EEE243 – Applied Computer Programming

Linked Lists

Outline

1. Struct and pointers
2. Struct and arrays
3. Linked Lists
4. LL Creation
5. Traversing
6. Adding and deleting nodes
7. API

Types derived from structs

- structs are very expressive complex types
- we can create derived types from these complex types
  - pointers to structs

```
StructName *pointer_name;
```

  - arrays of structs

```
StructName array_name[X];
```

Recall our Student struct type

```
typedef struct{
    char first_name[15];
    char last_name[25];
    char college_number[6];
    float average;
  } Student;
```

Pointers inside a structure

- If I want to identify the lab partner inside my `Student` structure, I could include new fields:

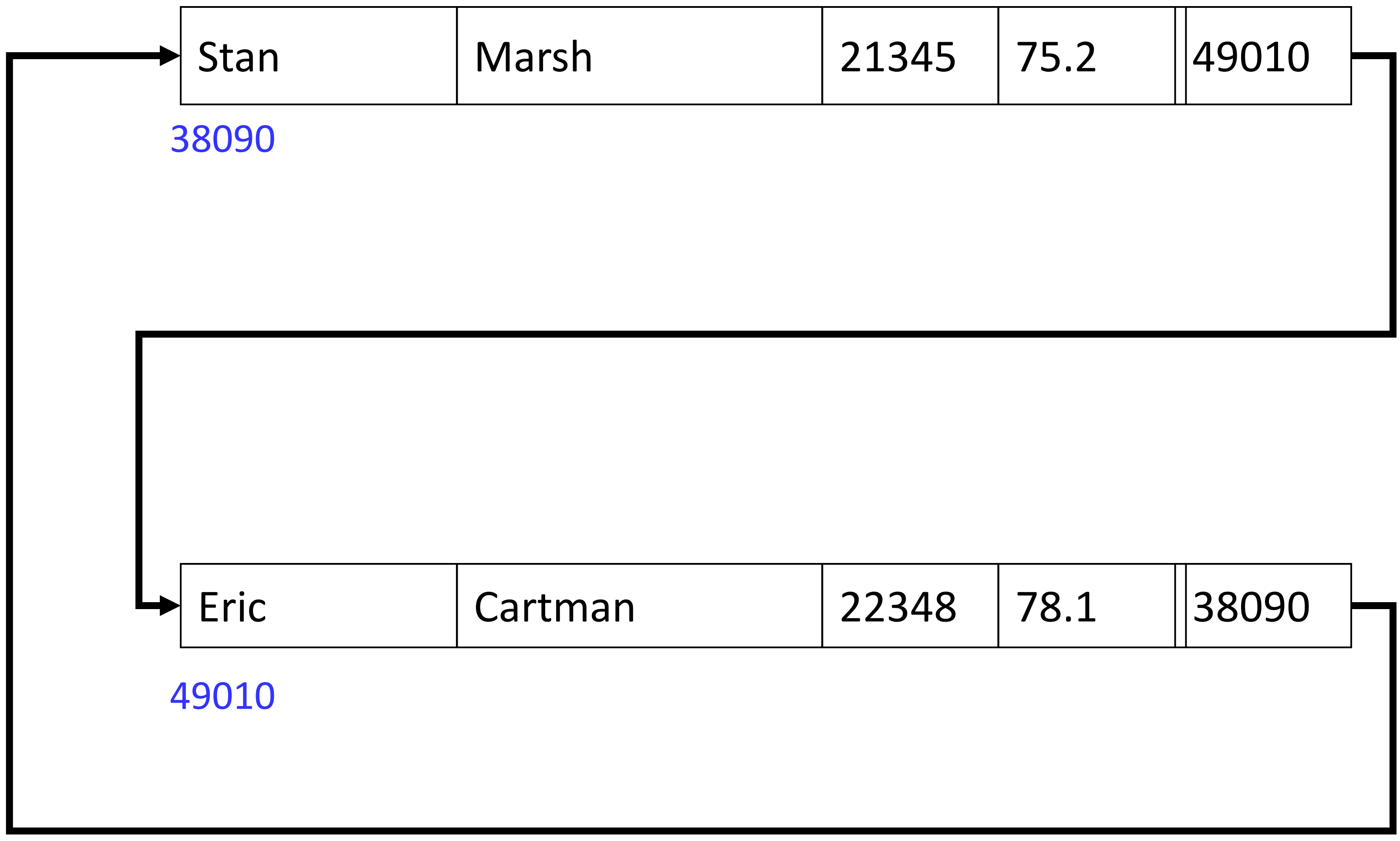
```
typedef struct {
  char first_name[15];
  char last_name[25];
  char college_number[6];
  float average;
  char partner_first_name[15];
  char partner_last_name[25];
} Student;
```

Pointers inside a structure

- but since I know the lab partner is also a student, a pointer might make more sense…

```
typedef struct STUDENT_TAG {
  char first_name[15];
  char last_name[25];
  char college_number[6];
  float average;
  struct STUDENT_TAG *lab_partner;
} Student;
```

Pointers inside a structure



Arrays of structures

- For example:

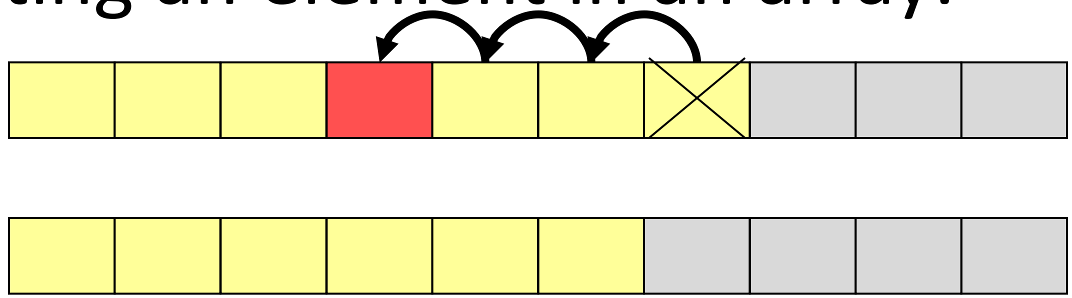
```
typedef struct{
     char first_name[15];
     char last_name[25];
     char college_number[6];
     float average;
  } Student;

Student students[20]; //array of students
```

Arrays of structures

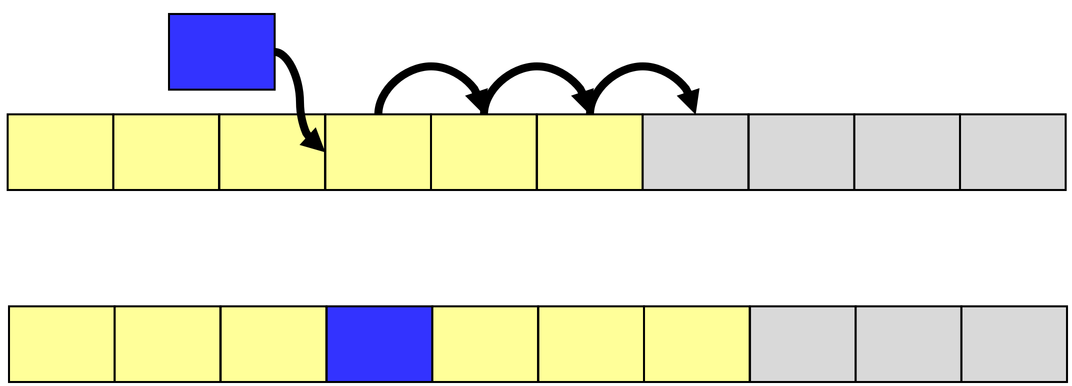
- Arrays of structures are powerful for storing a fixed quantity of complex data *that will not change often*
- However, arrays of structures have some limitations
  - must know required number before compile time
    - could use very large arrays that contain a maximum number of elements – this is a waste of memory
  - delete an element, must leave a “hole” in your array or move back the elements above the deleted position
  - inserting an element at a particular point in an array, e.g., alpha by `last_name` means moving many elements — inefficient

Arrays of structures

- Deleting an element in an array:



- Inserting an element in an array:

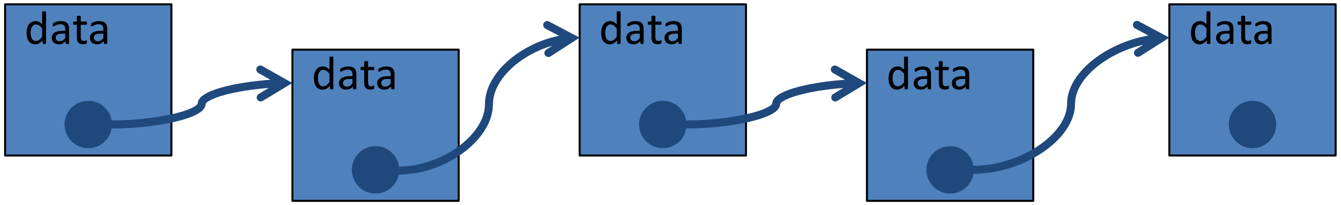


Linked Lists

- We can point from one structure to another structure…and *link them together*
  - so we can create another kind of data structure
- Instead of using arrays of structures, we could use pointers to ***link*** all the structures of the *same type* together
  - This new data structure is called a ***linked list***
  - The elements in a linked list are referred to as ***nodes***

Linked lists - Nodes

- Each node in the linked list contains two main components:
  - The ***data*** – i.e. our student information
  - The ***link*** – a *pointer to the next node* in the list
- A linked list is a chain of structures of the same type



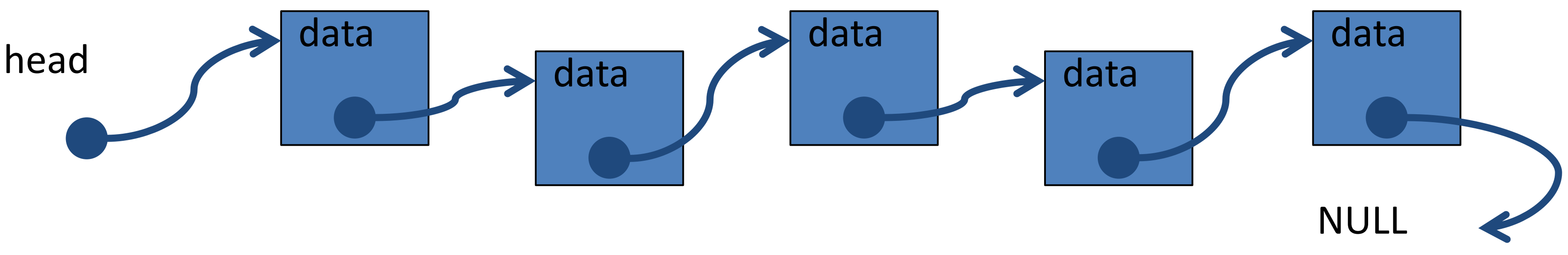
Linked list – Example

- I could declare my student node structure like this:

```
typedef struct STUDENT_TAG {
  char first_name[15];
  char last_name[25];
  char college_number[6];
  float average;
  struct STUDENT_TAG *p_next;
} StudentNode;
```

Linked lists - Head

- Linked lists grow and shrink as needed
- `malloc()` and `free()` to create/destroy nodes
- new nodes are attached to the chain
- declare a pointer to "hold" the *head* (start) of the linked list
- last node in the list points to NULL



Example – Create with 1 node

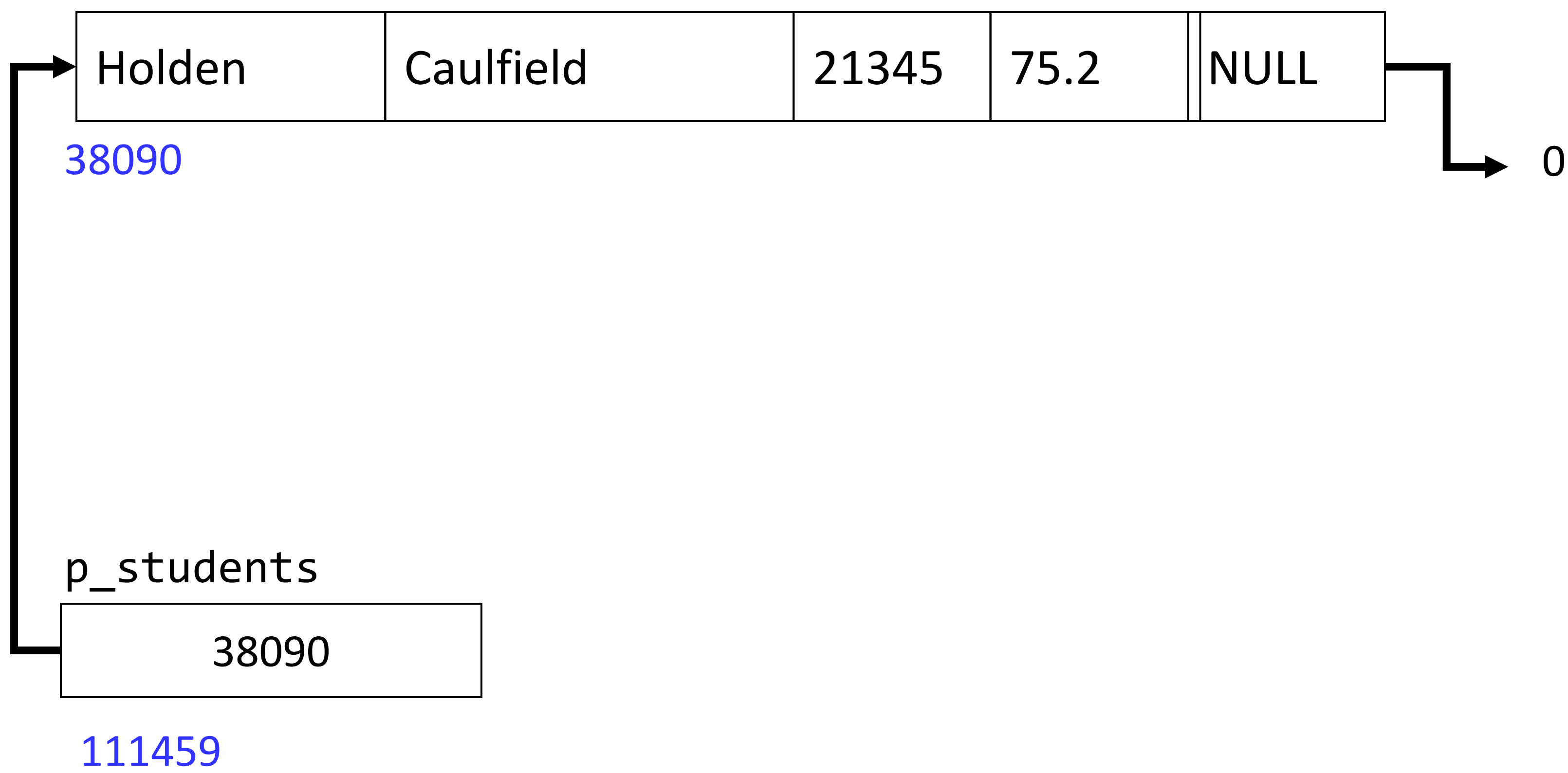
- The head of my linked list is a pointer of type `StudentNode`:

  ```
  StudentNode* p_students = NULL;
  ```

- add the first student in the list:

```
p_students =
   (StudentNode*)malloc(sizeof(StudentNode));
strcpy(p_students->first_name,"Holden");
strcpy(p_students->last_name,"Caulfield");
strcpy(p_students->college_number = "21345");
p_students->average = 74.2;

//first node is end of list, no next node
p_students->p_next = NULL;
```

Example – Create with 1 node

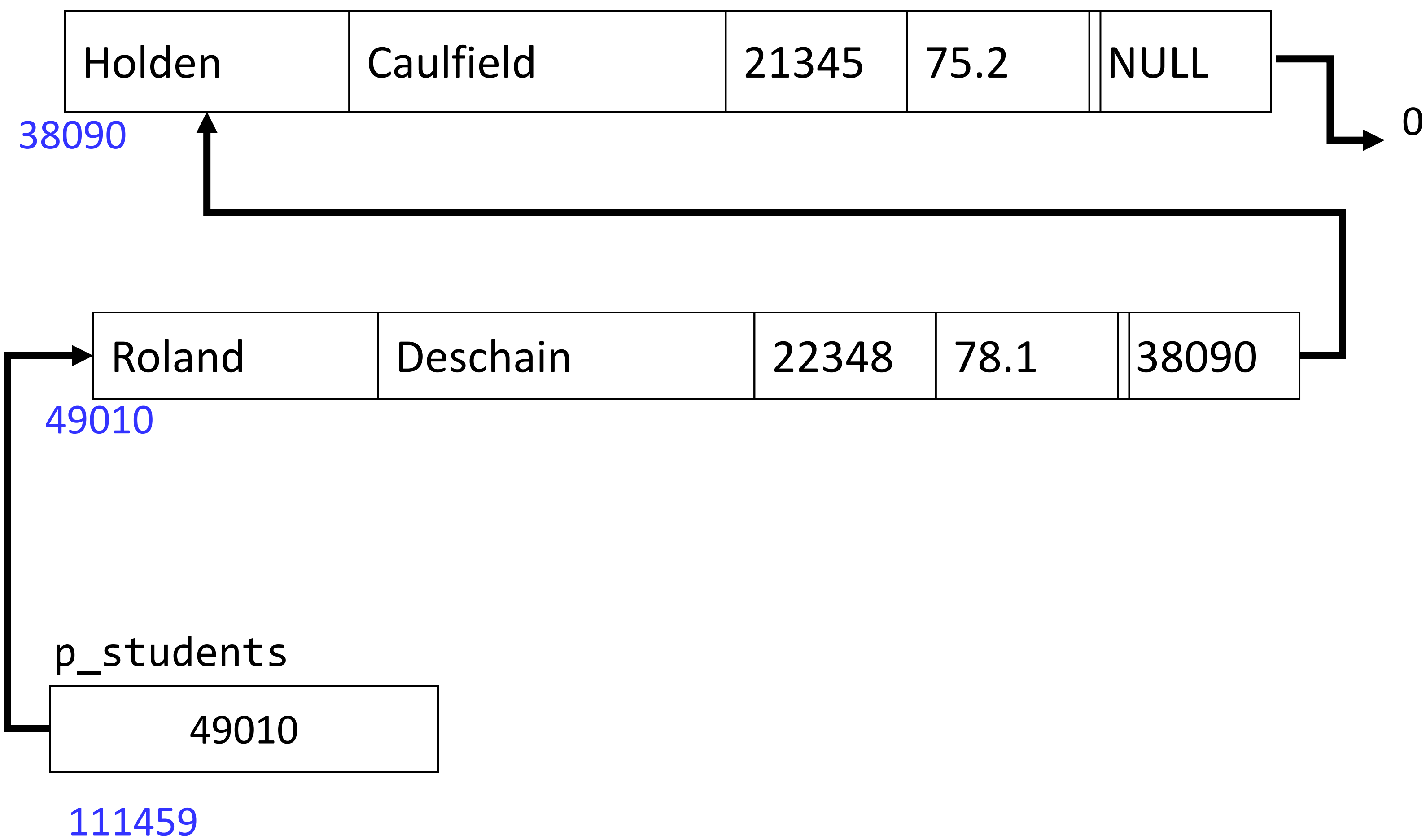


Example – add at beginning

- I could then add a second student in the list at the beginning (before Holden):

```
…
p_new = (StudentNode*)malloc(sizeof(StudentNode));
strcpy(p_new->first_name,"Roland");
strcpy(p_new->last_name,"Deschain");
strcpy(p_new->college_number, "22348");
p_new->average = 78.1;

p_new->p_next = p_students; // next node is old first
   node

p_students = p_new;  // head points to new first node
```

Example – Adding at beginning



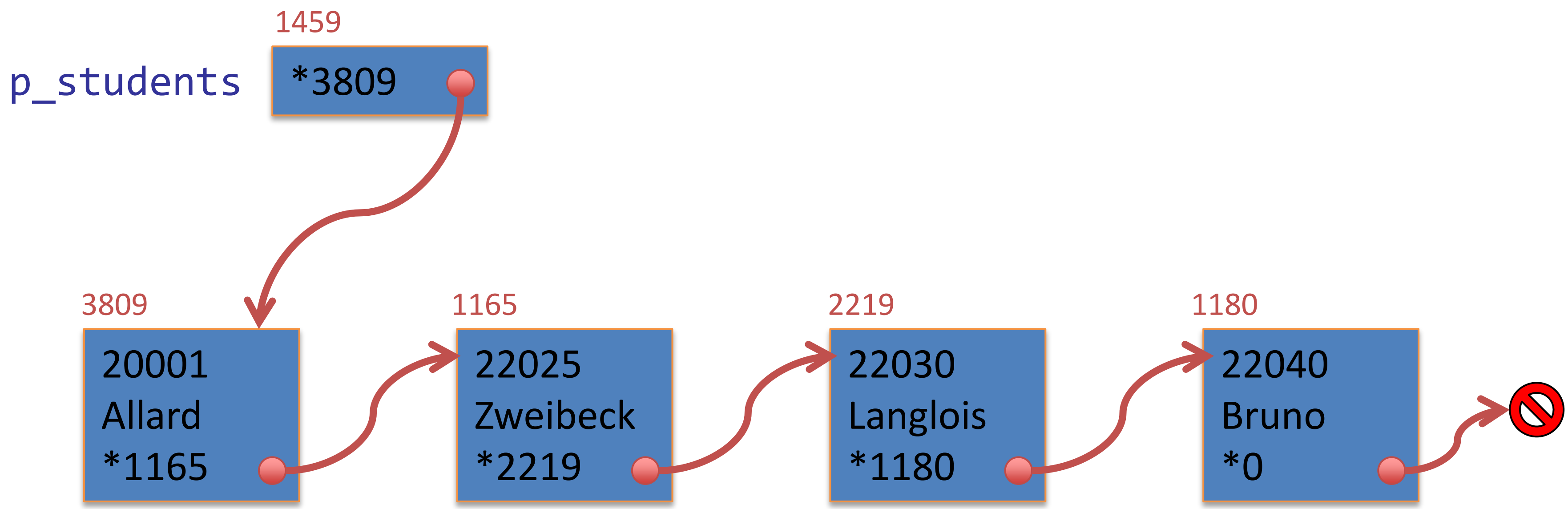
Traversing

- "traversing a linked list" means walking along the chain of links
- normally used to find a particular element in a list (linear search) based on some criteria
  - to display the element
  - to remove the element
  - to insert an element before or after
- also used to operate on all elements of a list (e.g., to count elements or to compute an overall average)

Traversing/Searching through a LL

```
typedef struct STUDENT_TAG {
  char first_name[15];
  char last_name[25];
  char college_number[6];
  float average;
  struct STUDENT_TAG *p_next;
} StudentNode;

StudentNode *p_students; // head
 pointer
```

Typical linked list structure



Traversing a linked list

Declare a pointer to the type-defined structure in your declaration section (say `p_walker`)

```
StudentNode *p_walker = NULL;
```

Set `p_walker` to the head of the list

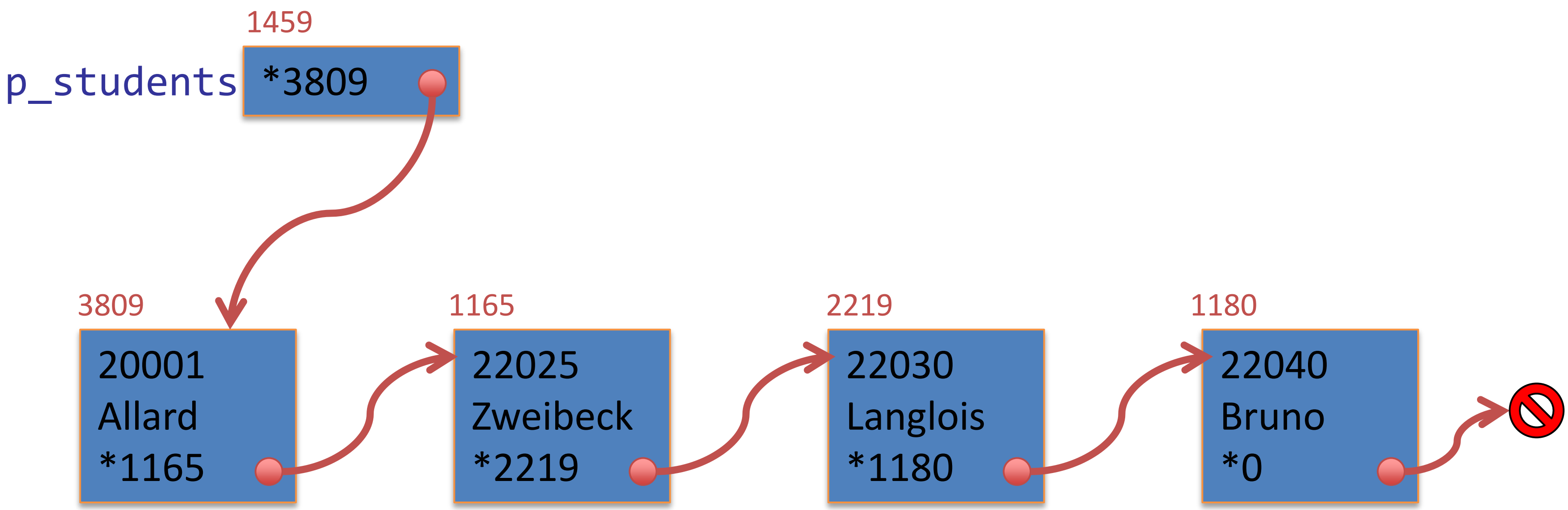
```
p_walker = p_students;
```

While `p_walker` isn't `NULL`

- Process the current node
- Set the pointer to the next node

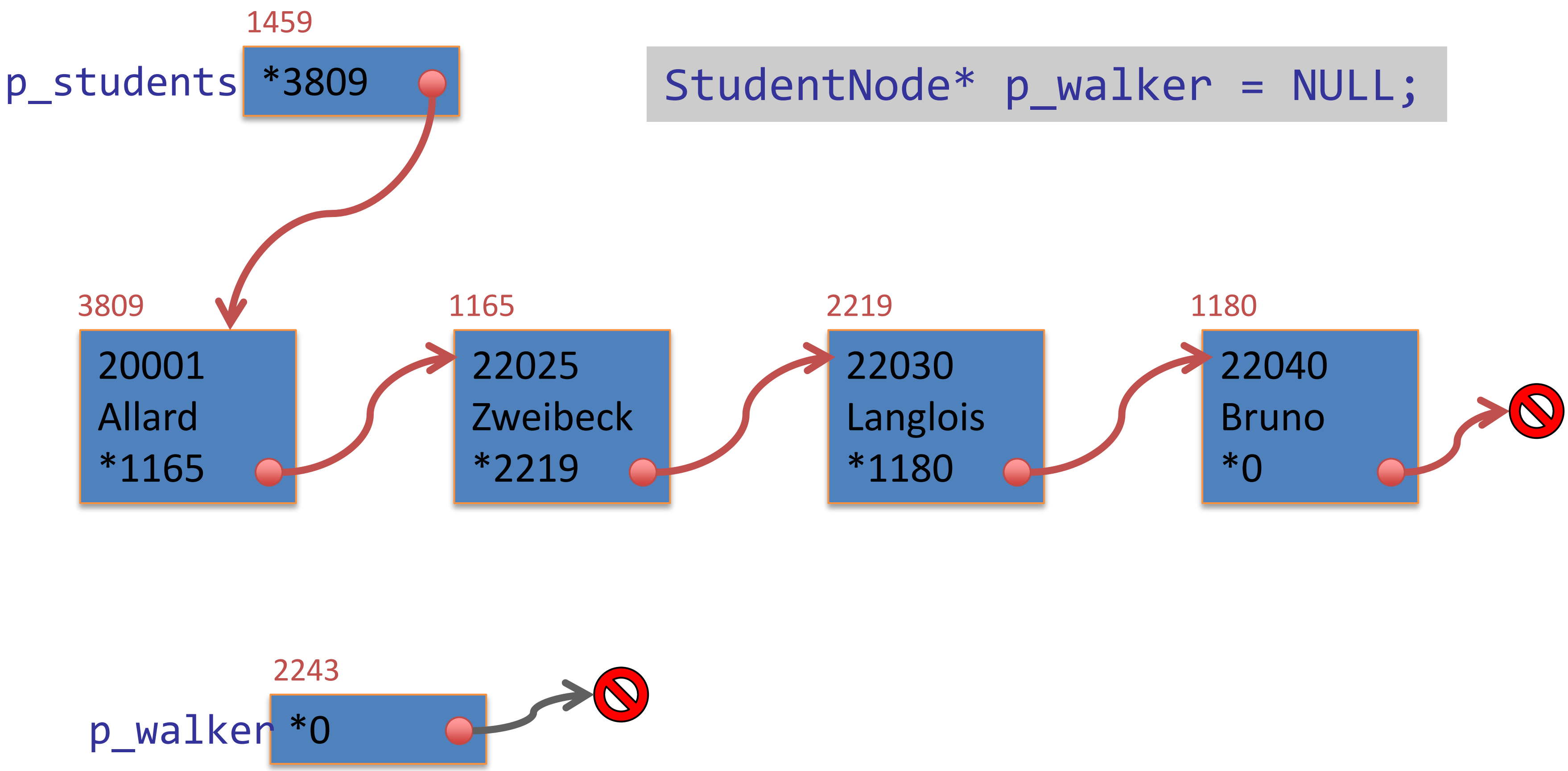
```
p_walker = p_walker->p_next;
```

Traversing a linked list



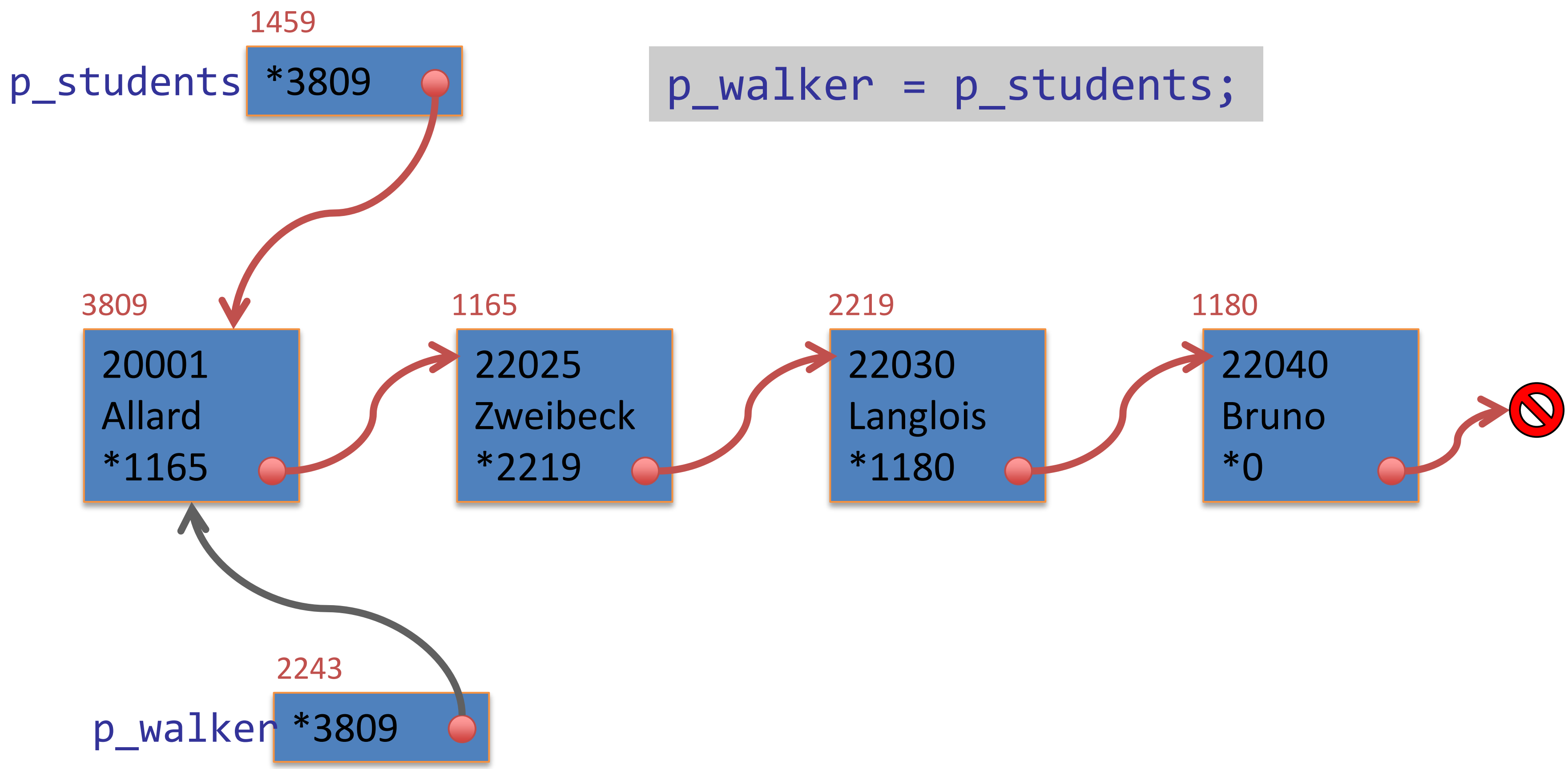
Traversing a linked list

```
StudentNode* p_walker = NULL;
```



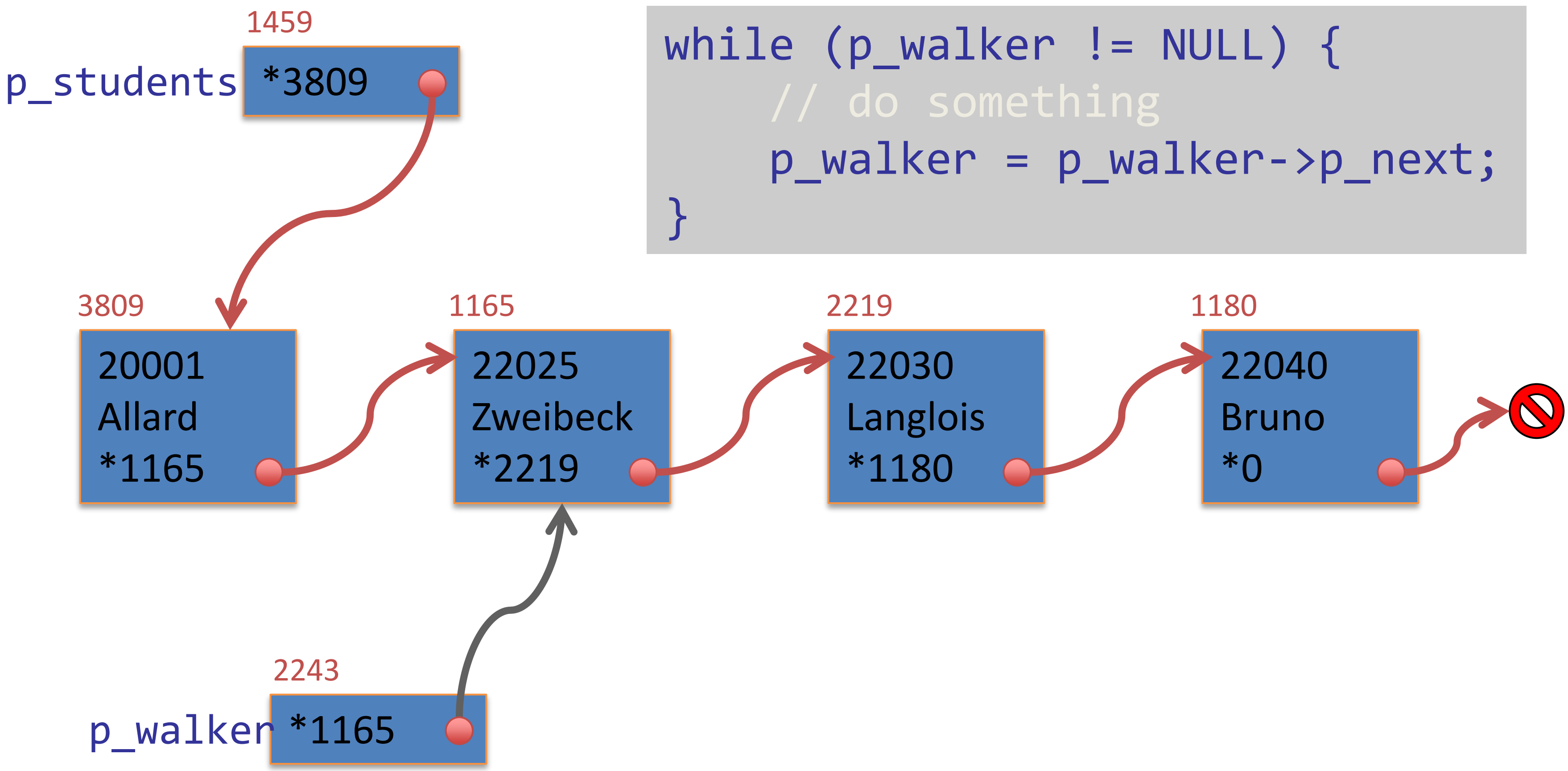
Traversing a linked list

```
p_walker = p_students;
```



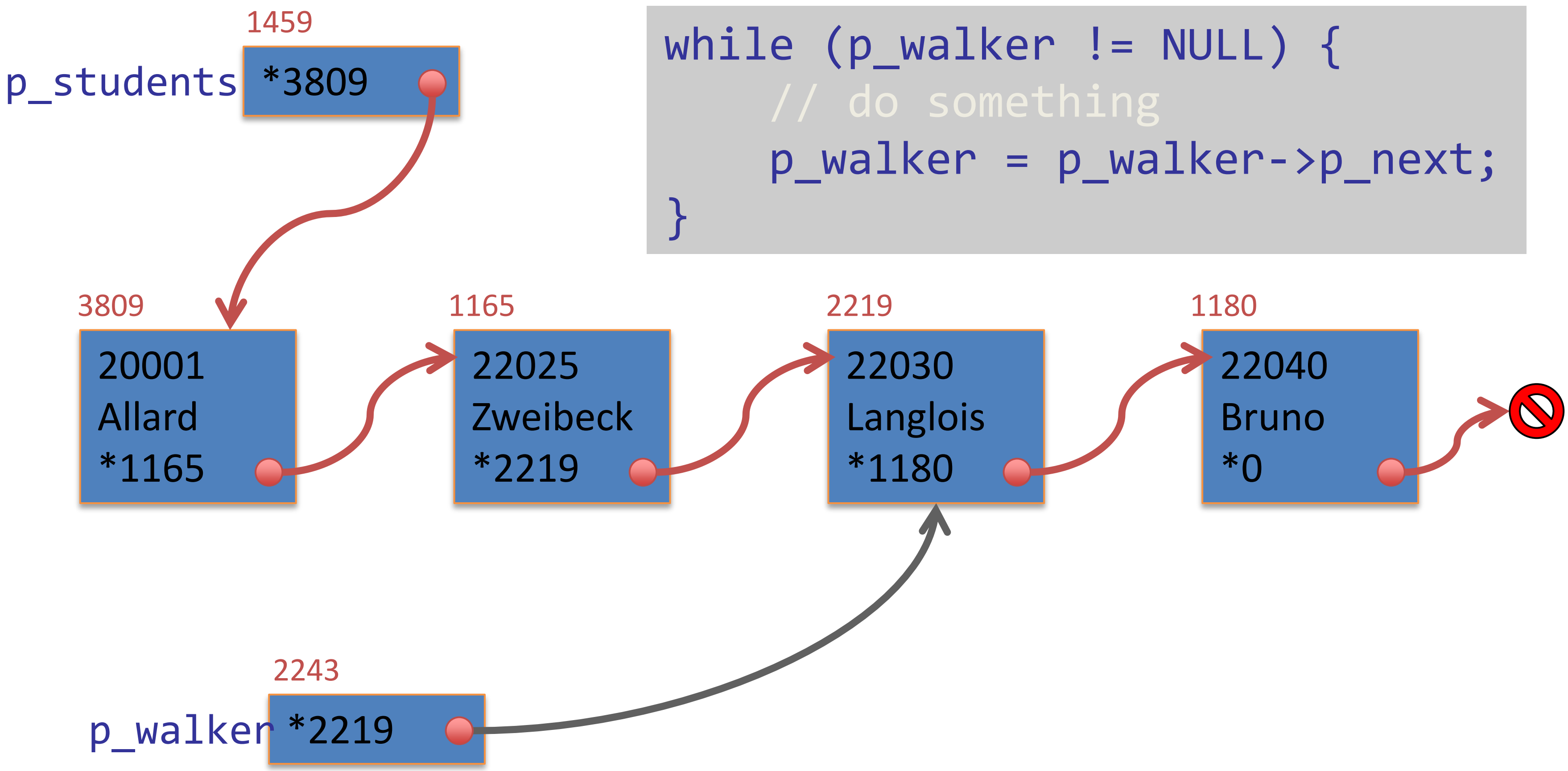
Traversing a linked list

```
while (p_walker != NULL) {
    // do something
    p_walker = p_walker->p_next;
}
```



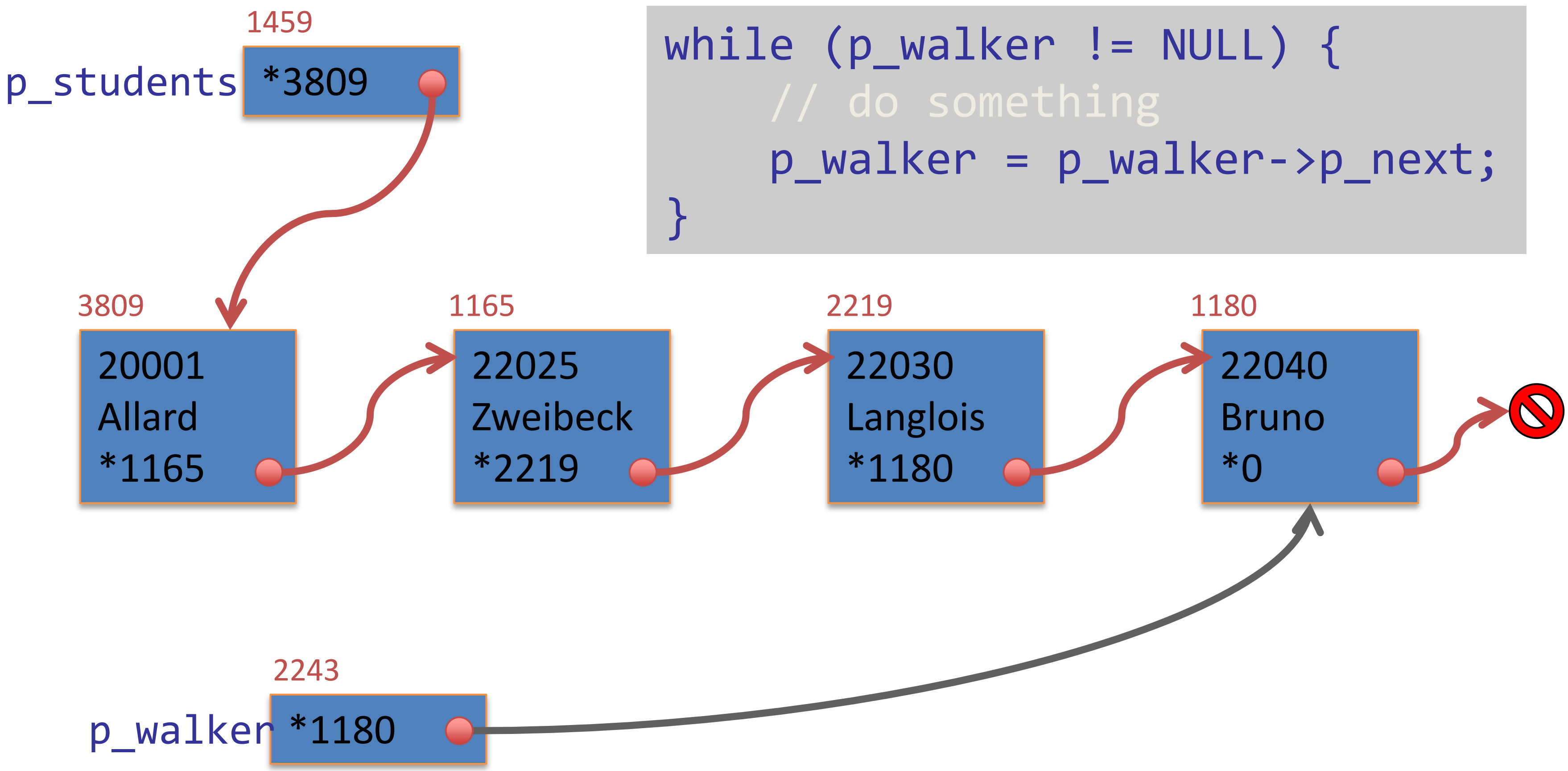
Traversing a linked list

```
while (p_walker != NULL) {
    // do something
    p_walker = p_walker->p_next;
}
```



Traversing a linked list

```
while (p_walker != NULL) {
    // do something
    p_walker = p_walker->p_next;
}
```



Traversing a linked list

```
// count number of elements in the list
int count = 0;
StudentNode* p_walker = p_students;

while (p_walker != NULL) {
  // process the current node
  count++;
  // move on to the next one
  p_walker = p_walker->p_next;
}
```

Inserting a node

- We saw how to insert the a node at the beginning of a linked list

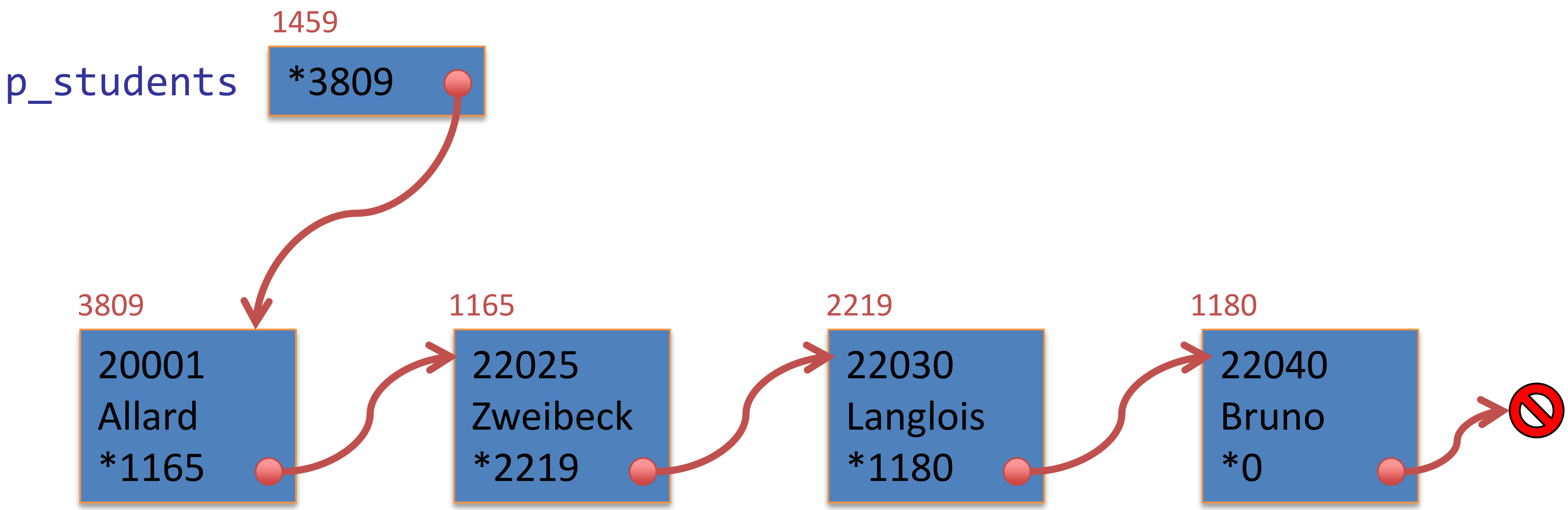
- if we want to insert a node at a particular location (e.g., alphabetical order) we must traverse the list to find the insertion point

Creating a node

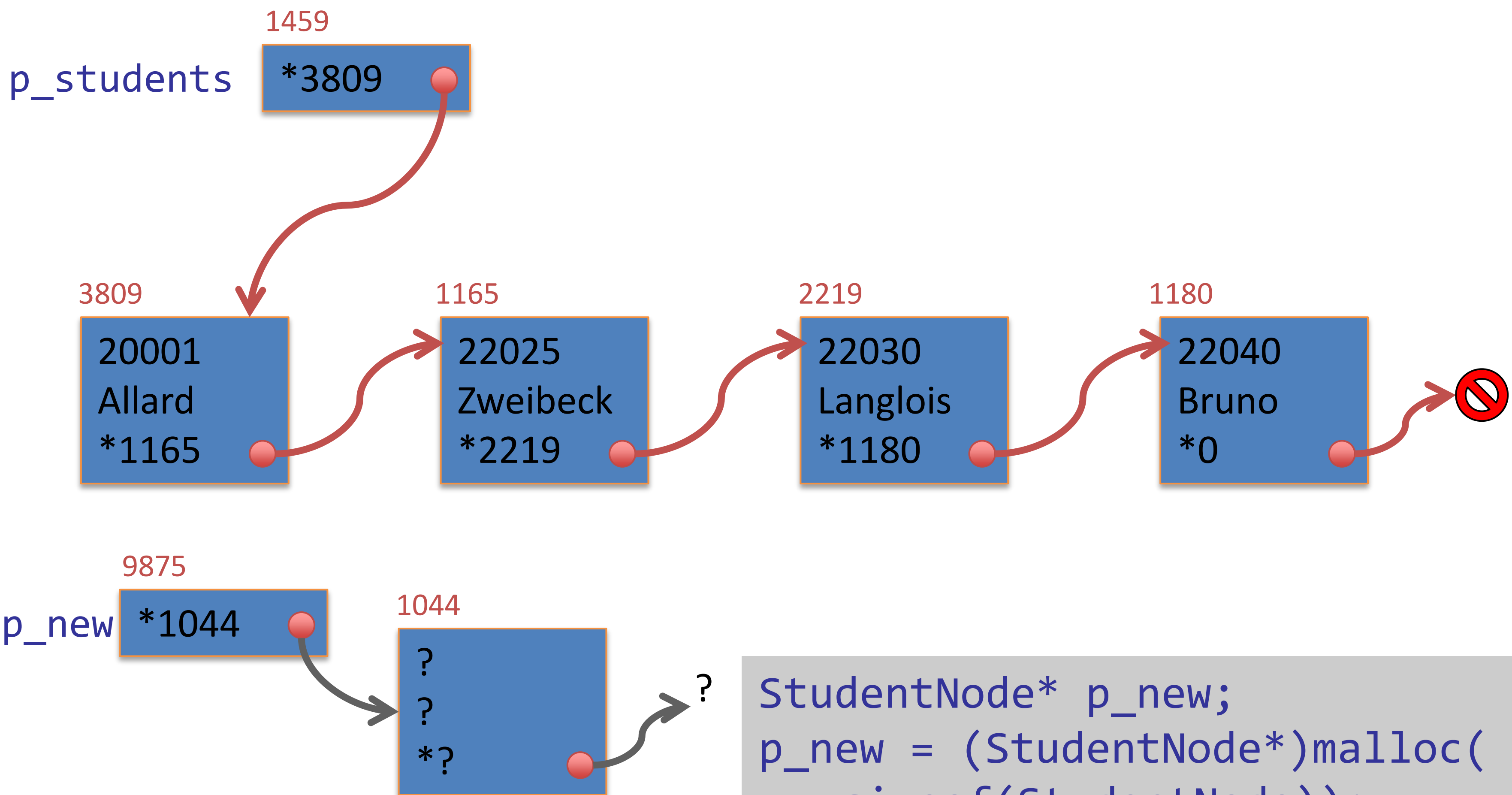
1. Declare a new node using a temporary variable (`p_temp`) and request memory through `malloc`
   – verify that there was enough memory available

2. Initialize all the fields of the new node with appropriate values

Create new node

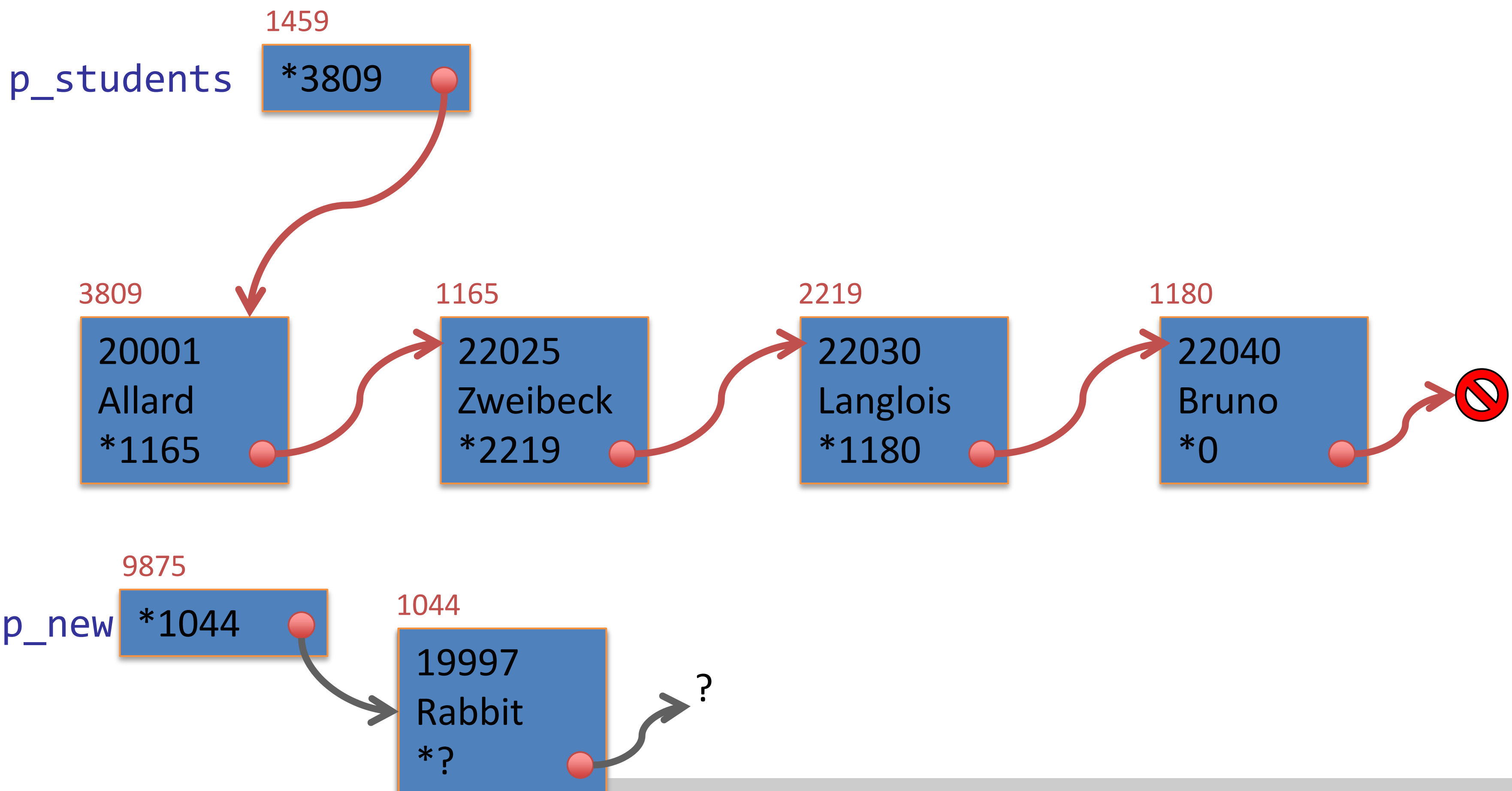


Create new node




```
StudentNode* p_new;
p_new = (StudentNode*)malloc(
    sizeof(StudentNode));
if (p_new == NULL){
    exit(EXIT_FAILURE);
}
```

Initialize values




```
strcpy(p_new->first_name, "Jack");
strcpy(p_new->last_name, "Rabbit");
strcpy(p_new->college_number, "19997");
p_new->average = 99.9;
```

Special case: insert at the start

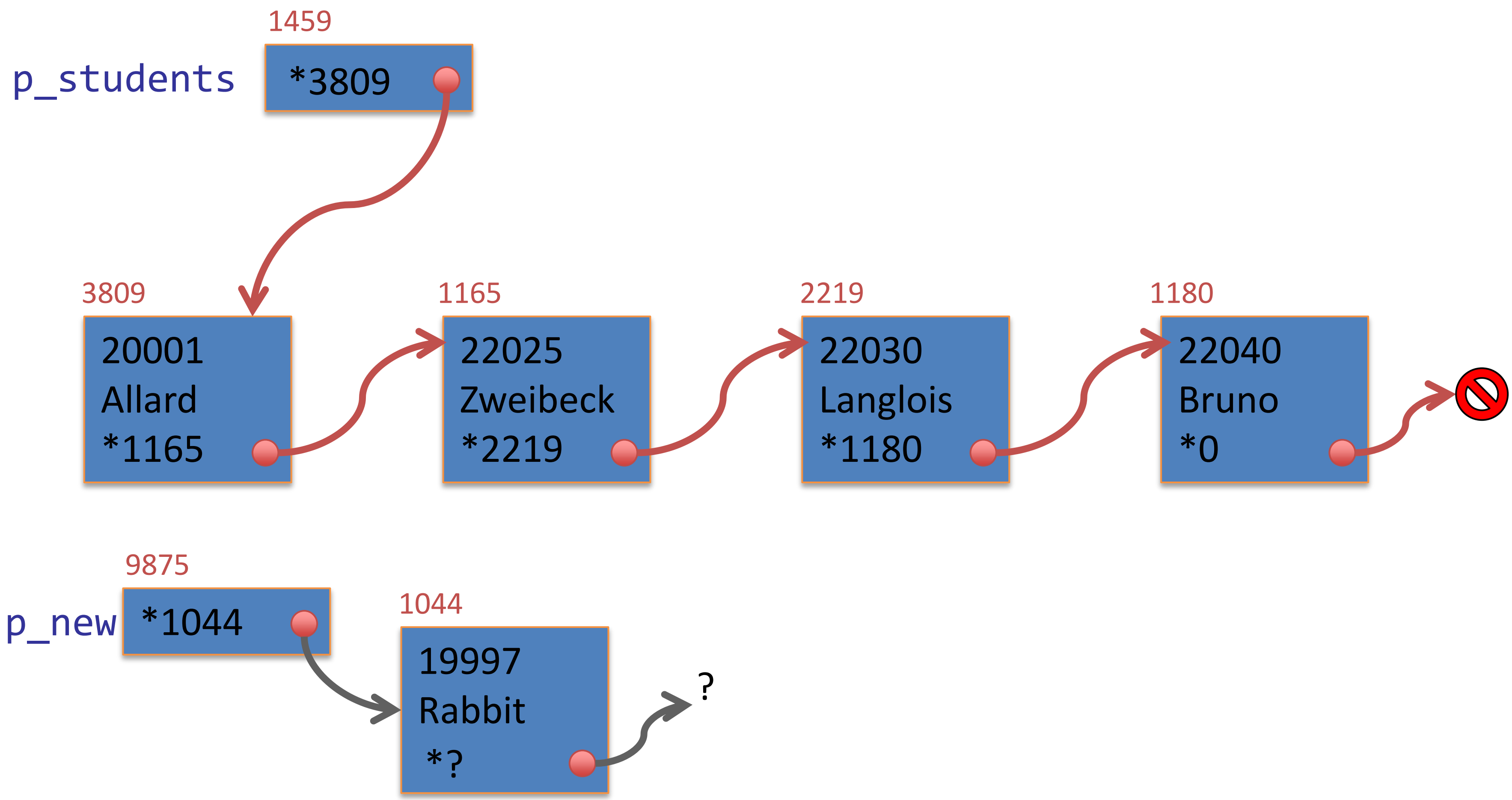
1. Set the new node's next pointer to the old first node:

```
p_new->p_next = p_students;
```

1. Set the head pointer to the new node:

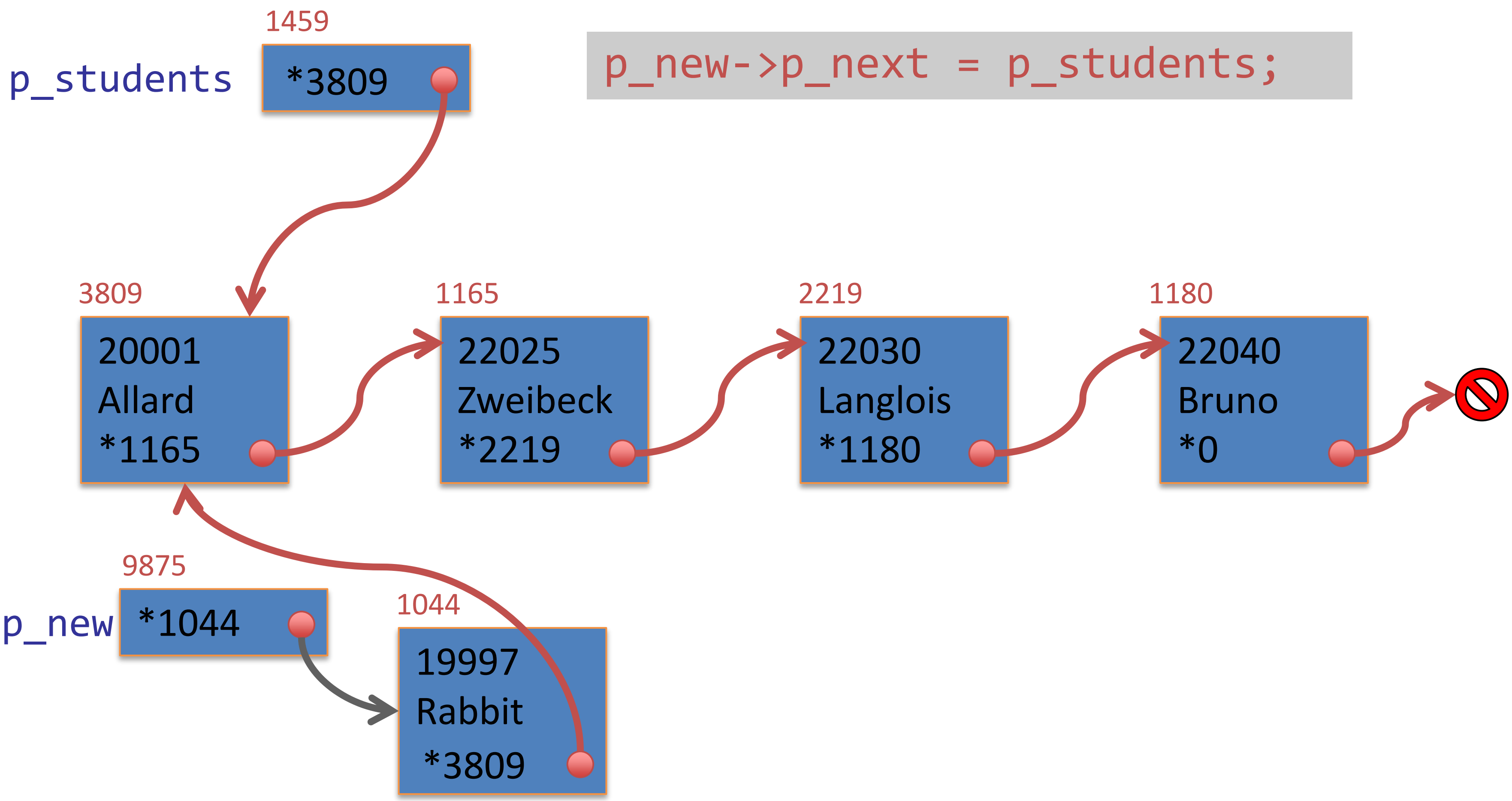
```
p_students = p_new;
```

Insert at beginning of list



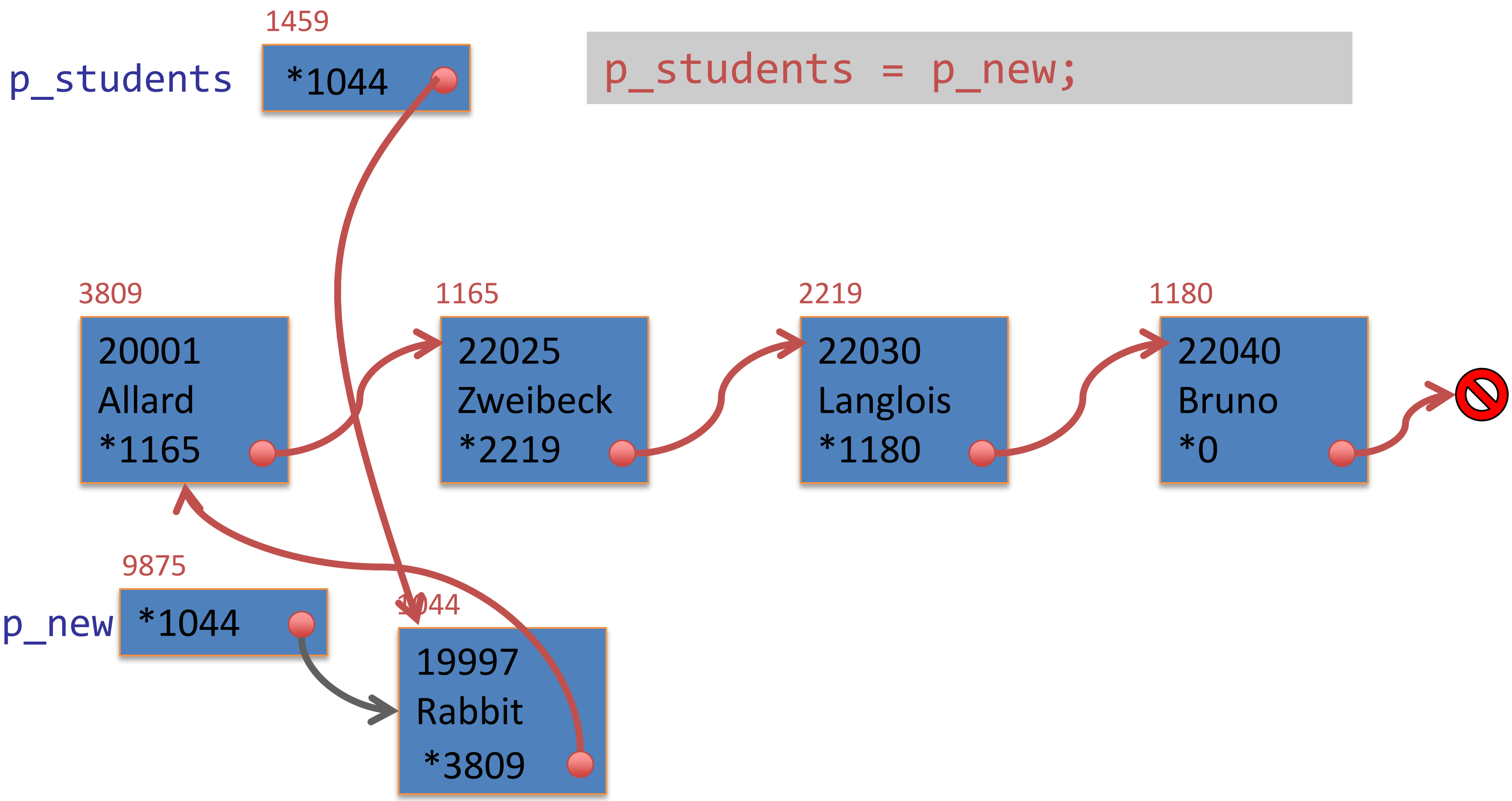
Insert at beginning of list

```
p_new->p_next = p_students;
```



Insert at beginning of list

```
p_students = p_new;
```



Create and insert a node at start of LL

```c
//1. Create new node and check for available memory
StudentNode* p_new= NULL;
p_new= (StudentNode*)malloc(sizeof(StudentNode));
if (p_new== NULL){
   exit(EXIT_FAILURE);
}
//2. Initialize new node
strcpy(p_new->first_name, "Jack");
strcpy(p_new->last_name, "Rabbit");
strcpy(p_new->college_number, "19997");
p_temp->average = 99.9;

//3a. Insert node at head of list
p_new->p_next = p_students;
p_students = p_new;
```

Inserting a node elsewhere in LL

1. Traverse the list using `p_walker` to find the insertion point.
   – here, assume we want to insert in college_number order, ascending
2. Make the new node point where `p_walker` points

```
p_new->p_next = p_walker->p_next;
```

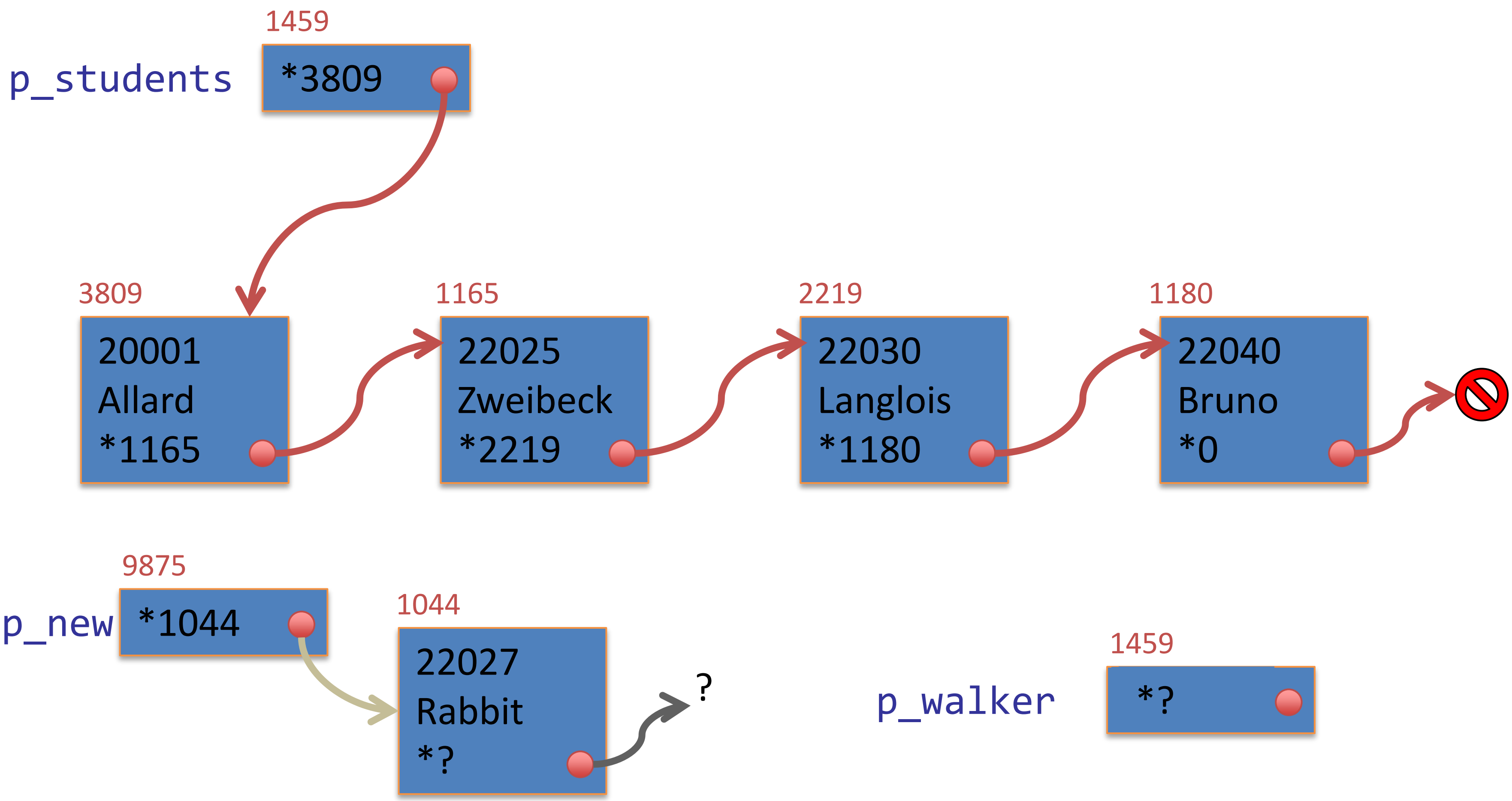
3. Make `p_walker` point to the new node

```
p_walker->p_next = p_new;
```

Aside: a goes_later function

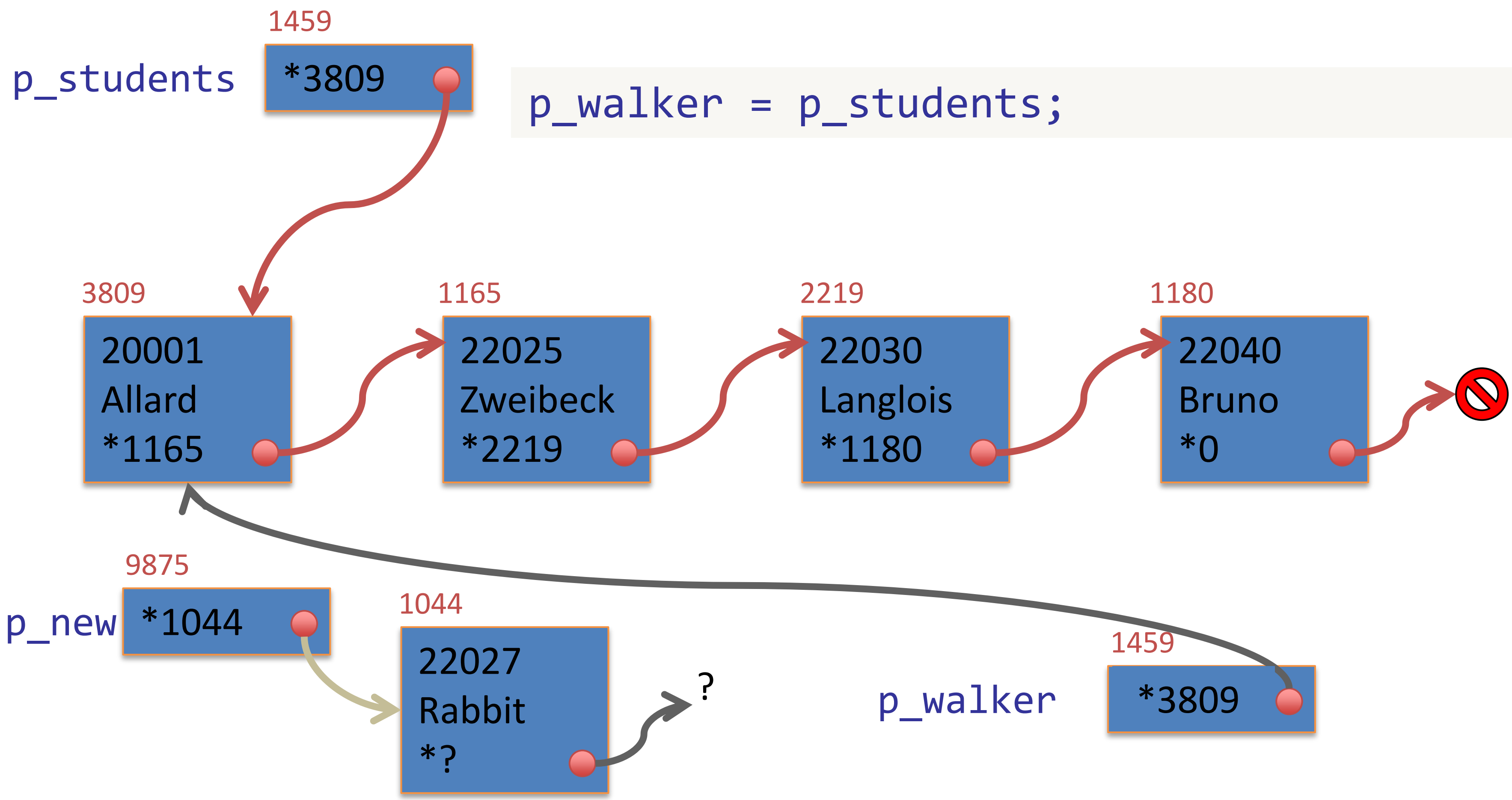
```
// to find the insertion point could do this: UGLY
while (p_walker->p_next != NULL &&
       strcmp(p_walker->p_next->college_number,
       p_new->college_number) <= 0) {
   p_walker = p_walker->p_next;
}

// meaning would be clearer if we could just do this
while (goes_later(p_walker, p_temp)) {
   p_walker = p_walker->p_next;
}

// so define a function…
bool goes_later(StudentNode* s1, StudentNode* s2) {
    return s1->p_next != NULL &&
        strcmp(s1->p_next->college_number,
        s2->college_number) <= 0;
}
```

Traverse to find insertion point

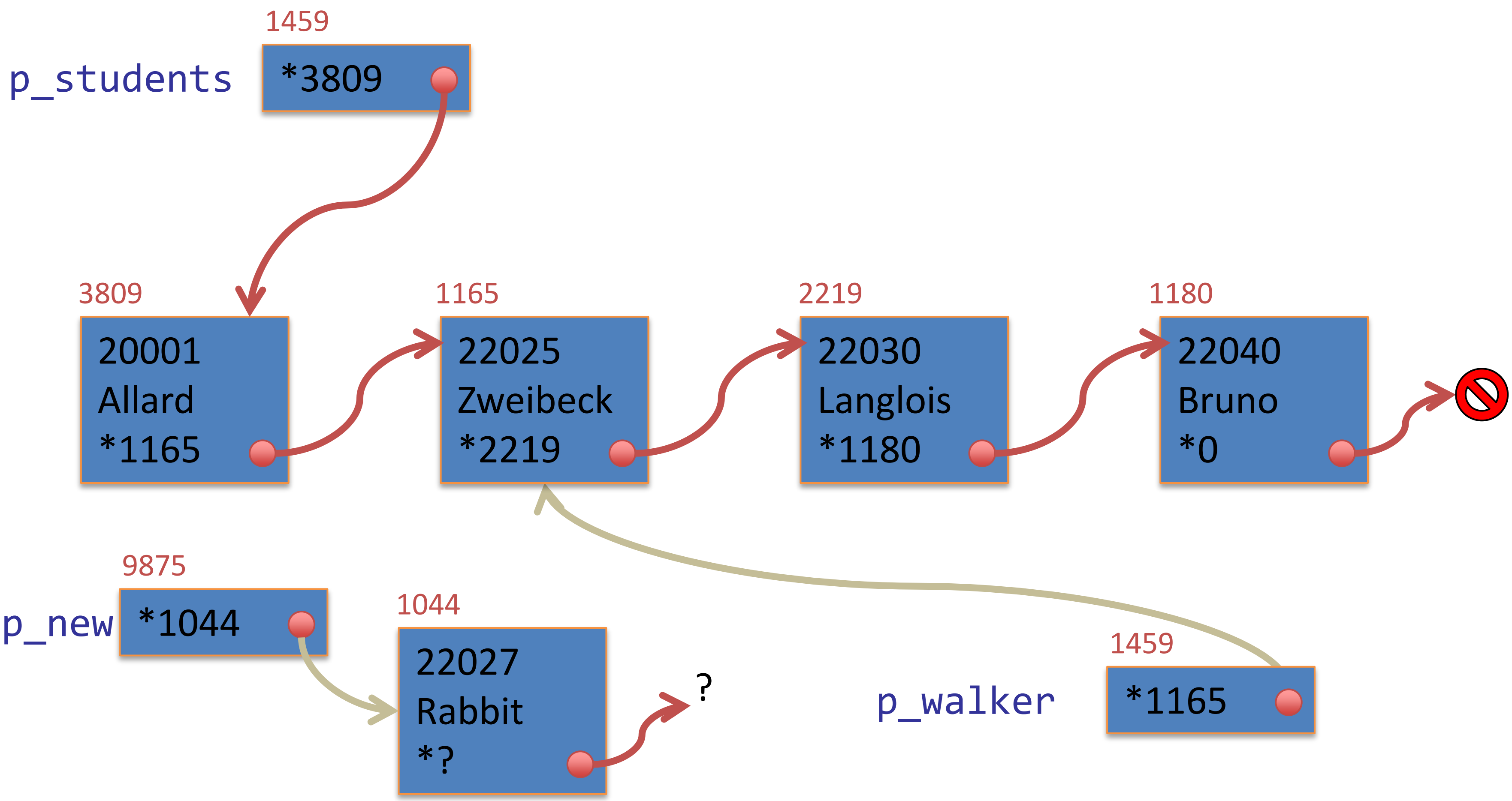


Traverse to find insertion point

```
p_walker = p_students;
```

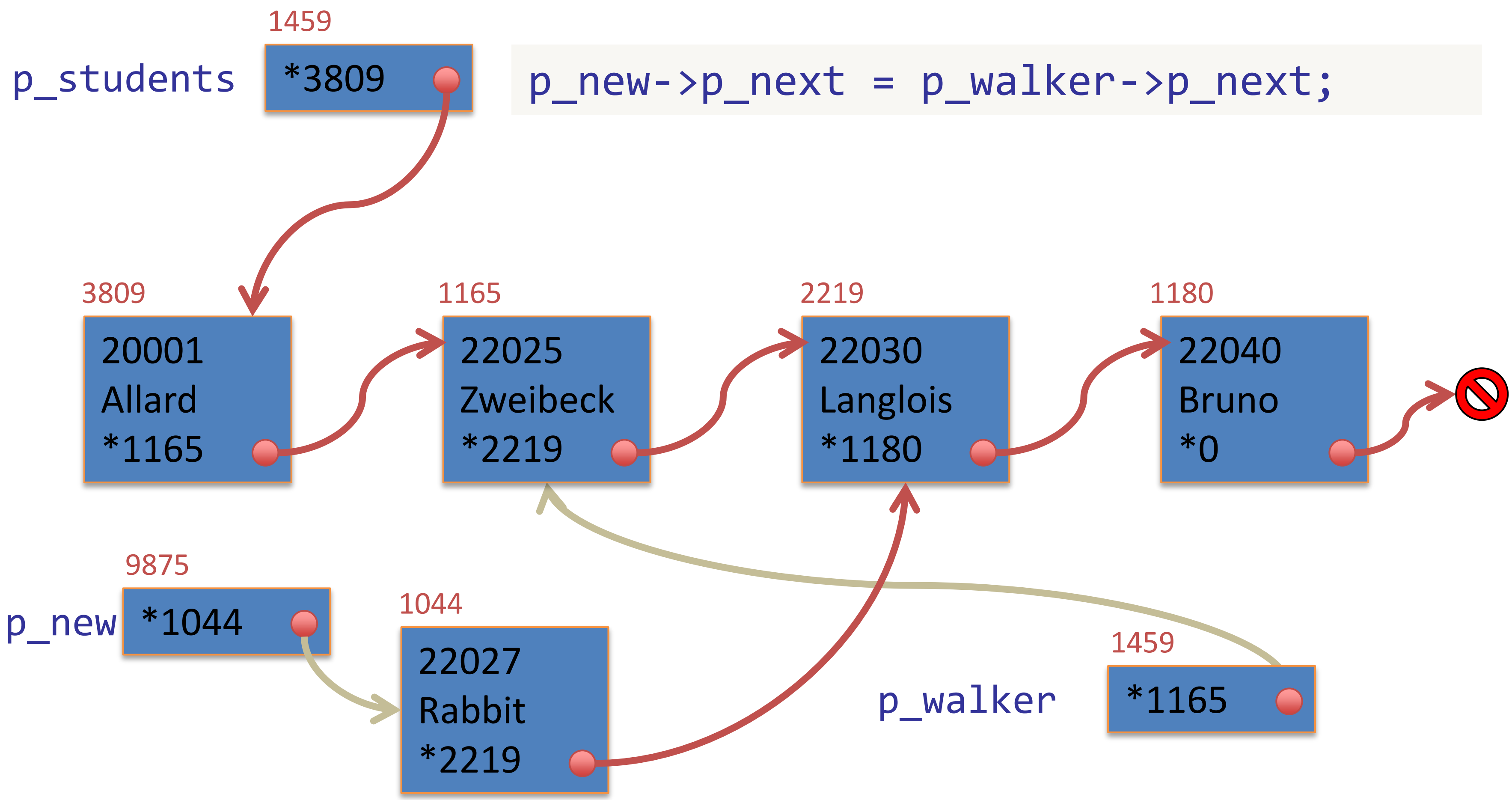


Insert node into list



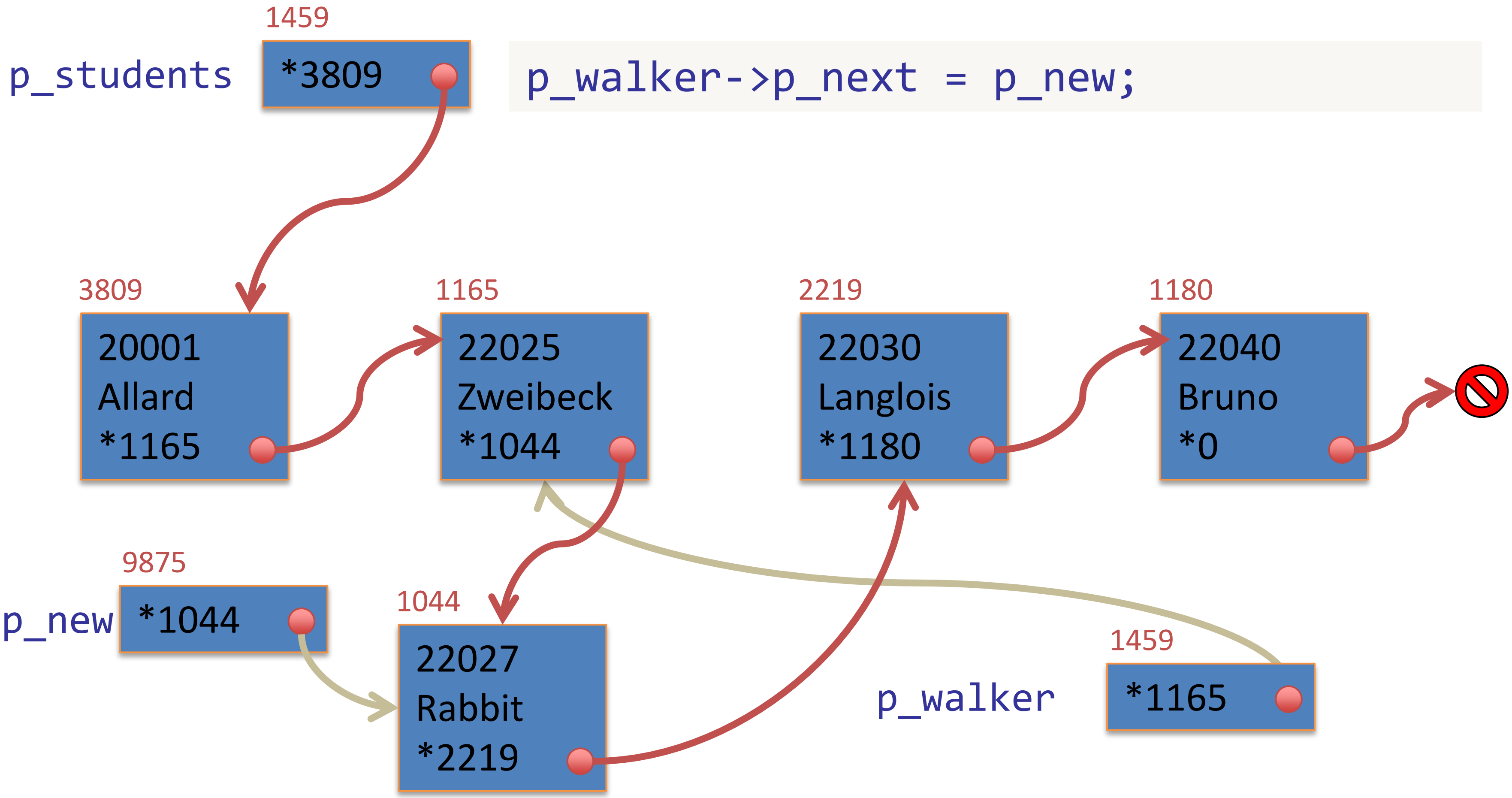
Insert node into list

```
p_new->p_next = p_walker->p_next;
```



Insert node into list

```
p_walker->p_next = p_new;
```



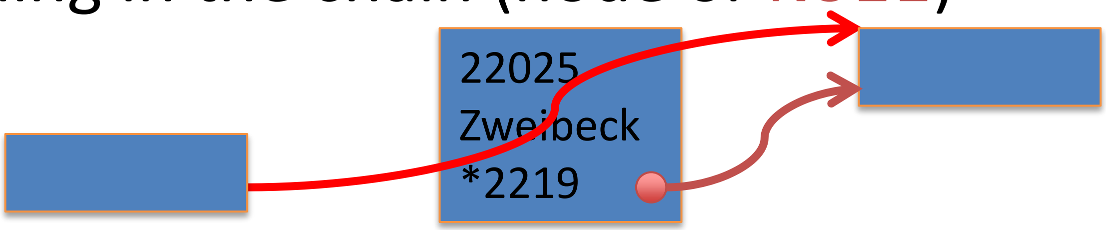
Inserting a node somewhere in LL

```
void insert_by_college_number(StudentNode* p_new) {
    StudentNode* p_walker;

    if (p_students == NULL) { //special case: empty list
        p_students = p_new;
        return;
    }
    // special case: new node goes at beginning
    if (!goes_later(p_students, p_new)){
        p_new->p_next = p_students;
        p_students = p_new;
        return;
    }
    // 3. general case: new node goes later
    p_walker = p_students;
    while (goes_later(p_walker, p_new)) {
        p_walker = p_walker->p_next;
    }
    p_new->p_next = p_walker->p_next;
    p_walker->p_next = p_new;
    return;
}
```

Deleting a node in a LL

- general idea: to delete a node…

  - set the pointer that points to it, to point at the next thing in the chain (node or `NULL`)



  - then `free` the node

Deleting a node at start of LL

1. Declare a pointer `p_walker` and point it at the first node in the list

   ```
   StudentNode* p_walker = p_students;
   ```

2. Check if the list is empty, if so print an error or do nothing….
3. Delete first node by setting the head pointer to the next node in the list

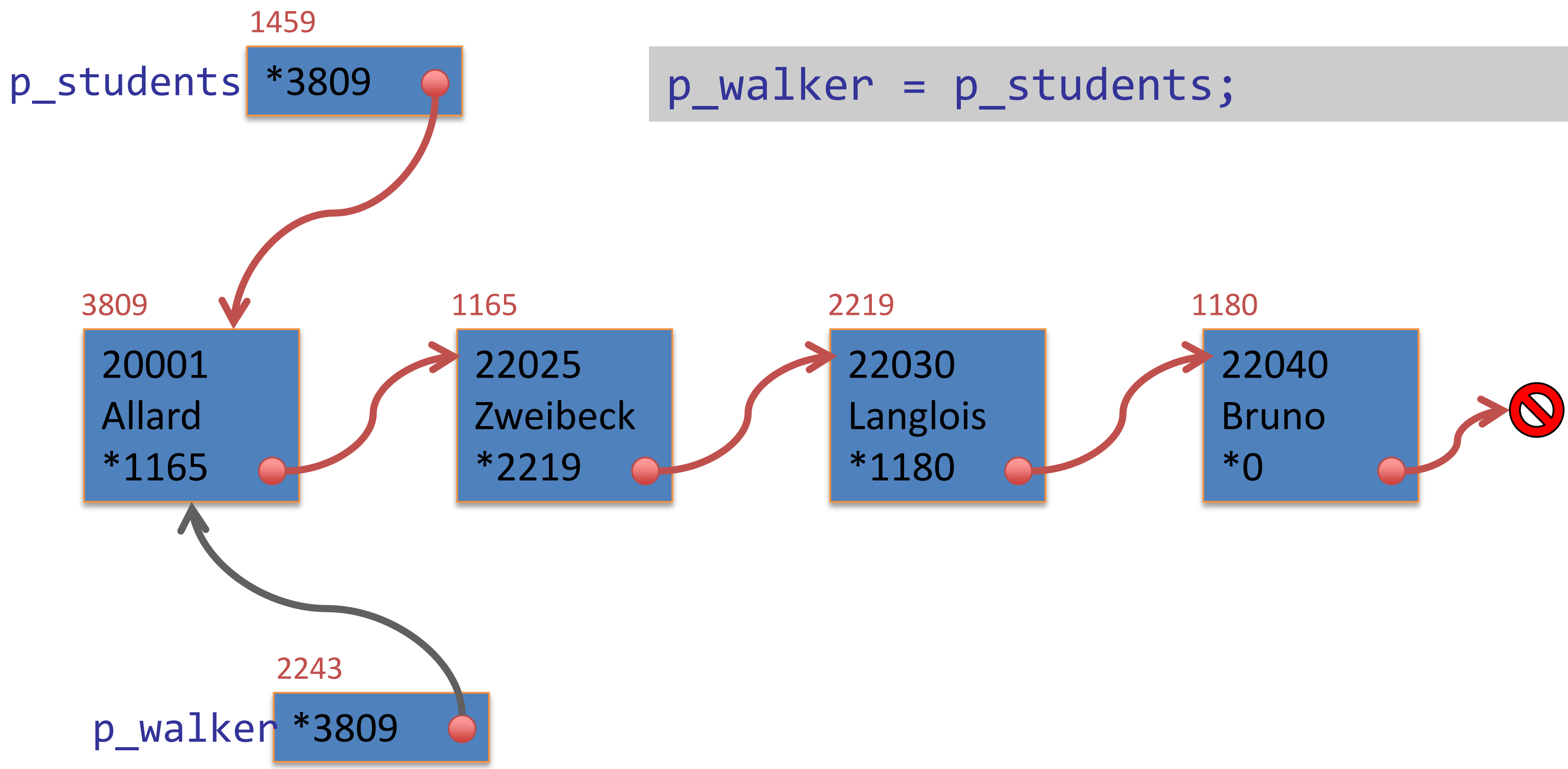
   ```
   p_students = p_students->p_next;
   ```

4. Free the memory for the deleted node

   ```
   free(p_walker);
   ```

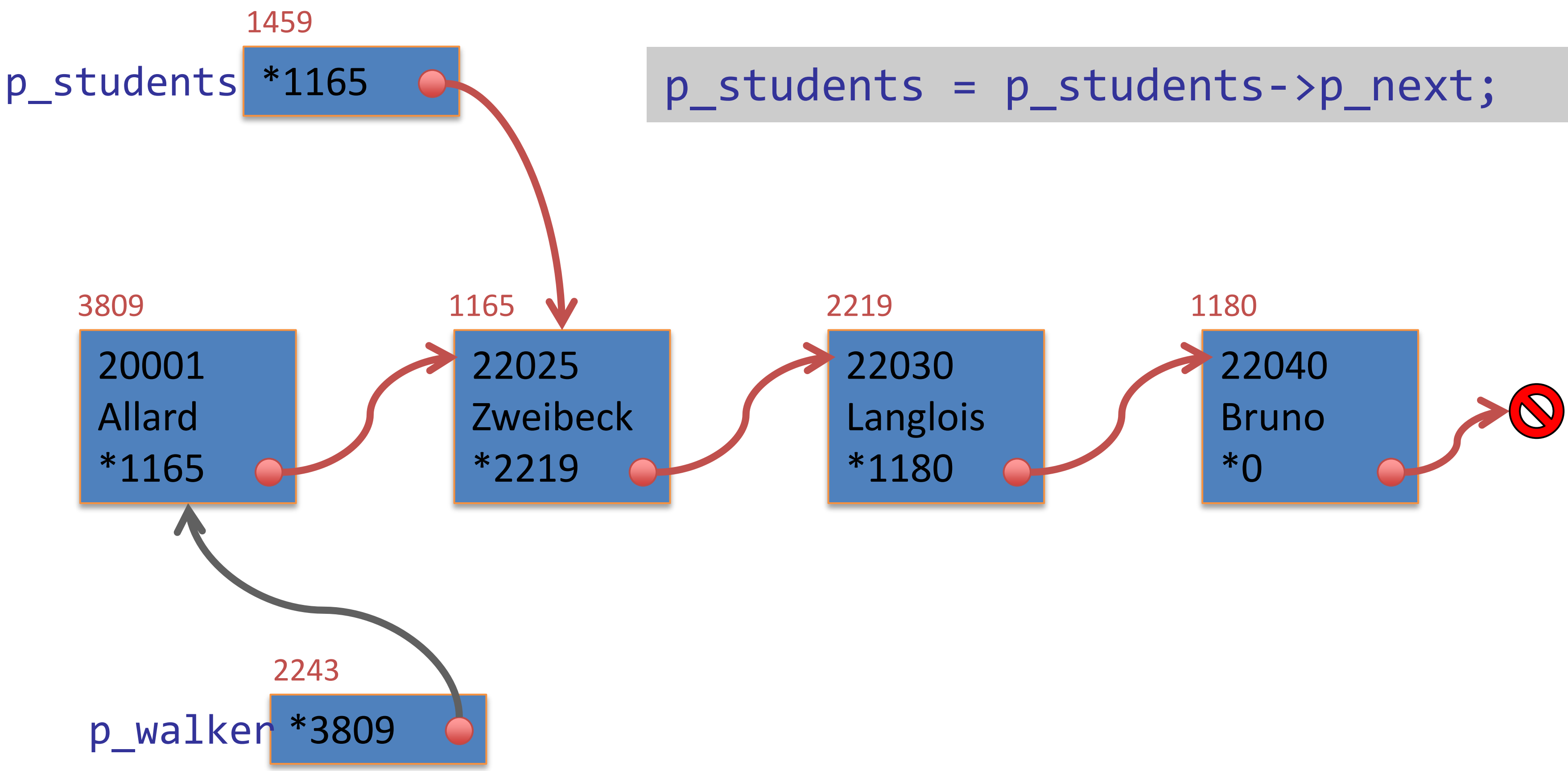
Deleting a node at the start

```
p_walker = p_students;
```

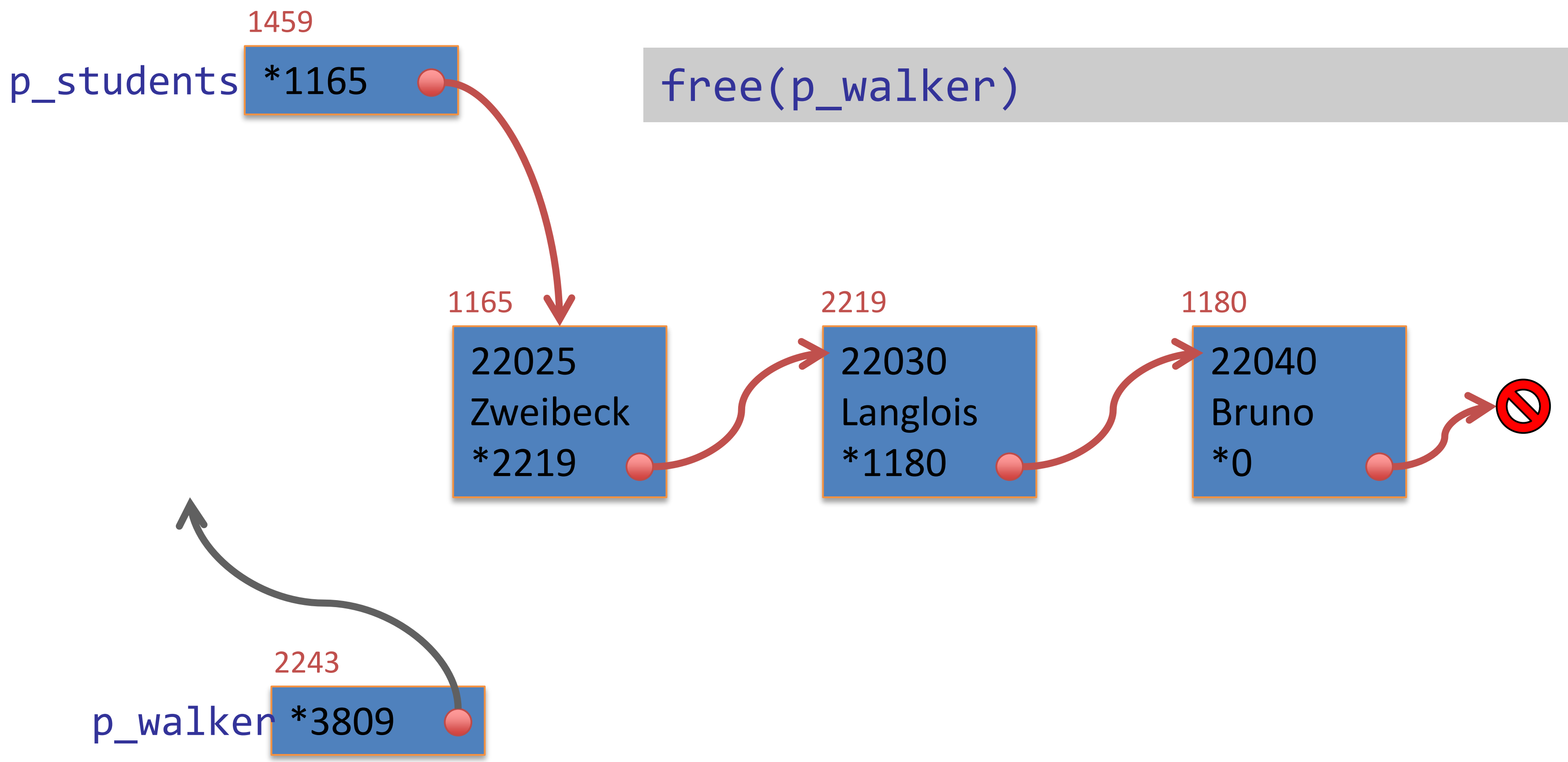


Deleting a node at the start

```
p_students = p_students->p_next;
```



Deleting a node at the start



Deleting a node at start of LL

```
StudentNode *p_walker = p_students;

if (p_students == NULL){
  printf("Can't delete from empty list!");
  return;
}

p_students = p_students->next;

free(p_walker);
```

Deleting a node at some unknown point in the linked list…

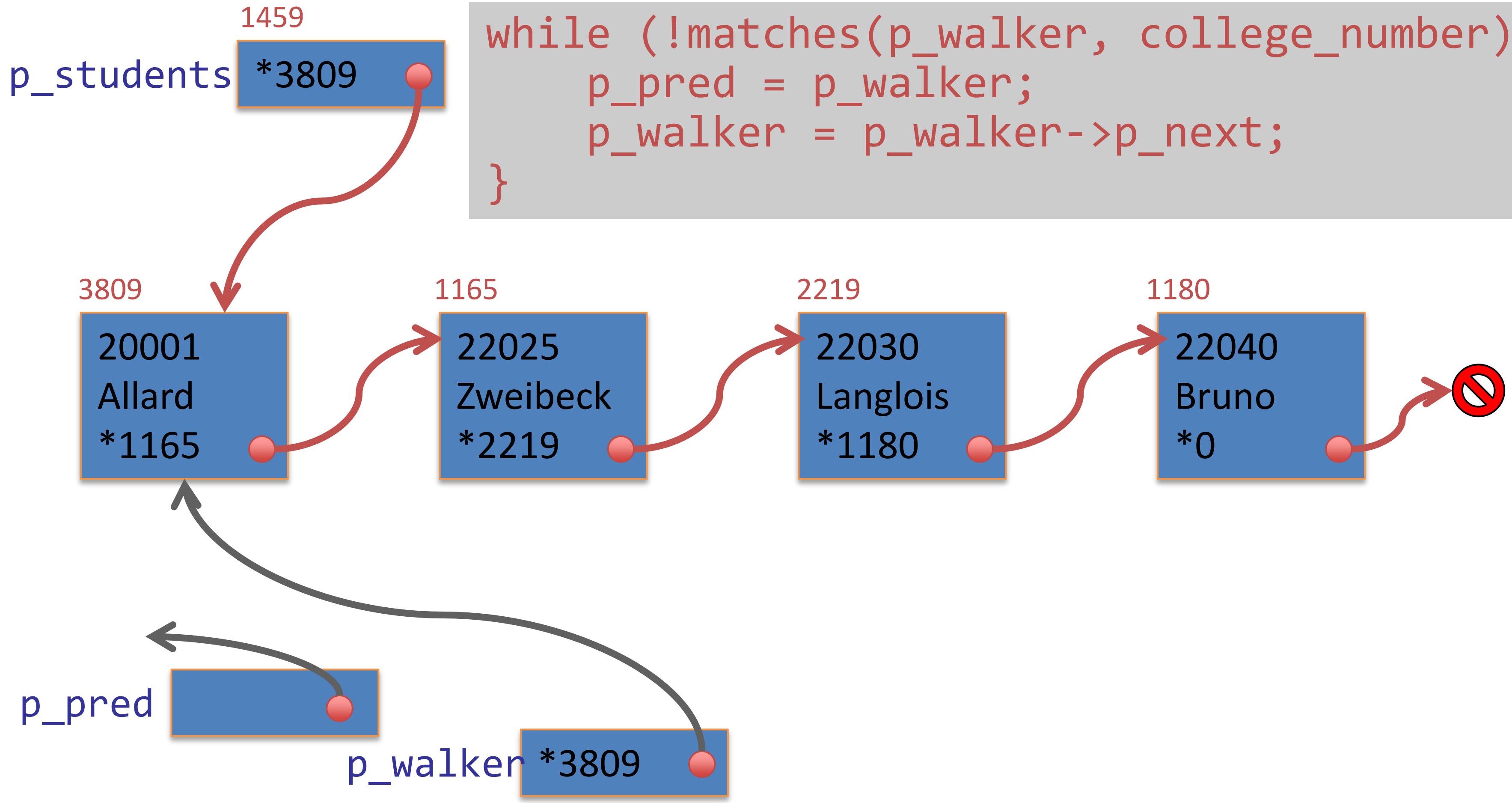
1. Declare two pointers `p_walker` and `p_pred`
2. Check if the node you want to delete is the first one, and if so, deal with as on previous slides.
3. If not, advance `p_walker` to next node, and keep `p_pred` one node behind it.
4. When `p_walker` points to the node you want to delete,
   1. point `p_pred->p_next` at the next node in the chain
   2. free `p_walker`

Aside: A matches function

```
// to find a match, could do this… UGLY
while (p_walker != NULL && strcmp(
   p_walker>college_number, college_number)) { …

//would look nicer like this
while (!matches(p_walker, college_number)) { …

// so define a function
// note the use of DeMorgan's law...
bool matches(StudentNode* node, char* college_number) {
    return (p_walker == NULL ||
        !strcmp(p_walker->college_number, college_number)
}
```

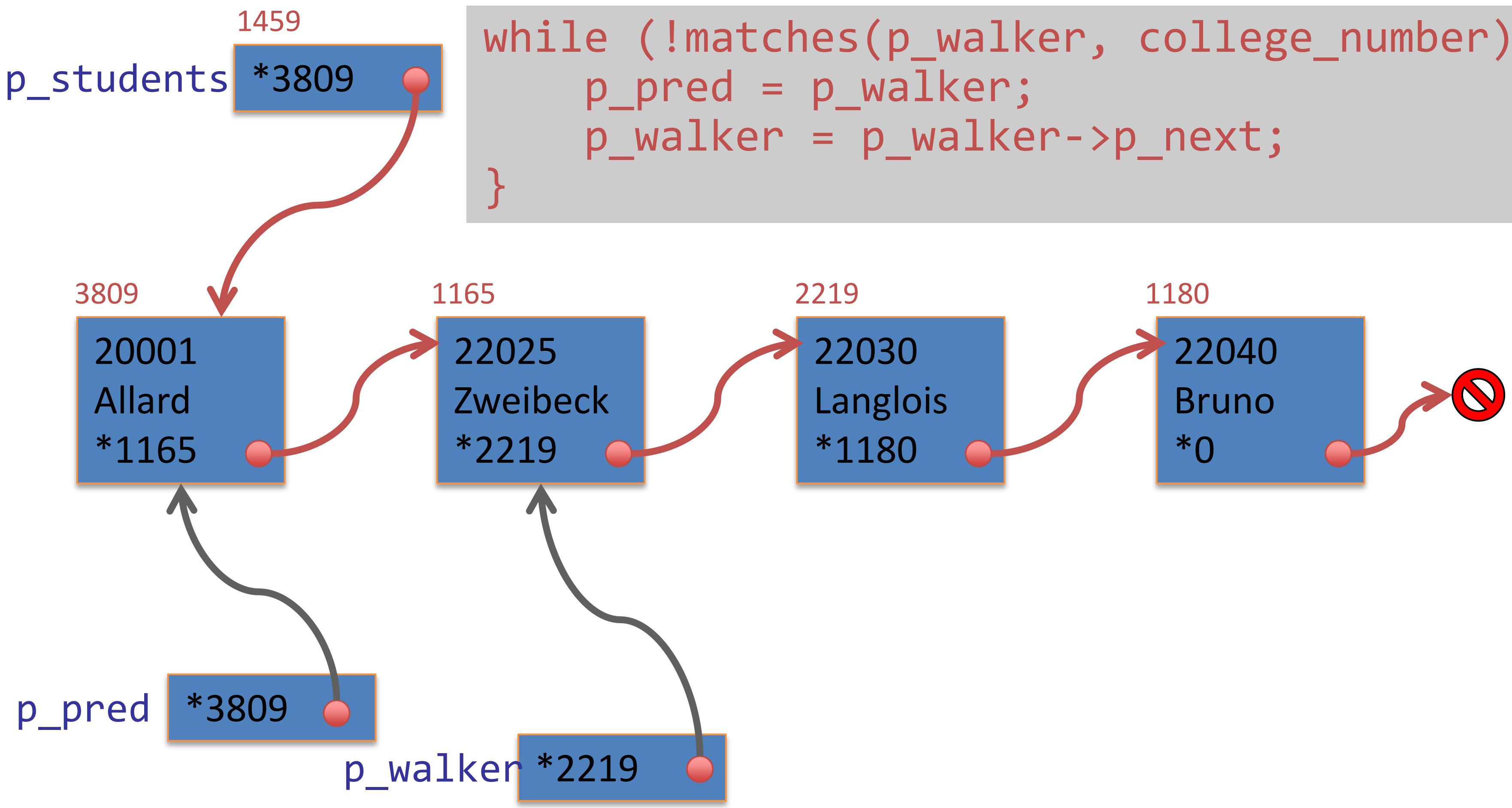
Deleting somewhere in the list

```
while (!matches(p_walker, college_number)){
    p_pred = p_walker;
    p_walker = p_walker->p_next;
}
```



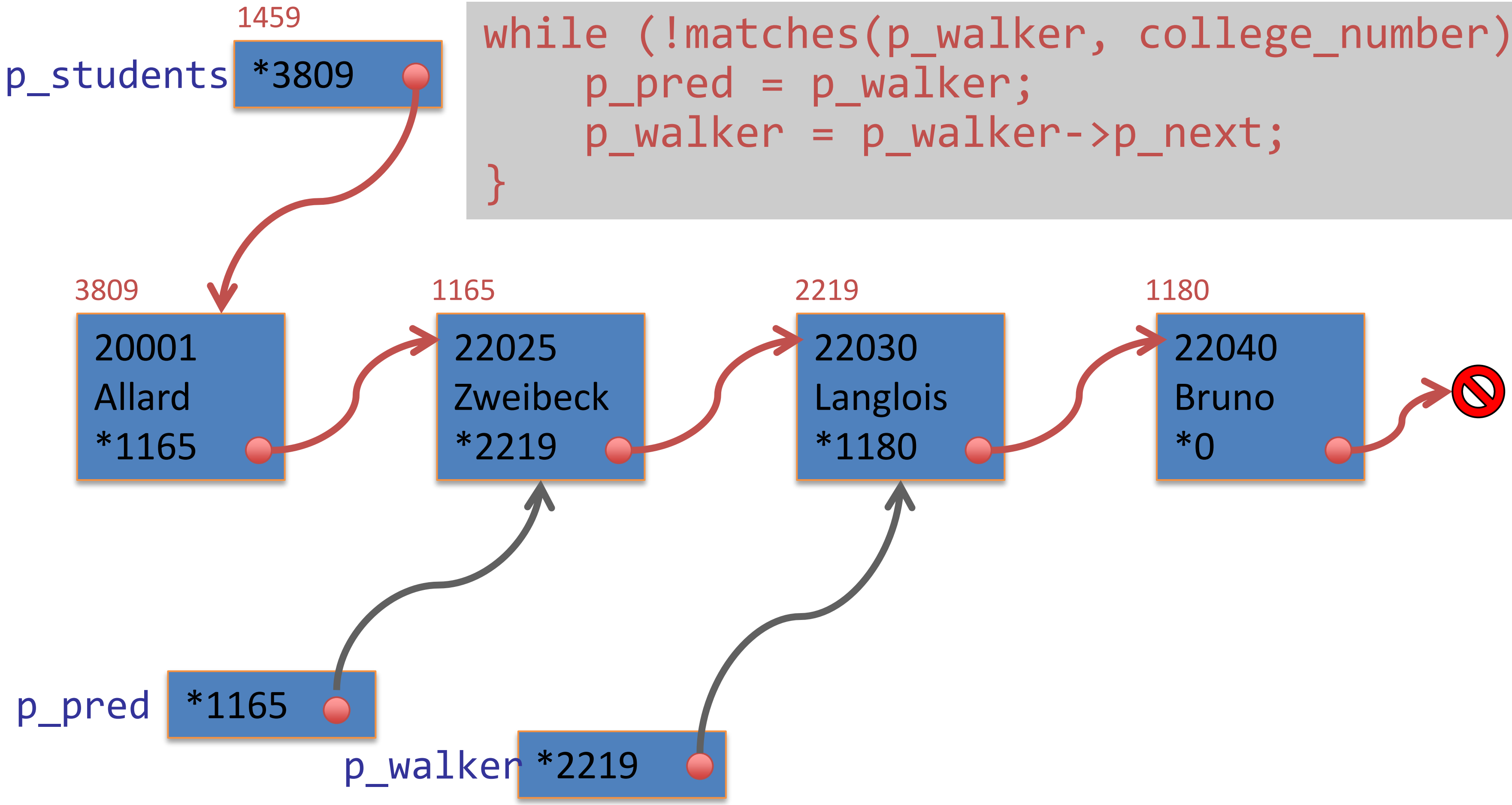
Deleting somewhere in the list

```
while (!matches(p_walker, college_number)){
    p_pred = p_walker;
    p_walker = p_walker->p_next;
}
```



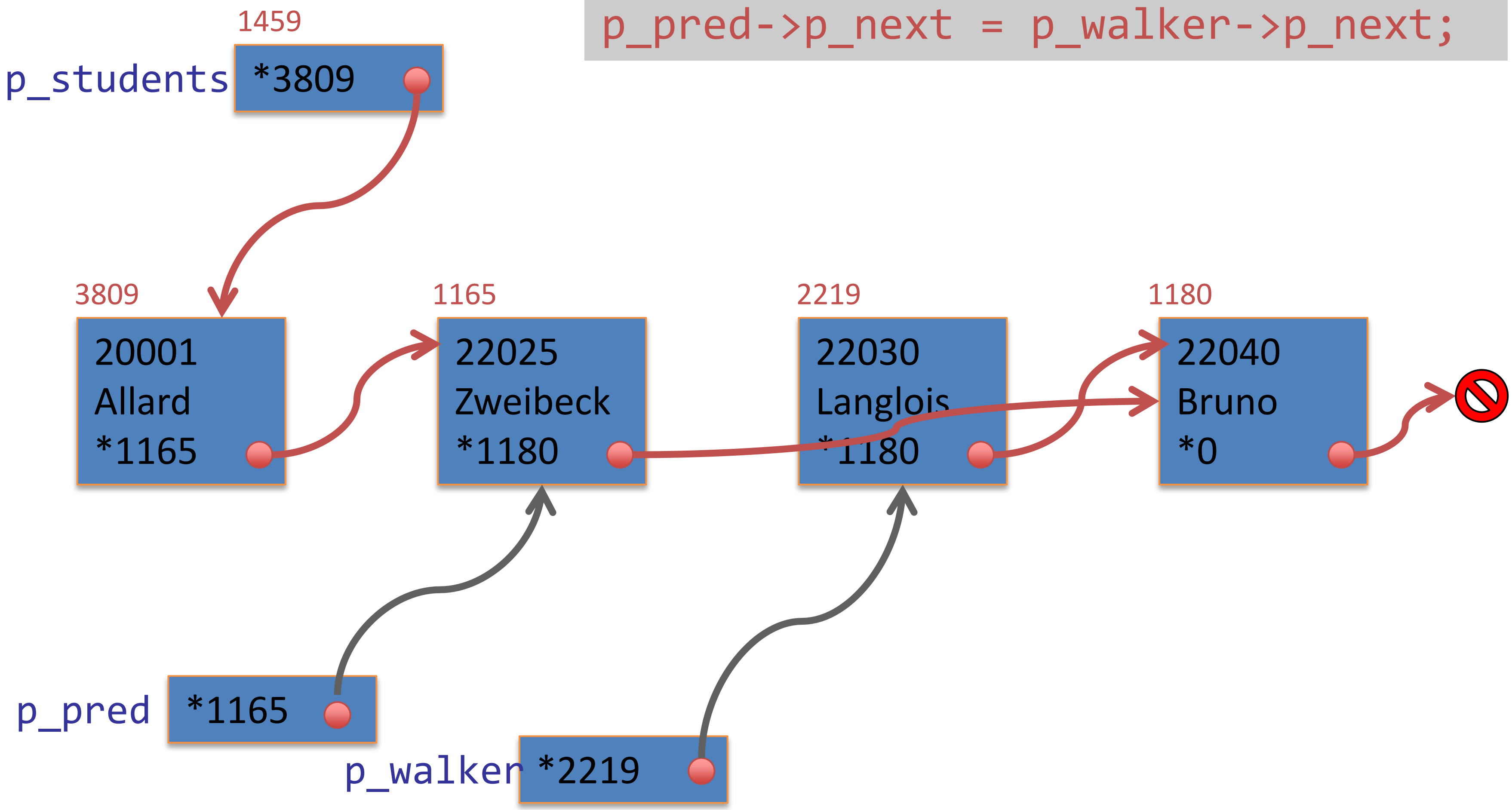
Deleting somewhere in the list

```
while (!matches(p_walker, college_number)){
    p_pred = p_walker;
    p_walker = p_walker->p_next;
}
```

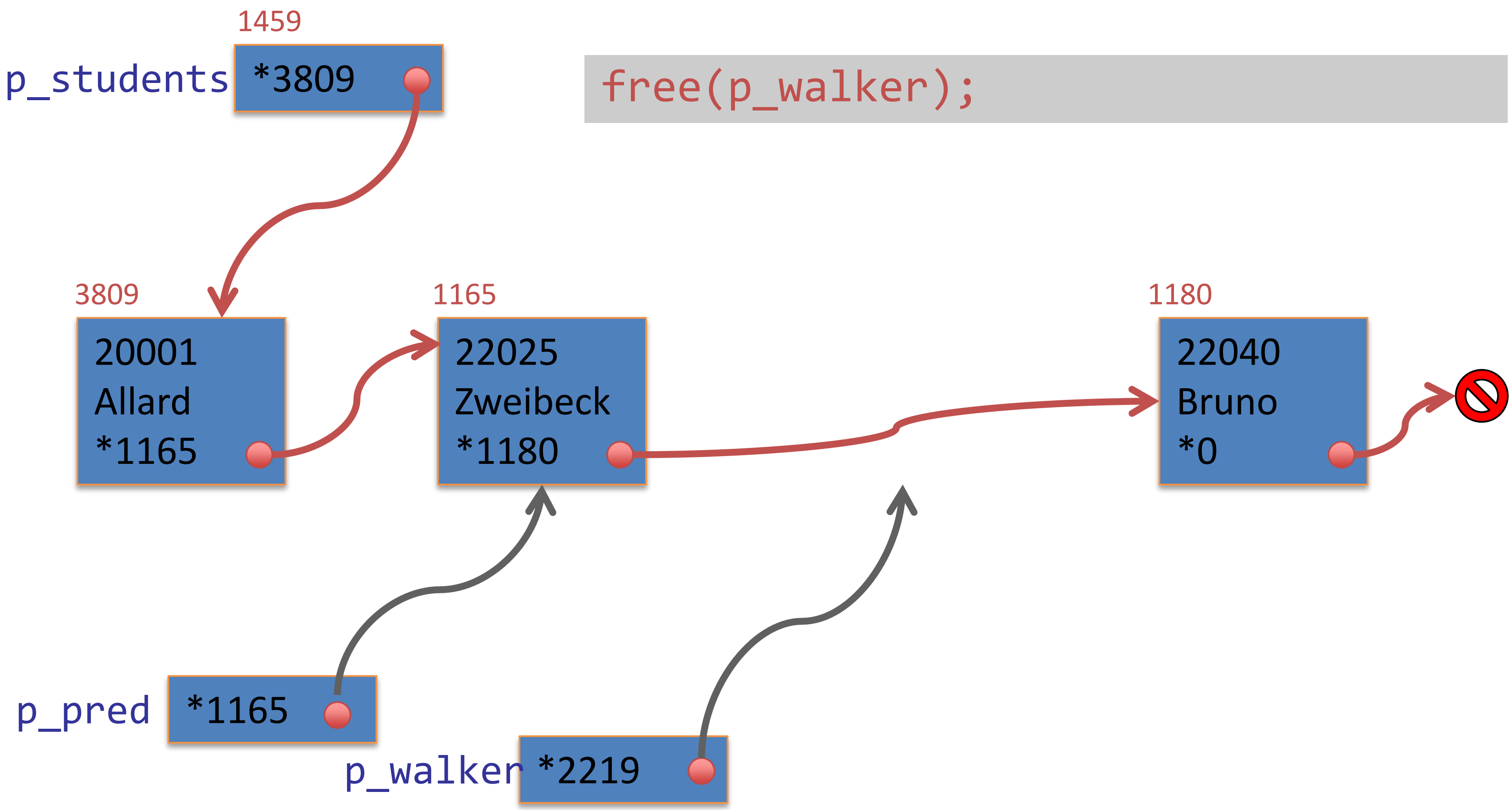


Deleting somewhere in the list

```
p_pred->p_next = p_walker->p_next;
```



Deleting somewhere in the list



Deleting somewhere in the list

```
void delete_by_college_number(char* college_number) {
    StudentNode* p_pred;
    StudentNode* p_walker = p_students;
    if (p_walker == NULL) return;
    if (matches(p_walker, college_number)) {
        p_students = p_walker->p_next;
        free(p_walker);
        return;
    }
    while (!matches(p_walker, college_number)) {
        p_pred = p_walker;
        p_walker = p_walker->p_next;
    }
    if (p_walker != NULL) {
        p_pred->p_next = p_walker->p_next;
        free(p_walker);
    }
    return;
}
```

Deleting a node elsewhere in LL using indirection

- You can also use a level of indirection to keep a hold of the predecessor to the node being deleted

```
p_walker->p_next->college_number
```

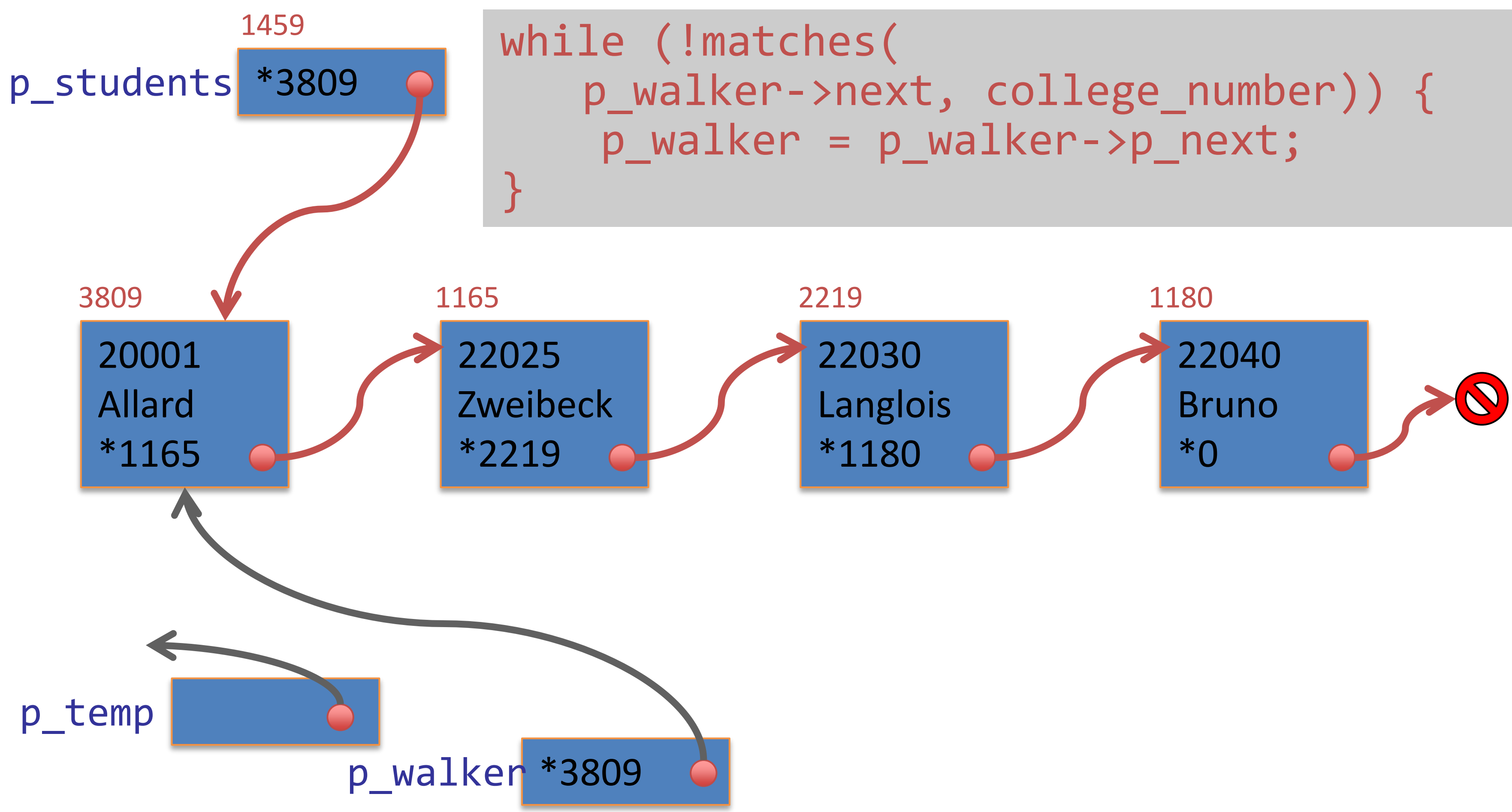
- We only keep one reference (`p_walker`) to walk through the list (`p_pred` is not used)
- You still need another pointer to do the deletion
- When using this kind of indirection be careful what you free!

Deleting a node elsewhere in LL using indirection

```
void function DeleteMatchingcollege_number(char* target) {
   StudentNode* p_walker = p_students;
   StudentNode* p_temp;
   if (p_walker == NULL) return;
   if (matches(p_walker, target) {
    p_students = p_walker();
    free(p_walker);
    return;
   }
   while (!matches(p_walker->p_next, target) {
    p_walker = p_walker->p_next;
   }
   if (p_walker->p_next != NULL) {
    p_temp = p_walker->p_next;
    p_walker->p_next = p_walker->p_next->p_next;
    free(p_temp);
   }
   return;
}
```

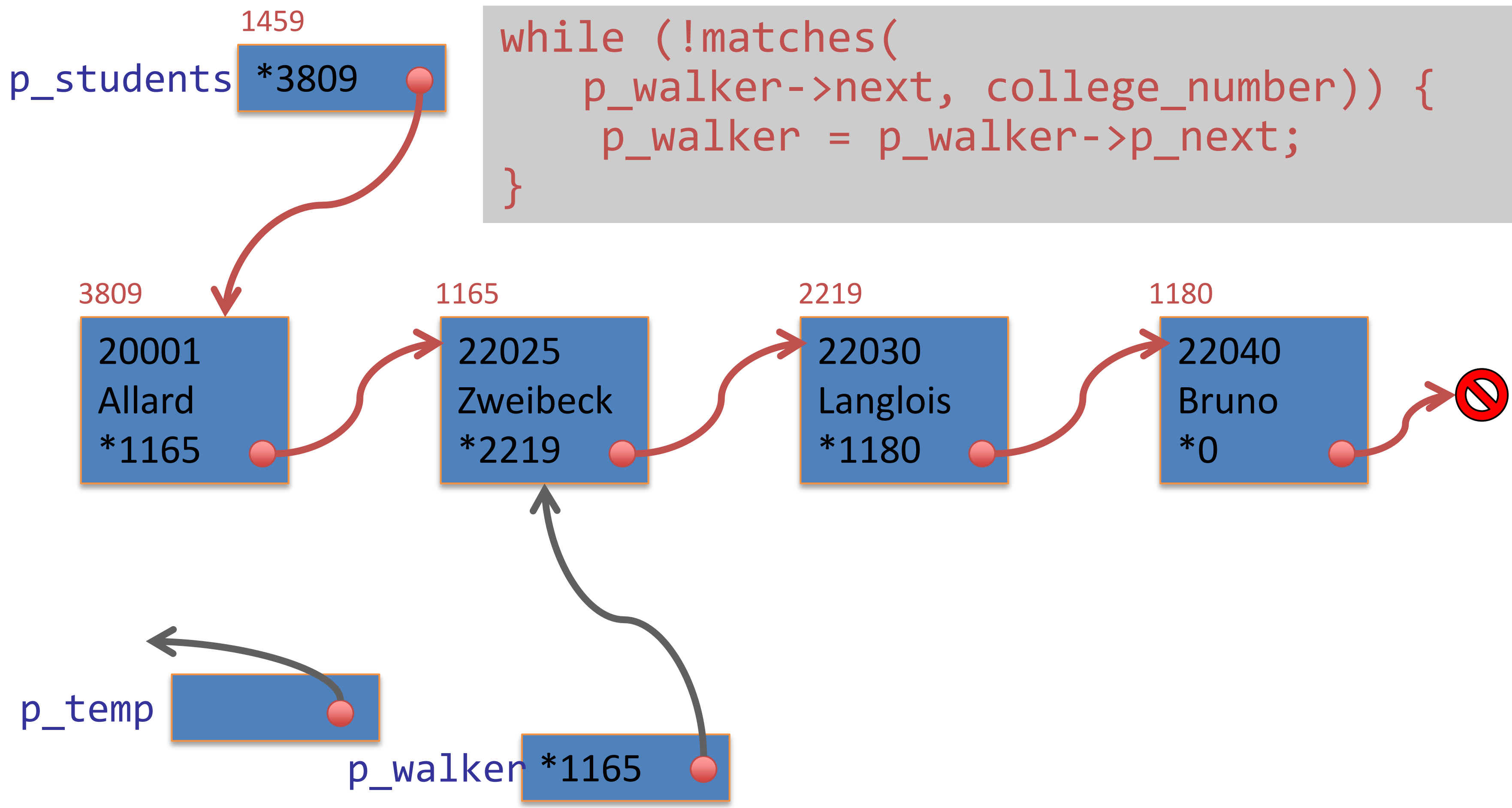
Deleting somewhere in the list

```
while (!matches(
    p_walker->next, college_number)) {
     p_walker = p_walker->p_next;
}
```



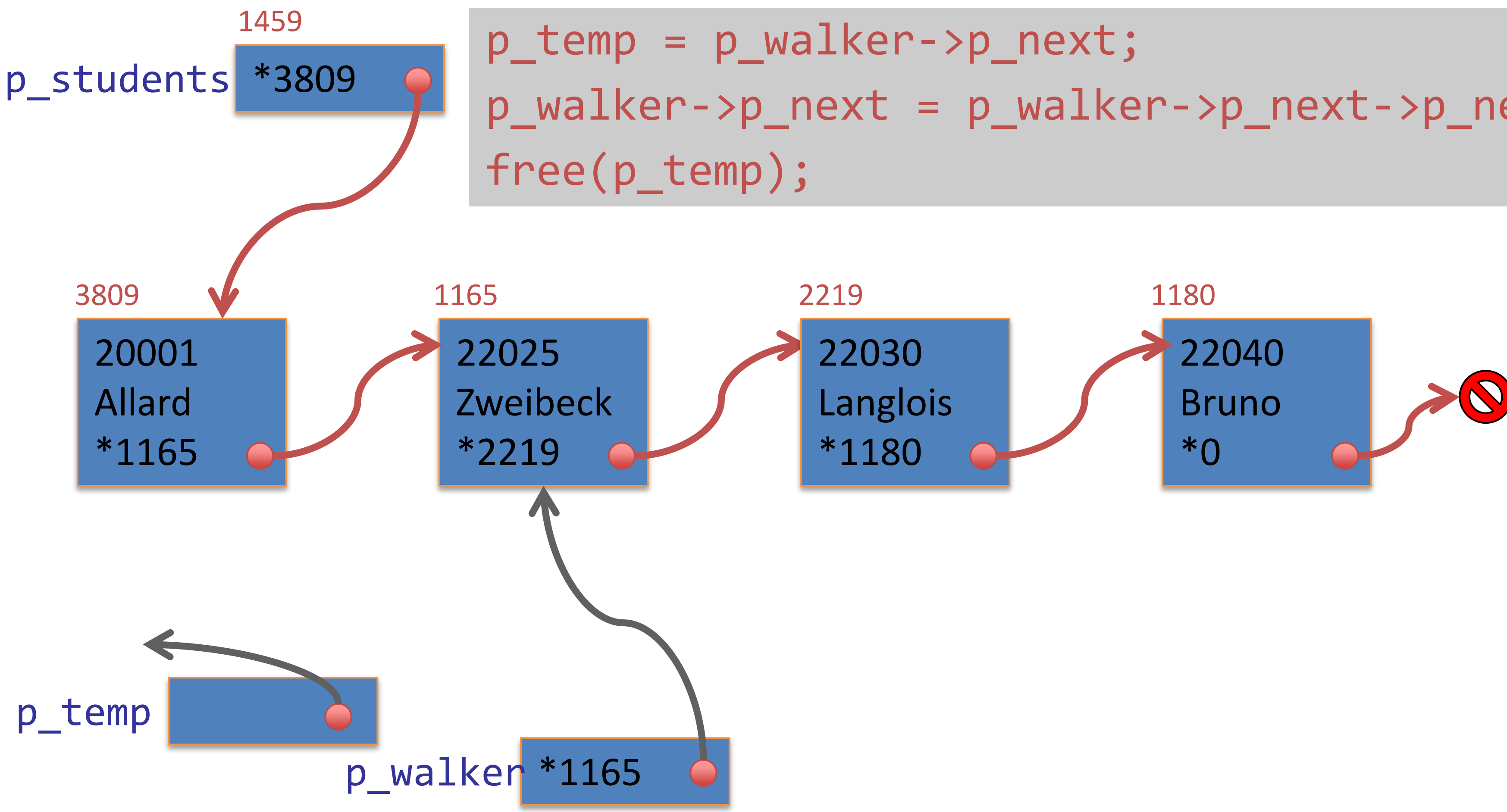
Deleting somewhere in the list

```
while (!matches(
    p_walker->next, college_number)) {
     p_walker = p_walker->p_next;
}
```



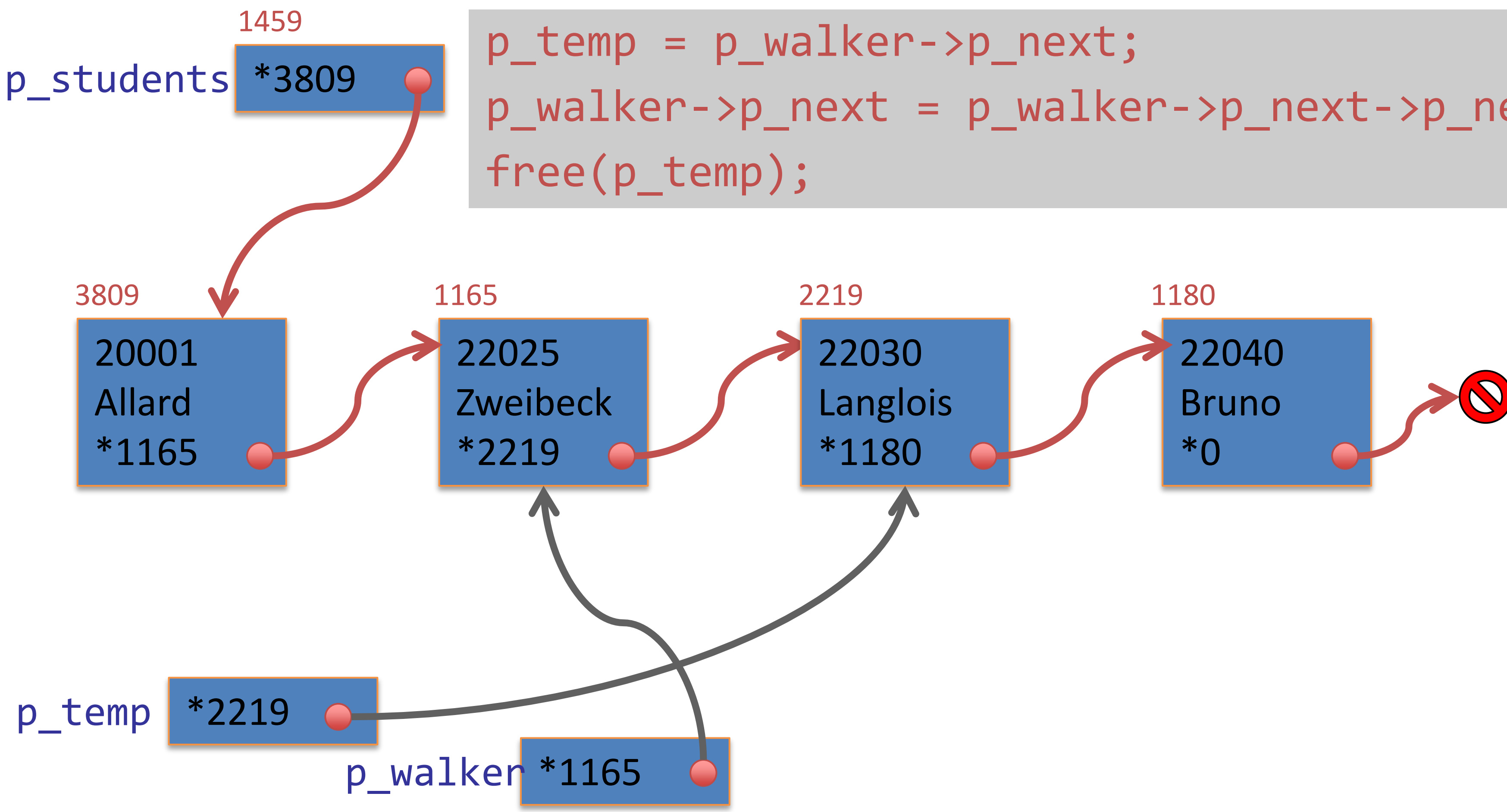
Deleting somewhere in the list

```
p_temp = p_walker->p_next;
p_walker->p_next = p_walker->p_next->p_next;
free(p_temp);
```



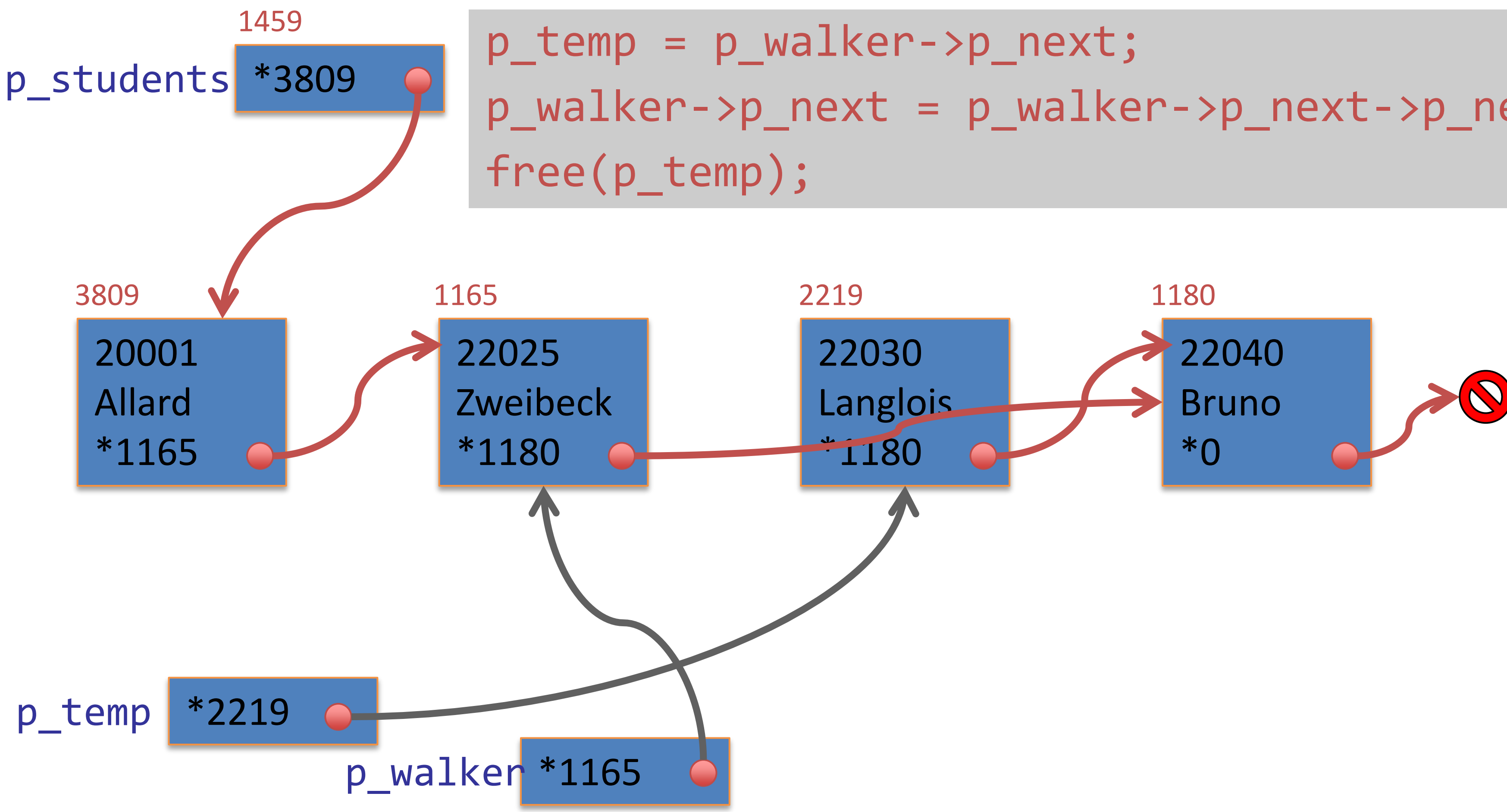
Deleting somewhere in the list

```
p_temp = p_walker->p_next;
p_walker->p_next = p_walker->p_next->p_next;
free(p_temp);
```



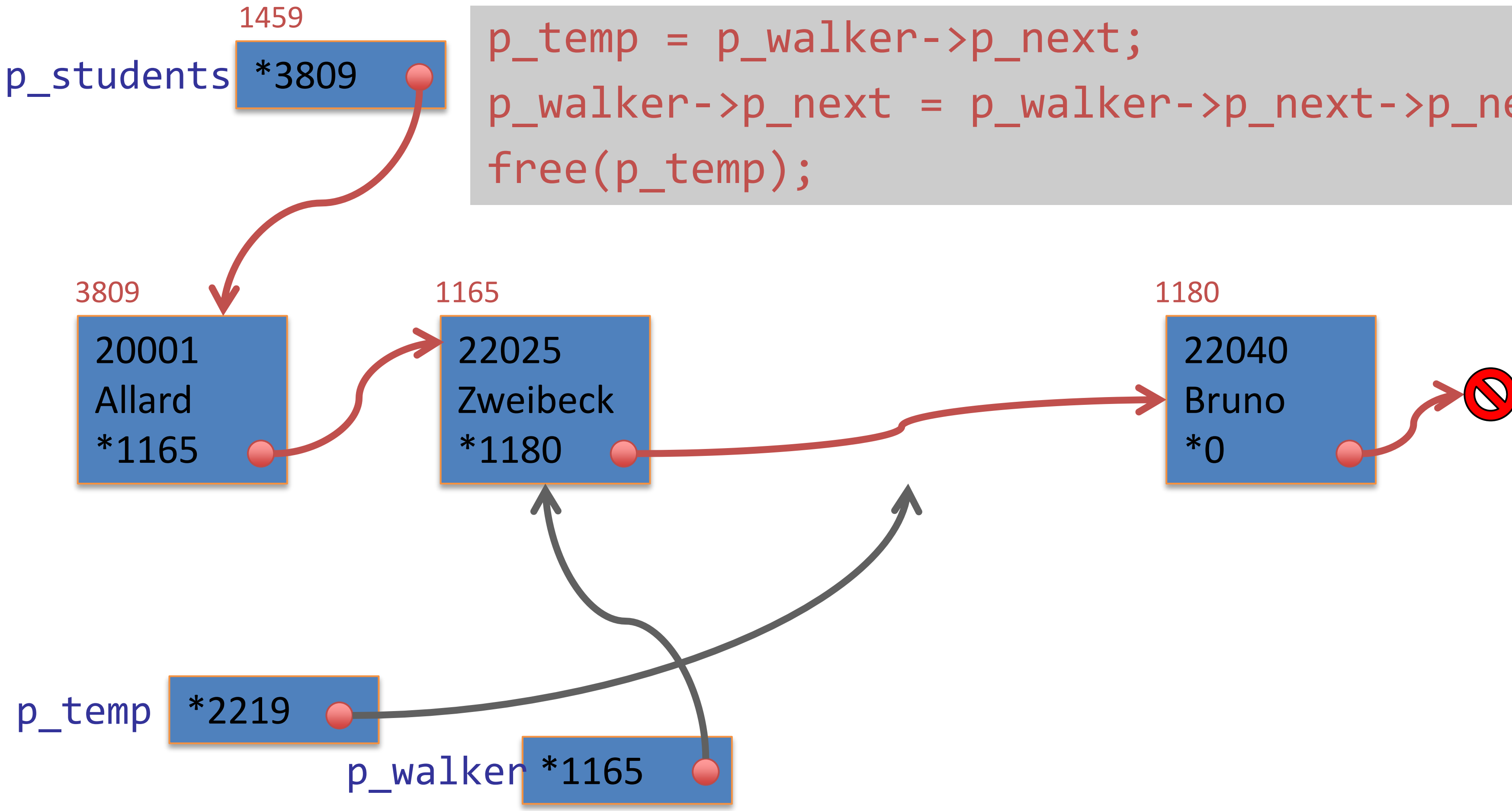
Deleting somewhere in the list

```
p_temp = p_walker->p_next;
p_walker->p_next = p_walker->p_next->p_next;
free(p_temp);
```



Deleting somewhere in the list

```
p_temp = p_walker->p_next;
p_walker->p_next = p_walker->p_next->p_next;
free(p_temp);
```



API design issue

- implement `push` as a function, e.g.,

```
void push(Node *p_new);
```

- but `push` needs to modify the list head, and the head isn't a local variable

- options:
  - make the list head a global and modify it inside the function
  - pass the list head to the `push` function by value and return it from the function
  - pass the list head to the `push` function by reference and modify it inside the function

p_head as a global

```
// global declaration of p
Node *p_head;

// prototype
void push(Node *p_new);

// use
push(p_node);

// implementation
void push(Node *p_new) {
    p_new->p_next = p_head;
    p_head = p_new;
    return;
}
```

pass and return p_head by value

```
// prototype
Node *push(Node *p_head, Node *p_new);

// use
p_head = push(p_head, p_node);

// implementation
Node *push(Node *p_head, Node *p_new) {
    p_new->p_next = p_head;
    return p_new;
}
```

pass p_head by reference

```
// prototype
void push(Node **pp_head, Node *p_new);

// use
push(&p_head, p_node);

// implementation
void push(Node **pp_head, Node *p_new) {
    p_new->p_next = *pp_head;
    *pp_head = p_new;
    return;
}
```

comparing the three

```
// p_head global
void push(Node *p_new);
push(p_node);

// pass and return p_head by value
Node *push(Node *p_head, Node *p_new);
p_head = push(p_head, p_node);

// pass p_head by reference
void push(Node **pp_head, Node *p_new);
push(&p_head, p_node);
```

Conclusion

- pass by reference is arguably the best option
  - harder to understand, since it uses a pointer to a pointer
- problem also applies to other linked list functions; comes up frequently in other contexts
- issue is dealt with very cleanly in object-oriented languages like Java, where the list will be an object:

```
myList.push(node);
```

Questions?