

Pointer

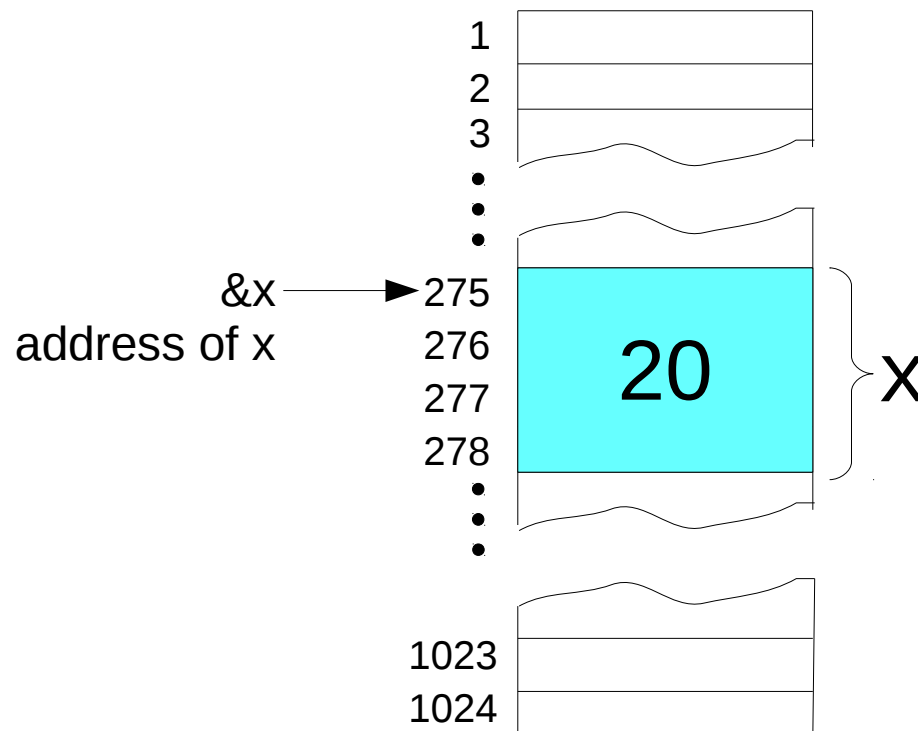
Memory of the computer is divided into bytes.
Each byte in memory has an address.

Suppose we declare a variable `int x`. (Assume that `sizeof(int)=4`)

- A block of **4 consecutive bytes are reserved** in memory which will be used to store the value of the variable `x`.
- The variable name `x` is associated with the starting address of the block. This address of the starting byte is called the **address of the variable `x`**.
Address of `x` is written as `&x` (In our example `&x` is 275)

```
int x;  
x = 20;  
  
cout << &x;
```

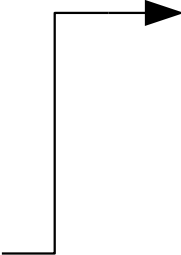
Output: 275



Let us define an integer variable

Suppose we want to store the address of **x** in another variable **a**

We need to first define the variable **a**



```
int x;  
int *a;  
a = &x;  
cout << a;
```

What should be the datatype of **a** ?

A variable which stores address of another variable is called a **pointer**

A pointer **a** which stores the address of **int** type variable is defined as,

int *a;

Datatype of **a** is **int ***

```
int x=20;
int *p;
p=&x;
```

Output: 275 20

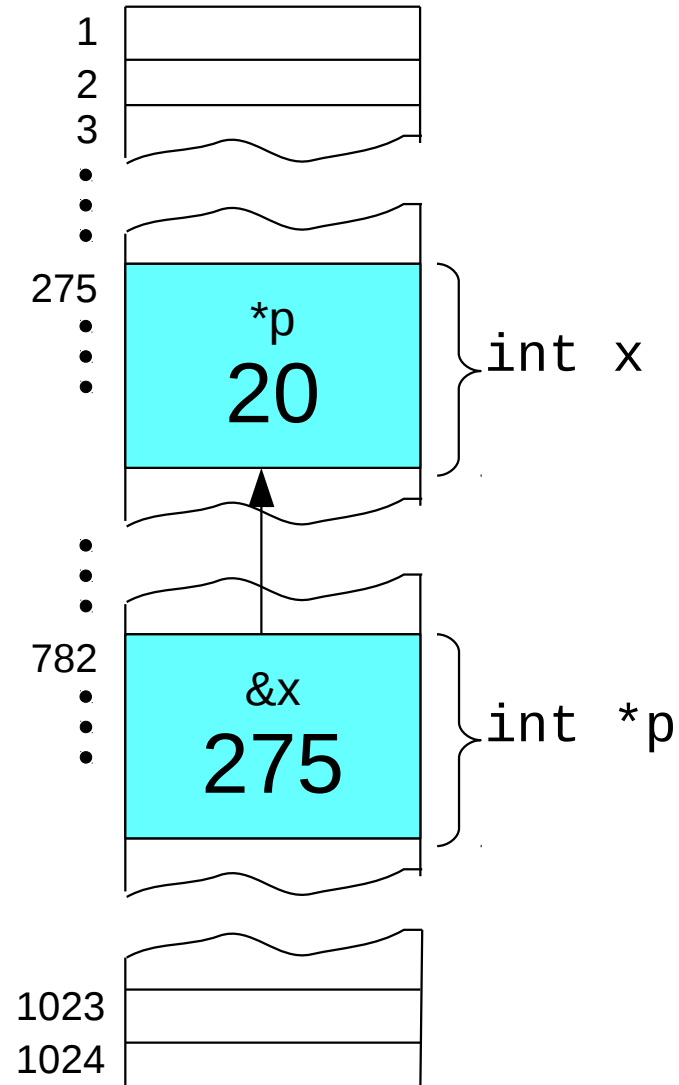
```
cout << p << " " << *p;
```



p is a pointer. So **p** stores an address
 Say **p** has a value α . So α is an address
 (in our example α is 275)
***p** denotes the value stored at address α

Datatype of **p** is **int ***

Datatype of ***p** is **int**



Call by Value

```
void f(int);

main()
{
    int x=20;
    f(x);
    cout<<x;
}

void f(int p)
{
    p=10;
}
```

Two different local variables- x in main() and p in f()

When we make the call f(x)

value of x is copied to p

p has the value 20

f() changes the value of p

x is undefined in f() so f() cannot access x

x remains unchanged

Output: 20

Call by Reference

```
void f(int *);

main()
{
    int x=20;
    f(&x);
    cout<<x;
}

void f(int *p)
{
    *p=10;
}
```

Two different local variables- x in main() and p in f()

When we make the call f(&x)

address of x is copied to p

p has the value &x

f() changes the value at location &x

x is undefined in f() but f() can modify the value at address &x.

x gets changed

Output: 10

Pointer to class and the -> operator

```
class student {  
public:  
    int roll;  
    float cpi;  
};
```

We use -> to access members from pointer to an object

```
student s, *p;  
s.roll = 16123011; s.cpi = 9.2;  
p = &s; //implies s is *p
```

These two statements are equivalent

```
cout << "Roll " << s.roll << " Cpi " << s.cpi << endl;  
cout << "Roll " << p->roll << " Cpi " << p->cpi << endl;
```

"this" Pointer

```
class student {  
public:  
    int roll;  
    float cpi;  
  
    void show(void)  
};
```

For every object a pointer named "**this**" is automatically defined which can be accessed by member functions.

"this" points to the object itself

These two are equivalent

```
void student::show(void)  
{  
    cout << "Roll " << roll << " Cpi " << cpi << endl;  
}
```

```
void student::show(void)  
{  
    cout << "Roll " << this->roll << " Cpi " << this->cpi << endl;  
}
```

We can assign a *special value* **NULL** to any pointer

It means the pointer is not holding any valid address
i.e. it is not pointing to anything

```
int *p;  
p = NULL;  
cout << p;
```

Output: 0x0

Reference Variable

Reference Variable

```
int x=10;
```

```
int &r = x;
```

```
r = 20;
```

```
cout << x;
```

Output: 20

r is a **reference** to x

r is **NOT** a **separate variable**

r is **just another name** for x

No difference between

r=20 and x=20

Type of r is int &

Reference Variable

Meaning of these
two '=' signs are completely different

`int &r=x` **creates a new name r** for x

`r=20` assigns a value to r
(which is now same as saying `x=20`)

```
int x=10;
```

```
int &r = x;
```

```
r = 20;
```

```
cout << x;
```

Output: 20

r is a **reference** to x
r is **NOT** a separate variable
r is **just another name** for x
No difference between
r=20 and x=20

Type of r is int &

This **&** is **NOT** the address operator
used with pointers.

Reference Variable

`int &r=x;` creates a new name `r` for `x`

It is a single operation.
It **cannot be broken down into**

`int &r;` **Error**
`r=x;`

`int x=10;`

`int &r = x;`

`r = 20;`

`cout << x;`

Output: 20

`r` is a **reference** to `x`
`r` is **NOT** a separate variable
`r` is **just another name** for `x`
No difference between
`r=20` and `x=20`

Type of `r` is **int &**

This **&** is **NOT** the address operator
used with pointers.

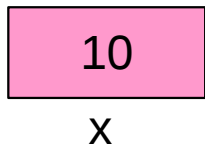
Reference Variable

```
➔ int x=10;  
   int &r = x;
```

```
   r = 20;  
   cout << x;
```

Output: 20

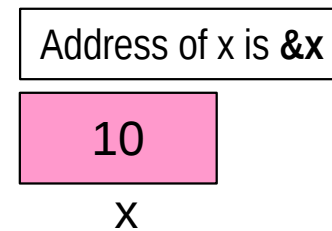
r is a reference to x
r is not a separate variable
r is just another name for x
No difference between
r=20 and x=20



Pointer

```
➔ int x=10;  
   int *p;  
   p = &x;  
   *p = 20;  
   cout << x;
```

Output: 20



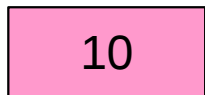
Reference Variable

```
int x=10;  
➔ int &r = x;
```

```
r = 20;  
cout << x;
```

Output: 20

r is a reference to x
r is not a separate variable
r is just another name for x
No difference between
r=20 and x=20



x

r (r and x are same now)

Pointer

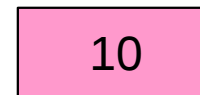
```
int x=10;  
➔ int *p;  
p = &x;  
*p = 20;  
cout << x;
```

Output: 20



p

Address of x is &x



x

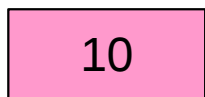
Reference Variable

```
int x=10;  
➔ int &r = x;
```

```
r = 20;  
cout << x;
```

Output: 20

r is a reference to x
r is not a separate variable
r is just another name for x
No difference between
r=20 and x=20



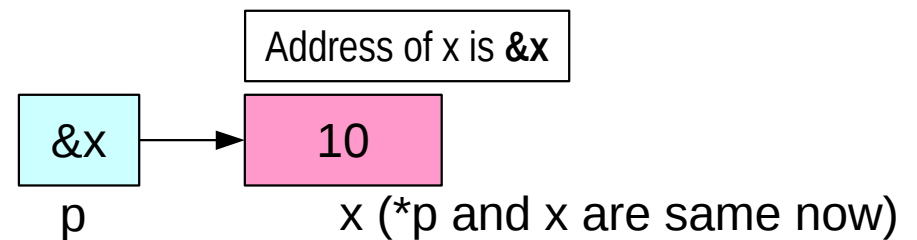
x

r (r and x are same now)

Pointer

```
int x=10;  
int *p;  
➔ p = &x;  
*p = 20;  
cout << x;
```

Output: 20



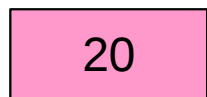
Reference Variable

```
int x=10;  
int &r = x;
```

```
➔ r = 20;  
cout << x;
```

Output: 20

r is a reference to x
r is not a separate variable
r is just another name for x
No difference between
r=20 and x=20



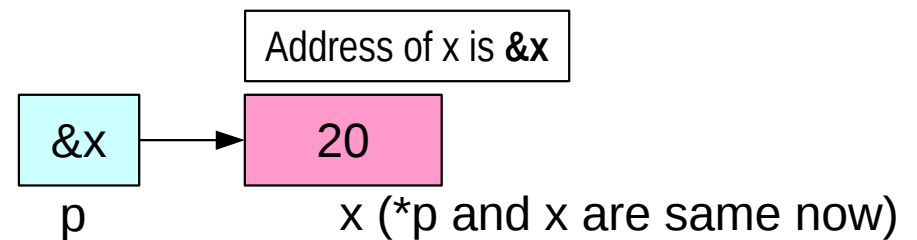
x

r (r and x are same now)

Pointer

```
int x=10;  
int *p;  
p = &x;  
➔ *p = 20;  
cout << x;
```

Output: 20



Reference Variable

```
int x=10,y=40;
```

```
int &r = x;  
r = 20;
```

```
r = y;  
r = 30;
```

```
cout<< x <<endl<< y;
```

What is the output ?

Pointer

```
int x=10,y=40;
```

```
int *p = &x;  
*p = 20;
```

```
p = &y;  
*p = 30;
```

```
cout<< x <<endl<< y;
```

What is the output ?

Reference Variable

➔ `int x=10,y=40;`

```
int &r = x;  
r = 20;
```

```
r = y;  
r = 30;
```

```
cout<< x <<endl<< y;
```

10

x

40

y

Pointer

➔ `int x=10,y=40;`

```
int *p = &x;  
*p = 20;
```

```
p = &y;  
*p = 30;
```

```
cout<< x <<endl<< y;
```

Address of x is &x

10

x

Address of x is &y

40

y

Reference Variable

```
int x=10,y=40;
```

```
➔ int &r = x;  
  r = 20;
```

```
  r = y;  
  r = 30;
```

```
cout<< x <<endl<< y;
```

10

x

r (r and x are same now)

40

y

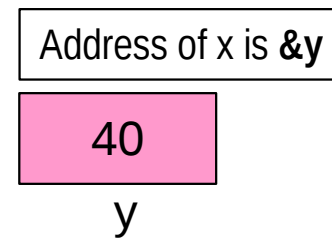
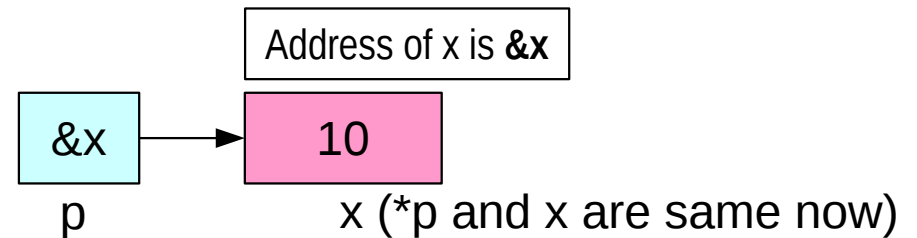
Pointer

```
int x=10,y=40;
```

```
➔ int *p = &x;  
  *p = 20;
```

```
  p = &y;  
  *p = 30;
```

```
cout<< x <<endl<< y;
```



Reference Variable

```
int x=10,y=40;
```

```
int &r = x;  
→ r = 20;
```

```
r = y;  
r = 30;
```

```
cout<< x <<endl<< y;
```

20

x

r (r and x are same now)

40

y

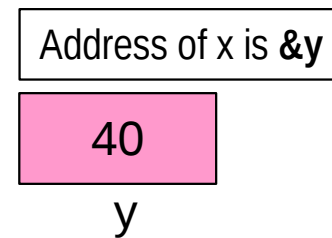
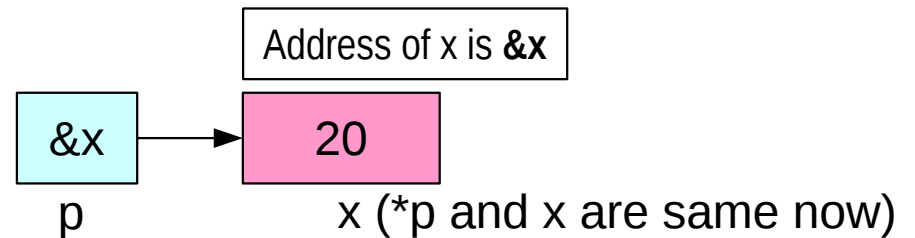
Pointer

```
int x=10,y=40;
```

```
int *p = &x;  
→ *p = 20;
```

```
p = &y;  
*p = 30;
```

```
cout<< x <<endl<< y;
```



Reference Variable

```
int x=10,y=40;
```

```
int &r = x;  
r = 20;
```

➔

```
r = y;  
r = 30;
```

```
cout<< x <<endl<< y;
```

40

x

r (r and x are same now)

40

y

Pointer

```
int x=10,y=40;
```

```
int *p = &x;  
*p = 20;
```

➔

```
p = &y;  
*p = 30;
```

```
cout<< x <<endl<< y;
```

&y

p

Address of x is &x

20

x

Address of x is &y

40

y (*p and y are same now)

Reference Variable

```
int x=10,y=40;
```

```
int &r = x;  
r = 20;
```

```
→ r = y;  
→ r = 30;
```

```
cout<< x <<endl<< y;
```

Output:

30
40

30

x

r (r and x are same now)

40

y

Pointer

```
int x=10,y=40;
```

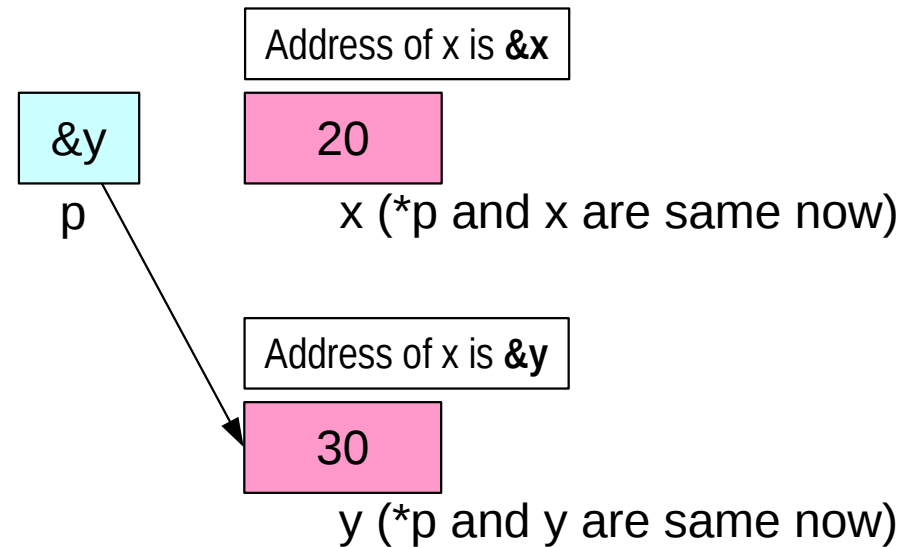
```
int *p = &x;  
*p = 20;
```

```
→ p = &y;  
→ *p = 30;
```

```
cout<< x <<endl<< y;
```

Output:

20
30



Function Arguments as Reference Variables

Call by Value

```
void f(int);

main()
{
    int x=20;
    f(x);
    cout<<x;
}

void f(int p)
{
    p=10;
}
```

When we make the call f(x)
value of x is copied to p
p has the value 20

f() changes the value of p

x remains unchanged

Output: 20

Call by Reference

```
void f(int *);

main()
{
    int x=20;
    f(&x);
    cout<<x;
}

void f(int *p)
{
    *p=10;
}
```

When we make the call f(&x)
address of x is copied to p
p has the value &x

f() changes the value at address &x

x gets changed

Output: 10

Function Arguments as Reference Variables

Call by Value

```
void f(int);

main()
{
    int x=20;
    f(x);
    cout<<x;
}

void f(int p)
{
    p=10;
}
```

When we make the call f (x)
value of x is copied to p
p has the value 20

f () changes the value of p

x remains unchanged

Output: 20

Call by Reference

```
void f(int *);

main()
{
    int x=20;
    f(&x);
    cout<<x;
}

void f(int *p)
{
    *p=10;
}
```

When we make the call f (&x)
address of x is copied to p
p has the value &x

f () changes the value at address &x

x gets changed

Output: 10

Call by Reference

```
void f(int &);

main()
{
    int x=20;
    f(x);
    cout<<x;
}

void f(int &p)
{
    p=10;
}
```

When we make the call f (&x)
NOTHING IS COPIED
p becomes a reference to x

f () changes the value of x

x gets changed

Output: 10


```
void f(int *);
```

```
main()
```

```
{
```

```
    int x=20;
```

```
    f(NULL);
```

Segmentation Fault
pointers may accidentally
point to invalid addresses

```
void f(int *p)
```

```
{
```

```
    *p=10;
```

```
}
```

```
void f(int &);
```

```
main()
```

```
{
```

```
    int x=20;
```

```
    f(x);
```

```
}
```

x is just a normal integer
we are not messing with
addresses

```
void f(int &p)
```

```
{
```

```
    p=10;
```

```
}
```

Function Returning Reference Variable

```
int &f(int [],int);
```

```
main()  
{  
    int x[3];  
    f(x,1)=10;  
    cout<< x[1] <<endl;  
}
```

f(x,1) becomes another name for x[1]

```
int &f(int p[],int i)  
{  
    return p[i];  
}
```

We are returning a reference to p[i]
We are NOT returning any pointer.

Output: 10

Dynamic Memory Allocation

Suppose,

p is any pointer of type **int*** and

p points here

then **p+1** points here

and **p+2** points here

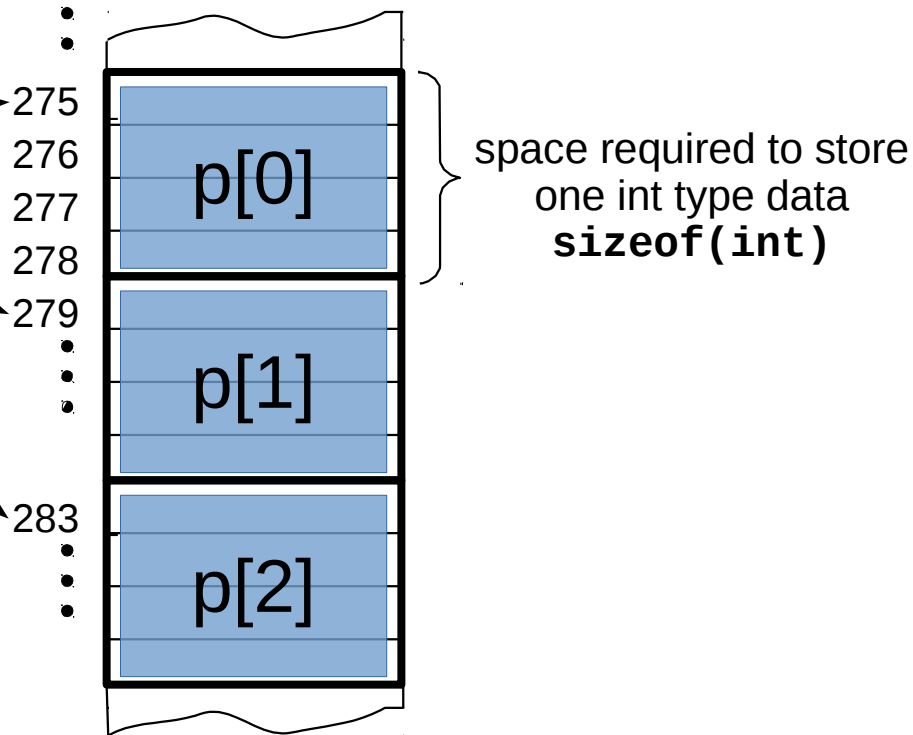
and so on...

p[0] is interpreted as ***p**

p[1] is interpreted as ***(p+1)**

p[2] is interpreted as ***(p+2)**

and so on...



p+i is interpreted as **(address stored in p) + i*sizeof(int)**

p[i] is interpreted as ***(p+i)**

In general, if we have a pointer **p** of type **<type> *** then,

p+i is interpreted as **(address stored in p) + i*sizeof(<type>)**

This is true even when **p** is pointing to some arbitrary location which is not allocated to your program
If you want to use **p[0]**,... MAKE SURE THAT THE SPACE IS ALLOCATED.

Is the following code correct ?

```
main()  
{  
    int *p;  
  
    *p=10;  
}
```

NO

no memory is allocated to store the value 10.
p contains a garbage value.
*p=10 is trying to store a value in a location
which is not allocated to this program

To make it work we need to do something like

```
int *p;  
<allocate memory to store one integer>  
<store the address of that memory location to p>  
*p=10;
```

The correct code should be the following

```
main()  
{  
    int *p;  
    p = new int;  
    *p=10;  
}
```



new keyword allocates memory and returns the address

```
main()
```

```
{
```

```
    int *p;
```

```
    p = new int;
```

```
    *p=10;
```

```
}
```

new int allocates memory for **one int type data**
and returns the address

```
main()
```

```
{
```

```
    int *p;
```

```
    p = new int[3];
```

```
    p[0]=5;p[2]=1;
```

```
}
```

new int[3] allocates memory for **3 int type data**
and returns the address

In general,

new <type> allocates memory for **one element of datatype <type>**
and returns the address

new <type>[n] allocates memory for **n elements of datatype <type>**
and returns the address

```
main()
```

```
{
```

```
    int *p;
```

```
    p = new int;
```

```
    *p=10;
```

```
    delete p;
```

```
}
```

delete <ptr> deallocates memory for **one element** pointed to by the pointer <ptr>

```
main()
```

```
{
```

```
    int *p;
```

```
    p = new int[3];
```

```
    p[0]=5;p[2]=1;
```

```
    delete[] p;
```

```
}
```

delete[] <ptr> deallocates memory for **multiple elements** pointed to by the pointer <ptr>

For dynamically allocated memory, you **must deallocate** it manually using **delete** when you no longer need it.

Otherwise your program may unnecessarily block large amount of memory and may even run out of memory. This is known as **memory leak**.

Example of dynamic allocation in mergesort

```
void merge(int A[],int p,int q,int r)
{
    int n1=q-p+1;
    int n2=r-q;

    int *L = new int[n1+1];
    int *R = new int[n2+1];

    //Copy A[p..q] to L and A[q+1..r] to R
    //Merge L and R into A[p..r]

    delete[] L; delete[] R;
}
```

```
void merge_sort(int A[],int p,int r)
{
    if(p<r)
    {
        merge_sort(A,p,(p+r)/2);
        merge_sort(A,(p+r)/2+1,r);

        merge(A,p,q,r);
    }
}
```

MERGE(A, p, q, r)

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```


Public and Private member

Classes can be thought of as generalisation of structure which can have **function as member**

Definition of class and member functions

```
class person
{
    public:
    string name;
    int age;

    void show(void)
    {
        cout<<name<<" "<<age<<endl;
    }
};
```

Some other function using the class

```
main()
{
    person p;
    p.name = "tom"
    p.age = 20;
    p.show();
}
```

Alternate Syntax

```
class person
{
    public:
    string name;
    int age;

    void show(void);
};

void person::show(void)
{
    cout<<name<<" "<<age<<endl;
}
```

```
main()
{
    person p;
    p.name = "tom"
    p.age = 20;
    p.show();
}
```

It is better to declare and define the member functions separately

Definition of class

```
class person
{
    public:
    string name;
    int age;

    void show(void);
};
```

Only declare the member function
inside class definition

Define the function outside. When defining,
write **<class_name>::** before the function name

Definition of member function

```
void person::show(void)
{
    cout<<name<<" "<<age<<endl;
}
```

Some other function using the class

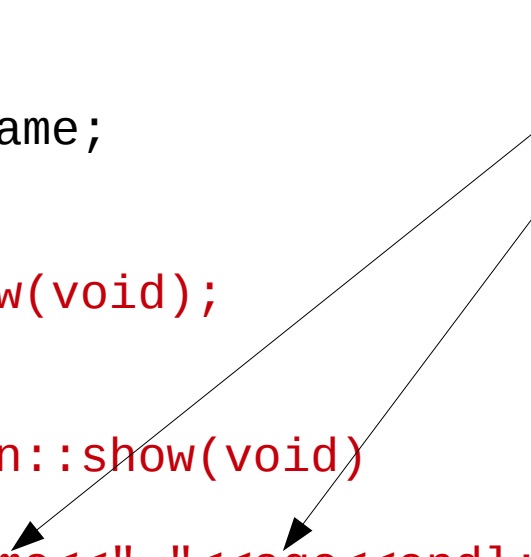
```
main()
{
    person p;
    p.name = "tom"
    p.age = 20;
    p.show();
}
```

Definition of class and its member functions

```
class person
{
    public:
    string name;
    int age;

    void show(void);
};

void person::show(void)
{
    cout<<name<<" "<<age<<endl;
}
```



All the data members are "available" to the member functions like global variables.

The member function accesses them by their names only (ex. 'name' , 'age').

Some other function using the class

```
main()
{
    person p;
    p.name = "tom"
    p.age = 20;
    p.show();
}
```

The user of the class creates objects of that class. (ex. here we create an **object p** of **class person**)

The user can access only public members.

Public members (both data and function) of that object are accessed by `<object_name>.<member_name>` (ex. 'p.name' , 'p.age')

Definition of class and its member functions

```
class person
{
    private:
    string name;
    int age;

    public:
    void setdata(string, int);
    void show(void);
};

void person::setdata(string s, int i)
{
    name = s; age = i;
}

void person::show(void)
{
    cout<<name<<" "<<age<<endl;
}
```

Members of a class can be **private**.

Member functions can access all other members (public or not)

User of the class can access only public members.

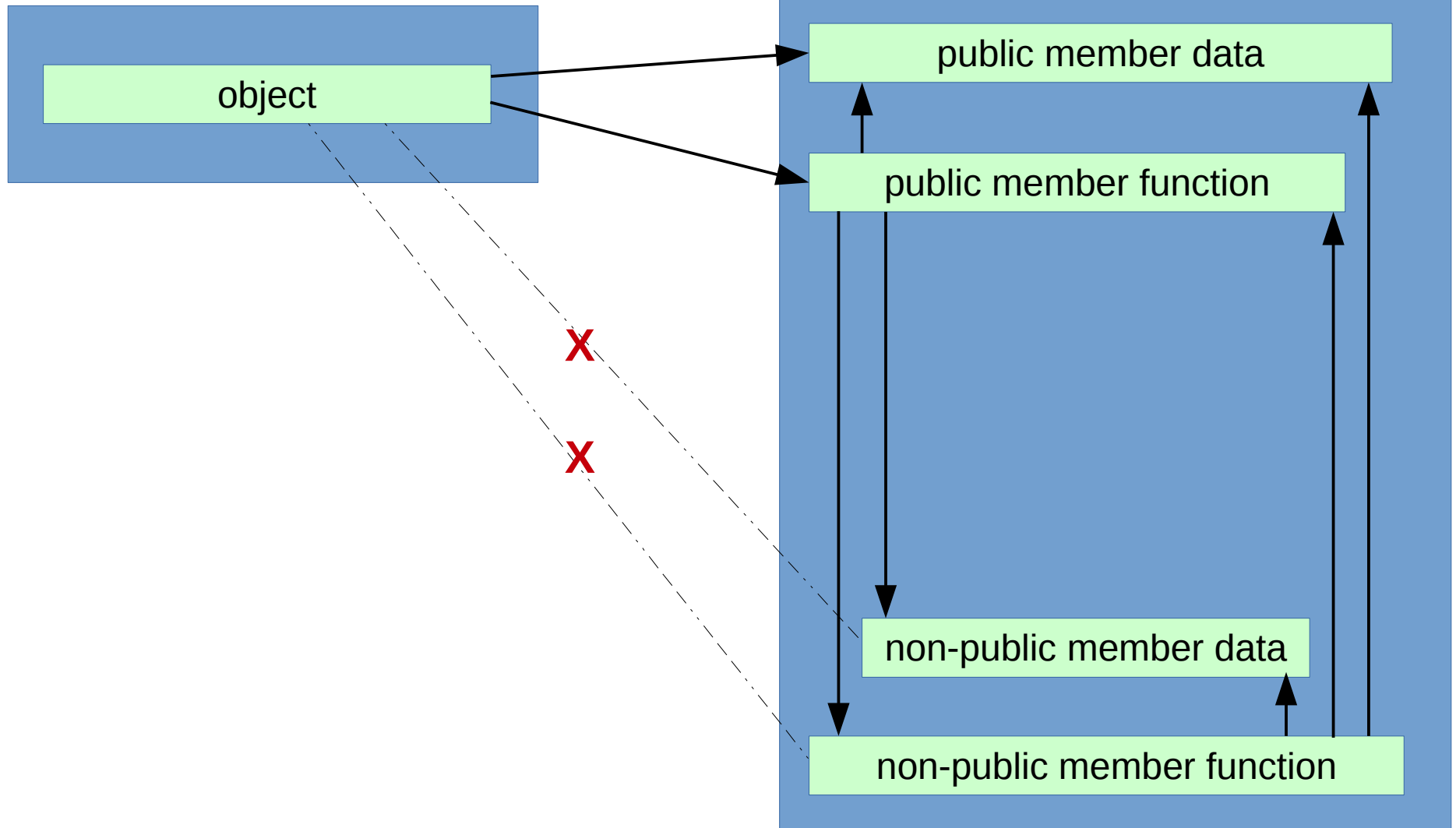
Here 'name' and 'age' are private members

Some other function using the class

```
main()
{
    person p;
    p.name = "tom" WRONG
    p.age = 20;      WRONG
    p.setdata("tom", 20);
    p.show();
}
```

Some function using the class

Class Definition



depicts that 'a' can access 'b'

depicts that 'a' cannot access 'b'

Constructor and Destructor

Constructor and destructor

When we define a class, two member functions are automatically defined

<class_name>() - This is called constructor
~<class_name>() - This is called destructor

These functions are automatically called
when an object of that class is created or destroyed

We may explicitly redefine them.

They do not have any return-type at all (not even void)

```
class T
{
    public:
        T();
        ~T();
};

T::T() {cout<<"Object Created\n";}
T::~~T() {cout<<"Object Destroyed\n";}

main()
{
    T x;
}
```


Output:
Object Created
Object Destroyed

Constructor and destructor

We explicitly define constructors typically for **initialising data** members. For example, setting some float / integer to a particular value, **dynamically allocate memory and initialise a pointer** with the address, etc.

We explicitly define destructors typically for "**clean-up**". For example, **deallocate memory** which is blocked by the object, etc.

Summery

- Classes can have both data and function as members.
- Members can be **public** or **private** (or protected). protected will be explained later
- Any member function can access any other member like global variable/function.
- User of the class can access only public members.
- Member functions access other member by just their name
- User of a class access a member of an object by **<object_name>.<member_name>**
- It is better to only declare a member function inside class definition and define the member function outside class definition.
- When defining a member function outside class definition write **<class_name>::** before the function name.
- **Constructor and destructors** are implicitly defined member functions which are automatically called when an object is created and destroyed respectively
- We may explicitly redefine constructor/destructor, but it is not mandatory.
- They always have the names **<class_name>** and **~<class_name>** respectively and they do not have any return-type (not even void).

Doubly linked list with sentinel

A node for a doubly linked list

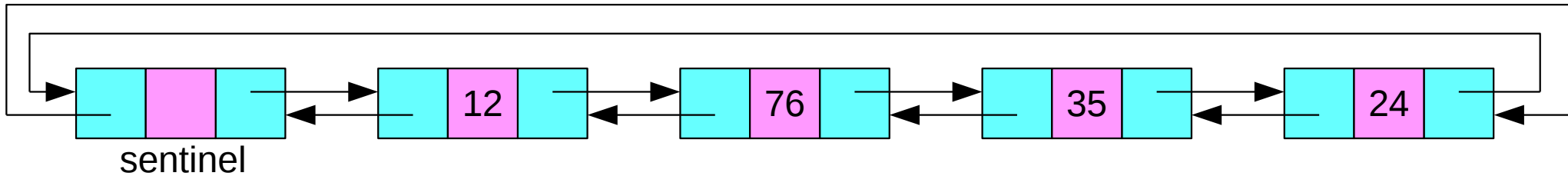


to simplify things
use shall use just
one integer as data

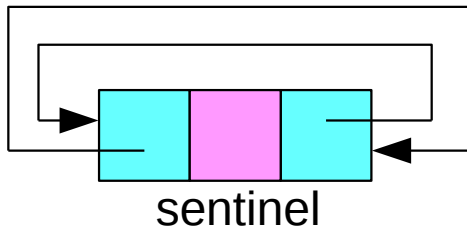
```
struct node
{
    int data;
    node *prev;
    node *next;
};
```

If we have pointer **n** to a node then,
n->prev and **n->next** are pointers to previous and next node respectively

A circular doubly linked list with sentinel (containing integer data 12, 76, 35, 24)



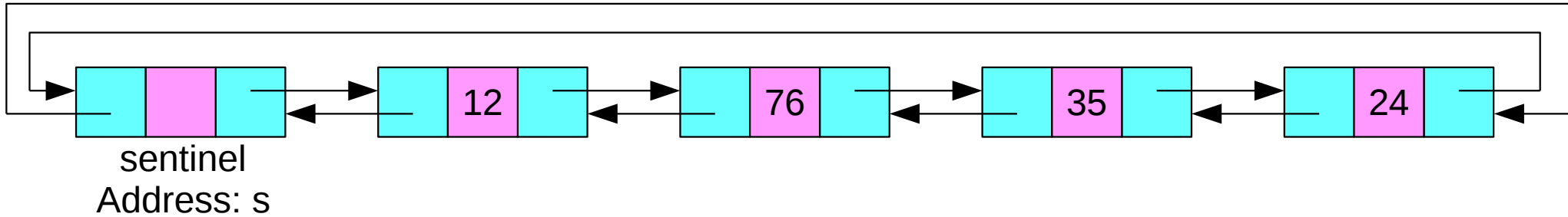
Empty list



Sentinel is a dummy node which does not store data.
It acts as a boundary between the first and last node.

It ensures that the list always (even when empty) contains at least one node.
We need not worry about handling NULL addresses separately.
(Trying to dereference NULL pointer leads to Segmentation fault)

Makes the coding lot easier.



To traverse the list or search the list for a node satisfying some criteria,

We maintain a pointer, say 't', to the "current" node and run a loop with t as the loop variable.

We start with the sentinel and continue until we reach the last node.

or

We start with the first node after sentinel and continue until we reach the sentinel

Notice that

t is s when we are at the sentinel.

t is s->next when we are at the first node.

t->next is s when we are at the last node.

t = t->next makes t point to the next node.

Let us create a class for our circular doubly linked list

We shall provide the following functionalities to the user as member functions

`push_front(x)` - should create a node, with data x, in front of the list.

`pop_front()` - should remove the node in front and return the integer contained in it.

What should be the declarations for these functions ?

```
void push_front(int); } and these should be public  
int pop_front(void); }
```

Similarly we may also declare,

```
void push_back(int);  
int pop_back(void);
```

In order to implement these functions we need to maintain a pointer to the sentinel.

This pointer will be used only by the member functions.

User doesn't need to access this pointer directly.

```
node *s; (this should be private)
```

```
class list  
{  
    private:  
        node *s;  
  
    public:  
        void push_front(int);  
        int pop_front(void);  
        void push_back(int);  
        int pop_back(void);  
};
```

Let us create a class for our circular doubly linked list

```
void push_front(int);  
int pop_front(void);  
void push_back(int);  
int pop_back(void);
```

 } these should be public

In order to implement these functions we need to maintain a pointer to the sentinel.
This pointer will be used only by the member functions.
User doesn't need to access this pointer directly.

node *s; (this should be private)

s is just a pointer. When we create a list we should create the sentinel node (dynamically) and store the address of sentinel in **s**.

We need an **explicit constructor**.

list(); (should be public)

Definition of constructor

```
list::list()  
{  
    s = new node;  
    s->next = s; s->prev = s;  
}
```

We also need an **explicit destructor**

to delete all the nodes when the list is destroyed.

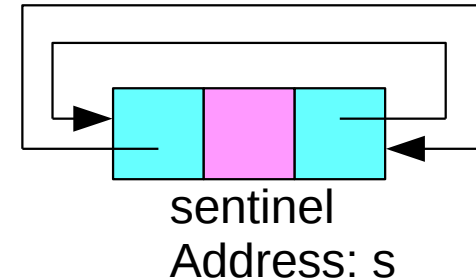
```
class list  
{  
    private:  
        node *s;  
  
    public:  
        void push_front(int);  
        int pop_front(void);  
        void push_back(int);  
        int pop_back(void);  
  
        list();  
        ~list();  
};
```


A function to check whether the list is empty

```
bool isempty(void);
```

```
bool list::isempty(void)
{
    if(s->next == s) return true;
    else return false;
}
```

Empty list



We may implement the destructor as follows

```
list::~~list(void)
{
    while(!isempty()) pop_front();
    delete s;
}
```

A function to print the entire list

```
void show(void);
```

```
bool list::show(void)
{
    for(node *t = s->next; t!=s ; t=t->next)
        cout << " " << t->data;
}
```

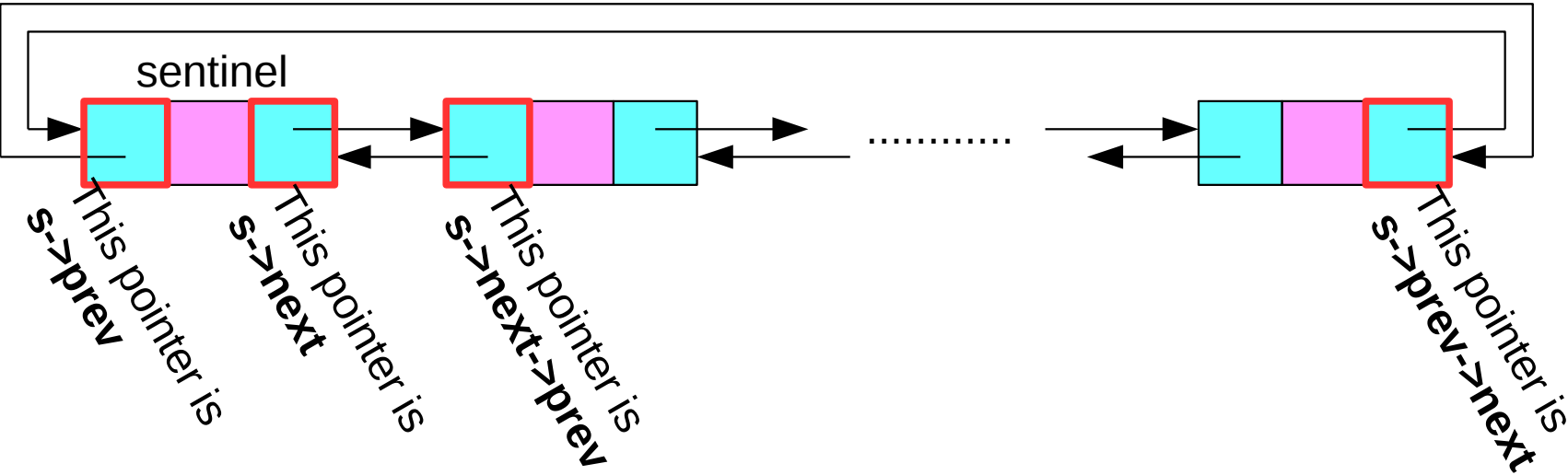
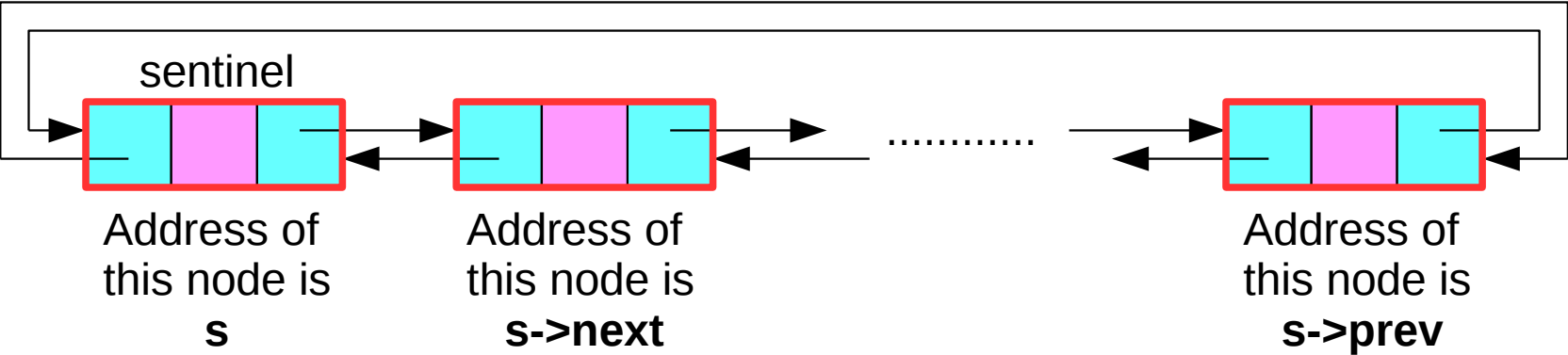
```
class list
{
    private:
        node *s;

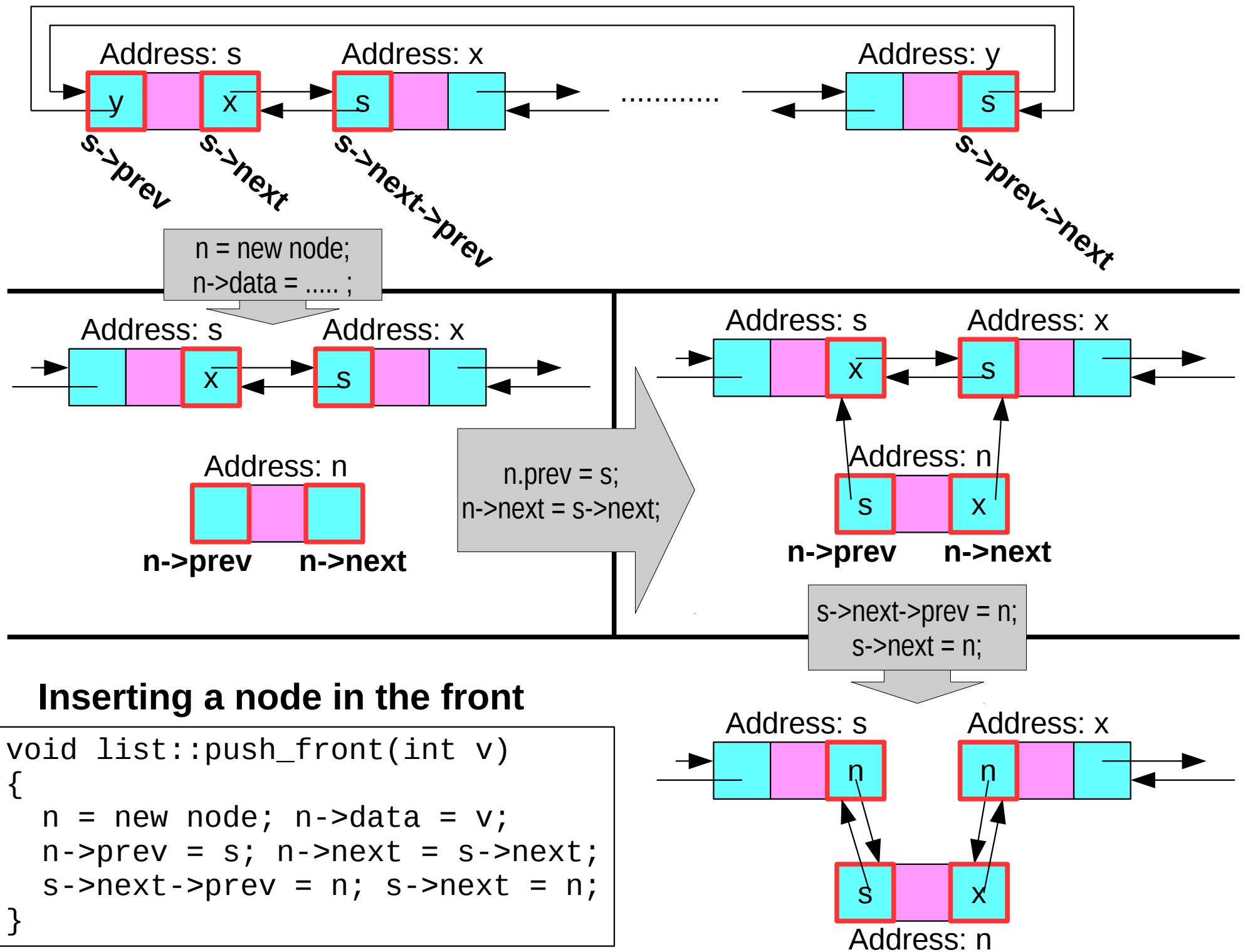
    public:
        void push_front(int);
        int pop_front(void);
        void push_back(int);
        int pop_back(void);

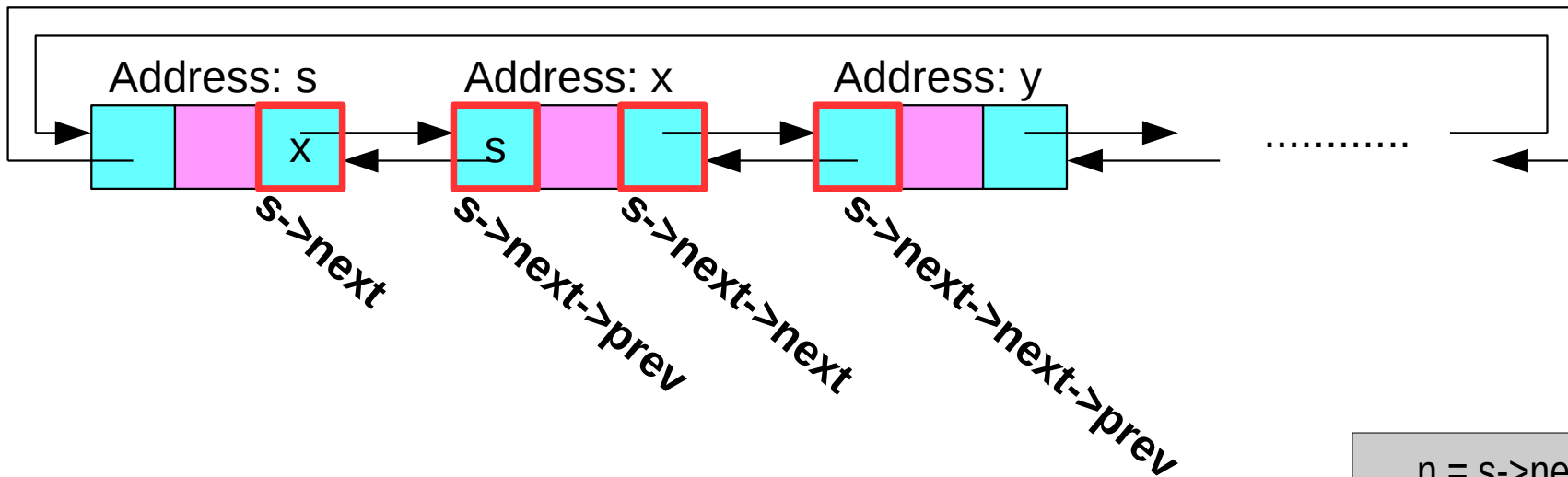
        list();
        ~list();

        void isempty(void);
        void show(void);
};
```

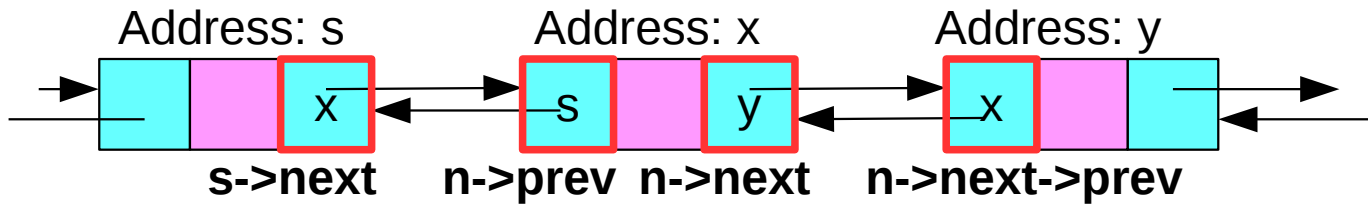
A circular doubly linked list with sentinel



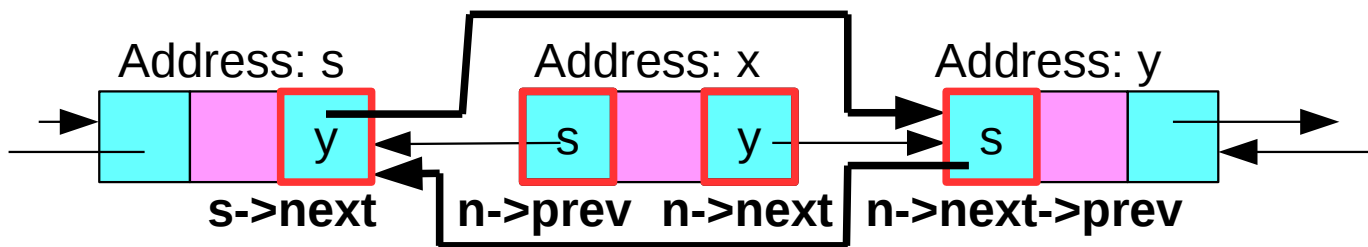




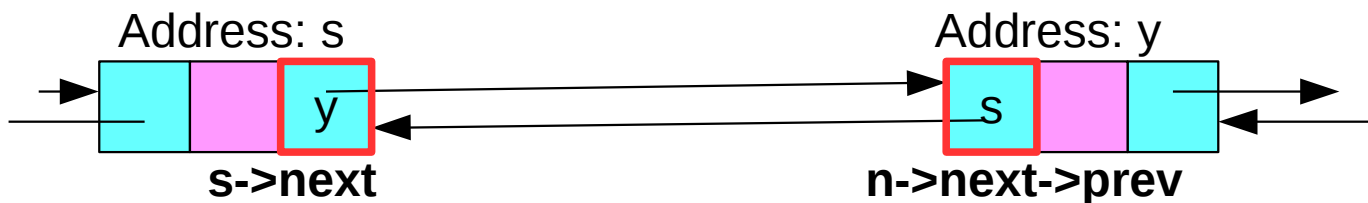
```
n = s->next;
int v = n->data
```



```
n->next->prev = s;
s->next = n->next;
```



```
delete n;
return v;
```

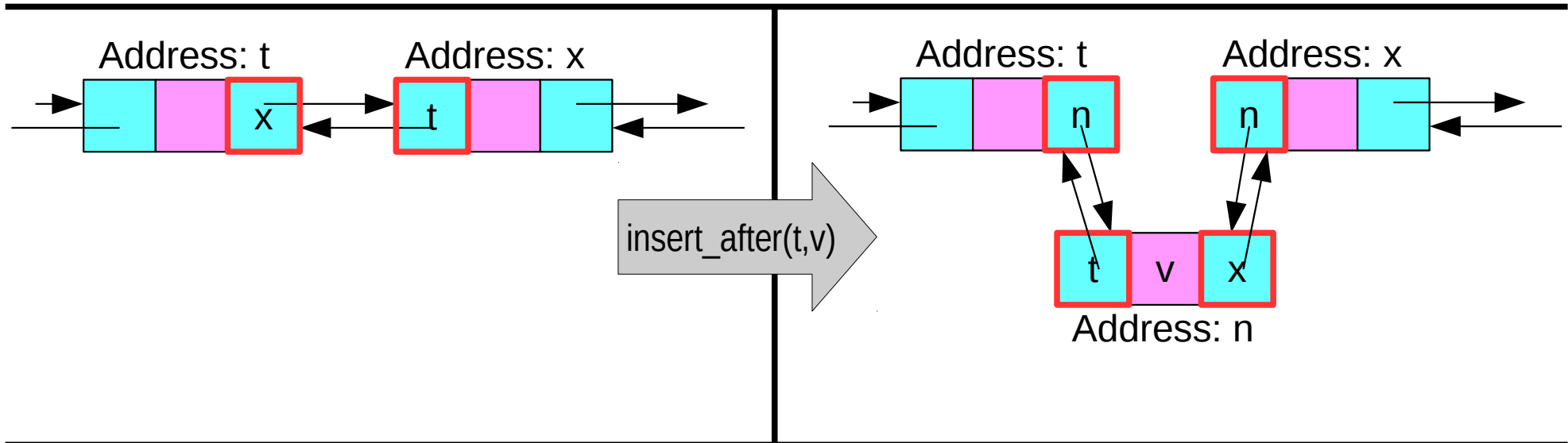


Deleting a node from the front

Inserting a new node after any arbitrary node

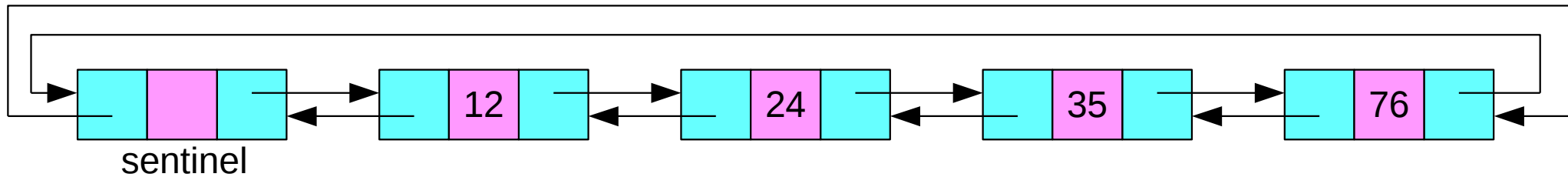
We may define **insert_after(node *t, int v)** which creates a new node with data 'v' and inserts that node after a node to which 't' points.

This is exactly same as push_front(), apart from the modification that we should just use the given pointer 't' instead of 's'.

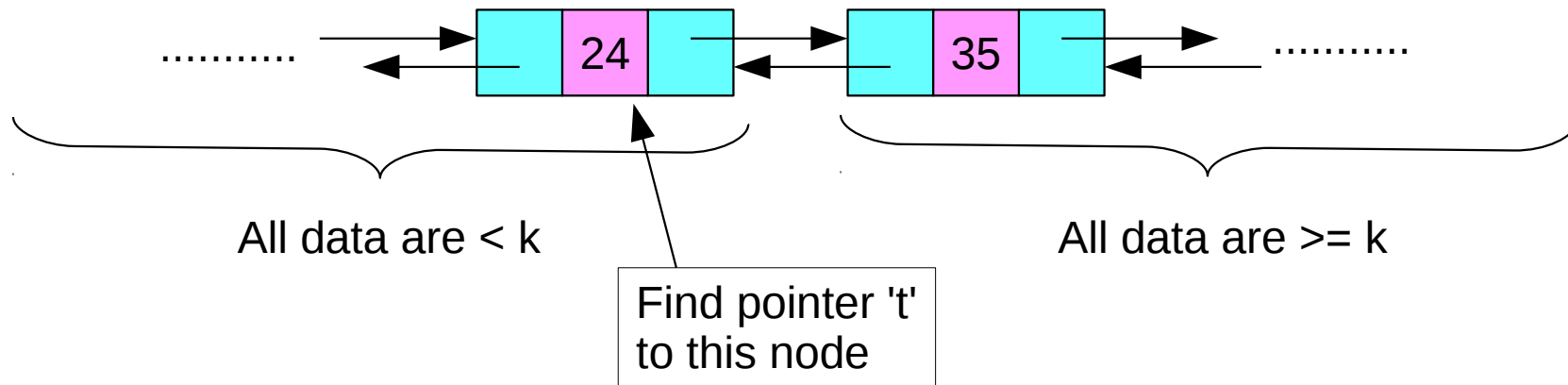


Insertion Sort with linked list

Suppose the data are already are already sorted in ascending order.



If we want to insert another integer 'k' (for example say $k=30$) then we traverse the list to **find the pointer 't'** to the last node such that $t \rightarrow \text{next} \rightarrow \text{data} \geq k$



Now do an `insert_after(t,k)`

