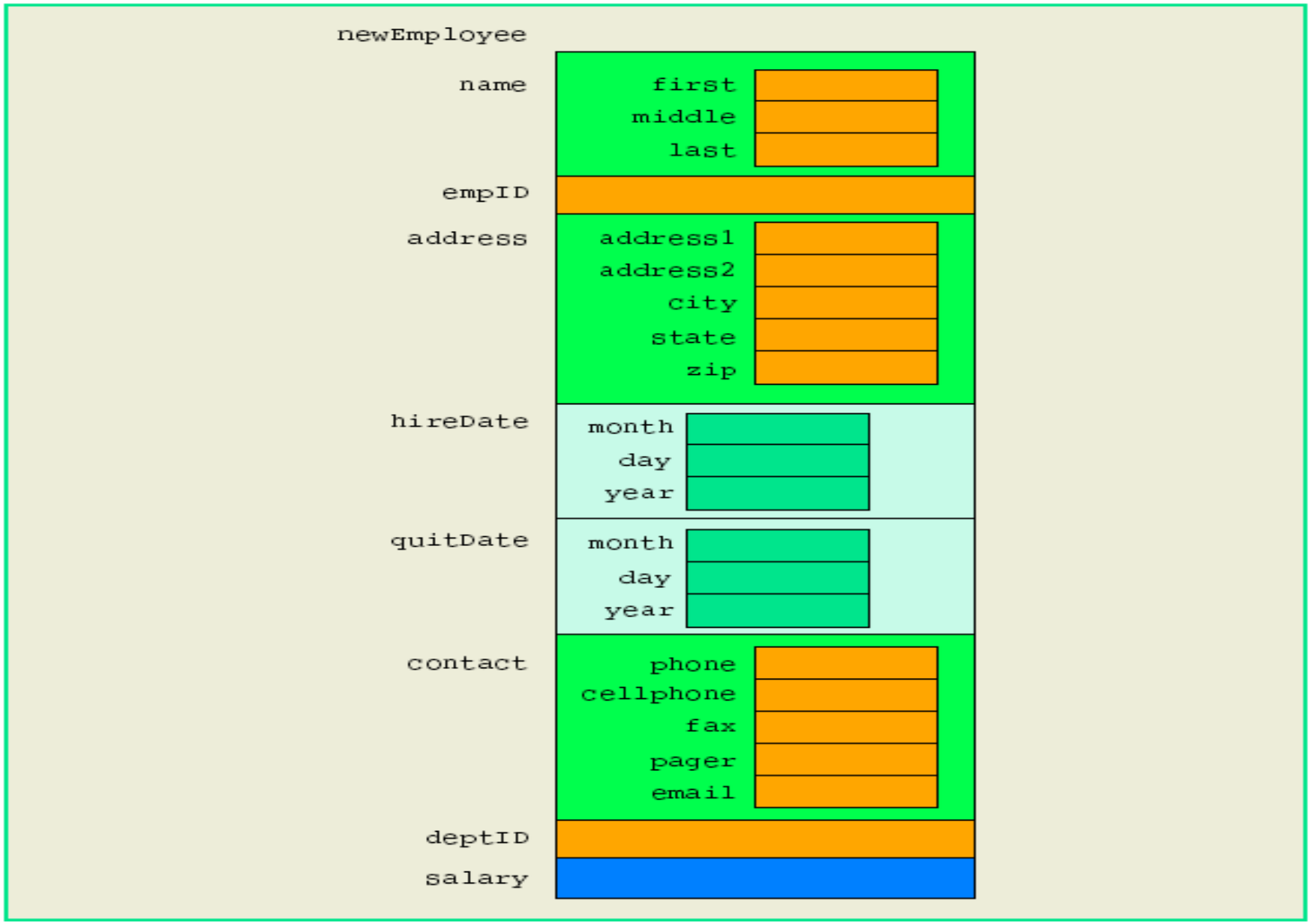
Arrays in structs

- Two key items are associated with a list:
  - Values (elements)
  - Length of the list
- Define a struct containing both items:

```
const arraySize = 1000;
struct listType
{
   int listElem[arraySize];   //array containing the list
   int listLength;               //length of the list
};
```



Array of employees



struct variable newEmployee

Summary

- Struct: collection of a fixed number of components
- Components can be of different types
- struct is a reserved word
- No memory is allocated for a struct; memory is allocated for struct variables when declared
- Components of a struct are called members

Summary (cont.)

- struct components are accessed by name
- Dot (.) operator is called the member access operator
- Members of a struct are accessed using the dot (.) operator
- The only built-in operations on a struct are the assignment and member access

Summary (cont.)

- Neither arithmetic nor relational operations are allowed on the entire structure
- structures can be passed by value or reference
- A function can return a structure
- A structure can be a member of another structure

List ADT

A general list of size N, of the form: $A_0, A_1, A_2, ..., A_{N-1}$.

N=0 -> empty list

For any non-empty list:

A_i follows (or succeeds) A_{i-1} $(i < N)$ and

A_{i-1} precedes A_i $(i > 0)$

We will not define the predecessor of A_0 or the successor of A_{N-1}

The **position** of element A_i in a list is i

The first element of the list is A_0, and the last element is A_{N-1}

List ADT
Set of Operations

- Insert
- Remove
- Find
- Next
- Previous
- Print
- Clear

List ADT
Set of Operations

Given list: 34, 12, 52, 16, 12

Find(52) -> returns 2

Insert(25, 2) -> 34, 12, 25, 52, 16, 12

Remove(52) -> 34, 12, 15, 16, 12

Next(2) -> 16 // next of given index

Previous(2) -> 12 // prev of given index

What does Find(8) returns?

The interpretation of what is appropriate for a function is entirely up to the programmer, as is the handling of special cases.

A Simple Array Implementation of Lists

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;              // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                          // Maximum size of list
  int listSize;                         // Current # of list items
  int curr;                             // Position of current element
```

Ref: OpenDSA Project
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ListArray.html

A Simple Array Implementation of Lists

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;              // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                          // Maximum size of list
  int listSize;                         // Current # of list items
  int curr;                             // Position of current element
```

```
public:
  // Constructors
  // Create a new list object with maximum size "size"
  AList(int size = DEFAULT_SIZE) : listSize(0), curr(0) {
    maxSize = size;
    listArray = new ListItemType[size];         // Create listArray
  }

  ~AList() { delete [] listArray; }      // destructor to remove array
```

Ref: OpenDSA Project
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ListArray.html

A Simple Array Implementation of Lists Operations

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;              // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                          // Maximum size of list
  int listSize;                         // Current # of list items
  int curr;                             // Position of current element
```

```
  public:
```

```
// Reinitialize the list
void clear() { listSize = curr = 0; }  // Simply reinitialize values
```

Ref: OpenDSA Project
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ListArray.html

A Simple Array Implementation of Lists Operations

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;              // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                          // Maximum size of list
  int listSize;                         // Current # of list items
  int curr;                             // Position of current element
```

```
public:
```

```
// Insert "it" at current position
bool insert(const ListItemType& it) {
  if (listSize >= maxSize) return false;
  for (int i = listSize; i > curr; i--)  // Shift elements up
    listArray[i] = listArray[i-1];       // to make room
  listArray[curr] = it;
  listSize++;                            // Increment list size
  return true;
}
```

Ref: OpenDSA Project
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ListArray.html

A Simple Array Implementation of Lists Operations

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;              // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                          // Maximum size of list
  int listSize;                         // Current # of list items
  int curr;                             // Position of current element
```

```
public:
```

```
// Append "it" to list
bool append(const ListItemType& it) {
  if (listSize >= maxSize) return false;
  listArray[listSize++] = it;
  return true;
}
```

Ref: OpenDSA Project
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ListArray.html

A Simple Array Implementation of Lists Operations

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;              // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                          // Maximum size of list
  int listSize;                         // Current # of list items
  int curr;                             // Position of current element
```

```
  public:
```

```
// Remove and return the current element
ListItemType remove() {
  if ((curr < 0) || (curr >= listSize)) // No current element
    throw std::out_of_range("remove() in AList has current of " + to_string(curr) + " and size of "
      + to_string(listSize) + " that is not a a valid element");
  ListItemType it = listArray[curr];     // Copy the element
  for(int i = curr; i < listSize-1; i++) // Shift them down
    listArray[i] = listArray[i+1];
  listSize--;                            // Decrement size
  return it;
}
```

Ref: OpenDSA Project
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ListArray.html

A Simple Array Implementation of Lists Operations

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;              // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                          // Maximum size of list
  int listSize;                         // Current # of list items
  int curr;                             // Position of current element
```

```
public:
```

```
// Set current list position to "pos"
bool moveToPos(int pos) {
  if ((pos < 0) || (pos > listSize)) return false;
  curr = pos;
  return true;
}
```

Ref: OpenDSA Project
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ListArray.html

A Simple Array Implementation of Lists Operations

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;              // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                          // Maximum size of list
  int listSize;                         // Current # of list items
  int curr;                             // Position of current element
```

```
  public:
```

```
    void moveToStart() { curr = 0; }        // Set to front
    void moveToEnd() { curr = listSize; }  // Set at end
    void prev() { if (curr != 0) curr--; } // Move left
    void next() { if (curr < listSize) curr++; } // Move right
    int length() { return listSize; }       // Return list size
    int currPos() { return curr; }          // Return current position
```

Ref: OpenDSA Project
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ListArray.html

A Simple Array Implementation of Lists Operations

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;              // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                          // Maximum size of list
  int listSize;                         // Current # of list items
  int curr;                             // Position of current element
```

```
public:
```

```
// Return true if current position is at end of the list
bool isAtEnd() { return curr == listSize; }
```

```
// Check if the list is empty
bool isEmpty() { return listSize == 0; }
```

Ref: OpenDSA Project
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ListArray.html

A Simple Array Implementation of Lists Operations

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;              // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                          // Maximum size of list
  int listSize;                         // Current # of list items
  int curr;                             // Position of current element
```

```
  public:
```

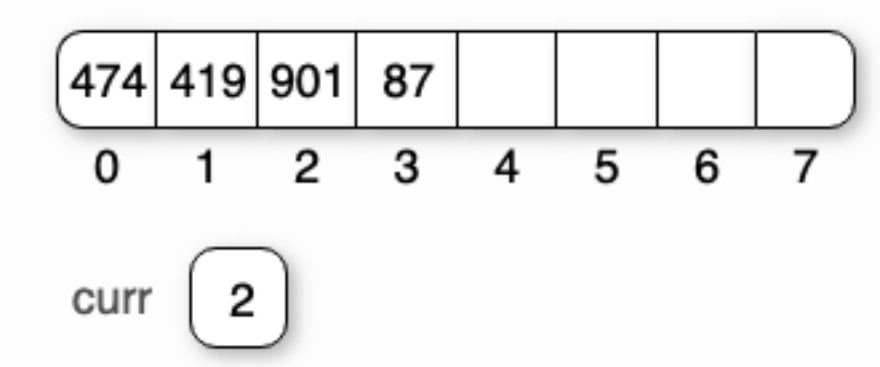
```
// Return the current element
ListItemType getValue() {
  if ((curr < 0) || (curr >= listSize)) // No current element
    throw std::out_of_range("getvalue() in AList has current of " + to_string(curr) +  + " and size of "
      + to_string(listSize) + " that is not a a valid element");
  return listArray[curr];
}
```

Ref: OpenDSA Project
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ListArray.html

Insert

```
// Insert "it" at current position
public boolean insert(Object it) {
  if (listSize >= maxSize) return false;
  for (int i=listSize; i>curr; i--)  // Shift elements up
    listArray[i] = listArray[i-1];   //   to make room
  listArray[curr] = it;
  listSize++;                        // Increment list size
  return true;
}
```

insert 512 to current position

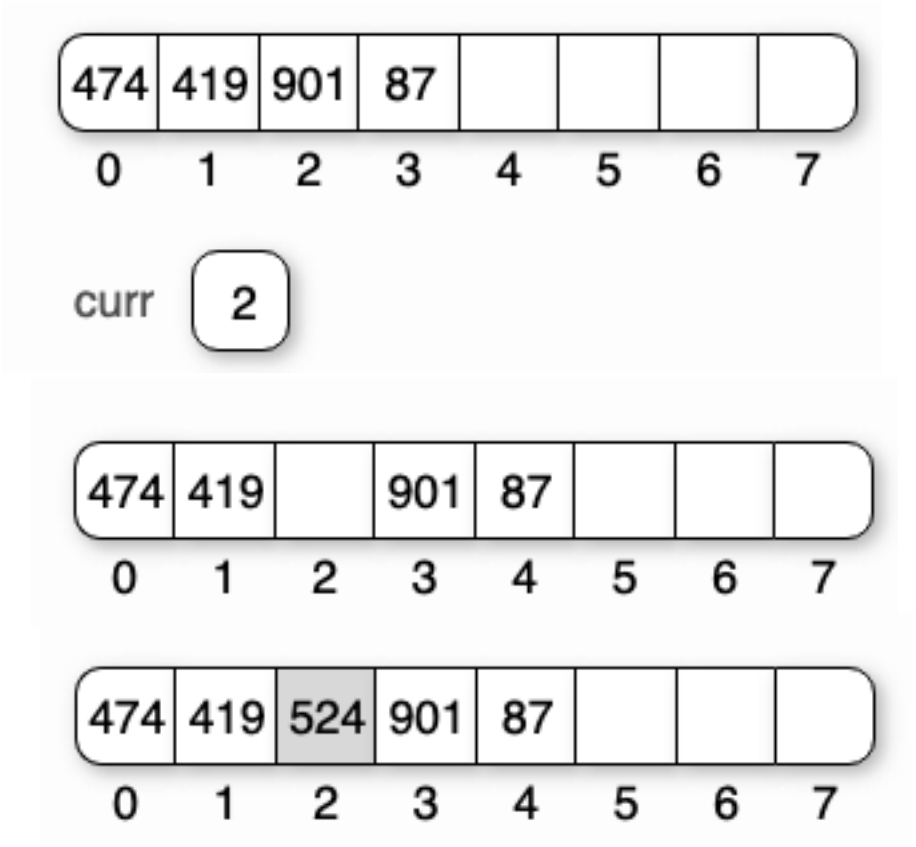


Ref: OpenDSA Project
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ListArray.html

Insert

```
// Insert "it" at current position
public boolean insert(Object it) {
  if (listSize >= maxSize) return false;
  for (int i=listSize; i>curr; i--)  // Shift elements up
    listArray[i] = listArray[i-1];   //   to make room
  listArray[curr] = it;
  listSize++;                        // Increment list size
  return true;
}
```

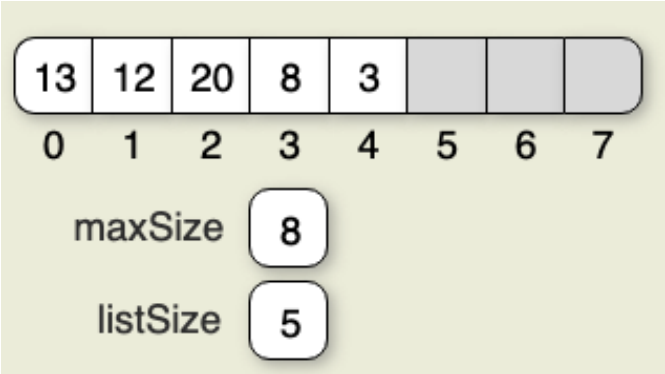
insert 512 to current position



Append

Append 23

```
// Append "it" to list
public boolean append(Object it) {
  if (listSize >= maxSize) return false;
  listArray[listSize++] = it;
  return true;
}
```

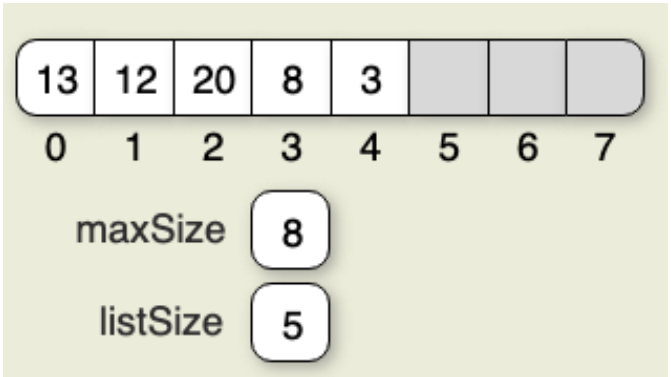


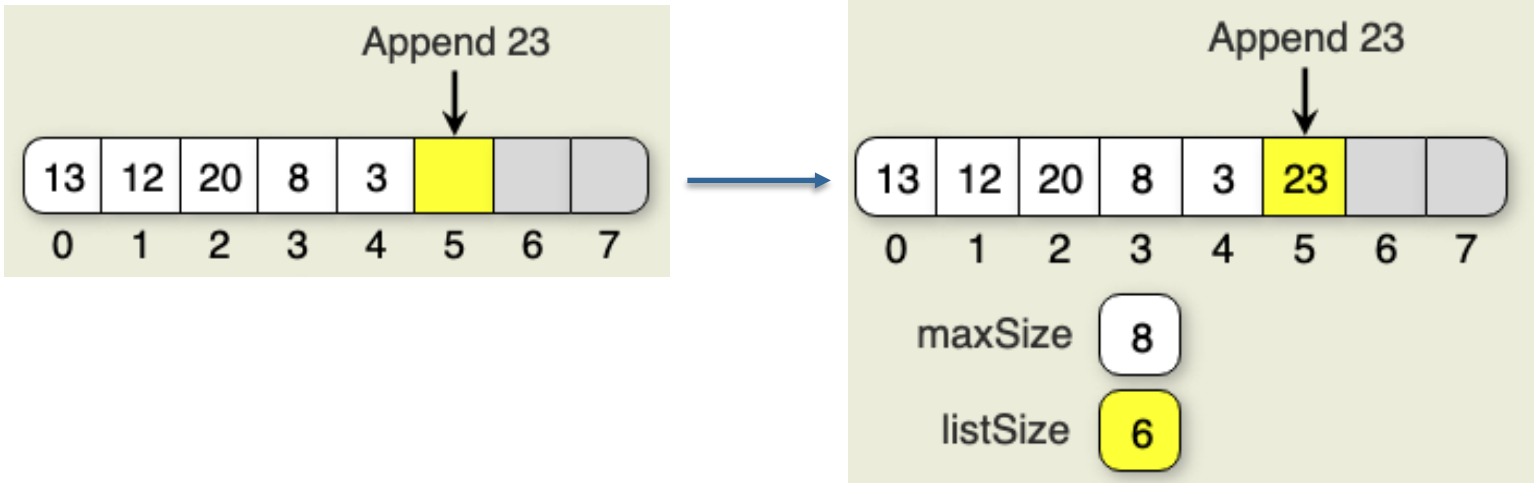
Ref: OpenDSA Project
https://opendsa-

Append

Append 23

```
// Append "it" to list
public boolean append(Object it) {
  if (listSize >= maxSize) return false;
  listArray[listSize++] = it;
  return true;
}
```

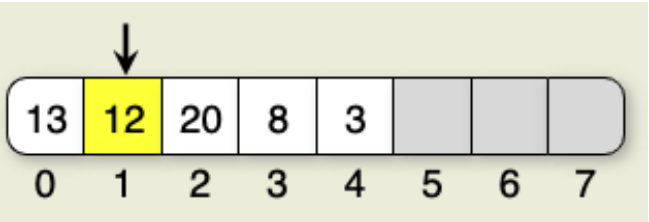




Remove

Remove 12 in position 1

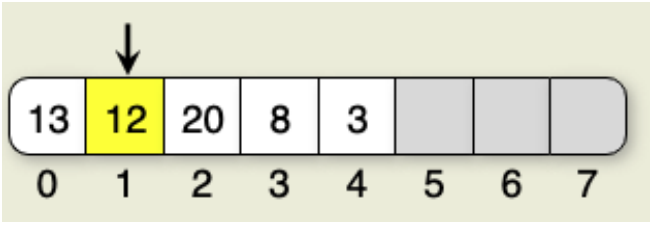
```
// Remove and return the current element
public Object remove() throws NoSuchElementException {
  if ((curr<0) || (curr>=listSize))  // No current eleme
    throw new NoSuchElementException("remove() in AList
      + listSize + " that is not a a valid element");
  Object it = listArray[curr];       // Copy the element
  for(int i=curr; i<listSize-1; i++) // Shift them down
    listArray[i] = listArray[i+1];
  listSize--;                        // Decrement size
  return it;
}
```

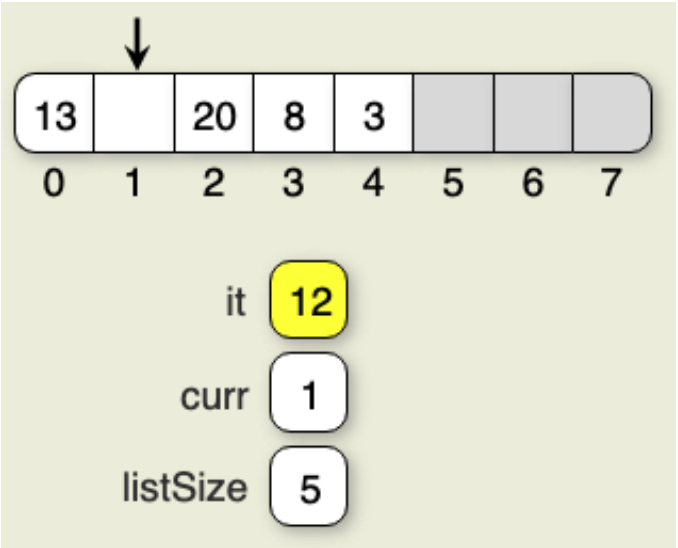


Remove

```
// Remove and return the current element
public Object remove() throws NoSuchElementException {
  if ((curr<0) || (curr>=listSize))  // No current eleme
    throw new NoSuchElementException("remove() in AList
      + listSize + " that is not a a valid element");
  Object it = listArray[curr];       // Copy the element
  for(int i=curr; i<listSize-1; i++) // Shift them down
    listArray[i] = listArray[i+1];
  listSize--;                        // Decrement size
  return it;
}
```

Remove 12 in position 1



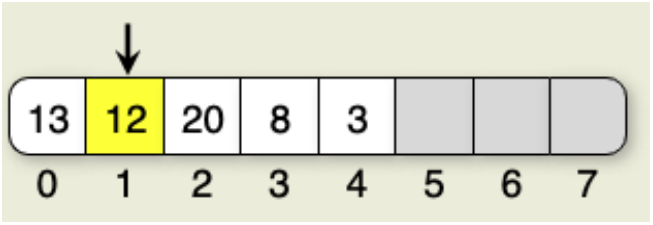


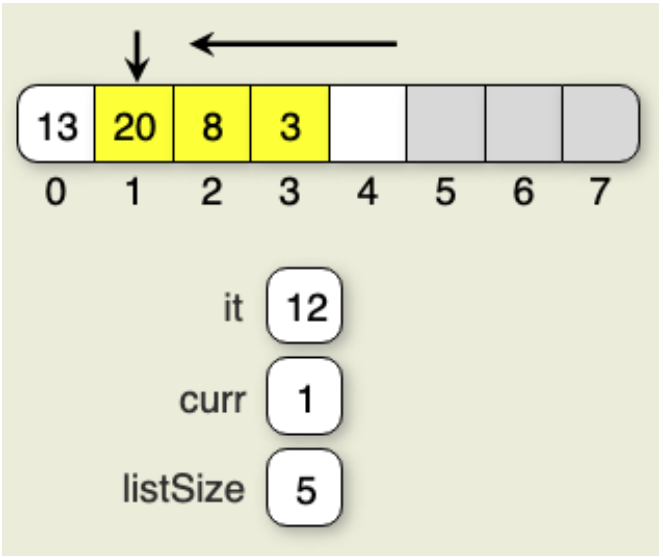
Ref: OpenDSA Project
https://opendsa-

Remove

```
// Remove and return the current element
public Object remove() throws NoSuchElementException {
  if ((curr<0) || (curr>=listSize))  // No current eleme
    throw new NoSuchElementException("remove() in AList
      + listSize + " that is not a a valid element");
  Object it = listArray[curr];       // Copy the element
  for(int i=curr; i<listSize-1; i++) // Shift them down
    listArray[i] = listArray[i+1];
  listSize--;                        // Decrement size
  return it;
}
```

Remove 12 in position 1



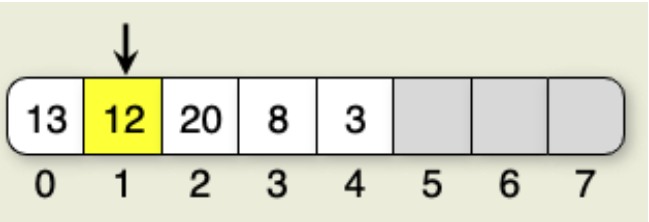


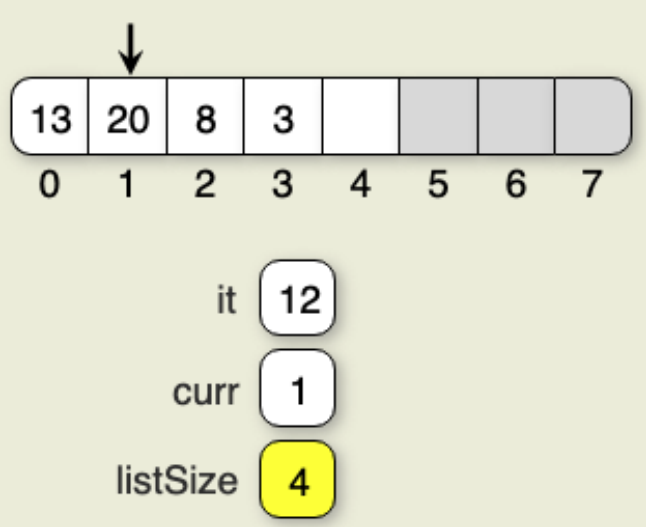
Ref: OpenDSA Project
https://opendsa-

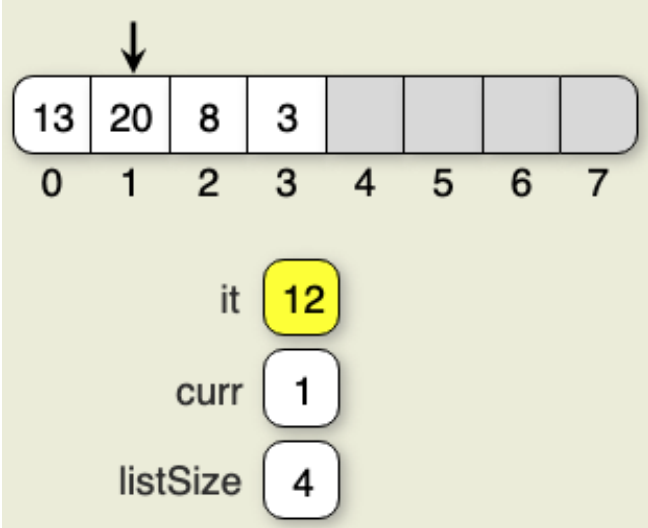
Remove

Remove 12 in position 1

```
// Remove and return the current element
public Object remove() throws NoSuchElementException {
  if ((curr<0) || (curr>=listSize))  // No current eleme
    throw new NoSuchElementException("remove() in AList
      + listSize + " that is not a a valid element");
  Object it = listArray[curr];       // Copy the element
  for(int i=curr; i<listSize-1; i++) // Shift them down
    listArray[i] = listArray[i+1];
  listSize--;                        // Decrement size
  return it;
}
```







Ref: OpenDSA Project
https://opendsa-

Computational Times

Printing list -> linear time

Empty list -> constant time

Insert & remove ops -> expensive

- depends on where the insertions and deletions occur

Computational Times

Insertion (worst case)

- Insert at the front of the list
- What is the complexity?

Insertion (best case)

- Insert at the end of the list
- What is the complexity?

Computational Times

Remove (worst case)

- Deleting the first element of the list
- What is the complexity?

Remove (best case)

- Deleting the last element of the list
- What is the complexity?

Remarks

There are many situations where the list is built up by insertions at the high end, and then only array accesses (i.e., getValue operations) occur. In such a case, the array is a suitable implementation.

If insertions and deletions occur throughout the list and in particular, at the front of the list, then the array is not a good option.