Lecture 5

- Dynamic data structures: singly-linked lists [13.1.1]
  (Enkellänkade listor) *– Lab2*

Info

- Lab lists have been digitalized
  - *http://www.itn.liu.se/~aidvi/courses/12/TNG033/Labs/Lab_groups.pdf*
- Dugga 1 -- posted through Lisam
  - Friday, 12:00 - Monday, 12:00
  - Instructions posted on the course web site before Friday
  - One source file (**`.cpp`**) for each exercise
  - Covers Fö 1 – Fö 5 + Lesson 1 + Lab 1
  - Correct output is <u>not</u> enough for **`G`**
    - Must show can use the concepts introduced in the course
  - Must follow the submission instructions
- Lesson 1 -- Wednesday
  - MT2a+MT2b, 10-12, K21, Aida Nordman
  - ED3, 10-12, TP44, Ehsan Miandji
  - Lesson 8-10 is cancelled!!

Allocation/Deallocation of Memory

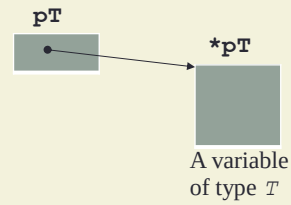
- To **allocate memory**
  - Reserve memory space for a variable
- To **deallocate memory**
  - Release memory space reserved for a variable
    - Released space can later be used for another variable
- Memory for variables can be allocated
  - **automatically** -- deallocated automatically
  - **explicitly (dynamically)** -- deallocated by explicit C++ instructions

Specific **C++** instructions are added to the program by the programmer to reserve (free) memory

Memory Allocation: **new**

Declarations of functions used to manage dynamic storage in C++

```
#include <new>
T *pT = new T;
*pT = …;
```



1. Allocate memory space for a variable of type *T*
2. Return the memory address of the first byte of the allocated memory, i.e. a pointer

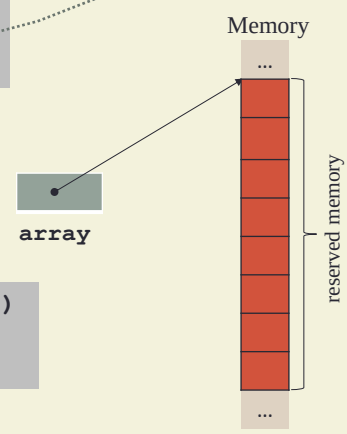
Note: There's no way to access the allocated memory, but through the pointer

- Sequences with variable length
  1. Ask the user how many (**howMany**) items to be stored in an array
  2. Allocate dynamically the memory for the array
     - Reserve space for an array with **howMany** slots

Memory Allocation: **new**

```
#include <new>
int howMany;
cout << "How many items: ";
cin >> howMany;
T *array = new T[howMany];
```

Allocate space for an array of *T*s



Read values and store them into the array

```
for(int i = 0; i < howMany; i++)
  cin >> array[i];
 //cin >> *(array+i)
```

Memory Deallocation: `delete`

```
#include <new>
T *pT = new T;
 …;
//*pT is no longer needed
delete pT;
```

Deallocate the memory pointed by pointer **pT**

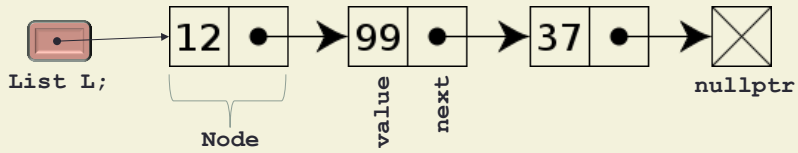
```
#include <new>
int howMany;
cout << "How many items: ";
cin >> howMany;
T *array = new T[howMany];
…
//*array is no longer needed
delete [] array;
```

Deallocate the memory for the array pointed by pointer **array**

Dynamic Data Structures

- Can grow and shrink depending on how many elements are stored
  - No need to ask in advance to the user how many items
  - New items can be added and removed one at time
- Use dynamic memory allocation
- *Lab 2* *(`List` with classes)*
- Basics for the **S**tandard **T**emplate **L**ibrary (**STL**)
- Used in future course **Data Structures, TND004**

Singly-linked lists



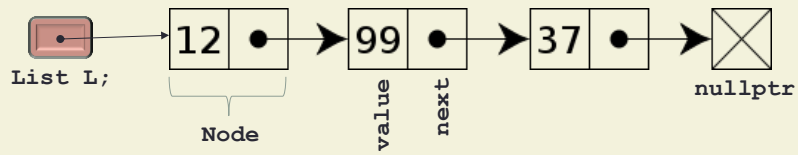

```
struct Node
{
   int value;
   Node* next;
};

typedef Node* List;
```

Pointer to the first node of a list

Empty List **L**

```
List L = nullptr;
//Node* L = nullptr;
```

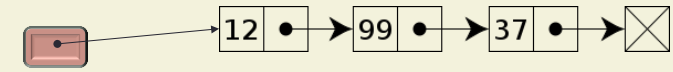

Singly-linked lists



Which operations should be available?

- **Insert** a new node
- **Remove** a node
- **Search** for a node

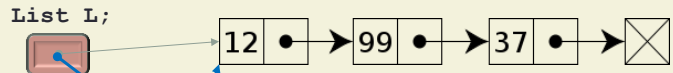
Insert a node as first node



List L;

5

Insert **5** at the beginning of the list



newNode

```
1. Node *newNode = new Node;
2. newNode->value = 5;
3. newNode->next = L;
4. L = newNode;
```

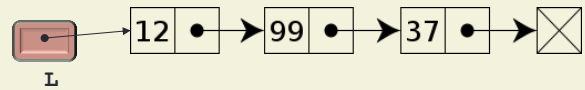
Do not swap lines 3. and 4.

Creating the list

```
void insert(List& l, int n)
{
    Node *newNode = new Node;
    newNode->value = n;
    newNode->next = l;
    l = newNode;
}
```

Insert a new node with value **n** in the beginning of the list

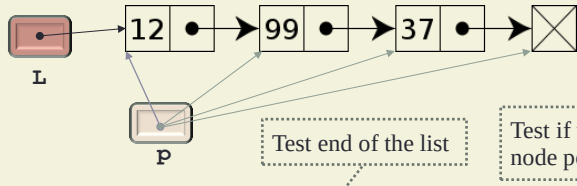
Why **List&** **l**?
Why not **List** **l**?



L

```
List L = nullptr; //empty list
insert(L,37);
insert(L,99);
insert(L,12);
```

Search a value **n**



Search for **5** in the list

Test end of the list

Test if value stored in the node pointed by **p** is **n**

```
Node *p = L;
while( (p != nullptr) && (p->value != n) )
       p = p->next;
if (!p) //if (p == nullptr)
  cout << "Not found!!";
else //p points to the node storing n
{
   …;
}
```

Important: do not swap the tests

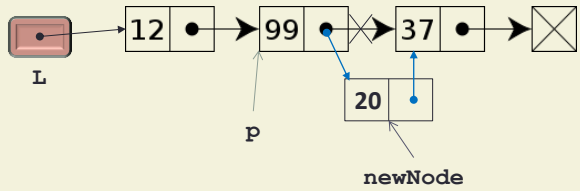
Testing if a pointer is **nullptr**

```
if (p != nullptr)
{
  //do someting with pointer p
}
```

nullptr is automotically converted to **false**
Any other pointer is converted to **true**

```
if (p)
{
  //do someting with pointer p
}
```

Insert after a node



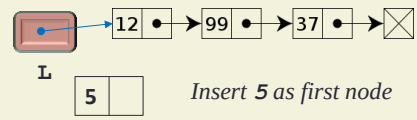
Insert 20 after 99

```
1. Node *newNode = new Node;
2. newNode->value = 20;
3. newNode->next = node->next;
4. p->next = newNode;
```

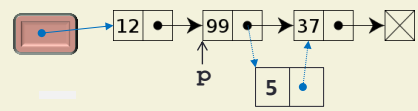
Lines 3 and 4 cannot be swapped

Insertion *(revisited)*

```
1. Node *newNode = new Node;
2. newNode->value = 5;
3. newNode->next = L;
4. L = newNode;
```



Insert 5 as first node

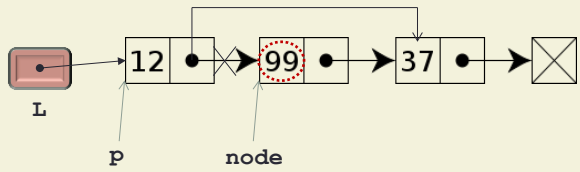


Insert 5 after 99

```
1. Node *newNode = new Node;
2. newNode->value = 37;
3. newNode->next = p->next;
4. p->next = newNode;
```

Inserting in the beginning of the list or inserting after another node requires different code

Remove a node



Remove **99**

```
1. //Make p point to the node before
   //the one to be removed
2. Node *node = p->next;  //node to remove
3. p->next = node->next;
4. delete node; //deallocate
```


Remove *(revisited)*

```
1. //Make p point to the node before
   //the one to be removed
2. Node *node = p->next;  //node to remove
3. p->next = node->next;
4. delete node; //deallocate
```

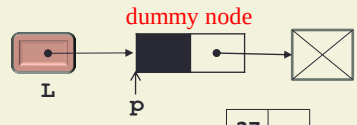
If node to be removed is not the first node

Remove first node

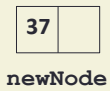
```
if ( !L && L->value == n )
{
  Node* node = L; //point to first node
  L = L->next;
  delete node; //delete first node
}
```

Lists with dummy nodes

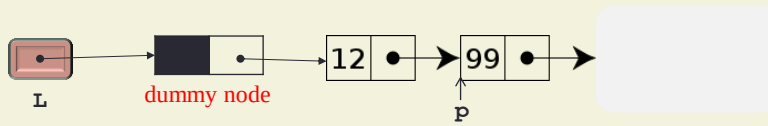
Empty list



Insert **37** as first node



Insert **37** after **99**




```
//insert 37
1. Node *newNode = new Node;
2. newNode->value = 37;
3. newNode->next = p->next;
4. p->next = newNode;
```

Dummy node avoids the special case of insertion (removal) in the beginning of the list, i.e. as first node

Every node has a node before it

Home work

Study carefully the code in the folders

- ***List1***: dummy nodes not used in the list's implementation
- ***List2***: dummy nodes used in the list's implementation
- Compare implementation of function **remove**

Preparation for Lab 2 -- uses classes

Read sec. 13.1.1

Next …

- Fö 6
  - References versus pointers [sec. 5.5.1]
  - Pointers to functions [sec. 5.4.7]
- Lesson 1 -- Wednesday
  - Do all exercises
  - If you do not try the exercises then there's no point to be at the lesson
  - Preparation for lab 2
- Lab 2 = singly linked list + classes