Module 8 Guidance Notes

Pointers, Dynamic Memory & Linked List

ENGG1340

Computer Programming II

Before We Start

- We will deal with only C++ in this module.
- C++: We will be using the C++ 11 standard, so make sure that your compiler option is set appropriately. We suggest to use the following command to compile your C++ program:

```
g++ -pedantic-errors -std=c++11 your_program.cpp
```

How to Use this Guidance Notes

- Use Powerpoint Slide Show Mode to view the slides.
- This guidance notes aim to lead you through the learning of the C/C++ materials. It also defines the scope of this course, i.e., what we expect you should know for the purpose of this course. (and which should not limit what you should know about C/C++ programming.)
- Pages marked with "Reference Only" means that they are not in the scope of assessment for this course.
- The corresponding textbook chapters that we expect you to read will also be given. The textbook may contain more details and information than we have here in this notes, and these extra textbook materials are considered references only.

How to Use this Guidance Notes

- We suggest you to copy the code segments in this notes to the coding environment and try run the program yourself.
- Also, try make change to the code, then observe the output and deduce the behavior of the code. This way of playing around with the code can help give you a better understanding of the programming language.

References

- cplusplus.com tutorial
  - Pointers
  - Dynamic Memory
- Textbook Chapters
  - C++: How to program (9th edition)
    Electronic version available from HKU library
  - Ch. 8 (on pointers)
  - Ch. 10.9 (on dynamic memory management)

Topics

Part I: Pointers

Part II: Dynamic Memory Management

Part III: Linked List

Part I

POINTERS

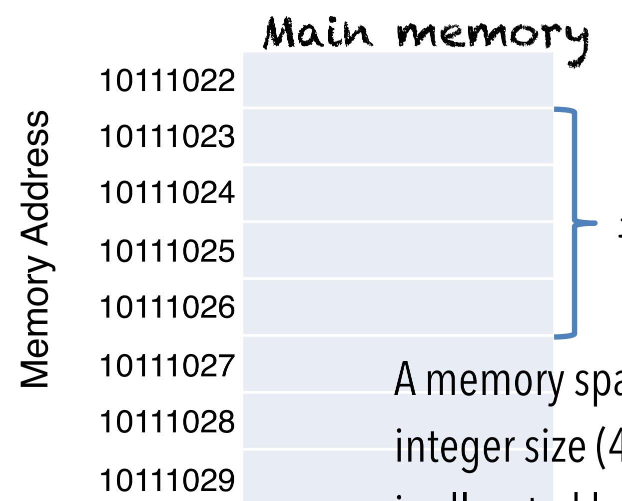
What are we going to learn?

- Memory addresses and pointers
- Pointers and arrays
- Pass-by-reference with pointers
- Dynamic variables
- Dynamic arrays
- Pointer operations

Memory Address

- The main memory of a computer can be regarded as a collection of consecutively numbered memory cells
- Each memory cell has a minimal size that the computer can manage (e.g., one byte)

- The unique number assigned to each memory cell is called its address, which is used to locate the memory cell in main memory



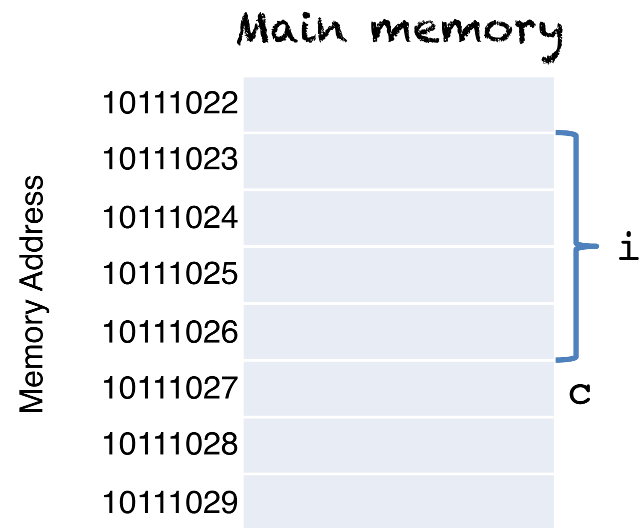
A memory space of an integer size (4 bytes) is allocated by the the variable declaration `int i;`

The address of `i` is `10111023`.

Address-of Operator

- The memory address of a variable can be obtained by placing the address-of operator & in front of the variable

```
int i;
char c;

cout << &i << ' ' << &c;
```




```
10111023 10111027
```

This is just the conceptual output, as memory addresses are by default output as hex. Check addressof.cpp

Pointer Variable

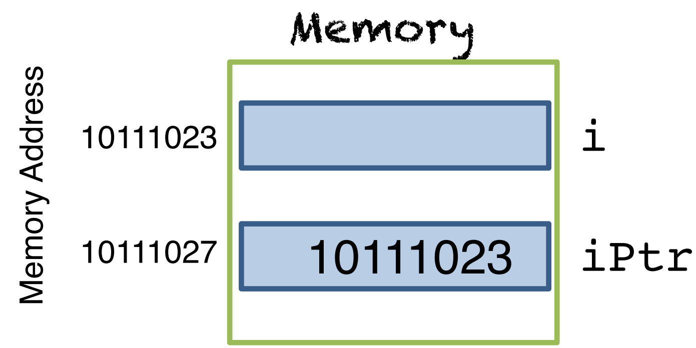
- We may declare a pointer variable to store the address of a variable

Creating a variable named `i` of type `int` that stores an integer

```
int i;
int * iPtr = &i;
```

Creating a variable named `iPtr` of type `int *` that stores the address of another integer variable.

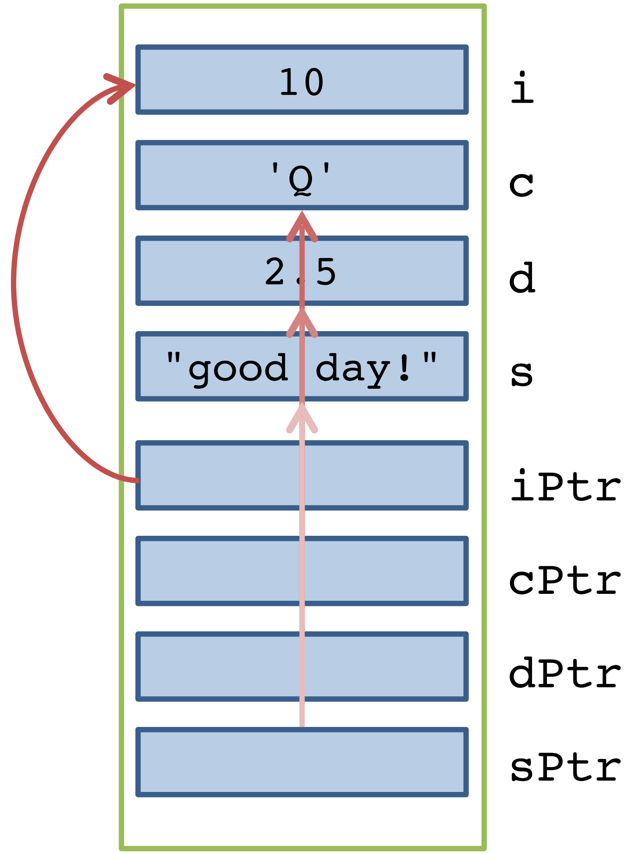
address of `i`



Usually represented as a diagram with an arrow pointing from the pointer variable to the memory address that it stores:

Pointer Variable

```
int i = 10;
char c = 'Q';
double d = 2.5;
string s = "good day!";

int * iPtr;
char * cPtr;
double * dPtr;
string * sPtr;

iPtr = &i;
cPtr = &c;
dPtr = &d;
sPtr = &s;
```



Pointer Variable

```
int * iPtr;
char * cPtr;
double * dPtr;
string * sPtr;
```

These are all pointers that point to variables of different types, and therefore the pointers are of different types

Hence, it is an error to assign to a pointer variable of one type with an address of another variable of a different type.

```
int * iPtr;
char c;

iPtr = &c;
```

✗

Compilation error!
`&c` is of type `char *`

Pointer Variable

- We can declare pointer variables and regular variables together in the same declaration statement:

```
int i, * iPtr;
char c, * cPtr;
double d, * dPtr;
string s, * sPtr;
```

How may we declare multiple pointers of the same type in a single statement?

```
int * iPtr1, * iPtr2, * iPtr3;
```

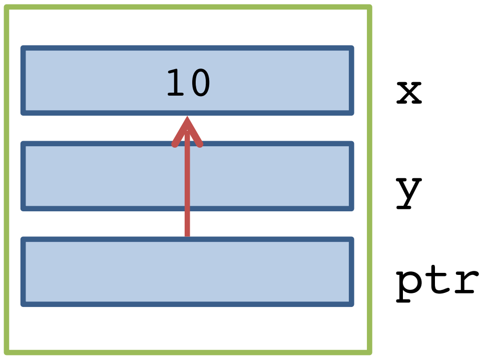
We need to place an asterisk * in front of each variable to indicate that each of them is a pointer.

Dereference Operator

- The memory location that a pointer points to can be accessed or modified using the dereference operator *.

```
int x = 10, y;
int * ptr = &x;

y = *ptr;
*ptr = 20;
cout << x << ' ' << y << endl;
cout << *ptr << endl;

ptr = &y;
(*ptr)++;
cout << x << ' ' << y << endl;
cout << *ptr << endl;
```

dereference.cpp



Dereference Operator

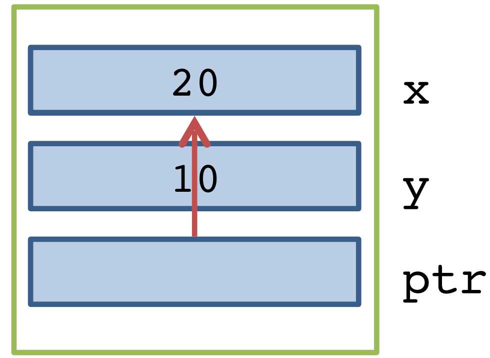
- The memory location that a pointer points to can be accessed or modified using the dereference operator *.

```
int x
int *

y = *ptr;
*ptr = 20;
cout << x << ' ' << y << endl;
cout << *ptr << endl;

ptr = &y;
(*ptr)++;
cout << x << ' ' << y << endl;
cout << *ptr << endl;
```

Accessing the contents of the memory location pointed to by `ptr`

Modifying the contents of the memory location pointed to by `ptr`

dereference.cpp



Screen output

```
20 10
20
```

Dereference Operator

- The memory location that a pointer points to can be accessed or modified using the dereference operator *.

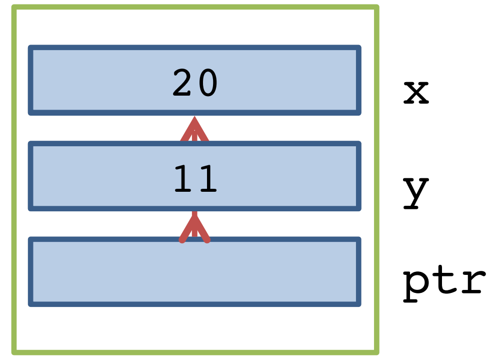
`*ptr` can be viewed as an alias (i.e., another name) of the variable that the pointer `ptr` points to.

Note that * is both used (1) to declare a pointer and (2) to dereference a pointer. It has different meanings in the two cases.

The parentheses are necessary since the ++ operator takes high precedence over *

```
cout << x <
cout << *p

ptr = &y;
(*ptr)++;
cout << x << ' ' << y << endl;
cout << *ptr << endl;
```

dereference.cpp



Screen output

```
20 10
20
20 11
11
```

```
int x = 10, y = 20;
string s = "abc";

int * ptr1, * ptr2;
int * ptr3;
string * ptr4;
```

What are the results of the followings?

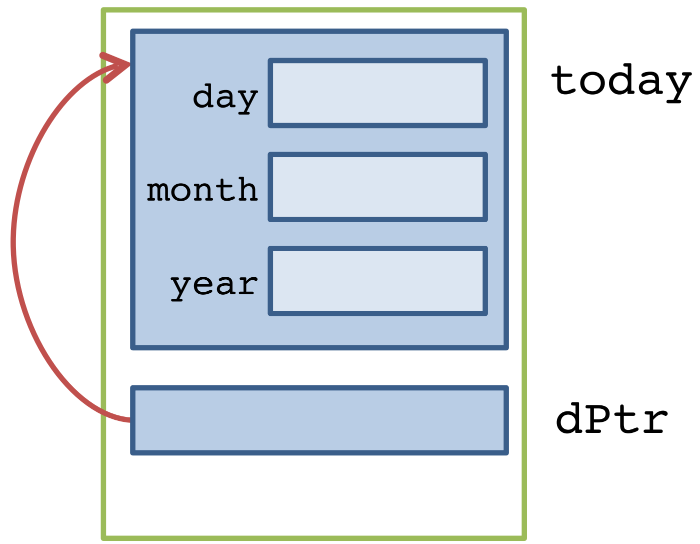
• `ptr1 = &x;`	`ptr1` points to `x`
• `ptr2 = &y;`	`ptr2` points to `y`
• `ptr3 = &y;`	`ptr3` also points to y
• `ptr4 = &y;`	Error! A pointer to `string` cannot store the address of an `int`
• `*ptr1 = *ptr2;`	`x` now stores `20`
• `*ptr3 = *&x - 10;`	`y` now stores `10`
• `cout << *ptr3;`	`10`

& and * are inverse of each other

Member Access Operator

- Consider a pointer that points to a compound data (e.g., a structure or a class):

```
struct Date
{
    int day;
    int month;
    int year;
};

Date today;

Date * dPtr = &today;
```



Declare a pointer to a structure of type `Date` and assign the address of `today` to it.

Member Access Operator

- Now we may access the members of the structure in the following ways:

```
today.year = 2015;
```

By using the dot operator of a structure

```
(*dPtr).year = 2015;
```

By first dereferencing the pointer to obtain a structure, then using the dot operator.
Note that the parentheses are necessary here, as . (dot) takes higher precedence over * (star)

```
dPtr->year = 2015;
```

By using the –> shorthand (which means member of pointer)

Check pointer_date.cpp

Member Access Operator

- Member functions of a class can also be accessed in the same ways.

```
string s = "good day!";
string * sPtr = &s;

cout << s.length() << endl;

cout << "1st word: " << (*sPtr).substr(0, 4) << endl;
cout << "2nd word: " << sPtr->substr(5, 3) << endl;

cout << "sixth letter: " << (*sPtr)[5] << endl;
```

pointer_string.cpp

`*sPtr` is like an alias to `s`

Dangling Pointers

- A pointer that does not point to a valid object is called a dangling pointer.
- Dereferencing a dangling pointer will lead to unpredictable result and sometimes may crash your program.

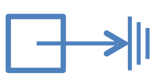
```
int * dangling_ptr;
cout << *dangling_ptr << endl;
```

What is the result?

Since `dangling_ptr` is not initialized, it stores an address which is just some garbage value.
The result of the statement depends on where `dangling_ptr` points to.

Null Pointer

- We may assign a zero value (using the keyword `nullptr`) to a pointer which means that the pointer points to nothing.
- The pointer is then called a null pointer or a zero pointer.

ptr 

```
int * ptr = nullptr;
cout << *ptr << endl;
```

Dereferencing a null pointer will crash the program

```
if ( ptr != nullptr )
    cout << *ptr << endl;
```

null_pointer.cpp

Check if a pointer is null before using it

`nullptr` is a constant that equals `0`, so we may use either `nullptr` or `0`. (Prior to C++11, the constant NULL is used instead.)

What's wrong with the following statements?

```
Date today;
Date * dPtr;

cout << dPtr->month;
```

`dPtr` is a dangling pointer. Accessing `dPtr->month` is error prone.

How to fix it?

Trial 1

```
Date today;
Date * dPtr = 0;

cout << dPtr->month;
```

`dPtr` is a null/zero pointer. Accessing `dPtr->month` will crash the program.

Trial 2

```
Date today;
Date * dPtr = &today;

cout << dPtr->month;
```

Trial 3

```
Date today;
Date * dPtr = 0;

if (dPtr != 0)
    cout << dPtr->month;
```

Pointers and Arrays

- The name of an array is indeed a pointer pointing to the first element of the array
- Hence, we may assign an array name to a pointer, and use the pointer to access the array elements

```
int x[10], i;

for ( i = 0; i < 10; ++i )
    x[i] = 2 * i;

int * p = x;

for ( i = 0; i < 10; ++i )
    cout << p[i] << ' ';
cout << endl;
```

Assigning an array name to a pointer of the same type as the array element

The pointer variable can be used just as an array name

However, it is invalid to assign a pointer to an array name (e.g., `x = p`), since an array name is a constant pointer variable.

Pointers and Arrays

```
int a[10], i;

for ( i = 0; i < 10; ++i )
    a[i] = 2 * i;

int * p = a;
for ( i = 0; i < 10; ++i )
    cout << p[i] << ' ';
cout << endl;

int * q = &a[0];
for ( i = 0; i < 10; ++i )
    cout << q[i] << ' ';
cout << endl;

p = &a[2];
cout << p[3] << endl;
```

Screen output

```
0 2 4 6 8 10 12 14 16 18
0 2 4 6 8 10 12 14 16 18
10
```

pointer_array.cpp

Exercise 1

- Write a function that takes an integer array and its size, and returns a pointer to the largest element in the array

Pass-by-reference with Reference Arguments

- We have learned pass-by-value and pass-by-reference for passing arguments to a function.
- Pass-by-reference enables the called functions to modify the values of the arguments passed from the caller.

```
void swap( int & x, int & y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Reference arguments

In the caller (e.g., the main function)

```
int a = 2, b = 3;

swap(a, b);
```

The values in `a` and `b` will be swapped after calling `swap()` because `x` and `y` are just aliases of `a` and `b`, respectively (i.e., they share the same memory locations)

Pass-by-reference with Pointers

- We can also achieve pass-by-reference by passing pointers as arguments.

```
void swap( int * x, int * y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

Pointer arguments

swap_by_pointers.cpp

In the caller (e.g., the main function)

```
int a = 2, b = 3;

swap(&a, &b);
```

Here we explicitly pass the memory addresses of a and b to swap(), so that swap() operates on these memory locations directly.

The values in `a` and `b` will be swapped after calling `swap()`.

Exercise 2

- Write a function void addOne(int &p) which adds 1 to the integer referenced by p
- Write a function void addOne(int *p) which adds 1 to the integer pointed to by p
- Note the difference in the function parameter. For each of the above, write the appropriate function call in the main body of your program.

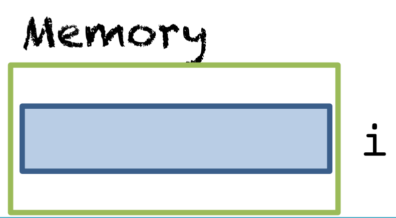
Part II

DYNAMIC MEMORY MANAGEMENT

Static Variables

- We have used only static variables in our programs so far, which means that:
  - The number of variables is fixed.
  - The life span of variable is determined by its scope; it is created (i.e., storage space is allocated) when it is declared and it is destroyed (i.e., storage space is released) when execution is out of scope.
  - Each variable is given a name when it is declared.

```
for (int i = 0; i < 10; ++i)
{
    cout << i << ' ';
}
```



The variable `i` only exists in the memory during the execution of the `for` loop.

Dynamic Variables

- Very often the number of variables that we need in a program is not known in advance. For example, processing student records without knowing the number of students beforehand.
- We can create dynamic variables in our program so that memory storage is dynamically allocated or released at runtime.

Unlike static variables, dynamic variables have no names!

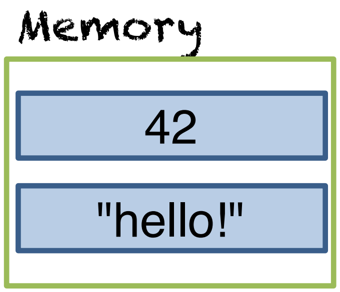
So how may we access dynamic variables?

Pointers!!!

Creating Dynamic Variables

- We use the new operator to create a dynamic variable:

```
new int (42);
new string ("hello!");
```

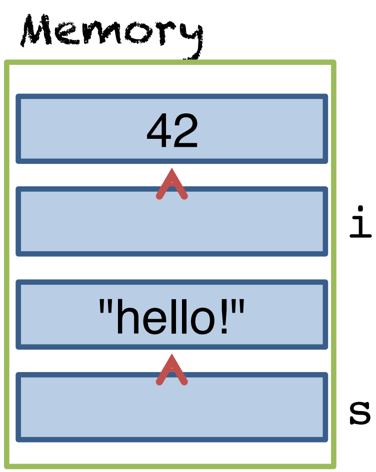


No names for these memory locations, and there's no way that you can access them

```
int * i = new int (42);
int * s = new string ("hello!");
```

this is equivalent to

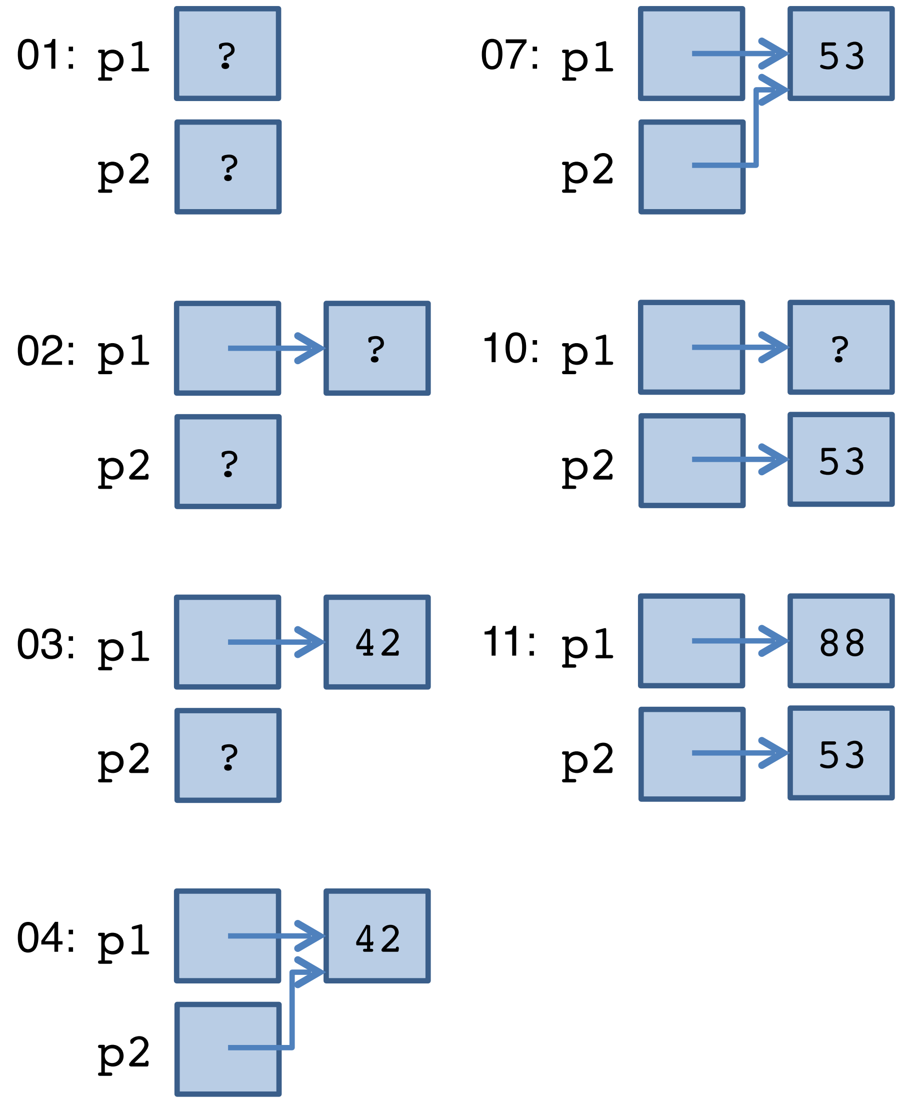
```
int * i = new int;
*i = 42;
string * s = new
string;
*s = "hello!";
```



Now we may access these memory locations via the pointers `i` and `s`.

dynamic.cpp

```
int *p1, *p2;
p1 = new int;
*p1 = 42;
p2 = p1;

cout << "*p1 = " << *p1 << ", ";
cout << "*p2 = " << *p2 << endl;

*p2 = 53;
cout << "*p1 = " << *p1 << ", ";
cout << "*p2 = " << *p2 << endl;

p1 = new int;
*p1 = 88;
cout << "*p1 = " << *p1 << ", ";
cout << "*p2 = " << *p2 << endl;
```



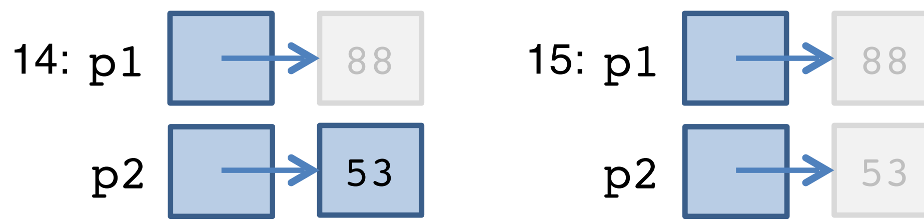
Screen output

```
*p1 = 42, *p2 = 42
*p1 = 53, *p2 = 53
*p1 = 88, *p2 = 53
```

Destroying Dynamic Variables

- Memory allocated to dynamic variables can be freed using the delete keyword:

```
14: delete p1;
15 : delete p2;
```



The pointer pointing to the memory location that needs to be freed.

The freed memory space can be re-used by the system.

Destroying Dynamic Variables

- It's a good practice to reset a pointer to zero after the memory location that it points to is freed.

```
int * p1 = new int (42);
cout << *p1;
delete p1;
p1 = 0;
```

- It's the programmer's responsibility to free up all dynamic variables that are no longer in use.

Failing to do so will lead to memory leak, i.e., having memory space that the system cannot reclaim, and the system may gradually run out of memory

Common Mistakes with Pointers

Dereferencing a pointer before it is initialized

```
int * p;
*p = 88;
```

Dereferencing a dangling pointer

```
int * p = new int;
*p = 88;
delete p;
cout << *p;
```

```
int * p1, * p2;
p1 = new int;
p2 = p1;
delete p1;
cout << *p2;
```

Deleting a pointer that does not point to a valid memory location

```
int * p1, * p2;
p1 = new int;
p2 = p1;
delete p1;
delete p2;
```

```
int * p;
p = new int;
…
delete p;
…
delete p;
```

Memory leak

```
int * p1, *p2;
p = new int;
q = new int;
q = p;
```

Dynamic Arrays

Recall

```
int a[10];
```

This declares an array of 10 integers. The size of the array is determined at compilation time.

What if we need more elements in the array during execution or the size of the array can only be known during runtime?

We may dynamically create an array at runtime using the new operator:

```
new int [10];
```

This allocates a dynamic array of 10 integers at runtime.

However, the dynamic array is without a name. So what's next?

Dynamic Arrays

- An example for the full cycle of a dynamic array

```
int n;
cin >> n;

int * a = new int [n];

for (int i = 0; i < n; ++i)
  a[i] = i;

…

delete [] a;
```

dynamic_array.cpp

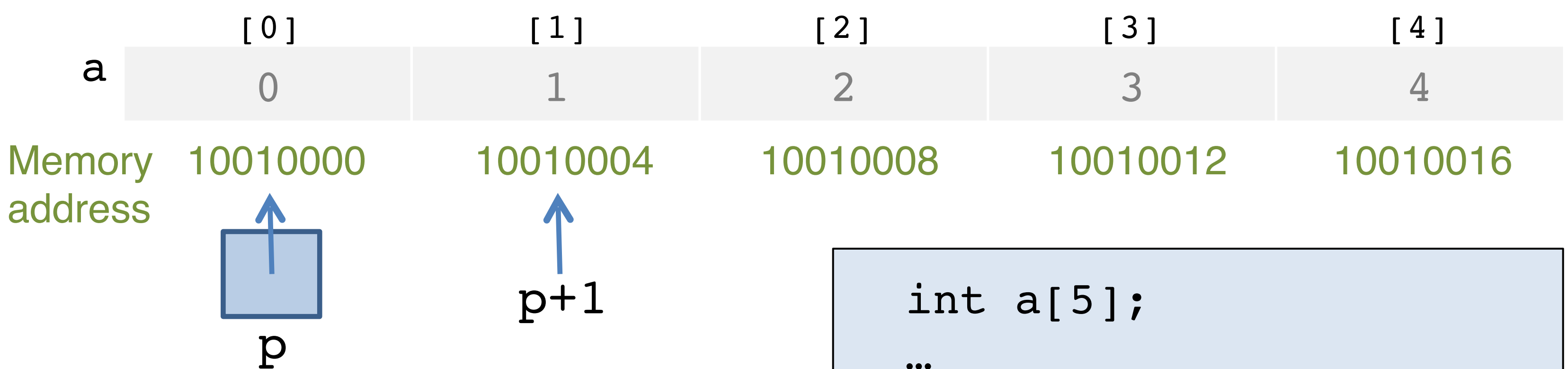
Create a dynamic array and use a pointer to point to it. Note that the value of `n` is only known at runtime.

Use the array pointer `a` to access the elements

Use `delete []` to free the dynamic array pointed to by `a`

Pointer Operations

- We may carry out addition and subtraction on pointers.
- Since they are actually memory addresses, the unit of addition and subtraction depends on the size of the data type to which they point.



Screen output

```
0 1
```

```
int a[5];
…
int * p = a;
cout << *p << ' ';
cout << *(p+1) << endl;
```

Pointer Operations

- We may also compare if two pointers are the same, i.e., if they point to the same memory location:

```
int a[5];
…

int * p = a, * q = a + 5;

while ( p != q ) {
  cout << *p << ' ';
  ++p;
}
```

pointer_operation.cpp

What does this program do?

Tutorial Problems – Dynamic Memory Management

PHONEBOOK MANAGER

Phonebook Manager

- We are going to work on a program that manages a phonebook that stores phone records. Functions provided by the phonebook are:
  - Load a phonebook from an external file
  - Print the records in a phonebook
  - Sort the records in a phonebook
  - Search in a phonebook
  - Save a phonebook to an external file
  - Add a record

No worry, most of these functions are implemented. But it is recommended that you take time (could be after the tutorial) to go through the codes and learn more about them.

`phonebook_incomplete.cpp` and **`phonebook.txt`** (a file containing phone records) are given to you.

phonebook.cpp provides the completed version of this tutorial problem. You may compile and run it to see the expected results first.

Phonebook Manager

- We focus ONLY on maintaining a dynamic array that stores the phone records so that the phonebook manager can handle as many phonebook records as the user requires, in a time/space efficient manner.
- You will be implementing a function called **`grow_phonebook()`** which enlarges the size of the dynamic array *(i.e., the size of phonebook)* when necessary.
- The phonebook is initially of size 3, i.e., it can hold 3 records at most:

In main():

```
int phonebook_size = 3;
PhoneRec * phonebook = new PhoneRec[phonebook_size];
```

Phonebook Manager

- Compile and run **`phonebook_incomplete.cpp`**.

Since we have not implemented **`grow_phonebook()`**, the program can only read in 3 records. (Note that there are 10 records in **`phonebook.txt`**.)

```
*******************************
* Welcome to Phonebook Manager *
*******************************
1. Load a phonebook.
2. Print all records.
3. Sort the records by ascending order of the name.
4. Search the records by partial match of the name.
5. Save the phonebook.
6. Add a new record.
0. Quit.
Please enter your choice: 1

Please enter the filename: phonebook.txt
--->phonebook size enlarged to hold a maximum of 3 records.
--->phonebook size enlarged to hold a maximum of 3 records.
--->phonebook size enlarged to hold a maximum of 3 records.
--->phonebook size enlarged to hold a maximum of 3 records.
--->phonebook size enlarged to hold a maximum of 3 records.
--->phonebook size enlarged to hold a maximum of 3 records.
--->phonebook size enlarged to hold a maximum of 3 records.

3 record(s) loaded.
```

Phonebook Manager

- After implemented **`grow_phonebook()`** correctly, the result should look like:

```
*********************************
* Welcome to Phonebook Manager *
*********************************
1. Load a phonebook.
2. Print all records.
3. Sort the records by ascending order of the name.
4. Search the records by partial match of the name.
5. Save the phonebook.
6. Add a new record.
0. Quit.
Please enter your choice: 1

Please enter the filename: phonebook.txt
---> phonebook size enlarged to hold a maximum of 6 records.
---> phonebook size enlarged to hold a maximum of 9 records.
---> phonebook size enlarged to hold a maximum of 12 records.

10 record(s) loaded.
```

When will **`grow_phonebook()`** be called?

- In `load_phonebook()` when the number of records read in exceeds the phonebook size.

```
int load_phonebook(...)
{
        ...
        if (i >= phonebook_size)
            grow_phonebook(...);

        ...
}
```

The variable **`i`** keeps track of the number of records read in so far, and **`phonebook_size`** stores the capacity of the phonebook, i.e., the size of the dynamic array for holding the records

- Before calling `add_record()` when the phonebook is already full.

```
int main()
{
        ...
        case '6':
            if (num_records >= phonebook_size)
                grow_phonebook(...);
        ...
}
```

The variable **`num_records`** stores the number of records that are kept in the dynamic array

What does **grow_phonebook()** do?

function prototype

```
void grow_phonebook(PhoneRec * &pb, int &pb_size, int n);
```

pb points to the dynamic array storing the phonebook

pb_size is the current size of the dynamic array

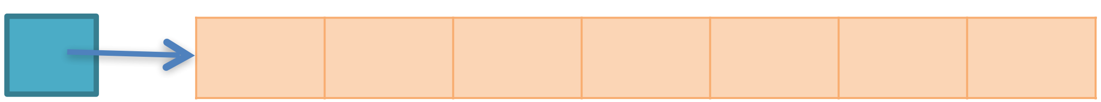
n is the size by which to increase the dynamic array. Hence, the new size of the array is **pb_size + n** after calling this function

Example:

phonebook **phonebook_size = 5**

phonebook_size is modified and hence it is passed as a reference parameter

After calling
`grow_phonebook( phonebook, phonebook_size, 2 );`



phonebook **phonebook_size = 7**

the new array occupies a new chunk of memory and hence the pointer **phonebook** needs also be modified; it is therefore passed as a reference parameter

Implementing `grow_phonebook()`

```
void grow_phonebook(PhoneRec * &pb, int &pb_size, int n);
```

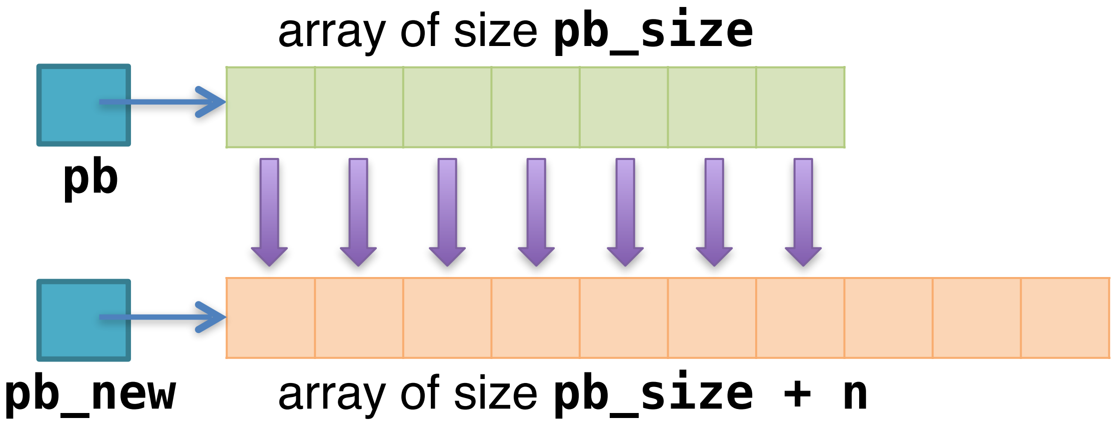
- Now, let's do the following steps for `grow_phonebook()`
- Step 1: create a new dynamic array with a new size equals **`pb_size + n`** dynamic array, pointed to by a pointer

What is the data type of **`pb_new`**?

How to create a dynamic array?

Implementing **grow_phonebook()**

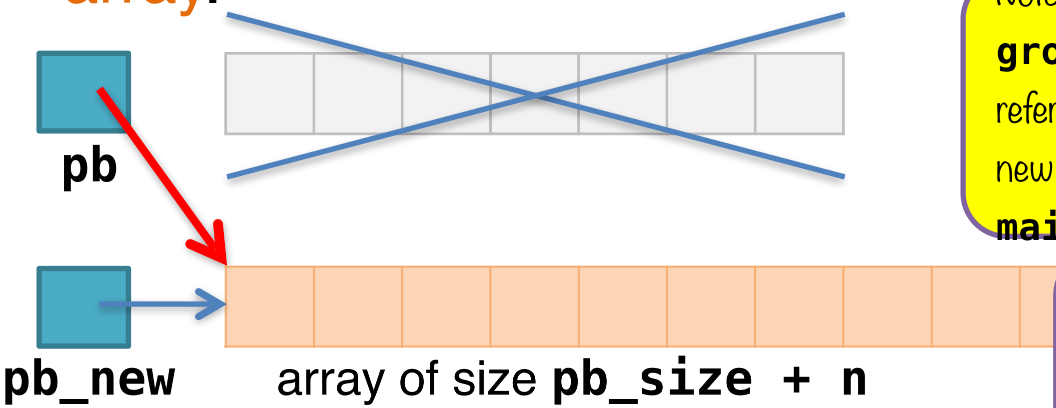
- Step 2: Copy all the records from the original array to the new array



You just need to treat it as ordinary copying of array elements. Remember that a pointer to array can be used as an array name for accessing the elements, e.g., you may write **pb[i]**, **pb_new[i]**

Implementing **grow_phonebook()**

- Step 3: Now that the new array is ready, we should deal with releasing the memory occupied by the old array. Delete the old dynamic array and points **pb** to the new array.



Note that **pb_new** is only local to **grow_phonebook**() and **pb** is the reference parameter that points to where the new array is in the calling function (i.e., **main()**)

If we forgot to update **pb** to point to the new array, the new array cannot be accessed in the main function and there is memory leak. Also, the pointer **phonebook** in the main function will become a dangling pointer.

- Step 4: update the phone book size pb_size to the new size and we are do

Try the program with the add record option from the main menu, and see the result.

A Question

- The program now works in such a way that the phonebook will grow whenever it is full, and we can control the size that it should grow every time (the parameter n in **`grow_phonebook()`**). But by how much?
- In this program, we just simply increase the size by a constant amount (now 3), independent of the original array size.
- What if we set a large **`n`**?
- What if we set a small **`n`**?
- Think about it first before turning to the next page for some suggestions.

A Question

- Having an `n` too large will result in wasted space in most of the time.
- Having an `n` too small will result in calling `grow_phonebook()` too frequent which is not time efficient, since it involves array copying.
- There is no right choice for `n` which works optimally in all cases, but a general practice is to double the array size every time when it needs to grow.

Part III

LINKED LIST

What are we going to learn?

- Modes of data access
  - Random access
  - Sequential access
- Linked lists
- Linked list operations
  - Traversing a linked list
  - Building a linked list
  - Inserting an item into a linked list
  - Deleting an item from a linked list

Mode of data access - Random Access

- Array is a container which allows random access to the items stored in it.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
data	23	56	92	38	12	76	35	89	10	62

We can directly access the 5^{th} item by writing `data[4]`

What if we want to access the 5^{th} smallest item?

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
data	10	12	23	35	38	56	62	76	89	92

Sort the array, and then access `data[4]` directly

Search can also be made fast (with a binary search) with a sorted array

Mode of data access - Random Access

What if we want to insert an item into a sorted array so that the array remains sorted?

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
data	10	12	23	35	38	56	62	76	89	92

For example, to insert 15 into data:

Step 1: Increase the array capacity if necessary

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
data	10	12	23	35	38	56	62	76	89	92	

Step 2: Shift all items larger than 15 to the right

What if this is a very very large array?

data	10	12		23	35	38	56	62	76	89	92

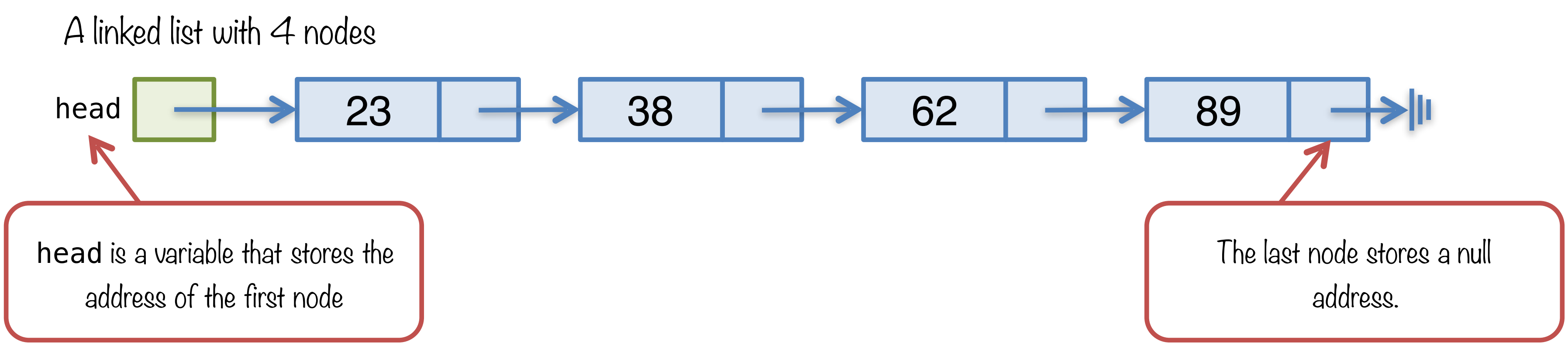
Step 3: Put 15 into the empty slot

data	10	12	15	23	35	38	56	62	76	89	92

Insertion and **deletion** using array is **not efficient**, because these involve data movement.

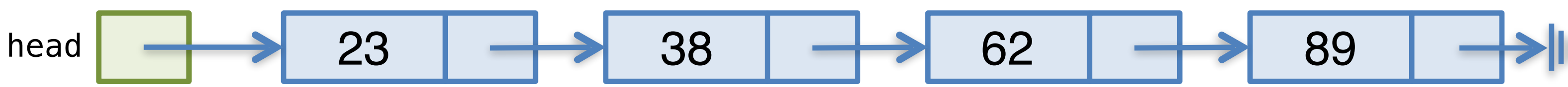
Linked Lists

- We need a data structure that can support efficient data insertion and deletion.
- Linked list is a collection of items called nodes.
- Each node stores a piece of data, as well as the address of the next node (except for the last node).

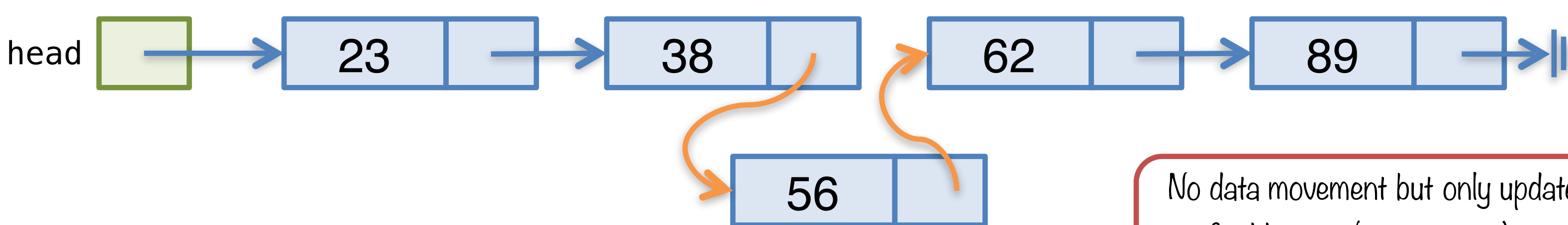


Linked Lists

- Linked list is a sequential access data structure
  - i.e., to go to a specific item in a linked list, you have to start from the head of the list and go through the item one by one until you hit that item you need.
- However, insertion and deletion of items can be done efficiently.



For example, to insert 56 into the linked list:



No data movement but only update of addresses (i.e., pointers) are needed.

Linked Lists vs. Arrays

Linked Lists

- Items need not be stored contiguously in memory
- Sequential access from the head of list for an item
- Insertion & deletion of items can be done efficiently (in constant time, i.e., independent of the number of items)

Arrays

- Items are stored contiguously in memory
- Random access that allows fast direct access to an item
- Insertion & deletion of items can be time consuming (in linear time in the number of items)

Implementation

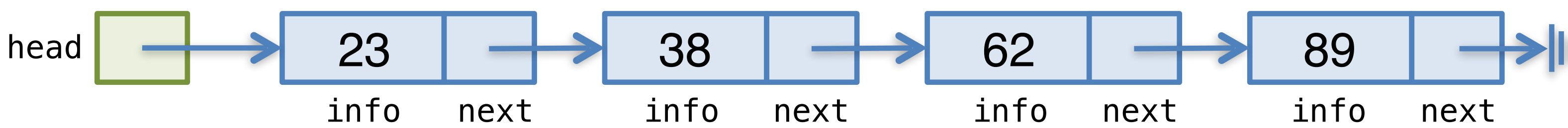
- A node can be implemented using a struct in C++.

```
struct Node
{
    int info;
    Node * next;
};
```

- The linked list is given as a pointer that points to the first node.

```
Node * head;
```

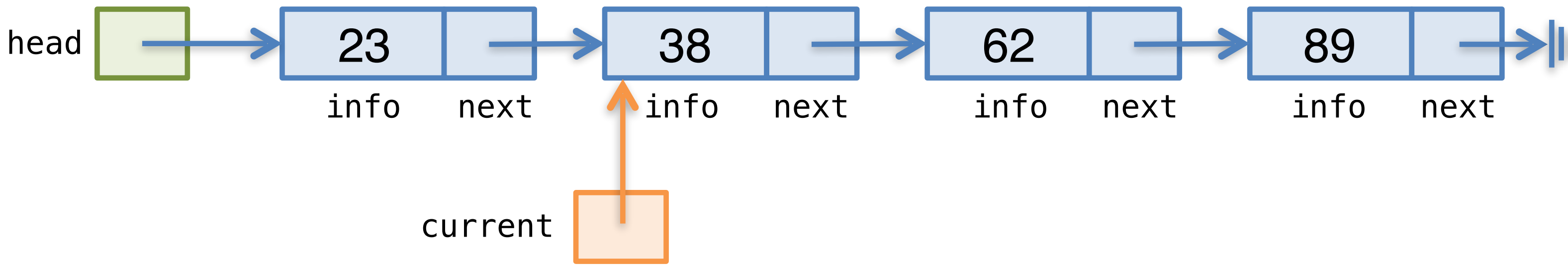
Implementation



What do the following expressions evaluate to?

head	address of the 1st node of the list
head->info	23
head->next	address of the 2nd node
head->next->info	38
head->next->next->info	62
head->next->next->next->info	89
head->next->next->next->next	0
head->next->next->next->next->info	does not exist, error!

Implementation



What do the following expressions evaluate to?

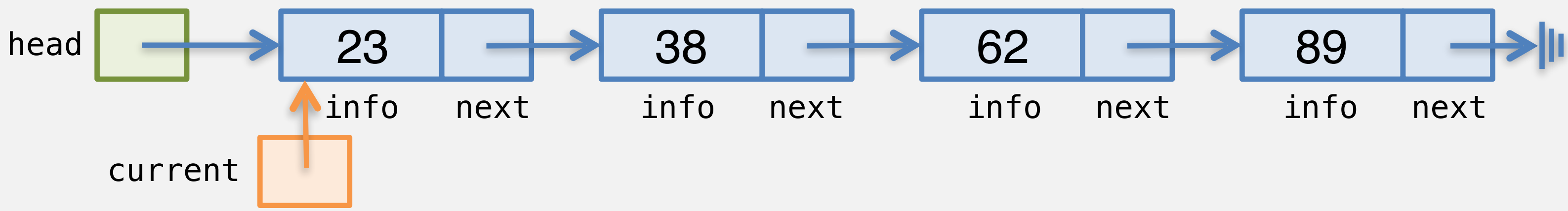
current	address of the 2nd node of the list
current->info	38
current->next	address of the 3nd node
current->next->info	62
current->next->next->info	89
current->next->next->next	0
current->next->next->next->next	does not exist, error!

A question: how may we move the `current` pointer to point to the previous node?

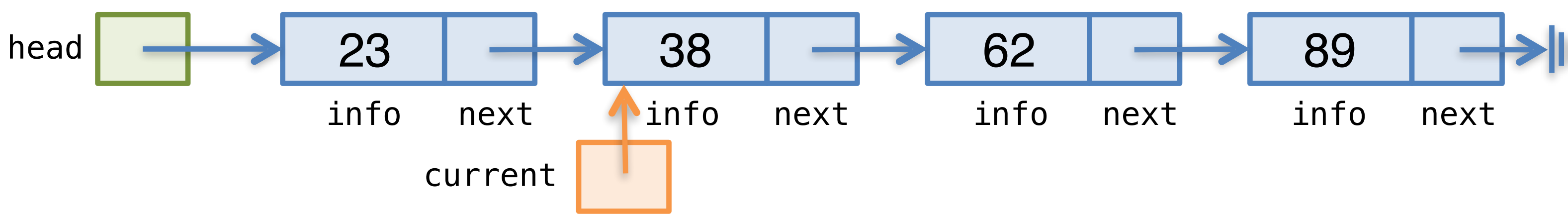
Traversing a Linked List

- Traversing: to go through the nodes in a linked list one-by-one, starting from the first node.

```
Node * current = head;
```

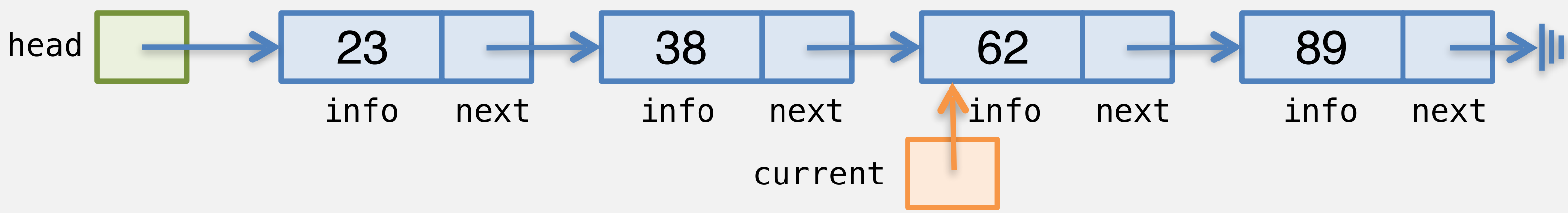



```
current = current->next;
```



Traversing a Linked List

```
current = current->next;
```



By advancing `current` to the next node repeatedly in this way, we may visit the nodes in the link list one by one.

How do we know the end of the linked list is reached and stop advancing to the next?

A standard while loop for traversing a linked list

```
Node * current = head;
while (current != NULL)
{
    // process the current node, e.g., print the content
    current = current->next;
}
```

print_list() function in build_list_backward.cpp

Traversing a Linked List

Why not traversing a list using the **head** pointer?

```
while (head != NULL)
{
    cout << head->info << endl;
    head = head->next;
}
```

NO!!! You should never do this.

If you modify the head pointer, the first node and therefore the entire linked list will be **lost**.

What happens if we forgot to advance the `current` pointer?

```
while (current != NULL)
{
    cout << current->info << endl;
}
```

This will go into an infinite loop, since current will never be equal to NULL, unless head points to an empty linked list initially.

Building a Linked List

- Starting from an empty list, new nodes may be created and inserted into the linked list.
- To build a linked list in a forward manner:
  - Always insert a new node at the end of the linked list
- To build a linked list in a backward manner:
  - Always insert a new node at the beginning of the linked list

We start by defining an **empty list**, i.e., a list without any node

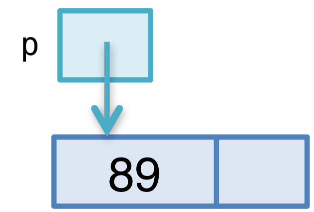
```
Node * head = NULL;
```

Building a Linked List Backward

- We now build a linked list in a backward manner by always inserting a new node at the beginning of the list.

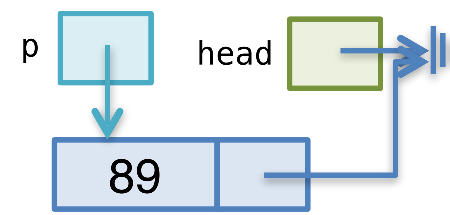
1. Create a new node and fill in the required info

```
Node * p = new Node;
p->info = 89;
```



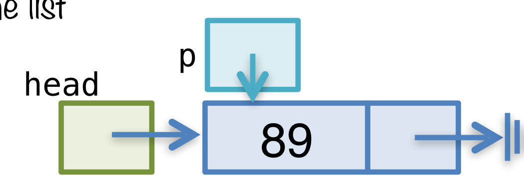
2. Have the `next` pointer of the new node points to the beginning of the list

```
p->next = head;
```



3. Update the head pointer to point to the new node, i.e., the new head of the list

```
head = p;
```



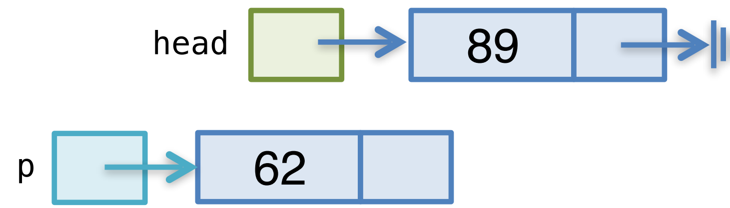
Now we have a list with one node.

Building a Linked List Backward

- Repeating the steps to insert one more node at the beginning:

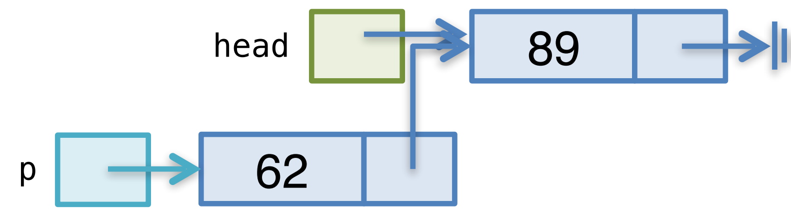
1.

```
Node * p = new Node;
p->info = 62;
```



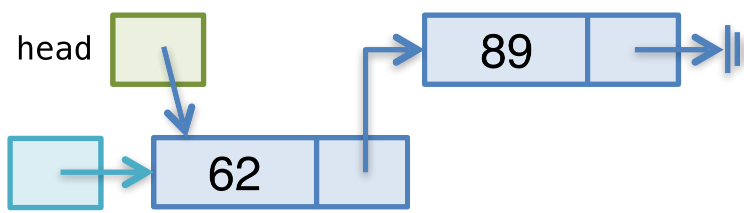
2.

```
p->next = head;
```



3.

```
head = p;
```



Now we have a list with two nodes.

Building a Linked List Backward

- Example: Suppose we want to build a linked list of numbers input by the user until he enters -999.

```
Node * head = NULL;
int num = 0;

cin >> num;
while ( num != -999 ) {
    head_insert(head, num);
    cin >> num;
}
```

build_list_backward.cpp

```
void head_insert(Node * & head, int num)
{
    Node * p = new Node;
    p->info = num;
    p->next = head;
    head = p;
}
```

The `head` pointer needs to be updated and hence is passed by reference

Building a Linked List Backward

- Example: Suppose we want to build a linked list of numbers input by the user until he enters -999.

Screen output

```
input integers (-999 to end): 23 56 14 45 98 -999

98 -> 45 -> 14 -> 56 -> 23 -> NULL
```

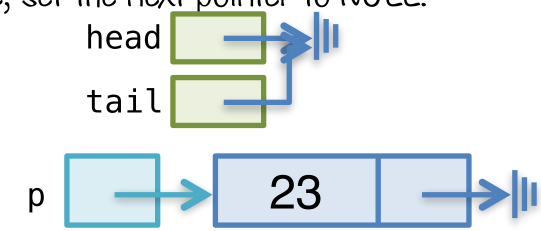
build_list_backward.cpp

Building a Linked List Forward

- To build a linked list in a forward manner, we always insert a new node at the end of the list.

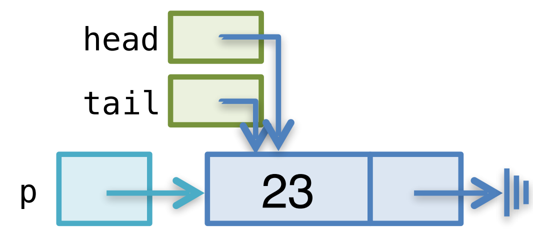
1. Create a new node and fill in the required info. Since this will be the last node, set the next pointer to NULL.

```
Node * p = new Node;
p->info = 23;
p->next = NULL;
```



2. If this is going to be the first node of the list, we point both head and last to it.

```
head =  p;
tail = p;
```



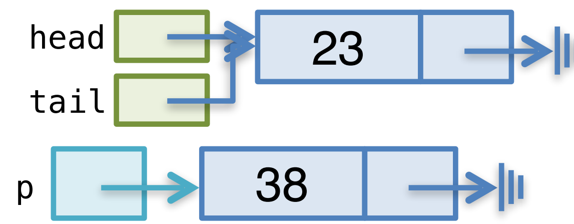
Since a new node is always inserted at the end, we need to maintain where the last node is using the pointer `tail`.

Building a Linked List Forward

- Repeating the steps to insert one more node at the end:

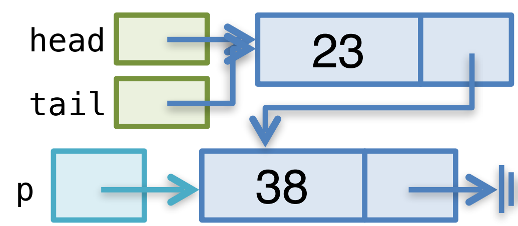
1. Create a new node and fill in the required info. Since this will be the last node, set the next pointer to NULL.

```
Node * p = new Node;
p->info = 23;
p->next = NULL;
```



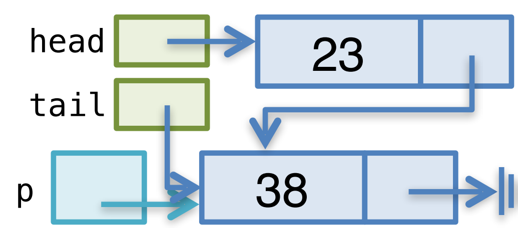
2. If this is **NOT** going to be the first node of the list, we link the last node of the list to the new node:

```
tail->next = p;
```



and set the last pointer to point to the new node:

```
tail = p;
```



Building a Linked List Forward

- Example: Suppose we want to build a linked list of numbers input by the user until he enters -999.

```
Node * head = NULL, * tail = NULL;
int num = 0;

cin >> num;
while ( num != -999 ) {
    tail_insert(head, tail, num);
    cin >> num;
}
```

build_list_forward.cpp

```
void tail_insert(Node * & head,
    Node * & tail, int num)
{
    Node * p = new Node;
    p->info = num;
    p->next = NULL;

    if (head == NULL) {
        head = p;
        tail = p;
    }
    else {
        tail->next = p;
        tail = p;
    }
}
```

Building a Linked List Forward

- Example: Suppose we want to build a linked list of numbers input by the user until he enters -999.

Screen output

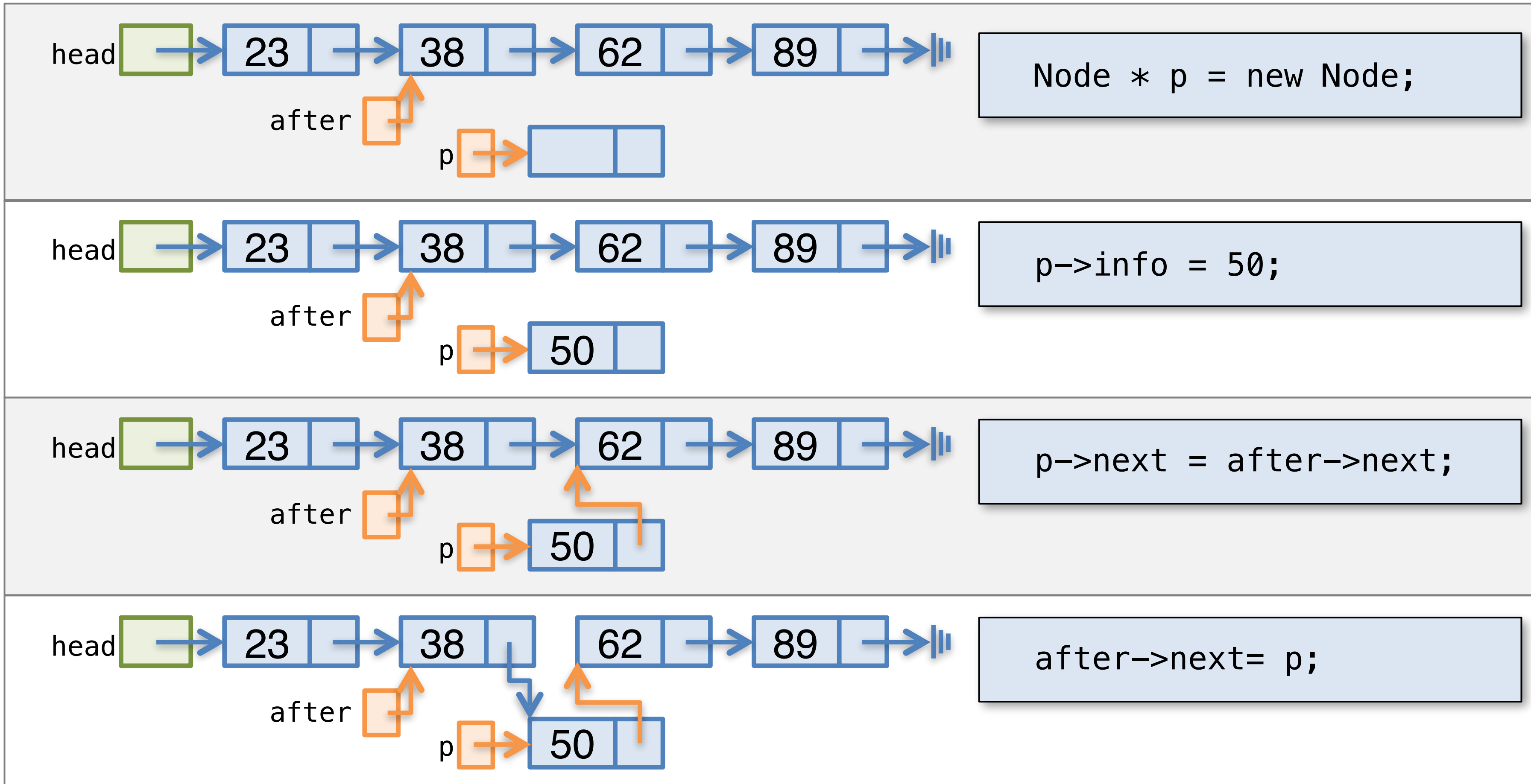
```
input integers (-999 to end): 23 56 14 45 98 -999

23 -> 56 -> 14 -> 45 -> 98 -> NULL
```

build_list_forward.cpp

Compare this with the output of build_list_backward.cpp

Inserting a Node

- Suppose the pointer `after` points to the node 38, and we want to insert 43 after it.



Inserting a Node

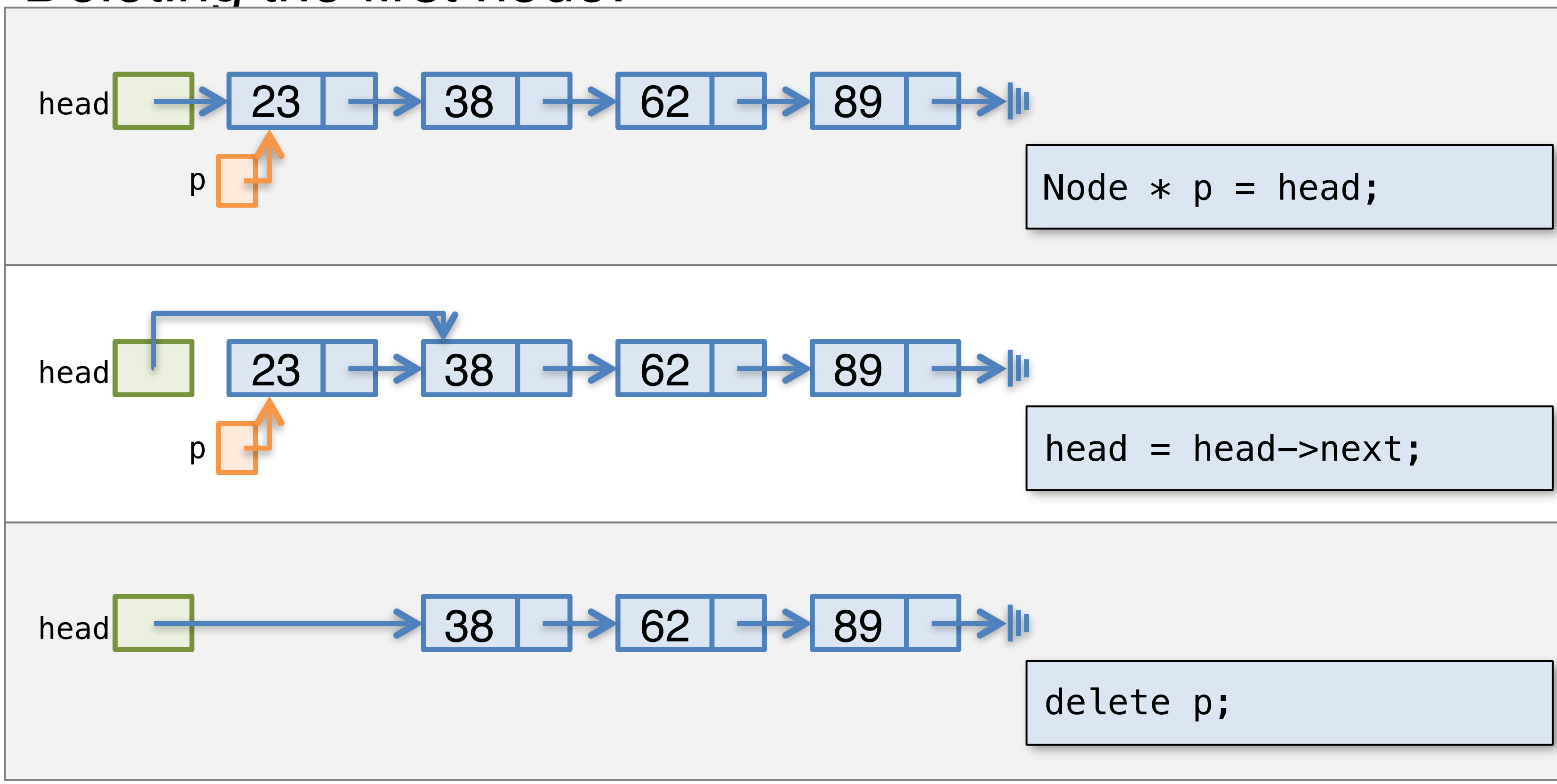
- A function to insert a number after the node pointed to by `after` in a linked list

```
// assume that after points to a node
// i.e., after not equals null
void insert( Node * after, int num )
{
    Node * p = new Node;
    p->info = num;
    p->next= after->next;
    after->next = p;
}
```

build_list_sorted.cpp

Deleting a Node

- Deleting the first node:



Deleting a Node

- A function to delete the first node in a linked list:

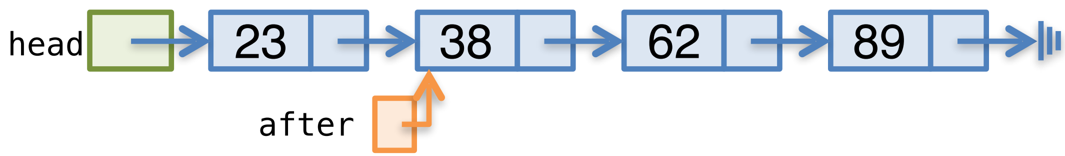
```
void delete_head( Node * & head)
{
    if (head != NULL) {
        Node * p = head;
        head = head->next;
        delete p;
    }
}
```

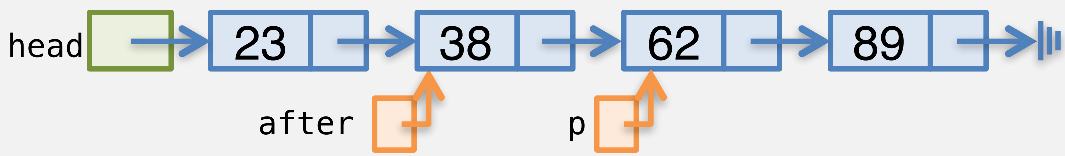
Make sure the list is not empty

build_list_sorted.cpp

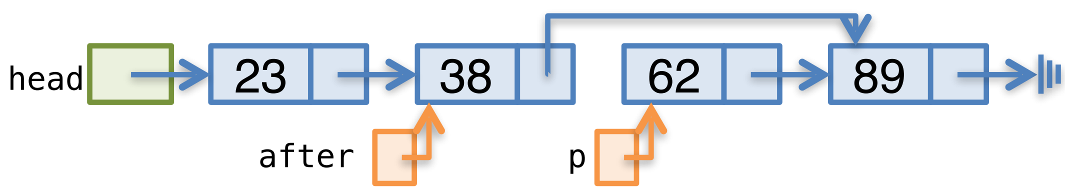
Deleting a Node

- To remove a node after the node pointed to by `after`, i.e. the node with number 62

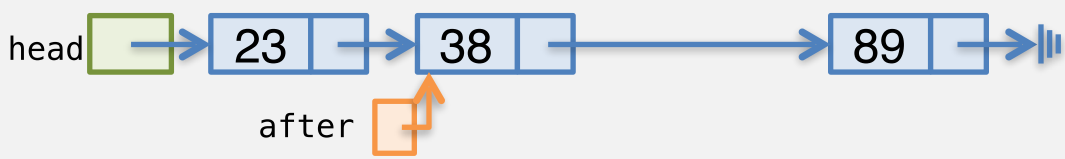





```
Node * p = after->next;
```




```
after->next = p->next;
```




```
delete p;
```

Deleting a Node

- A function to delete a number after the node pointed to by `after` in a linked list

```
// assume that after points to a node and
is // i.e., after not equals null
void delete_node( Node * after)
{
    Node * p = after->next;
    after->next = p->next;
    delete p;
}
```

build_list_sorted.cpp

Searching for a Node

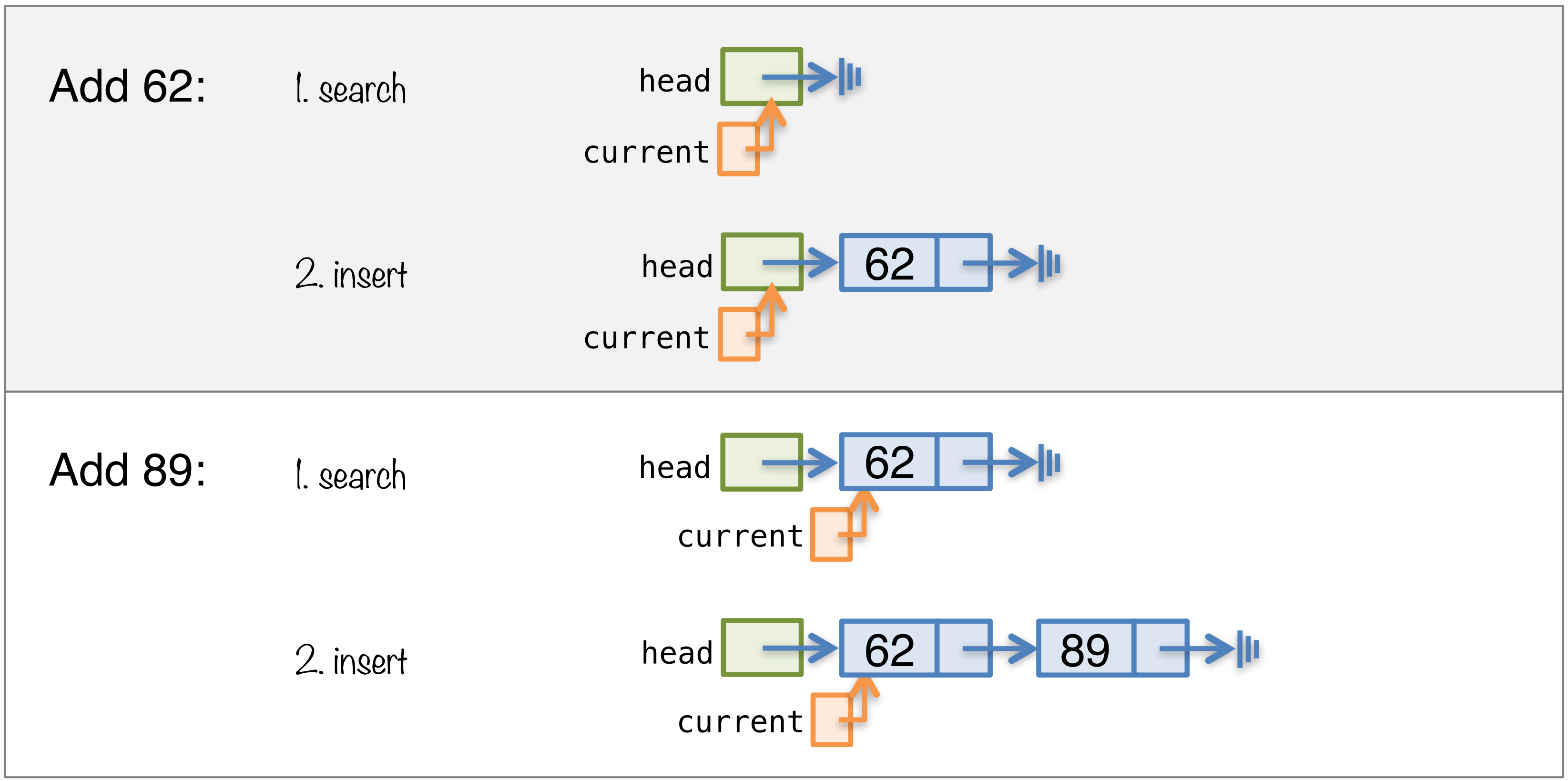
- To search for an item in a linked list is similar to traversing a list:
  - starting from the first node, we go through the items one by one
  - return the pointer to a found item, if found; or
  - ret m is not found

```
Node * find( Node * head, int num )
{
    Node * current = head;

    while (current != NULL) {
        if (current->info == num)
            return current;
        else
            current = current->next;
    }

    return NULL;
}
```

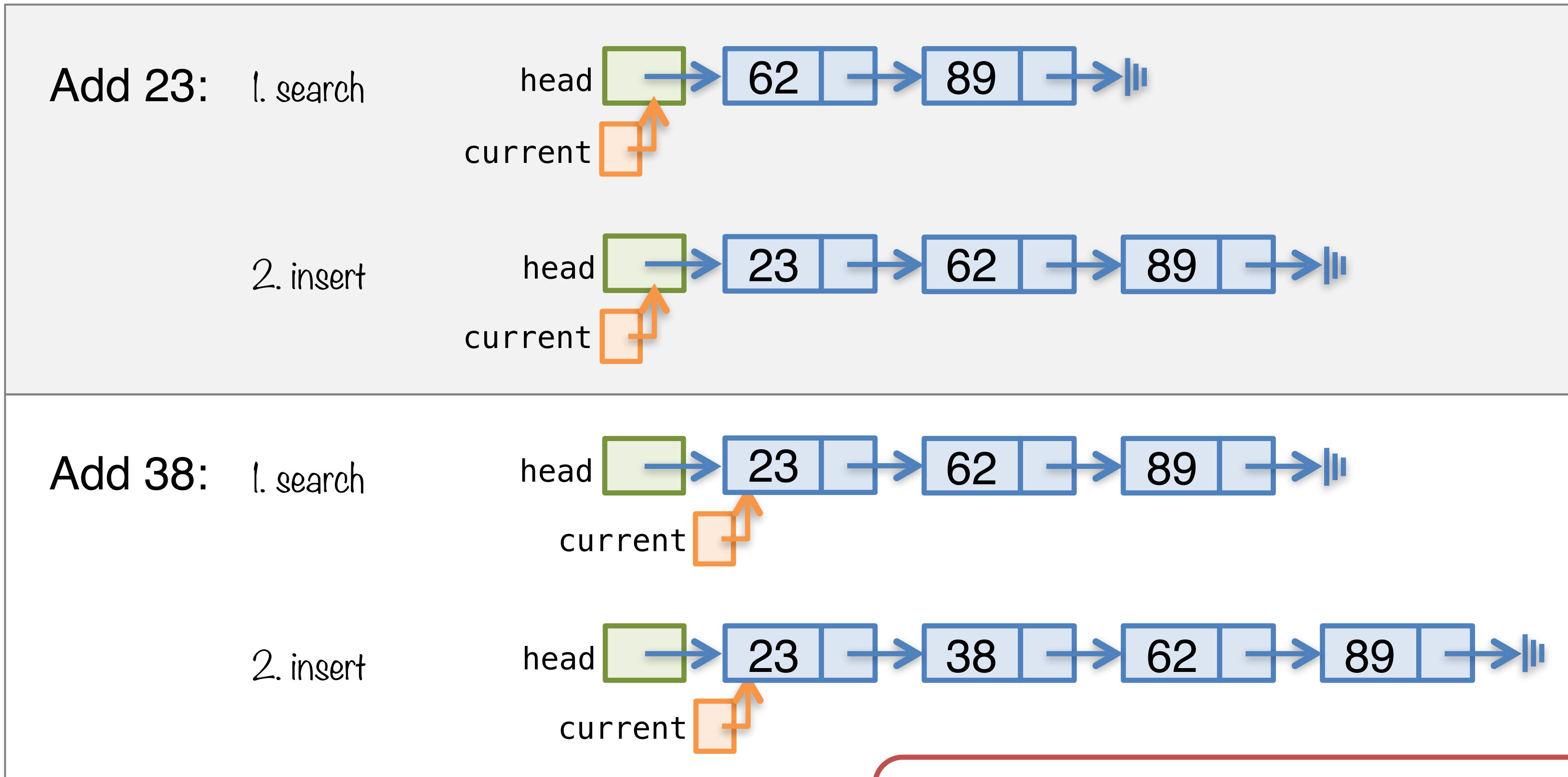
build_list_sorted.cpp

Building a Sorted Linked List

- To build a sorted linked list in which the items are always maintained in order, we need to search for an appropriate location to insert before adding any new item to the list



Building a Sorted Linked List



Note that `current` should always point to the previous node of where the new node is supposed to be

Building a Sorted Linked List

```
// return the node which is the last one in
// the list that is smaller than num
Node * find_prev( Node * head, int num )
{
    if (head == NULL || head->info >= num)
        return NULL;

    // at least one node in the list now
    Node * current = head;

    while (current->next != NULL) {
        if (current->next->info >= num)
            return current;
        else
            current = current->next;
    }

    return current;
}
```

build_list_sorted.cpp

Return `NULL` if the list is empty or the first item is not smaller than `num`

Compare the next item with `num`, `>=` makes sure that all items after `current` is larger than `num`

Execution reaches this point only when `num` is larger than all the existing items in the list

Compare this with the `find()` function

Building a Sorted Linked List

```
Node * head = NULL, * after_this;
int num = 0;

cin >> num;
while ( num != -999 ) {
    after_this = find_prev(head, num);

    if (after_this == NULL)
        head_insert(head, num);
    else
        insert(after_this, num);
    cin >> num;
}
```

The comparison in `find_prev()` determines whether the resulting list is in increasing order or in decreasing order

build_list_sorted.cpp

Deleting an Entire List

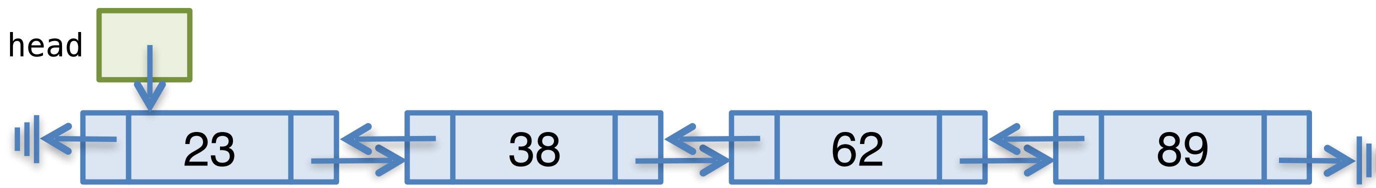
- To delete an entire linked list, we may iteratively delete the head node from it.

```
void delete_list(Node * & head)
{
    while ( head != NULL ) {
        delete_head(head);
    }
}
```

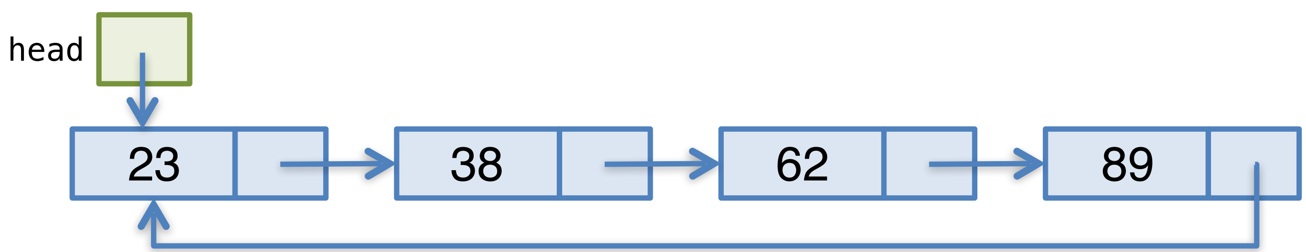
build_list_sorted.cpp

Variations of Linked Lists

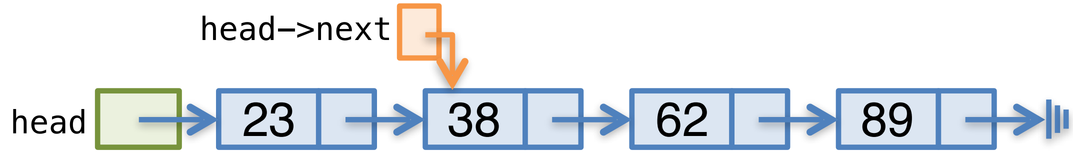
- Doubly-linked list



- Circularly-linked list



Printing a Linked List in Reverse
using Recursion



Recursive algorithm

To print a linked list pointed to by head in reverse

1. If linked list is empty, print nothing.
2. Otherwise,
   a) Print the linked list pointed to by head->next in reverse
   b) Print the node pointed to by head

```
void print_list_reverse(Node * head)
{
     if (head == NULL)
          cout << "NULL" << endl;

     else {
          print_list_reverse( head->next );
          cout << " <- " << head->info;
     }
}
```

print_list_reverse.cpp

Compare this to the iterative function for traversing a linked list [here](#).

Exercise 3

How to sort a linked list?

Idea: remove a node from the given list one by one, and built a new sorted linked list. You should have all the functions ready from the previous discussions.

Change the program build_list_backward.cpp and build_list_forward.cpp so that after a list is built, sort the list and output the contents

Exercise 4

Add a function `reverse()` to `build_list_sorted.cpp` to reverse a linked list. Add a user option in the main function to test this new function. A sample call of the function is (where head is the pointer to the first node of a linked list:

```
reverse(head);
```

Solution: ex4ex5.cpp

Exercise 5

Add a function `get_item()` to `build_list_sorted.cpp` to return the pointer to the kth item in the linked list. If no such item exists, return `NULL`. Add a user option in the main function to test this new function. A sample call of the function is:

```
Node * p = get_item(head, k);
if (p != NULL)
    cout << p->info << endl;
else
    cout << "Item does not exist." << endl;
```

Solution: ex4ex5.cpp

Exercise 6

Add a function to `build_list_forward.cpp` to divide the linked list into two sublists of almost equal sizes. For example, if a list points to the elements `1 -> 2 -> 3 -> 4 -> 5 -> NULL`, after division, the first list should be `1 -> 2 -> 3 -> NULL` and the second list should be `4 -> 5 -> NULL`. Modify the main function to call this new function and print out the two resulting lists. A sample call of the function is:

```
divide(head, second);
```

where `head` points to a linked list to be divided, and after completion of the function, `head` points to the first sublist, and `second` points to the second sublist.

Solution: ex6.cpp

Tutorial Problems – Linked Lists

LARGE NUMBERS

Large Numbers

- The largest integer that can be stored using a 32-bit **`int`** data type is 2,147,483,647.
- We are going to implement a linked list to store an arbitrarily large number. Making use of the linked list data structure, we write a program to determine if a larger number is bigger than the other one.

The two input numbers are separated by " > "

Sample output (user input in orange):

```
expr> 379821468310123801270301238974908123098 > 232378221392038248429490840198341389
Yes, 379821468310123801270301238974908123098 is larger.
```

```
expr> 232378221392038248429490840198341389 > 379821468310123801270301238974908123098
No, 379821468310123801270301238974908123098 is not larger.
```

- A template program largenum_incomplete.cpp is provided to you.

largenum.cpp provides the completed version of this tutorial problem.
You may compile and run it to see the expected results first.

Dynamic Array Input

- We obtain a large number from the user input as an arbitrarily long string of digits. Hence, we need an array that can grow its capacity dynamically to store the arbitrarily long string.
- The following helper function has been written for you to read in a long string of digits and store it in a dynamic arra

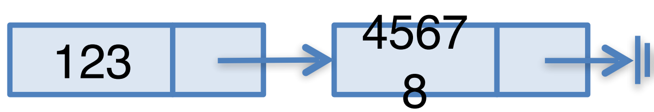
```
// get a number from a user
// by reading character by character until a space is hit
// use dynamic array to store the digits
// digits:  character array that stores the digits of the number
// numDigits: number of digits read from input
void input_num(char * & digits, int & numDigits);
```

Linked Lists for Large Numbers

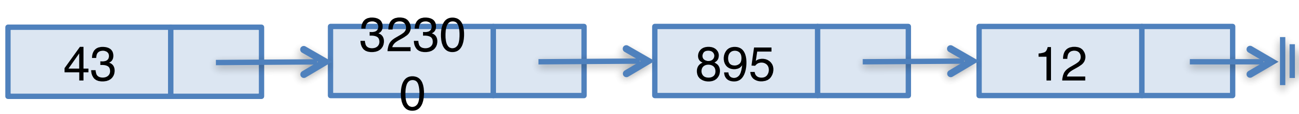
- A large number is segmented into chunks of 5 digits, starting from the least significant digit. The value of each chunk is then stored in a node of a linked list.

Example:

For the number 12345678, the linked list looks like:



For the number 4332300008950012, the linked looks like:



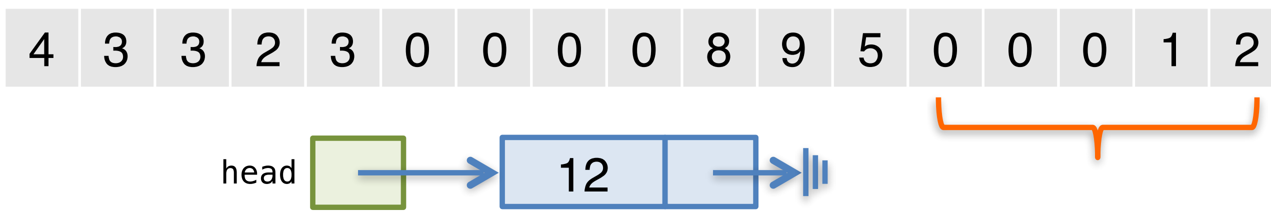
- We define a node structure as:

```
struct Node
{
     int value;
     Node * next;
};
```

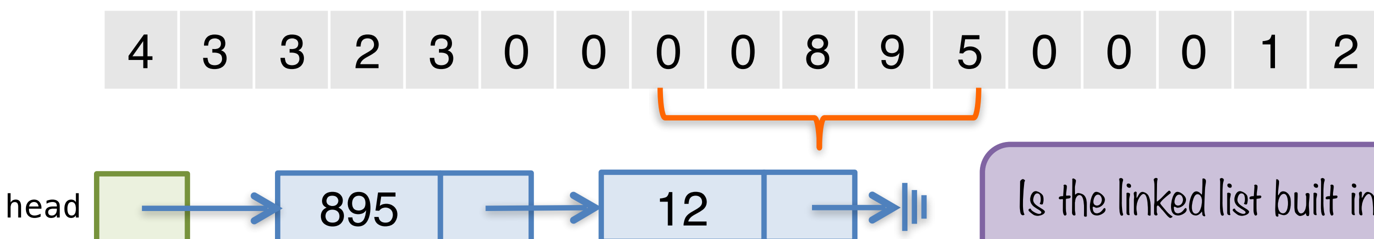
Linked Lists for Large Numbers

How to build such a list?

Scan the array of digits in reverse, and create a node for every 5 digits.

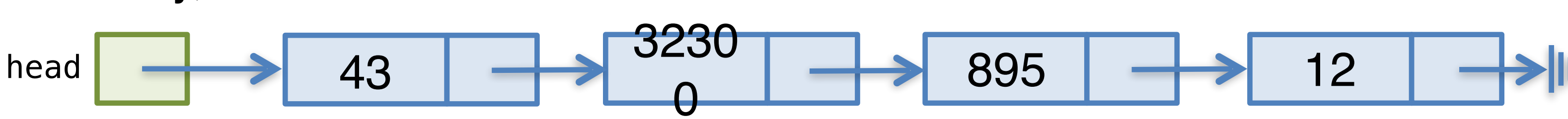


Create a node for the next 5 digits and insert it to the head of the linked list.



Is the linked list built in a forward or a backward manner?

Eventually, the linked list is built:



- Now, we write a function **`Node * create_num_list()`** to create a linked list for a number:

TASK 1a: Declare a pointer pointing to the head node of the linked list and initialize it as a null pointer

```
// get a large integer from user input
// and store in a linked list of Node
// each node stores the value of a trunk of 5 digits
// e.g., if the input is 43323000089500012, the linked list is
// 43 -> 32300 -> 895 -> 12 -> NULL
//
Node * create_num_list()
{
    // TASK 1a: declare a pointer pointing to the head of the link list



    string str;
    char * digits = NULL;  // a dynamic array for storing an input number
    int numDigits;
    int val;

    // get a number from the user
    input_num( digits, numDigits);

    ...
}
```

We call **`input_num()`** here to get a number from the user and store it in a dynamic array

- Still working in **`Node * create_num_list()`** …

```
Node * create_num_list()
{
    ...

    // scan the digits in reverse, and create a list of nodes for
    // the value of every 5 digits
    str.clear();
    for (int i = numDigits-1; i >=0; --i) {
        str = digits[i] + str;
        if (str.length()==5) {
            val = atoi(str.c_str());

            // TASK 1b: insert a value as a node to the head of the list


            str.clear();
        }
    }
    ...
}
```

scan the digits in reverse from the array

compose a substring for the digits

for every 5 digits scanned,

take the value of the string (`atoi` converts a `C-string` to an `int`)

TASK 1b: insert a value to the head of list.
Copy and modify a function that you learned in class for this.

- We are not done with **Node * create_num_list()** yet…

```
Node * create_num_list()
{
    ...

    // the digits array is scanned and there are still digits
    // stored in str that are not inserted into the list yet
    if (!str.empty()) {
        val = atoi(str.c_str());

        // TASK 1c: insert a value as a node to the head of the linked list


    }

    if (digits != NULL) {
        delete [] digits;
    }

    // TASK 1d: return the pointer to the linked list


}
```

After the execution of the preceding for loop, there may still be digits that are not inserted into the linked list (think of the most significant digits "43" in slide #6)

take the value of the residual string

TASK 1c: insert the value to the head of list. **This is essentially the same statement that you write for TASK 1b.**

free the dynamic array storing the digits

TASK 1d: return the pointer to the linked list

Checking: Printing the Linked Lists

- Now, check that the linked lists are correctly built by calling the **`void print_list(Node * )`** function.

```
int main()
{
    Node * n1, * n2; // linked lists for large numbers

    cout << "expr> ";
    n1 = create_num_list();
    cin.get();          // skip the '>' sign
    cin.get();          // the space after the '>' sign
    n2 = create_num_list();

    // TASK 2: call print_list() on n1 and n2 for checking


    ...
}
```

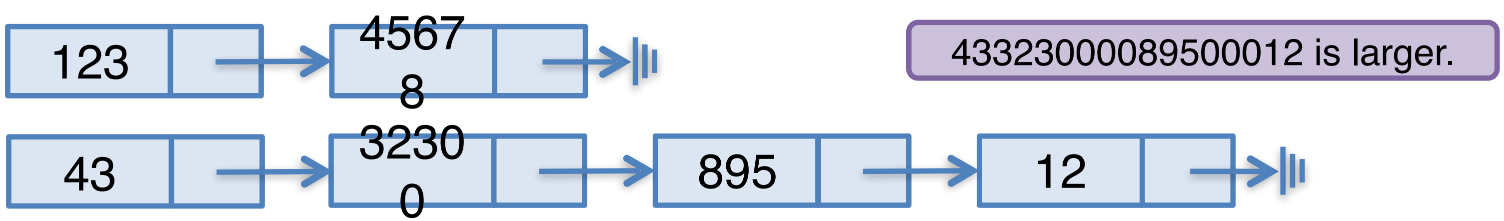
Pointers pointing to two linked lists

Get input numbers and create linked lists

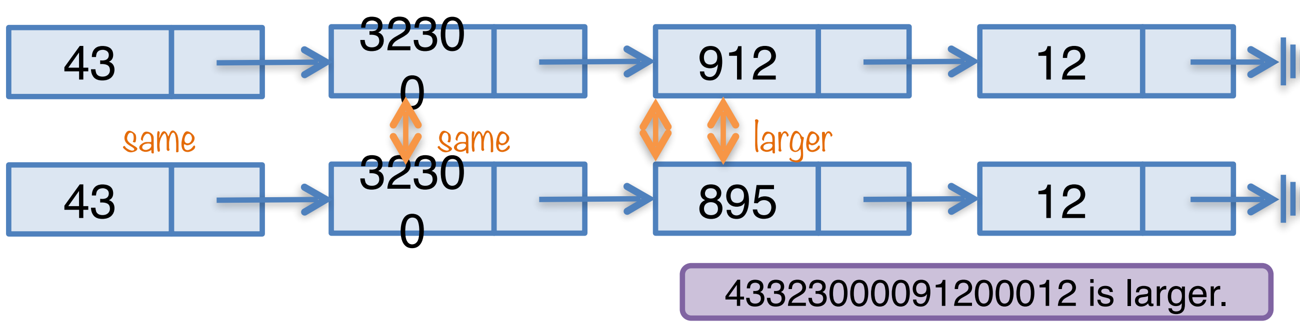
TASK 2: Call
`void print_list(Node *)`
in the main function and check if the linked lists are correct.

Comparing Two Large Numbers

- Now that we have two linked lists representing two numbers, we can compare which one is larger.
- A number n1 is larger than a number n2, if
  - the linked list of n1 is longer than the linked list of n2



  - or, the length of the linked lists are the same, and if we compare the nodes of the two linked lists starting from the first nodes in parallel, we should encounter a pair of nodes such that the value of the node in n1 is larger than the value of the node in n2



- We need a function to determine the length of a linked list

```
// return the length of a linked list
int list_length(Node * head)
{
    // TASK 3: Modify this print function to one that
    // count the number of nodes in a linked list

    Node * current = head;
    while (current != NULL)
    {
        // process the current node, e.g., print the content
        cout << current->value << " -> ";
        current = current->next;
    }
    cout << "NULL\n";
}
```

TASK 3: This is a print list function.
Modify it so that it counts the number of nodes in a linked list

- Next, we need a function to determine if a number is larger than another.

```
// return if the number n1 is larger than n2
bool larger(Node * n1, Node * n2)
{
    int len1 = list_length(n1);
    int len2 = list_length(n2);

    // TASK 4a: handle the case
    // when the list lengths are different



    // the two lists are of equal length

    Node * curr1 = n1, * curr2 = n2;

    while (curr1 != NULL) {
        if (curr1->value > curr2->value)
            return true;

        // TASK 4b: advance curr1, curr2
        // to point to the next nodes

    }

    return false;
}
```

TASK 4a: What should we do if the linked list for **n1** is longer? What if otherwise?

Compare the values of the corresponding nodes in n1 and n2 starting from the most significant values. The number n1 is larger if the value of its node is larger than its corresponding node in n2.

TASK 4b: advance the current pointers

Putting All Together in main()

```
int main()
{
    Node * n1, * n2;

    cout << "expr> ";
    n1 = create_num_list();
    cin.get();          // skip the '>' sign
    cin.get();          // the space after the '>' sign
    n2 = create_num_list();

    // TASK 2: call print_list() on n1 and n2 for checking

    if (larger(n1, n2)) {
        cout << "Yes, ";
        print_num(n1);
        cout << " is larger." << endl;
    }
    else {
        cout << "No, ";
        print_num(n1);
        cout << " is not larger." << endl;
    }

    // TASK 5: free the linked lists


    return 0;
}
```

Pointers pointing to two linked lists

Get input numbers and create linked lists

Output appropriate message depending on whether **n1** is bigger than **n2**.

`print_num()` outputs the number stored in the linked list in the ordinary number format

TASK 5: Free the linked lists by calling **delete_list()**

PROBLEMS

Problem 1

What is the output of the following program? Explain.

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4      int i1, i2;
5      int *p1 = &i1, *p2 = &i2;
6      *p1 = 10;
7      *p2 = 20;
8      cout << *p1 << " " << *p2 << endl;
9      p1 = p2;
10     cout << *p1 << " " << *p2 << endl;
11     *p1 = 30;
12     cout << *p1 << " " << *p2 << endl;
13 }
```

How would the output change if line 9 is replaced with *p1 = *p2? Explain.

Problem 2

What unfortunate misinterpretation can occur with the following declaration?

```
int* int_ptr1, int_ptr2;
```

Problem 3

Write a program that will first read in a positive integer N, and then N numbers provided by the user. Your program should store the numbers in a dynamically allocated array. Your program then prints the numbers in reverse.

Sample output (*user input is underlined*)

```
6
3.2 4.8 -1 9.2 5.5 -6
-6 5.5 9.2 -1 4.8 3.2
```

Problem 4

One problem with dynamically allocated arrays is that once the array is created using the new operator, the size cannot be changed. For example, you might want to add or delete entries from the array. This problem asks you to create functions that uses dynamically allocated arrays and allows add and deletion of entries to the array.

Complete the following two functions

```
string* addEntry (string *array, int &size, string newEntry);
```

This function should create a new dynamically allocated `array`, one element larger than `array`, copy all elements from array into the new array, add the new entry onto the end of the new array, increment `size`, delete `array`, and return the new dynamic array.

```
string* deleteEntry(string *array, int &size, string entryToDelete);
```

This function should search `array` for `entryToDelete`. If not found, the request should be ignored and the unmodified `array` returned. If found, create a new dynamically allocated array one element smaller than `array`. Copy all elements except `entryToDelete` into the new array, delete `array`, decrement `size`, and return the new dynamically allocated array.

Optional.

For those who would like to challenge yourselves.
Even for those of you who are beginners in C++ programming, it’s highly recommended for you to take a look at these problems and try to tackle them as well.

You are welcome to discuss these problems in the Moodle forum.

CHALLENGES

Challenge 1

This question is on file I/O and string manipulation.

Write a program to read a file and report the number of occurrences of each word in the file. Your program will not distinguish between upper- and lower-case words. E.g., “hello”, “Hello”, “HELLO” are the same. The words must be output in lowercase and sorted in ascending order of their frequencies. Words with same frequency are ordered lexicographically.

Example:

Contents of input file:

```
Hello, where are you going to?
Are you going to school to … say hello?
```

Your program output:

```
say 1
school 1
where 1
are 2
going 2
hello 2
you 2
to 3
```

Hint: By including `<cctype>` you can use the function `ispunct()` to test whether a character is a punctuation or not.