Coercion and Type Casting

Unions

Linked Lists

Coercion and Type Casting

- Coercion and Type-Casting are concepts in C that deal with data type conversions.

Coercion:

- Implicit conversion automatically performed by compiler
- Happens during expressions involving different data types
- Convert the data type without losing the actual meaning

```c
int num = 10;
float result = num / 2.0;
int intValue = 50;
double doubleValue = intValue; // Implicit Type Casting
```

Type-Casting

- Explicit : Programmer control over conversions and potentially changes data value
- Use () to cast the variable to another type

```
int x = 4, y;
float a = 12.4;
y = (int)a + x;
double value = 3.14;
int intValue = (int)value; // Explicit Type-Casting
```

Potential Issues

- May lead to loss of data or unexpected results.

```
int number = 1000;
char letter = (char)number; // Potential loss of data
```

Type Casting

- C allows this sing the cast operator (). So:

```
int integernumber;
float floatnumber = 9.87;
char letter='A';

integernumber = (int)floatnumber;

integernumber = 10;
floatnumber = (float)integernumber;

integernumber = (int)letter;

floatnumber = (float)internumber / (float)anotherint;
```

Unions

- A derived data type, like a structure
- Members can be any data type
- Increases memory efficiency
- Allows storing different data types
- Only one field can be used at a time
- Fields overlay the same memory address
- Size is equal to largest data member

Defined Union

```
union data {
  int i;
  float f;
  char str[20];
};

union data myData;

printf("%p\n", &myData);
myData.i = 10;
printf("%p\n", &myData.i);
printf("%d\n", myData.i);
// or
myData.f = 220.5;
printf("%p\n", &myData.f);
printf("%.3f\n", myData.f);
// or
strcpy(myData.str, "Hello");
printf("%p\n", myData.str);
printf("%s\n", myData.str);
```

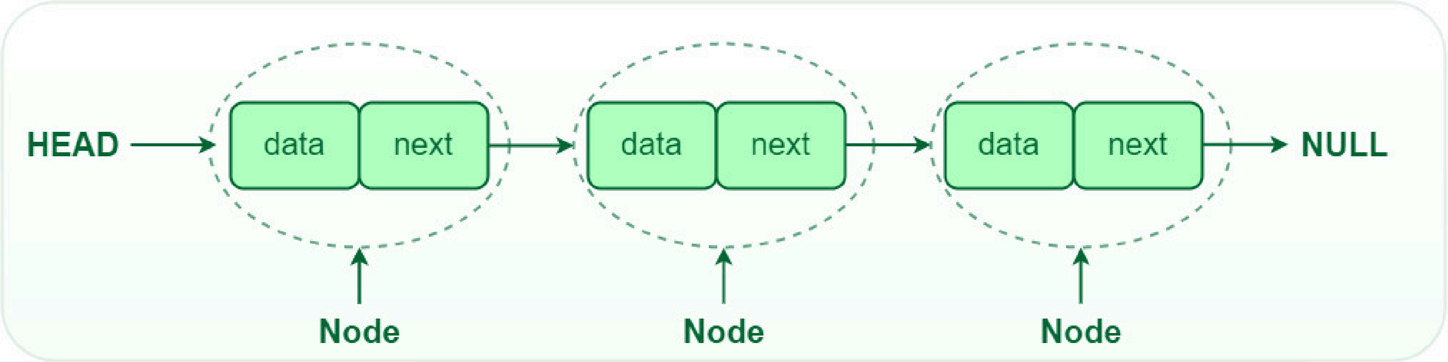
```c
#include <stdio.h>

union number {
  int x;
  double y;
}

int main() {
  union number value;

  value.x = 100;
  printf("Print value of x %d and y %f\n",
   value.x, value.y);
  value.y = 100.0;
  printf("Print value of x %d and y %f\n",
   value.x, value.y);
  printf("Address of x %p and y %p\n",
   &value.x, &value.y);
  return 0;
}
```

Linked Lists

- A linear data structure each element points to the next
- Useful for inserting and deleting elements efficiently
- Elements not stored contiguously like arrays
- Element is a separate object contains data and pointers
- Keeps memory costs low on removal and additions
- Self-referential structure contains a member that’s a pointer to the same structure type

Defined Linked List

```
struct node {
  int data;
  struct node* next;
};
```



```c
struct node* head = NULL;
struct node* second = NULL;
struct node* third = NULL;

// Allocate 3 nodes
head = (struct node*)malloc(sizeof(struct node));
second = (struct node*)malloc(sizeof(struct node));
third = (struct node*)malloc(sizeof(struct node));

head->data = 1;
head->next = second;

second->data = 2;
second->next = third;

third->data = 3;
third->next = NULL;
```

Types of Linked List

- Singly linked list
- Doubly linked list
- Circular linked list

```c
node* current = head;

while (current != NULL) {
  printf("%d ", current->data);
  current = current->next;
}
```

Linked List Advantages

- Dynamic size: Grow and shrink by allocating/deallocating
    - Arrays have a fixed capacity specified on initialization
- Ease of insertion/deletion: By modifying node links
    - Arrays requires shifting elements, time consuming
- No memory overhead: Occupy any available memory
    - Arrays overhead fragmentation fixed sizes or padding
- Less memory waste: only use memory for nodes present
    - Unused array elements still take up space
- Memory efficiency for sparse data: save with pointers
    - Array elements frequent NULL value

Disadvantages

- Lower access time
- Greater complexity for maintenance due to pointers
- No random access which arrays permit using indices
- Concept which can be difficult to understand

Questions?