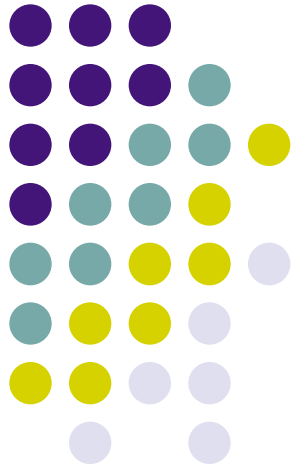
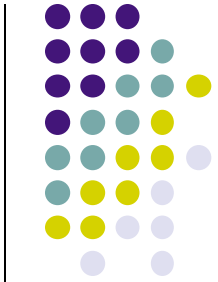
Linked list

2019 Spring

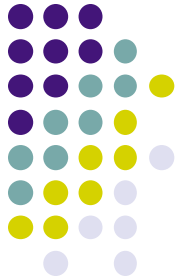


Contents

- Self referenced structure
- Implementing Data Structures in C



자기 참조 구조체

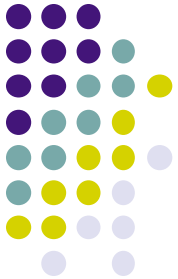


- 여러 권의 책에 대한 정보를 어떻게 저장할까?
  - 구조체 배열을 이용: 배열의 크기를 미리 정해야 함
  - 보다 유연한 데이터 표현 방법이 필요함

- 구조체 내에 자신을 가리킬 수 있는 포인터 변수를 선언

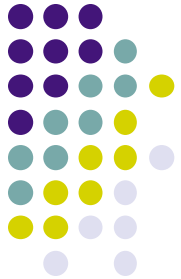
```
struct book {
    char title[50];
    char author[20];
    char publisher[20];
    struct date pub_day;
    struct book *next;
};
```

자기참조 구조체 사용



```
struct book *ptr;

// 첫 번째 책을 위한 구조체 생성
ptr = (struct book *) malloc(sizeof(struct book));
...                                    // 첫 번째 책 자료 대입

// 두 번째 책을 위한 구조체 생성
ptr->next = (struct book *) malloc(sizeof(struct book));
ptr = ptr->next;                       // 두 번째 책 구조체를 포인터
...                                    // 두 번째 책 자료 대입

// 세 번째 책을 위한 구조체 생성
ptr->next = (struct book *) malloc(sizeof(struct book));
ptr = ptr->next;                       // 세 번째 책 구조체를 포인터
...                                    // 세 번째 책 자료 대입
```

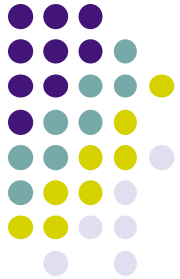
자기 참조 구조체의 용도 (1/2)



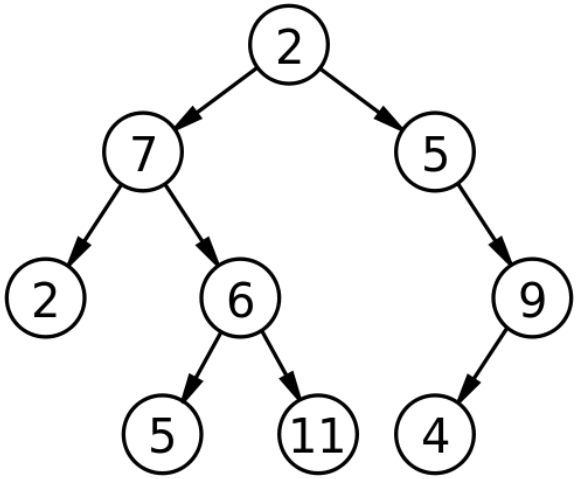
- 자기 참조 구조체란?
  - 구조체의 필드에 구조체 포인터 타입이 사용된 구조체
  - 해당 필드를 통해 자신과 같은 형태의 구조체를 참조할 수 있음

- 자기 참조 구조체가 필요한 이유
  - 유연한 개수의 레코드를 저장해야 할 때 사용함(연결 리스트)
  - 컨테이너 구조가 복잡하게 얽혀 있는 경우에 사용함(트리나 그래프)

자기 참조 구조체의 용도 (2/2)



- 자기 참조 구조체의 장점 및 단점
  - 장점: 데이터 구조를 마음대로 구성할 수 있음
  - 단점: 포인터를 사용해야 하기 때문에 데이터 구조가 불완전하게 생성될 가능성이 있으므로 주의해야 함



연결 리스트

- 연결 리스트(linked list)
  - 데이터 항목 여러 개를 포인터를 통해 연결해 둔 리스트
  - 헤드(head): 연결리스트 맨 앞 항목을 가리키는 포인터
  - 맨 끝 항목의 포인터에는 NULL이 저장됨

- 연결 리스트 예
  - 학생 데이터 연결 리스트

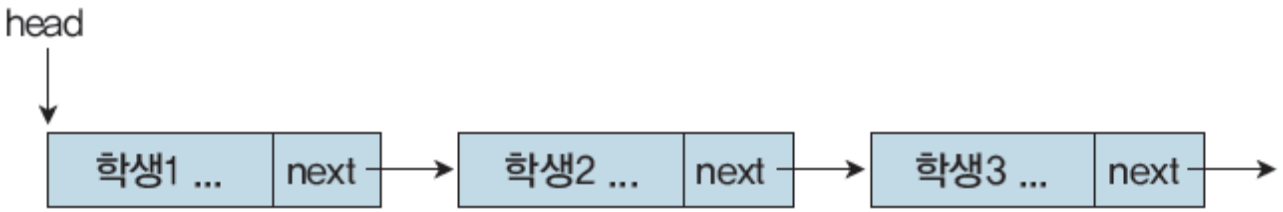


그림 13.4 학생1, 학생2, 학생3의 데이터가 차례로 연결된 연결 리스트

연결 리스트 관리

- 연결 리스트 관리의 필요성
  - head 포인터를 리스트 앞에 레코드를 추가하기는 쉽지만 리스트 뒤에 레코드를 추가하기는 어려움
  - 그래서 tail 포인터도 관리하는 것이 좋음
  - 큐 형태의 연결 리스트 예

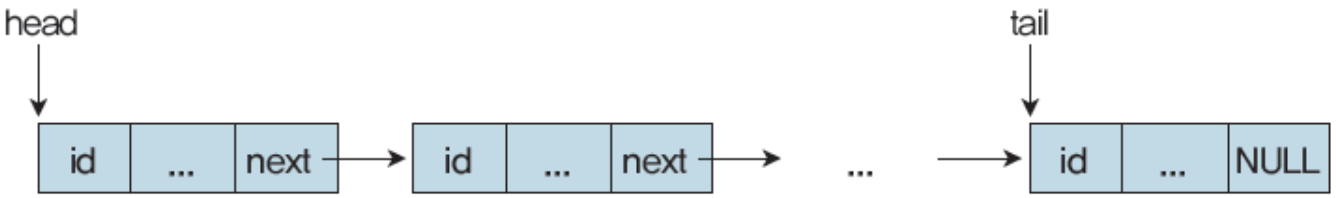
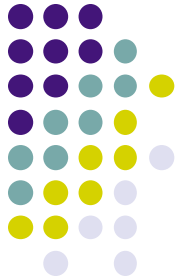
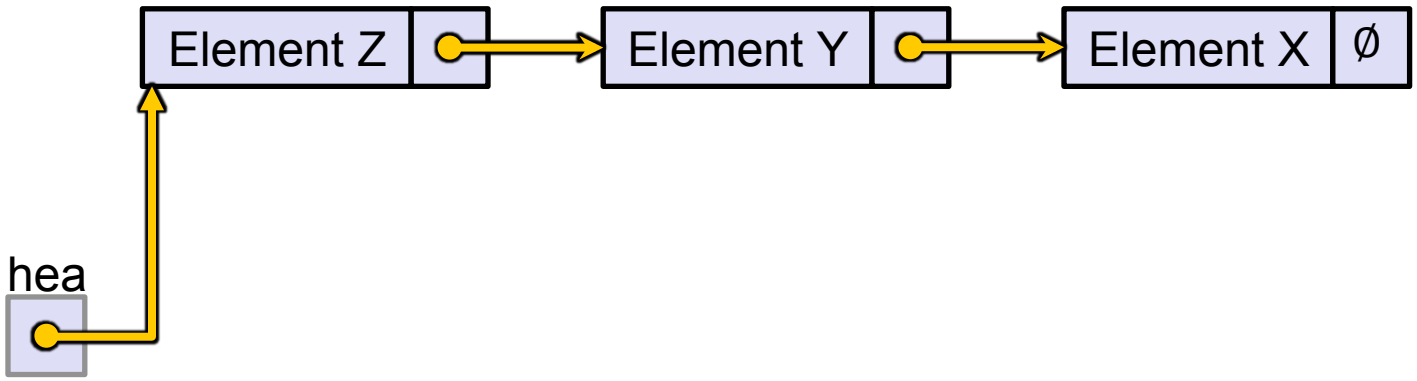


그림 13.5 큐 형태의 연결 리스트

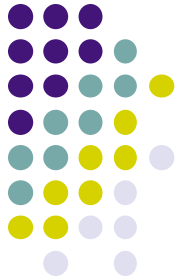
Simple Linked List in C



- Each node in a linear, singly-linked list contains:
  - Some element as its payload
  - A pointer to the next node in the linked list
    - This pointer is NULL (or some other indicator) in the la st node in the list



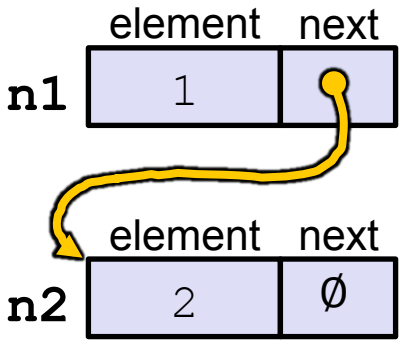
Linked List Node



- Let's represent a linked list node with a struct
  - For now, assume each element is an `int`

```c
#include <stdio.h>

typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

int main(int argc, char** argv) {
  Node n1, n2;

  n1.element = 1;
  n1.next = &n2;
  n2.element = 2;
  n2.next = NULL;
  return 0;
}
```

manual_list.c



Push Onto List(1/14)

Arrow points to *next* instruction.

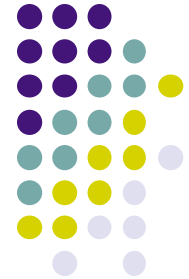
(main) `list` Ø

```
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

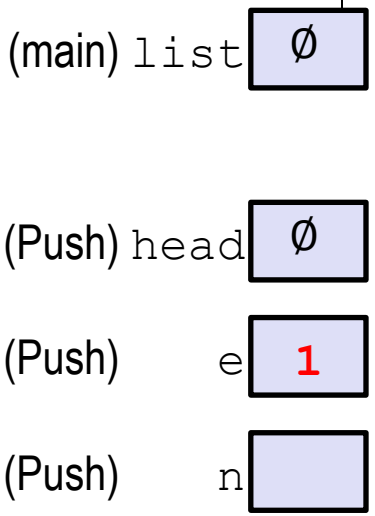
push_list.c

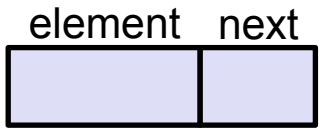
Push Onto List(2/14)

Arrow points to *next* instruction.



```
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

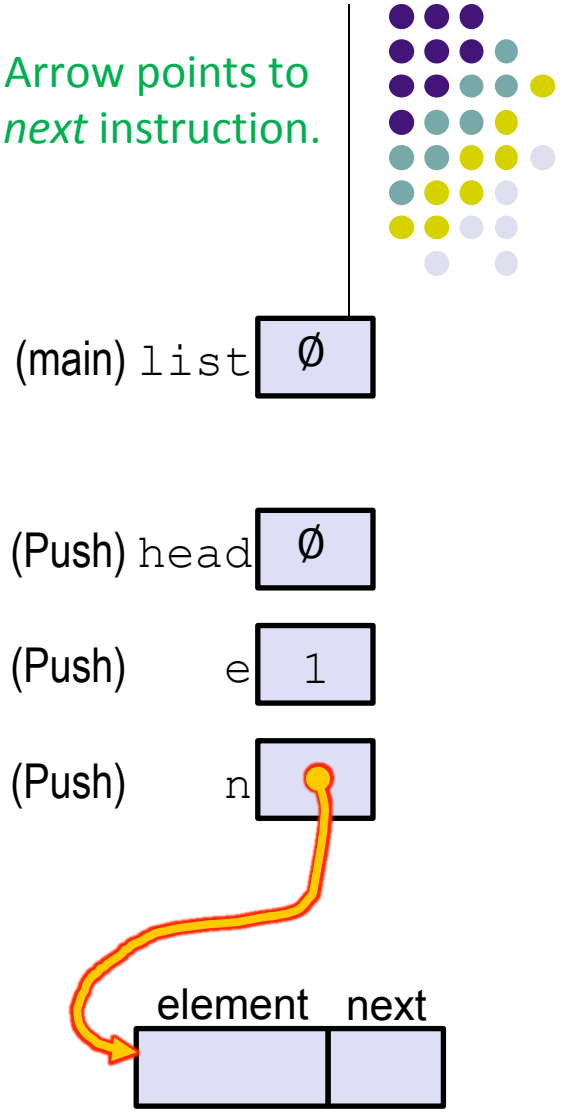




push_list.c

Push Onto List(3/14)

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

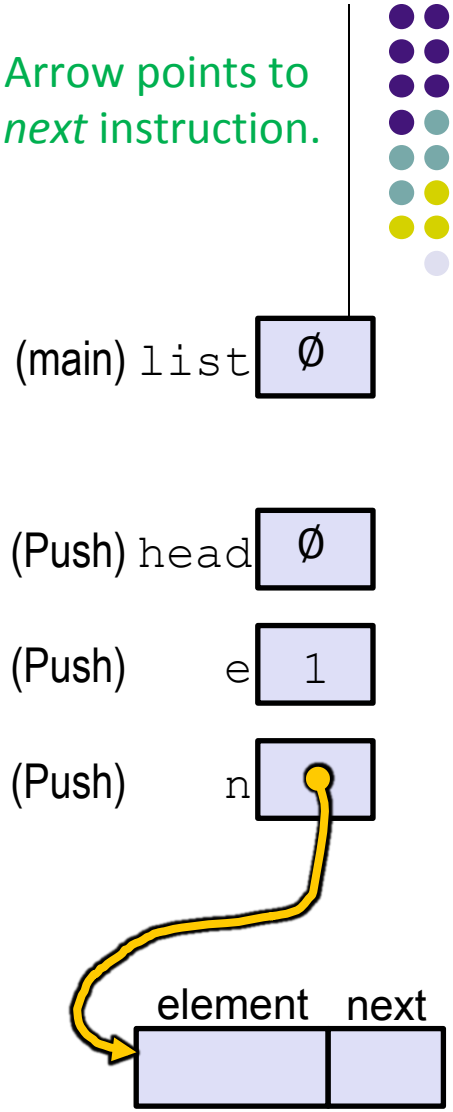


push_list.c

Push Onto List(4/14)

```
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

Arrow points to *next* instruction.

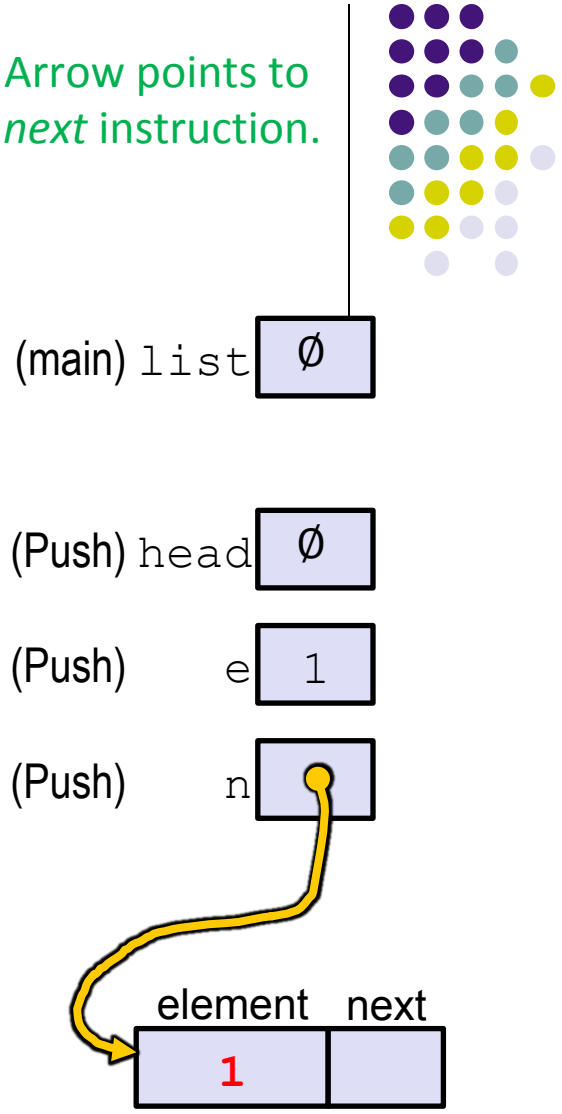


push_list.c

Push Onto List(5/14)

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

Arrow points to *next* instruction.

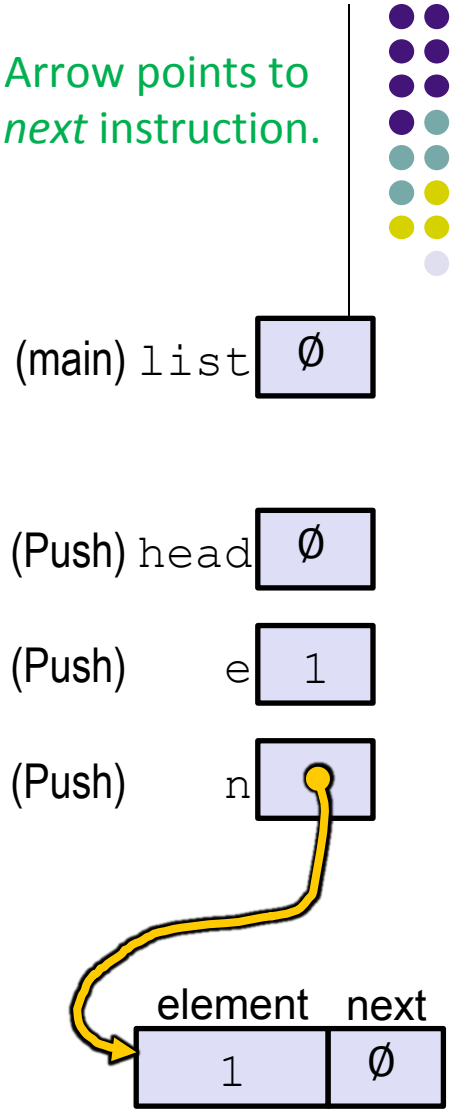


push_list.c

Push Onto List(6/14)

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

Arrow points to *next* instruction.



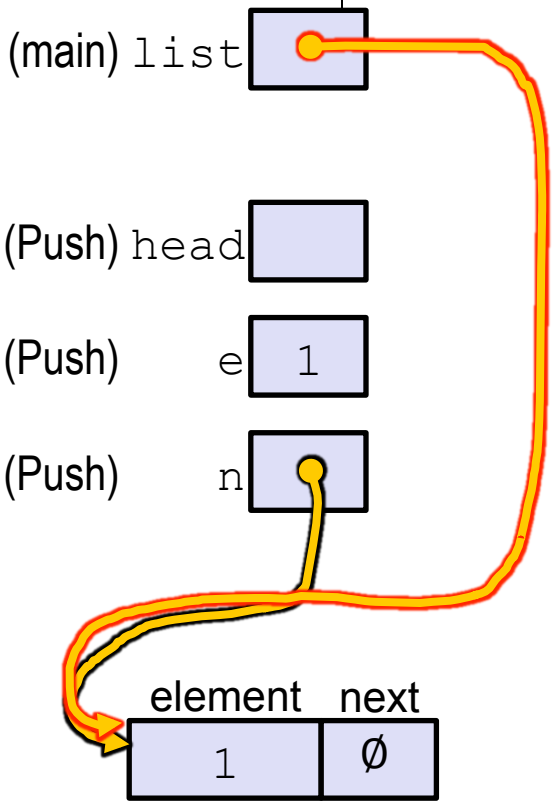
push_list.c

Push Onto List(7/14)

```
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

push_list.c

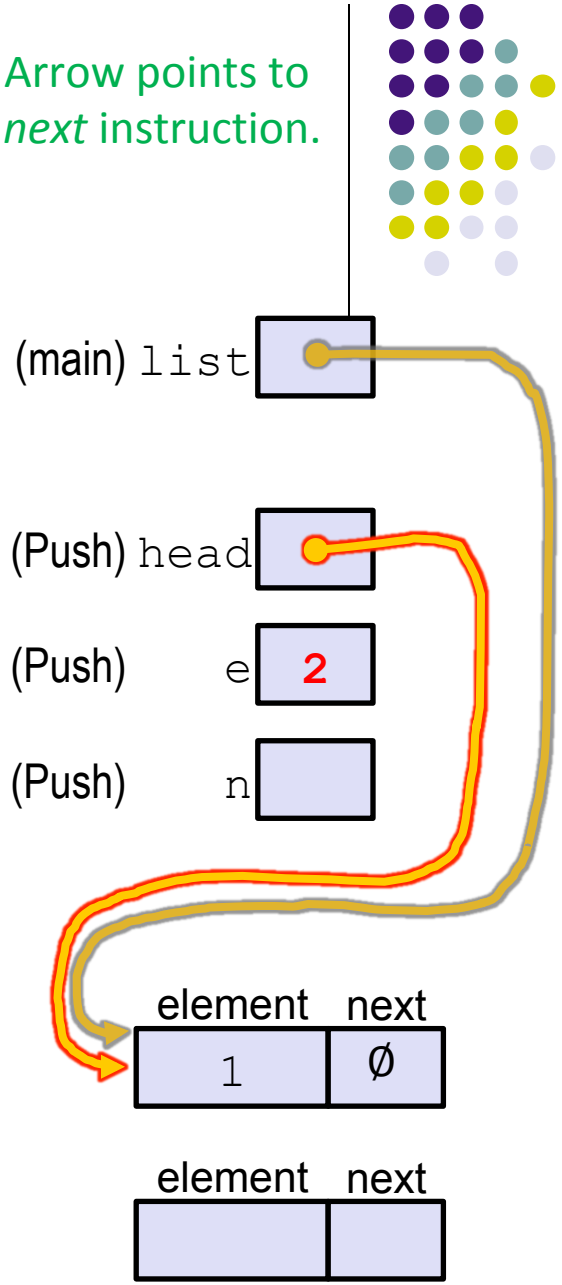
Arrow points to *next* instruction.



Push Onto List(8/14)

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

Arrow points to *next* instruction.

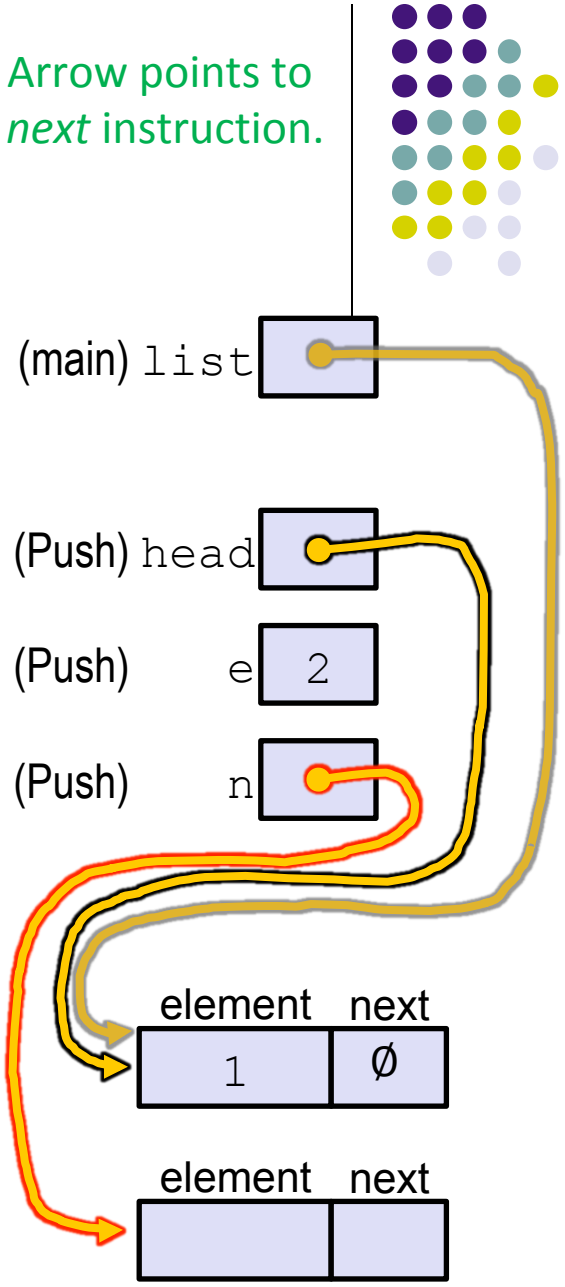


push_list.c

Push Onto List(9/14)

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

Arrow points to *next* instruction.

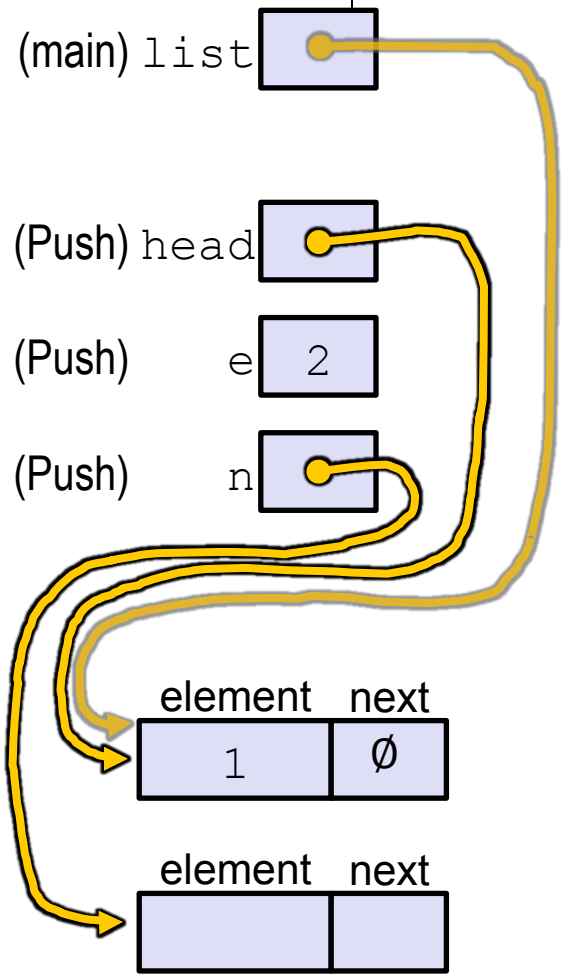


push_list.c

Push Onto List(10/14)

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

Arrow points to *next* instruction.

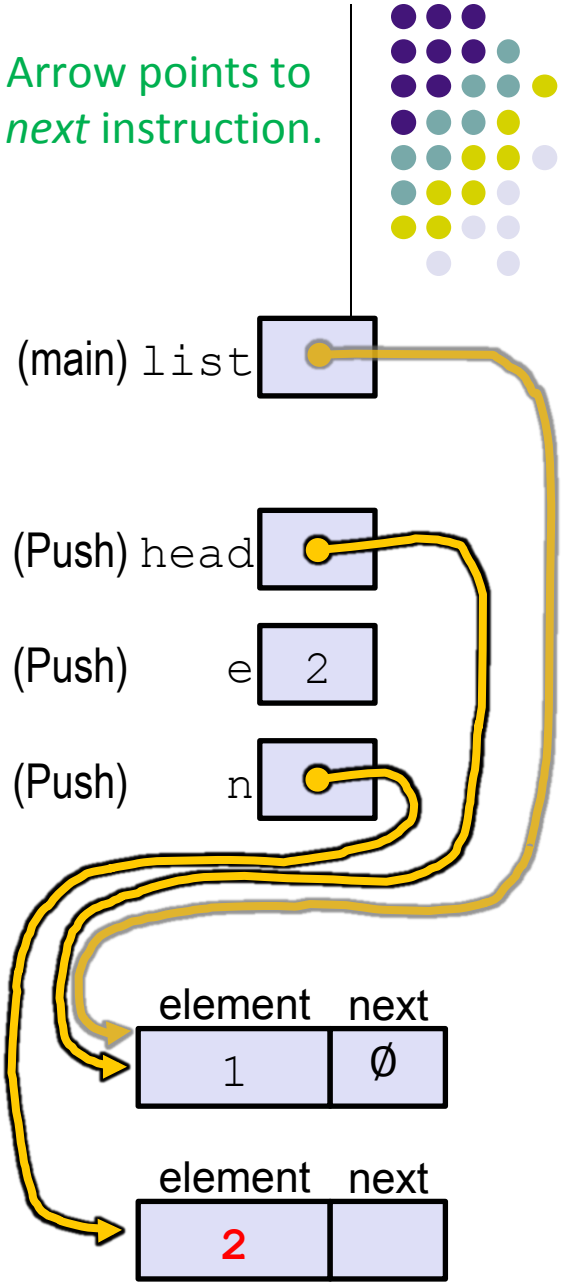


push_list.c

Push Onto List(11/14)

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

Arrow points to *next* instruction.

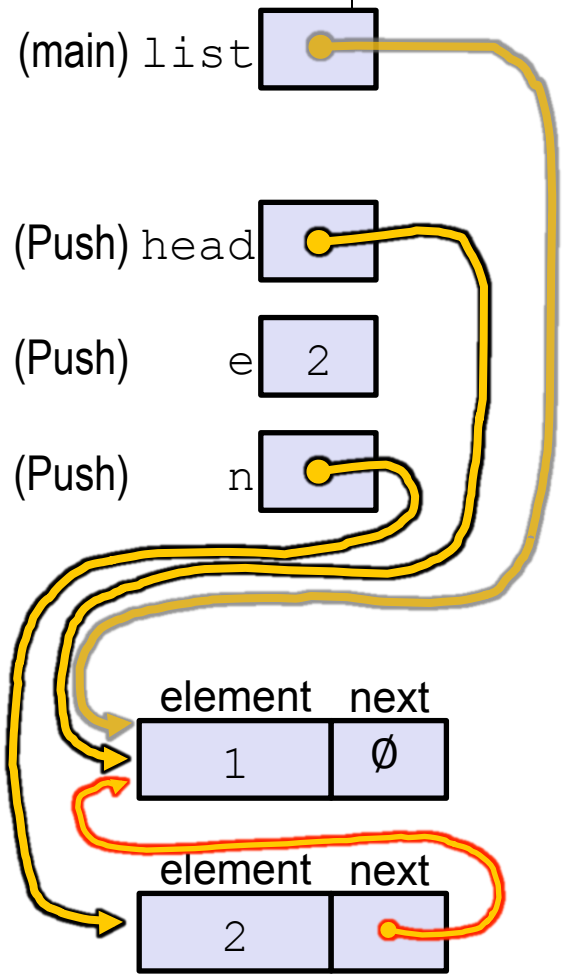


push_list.c

Push Onto List(12/14)

```
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

Arrow points to *next* instruction.



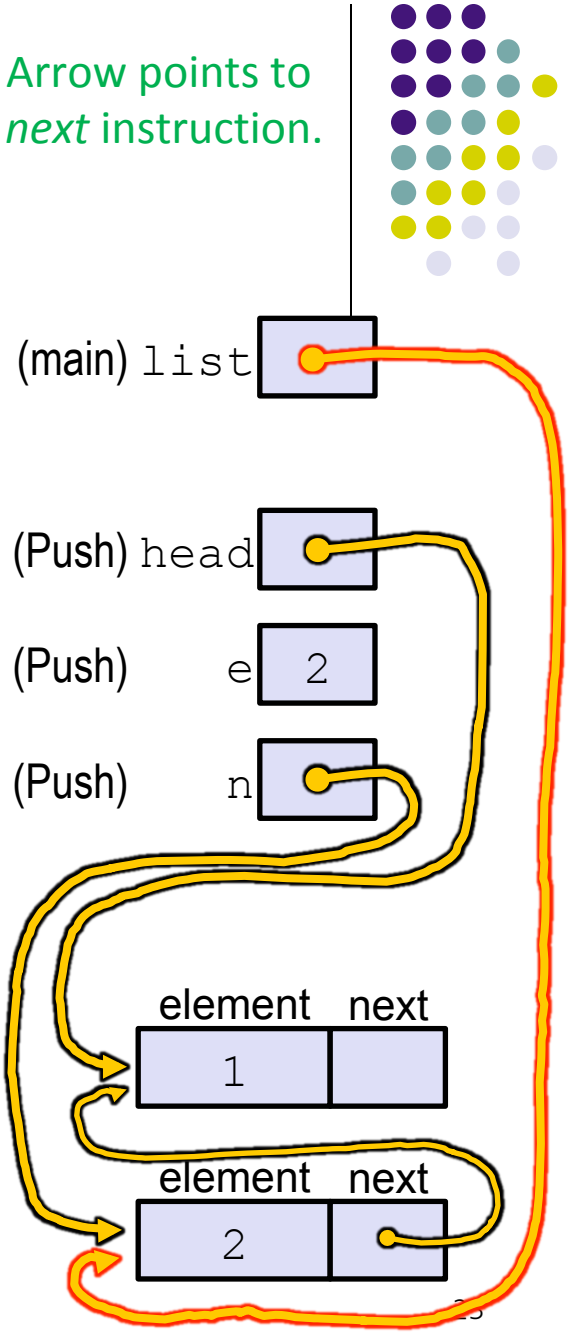
push_list.c

Push Onto List(13/14)

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

push_list.c

Arrow points to *next* instruction.



Push Onto List(14/14)

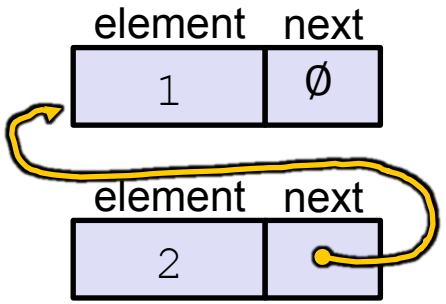
Arrow points to *next* instruction.

```
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

push_list.c

A (benign) memory leak!
Try running with Valgrind:

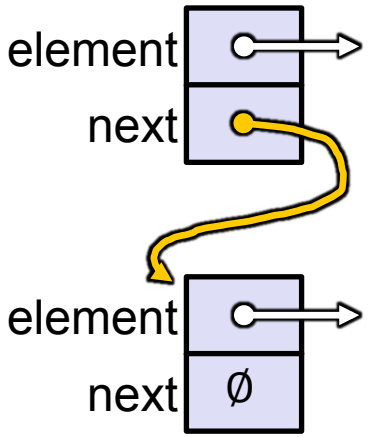
```
bash$ gcc -Wall -g -o
push_list push_list.c

bash$ valgrind --leak-
check=full ./push_list
```



A Generic Linked List

- Let's generalize the linked list element type
  - Let customer decide type (instead of always `int`)
  - Idea: let them use a generic pointer (*i.e.* a `void*`)

```
typedef struct node_st {
  void* element;
  struct node_st* next;
} Node;

Node* Push(Node* head, void* e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}
```



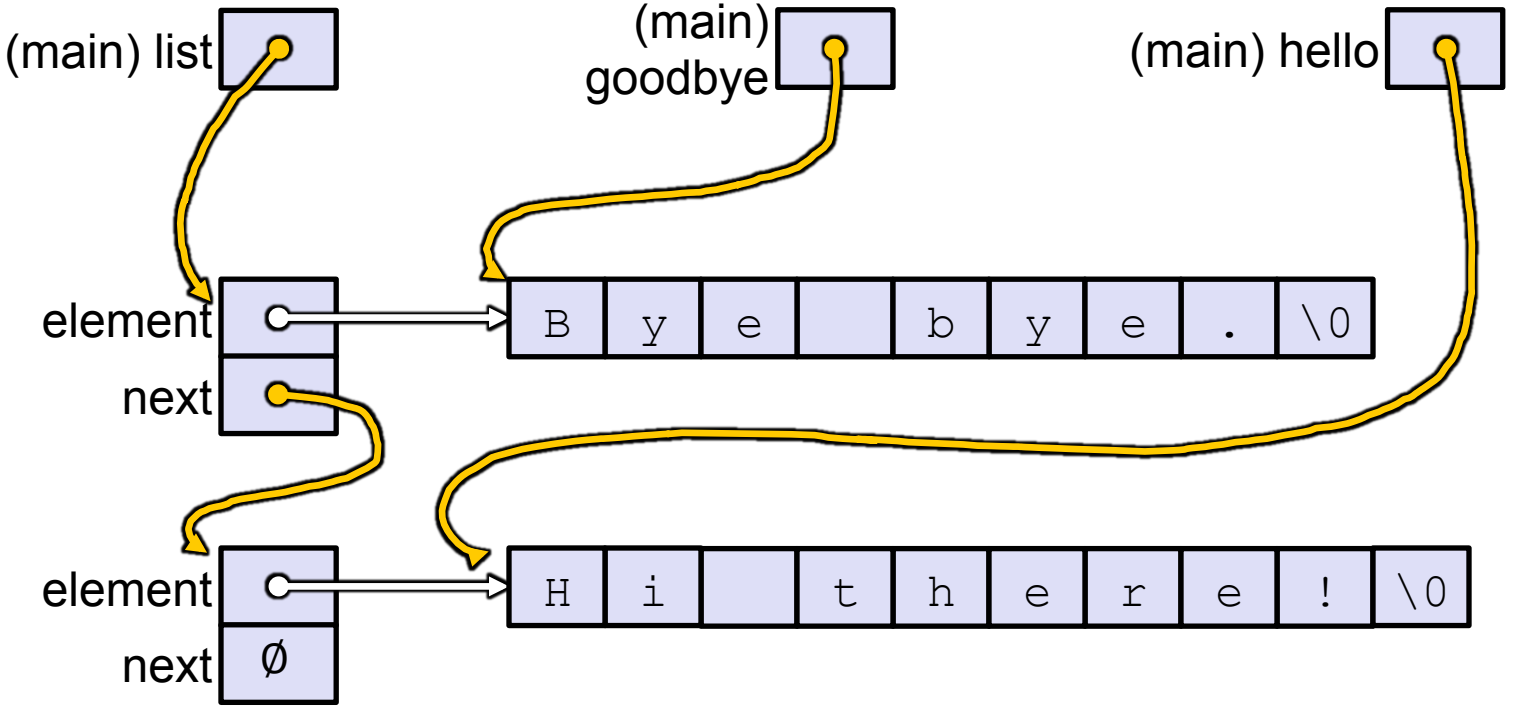
Using a Generic Linked List

- Type casting needed to deal with `void*` (raw address)
  - Before pushing, need to convert to `void*`
  - Convert back to data type when accessing

```
typedef struct node_st {
  void* element;
  struct node_st* next;
} Node;

Node* Push(Node* head, void* e);    // assume last slide's code

int main(int argc, char** argv) {
  char* hello = "Hi there!";
  char* goodbye = "Bye bye.";
  Node* list = NULL;

  list = Push(list, (void*) hello);
  list = Push(list, (void*) goodbye);
  printf("payload: '%s'\n", (char*) ((list->next)->element) );
  return 0;
}
```


manual_list_void.c

Resulting Memory Diagram



Questions?