ABSTRACT DATA STRUCTURE

PROGRAMMING TECHNIQUES

ADVISOR: Trương Toàn Thịnh

CONTENTS

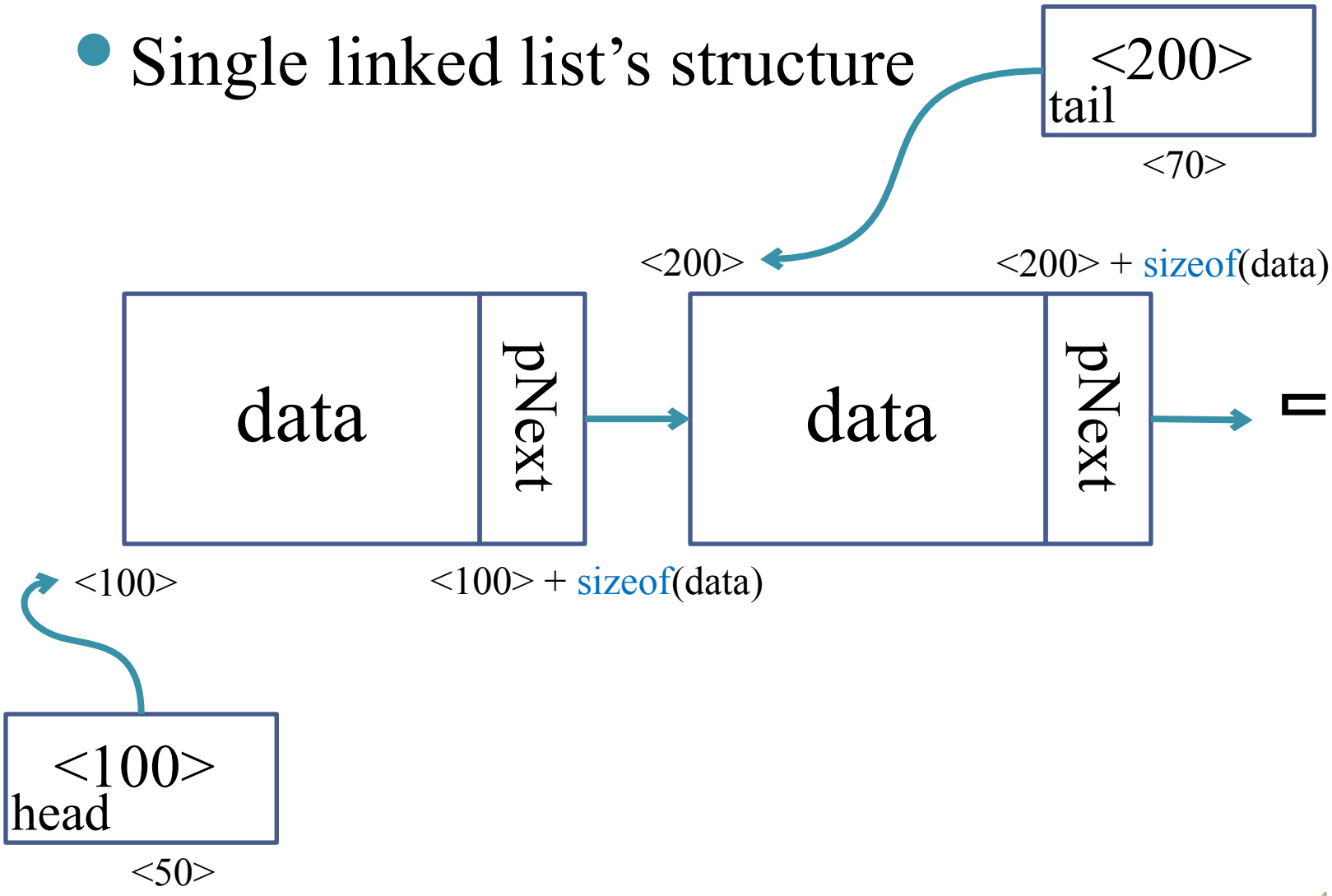
- Abstract linked list
- Abstract queue
- Abstract stack
- Other data structure

ABSTRACT LINKED LIST

- Review some knowledge
  - The static array's drawback is fixed-length
  - The dynamic array's drawback is lack of required memory
  - The array's advantage is quickly random access
  - The linked list's advantage:
    - Need not a fixed length
    - Suitable for fragmented memory
  - The linked list's disadvantage: sequential access

ABSTRACT LINKED LIST

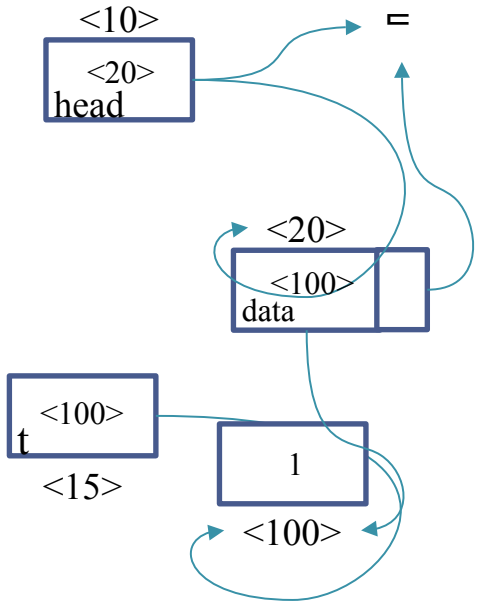
- Single linked list's structure



ABSTRACT LINKED LIST

- Example: create the first node
  - struct Node{void* data; Node* pNext;};
  - template <class T>
  - struct Node{T data; Node* pNext;};

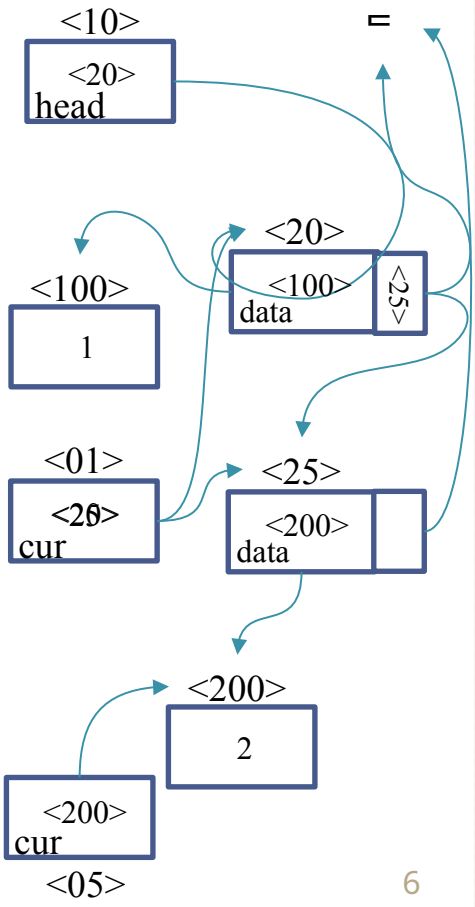
Lines	C	C++
1	void main(){	void main(){
2	Node* head = NULL;	Node<int>* head = NULL;
3	head = new Node;	head = new Node<int>;
4	head->data = new int;	
5	*(int*)(head->data) = 1;	head->data = 1;
6	head->pNext = NULL;	head->pNext = NULL;
7	}	}



ABSTRACT LINKED LIST

- Create next node-use temporary pointer cur

Lines	C	C++
1	void main(){	void main(){
2	Node* head = NULL;	Node<int>* head = NULL;
3	head = new Node;	head = new Node<int>;
4	head->data = new int;	
5	*(int*)(head->data) = 1;	head->data = 1;
6	head->pNext = NULL;	head->pNext = NULL;
7	Node* cur = head;	Node<int>* cur = head;
8	cur->pNext = new Node;	cur->pNext = new Node<int>;
9	cur = cur->pNext;	cur = cur->pNext;
10	cur->data = new int;	
11	*(int*)(cur->data) = 2;	cur->data = 2;
12	cur->pNext = NULL;}	cur->pNext = NULL;}

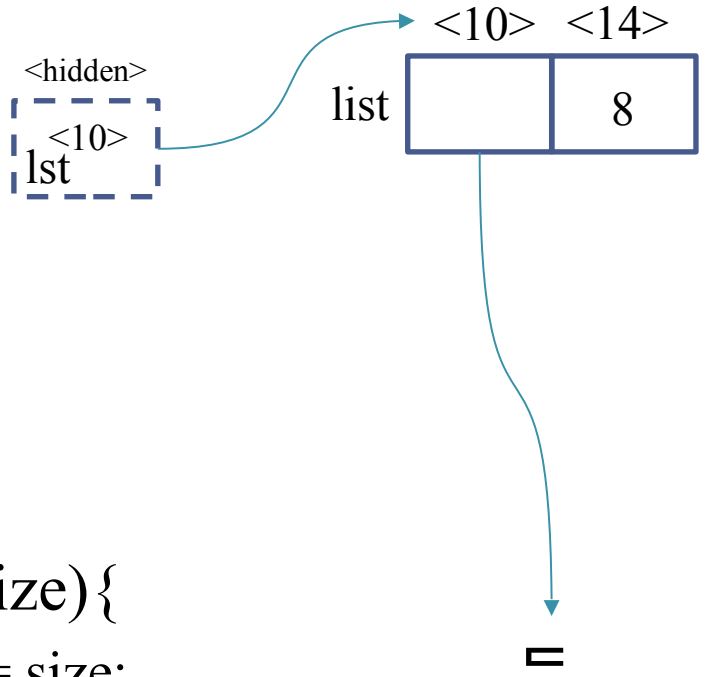


ABSTRACT LINKED LIST

- There are 2 ways creating a function of adding node at the start of the linked list
  - Using template of C++: simple, natural code
  - Using void* of C:
    - Complex code but quick speed
    - Note to free memory of data before deleting the node itself
- With void*, we need to build struct List to manage the information of sizeof(<datatype>)

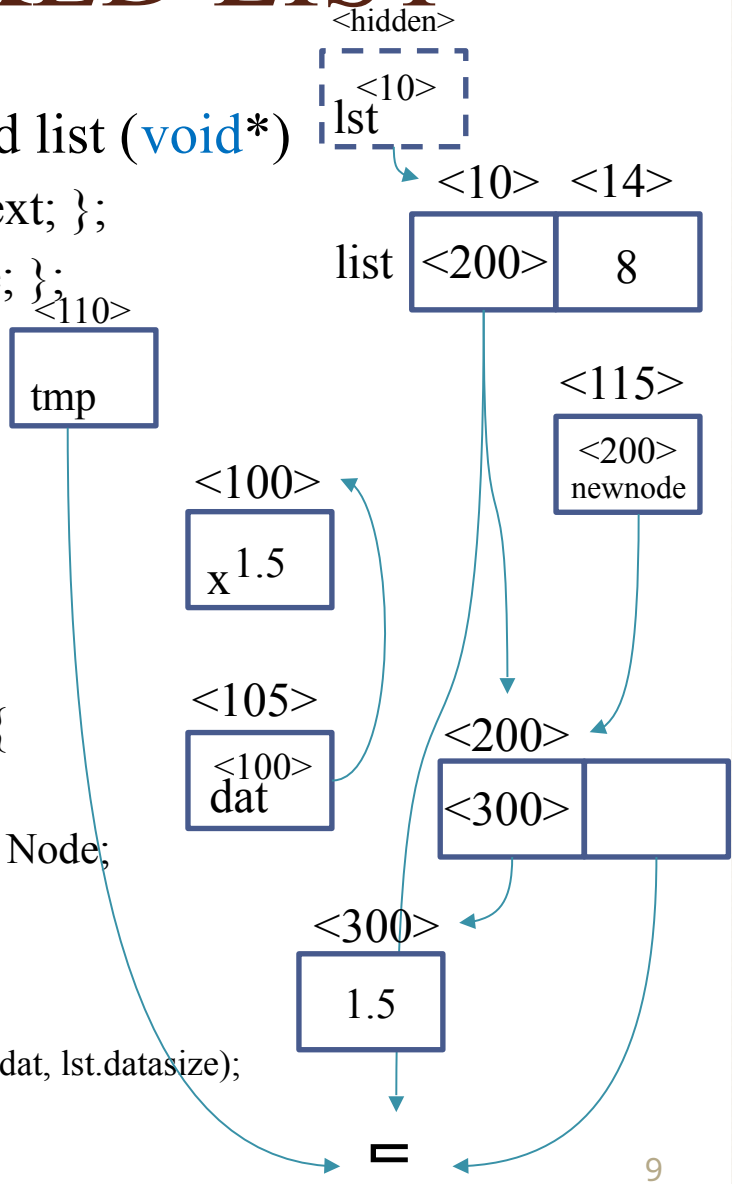
ABSTRACT LINKED LIST

- Ex: add a node at the start of linked list (void*)
  - struct Node { void* data; Node* pNext; };
  - struct List { Node* head; int datasize; };
  - void main(){
    - double x = 0; List list;
    - **initList**(list, sizeof(double));
    - x = 1.5; insertFirst(list, &x);
    - x = 2.6; insertFirst(list, &x);
    - freeList(list);
  - }
  - void **initList**(List& lst, int size){
    - lst.head = NULL; lst.datasize = size;
  - }



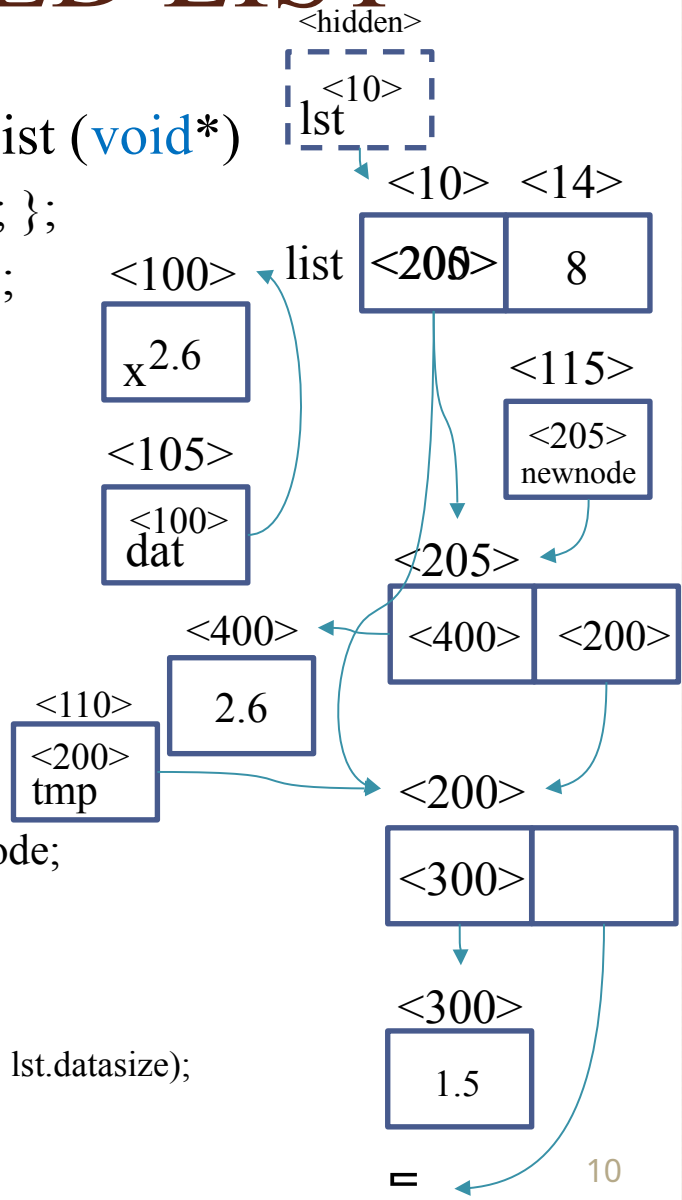
ABSTRACT LINKED LIST

- Ex: add a node at the start of linked list (void*)
  - struct Node { void* data; Node* pNext; };
  - struct List { Node* head; int datasize; };
  - void main(){
    - //…
    - x = 1.5; **insertFirst**(list, &x);
    - x = 2.6; insertFirst(list, &x);
    - freeList(list);
  - }
  - void insertFirst(List& lst, void* dat){
    - if(!dat) return;
    - Node* tmp = lst.head, *newnode = new Node;
    - if(newnod){
      - lst.head = newnod; newnod->pNext = tmp;
      - newnod->data = new char[lst.datasize];
      - if(newnod->data) memmove(newnod->data, dat, lst.datasize);
    - }
  - }



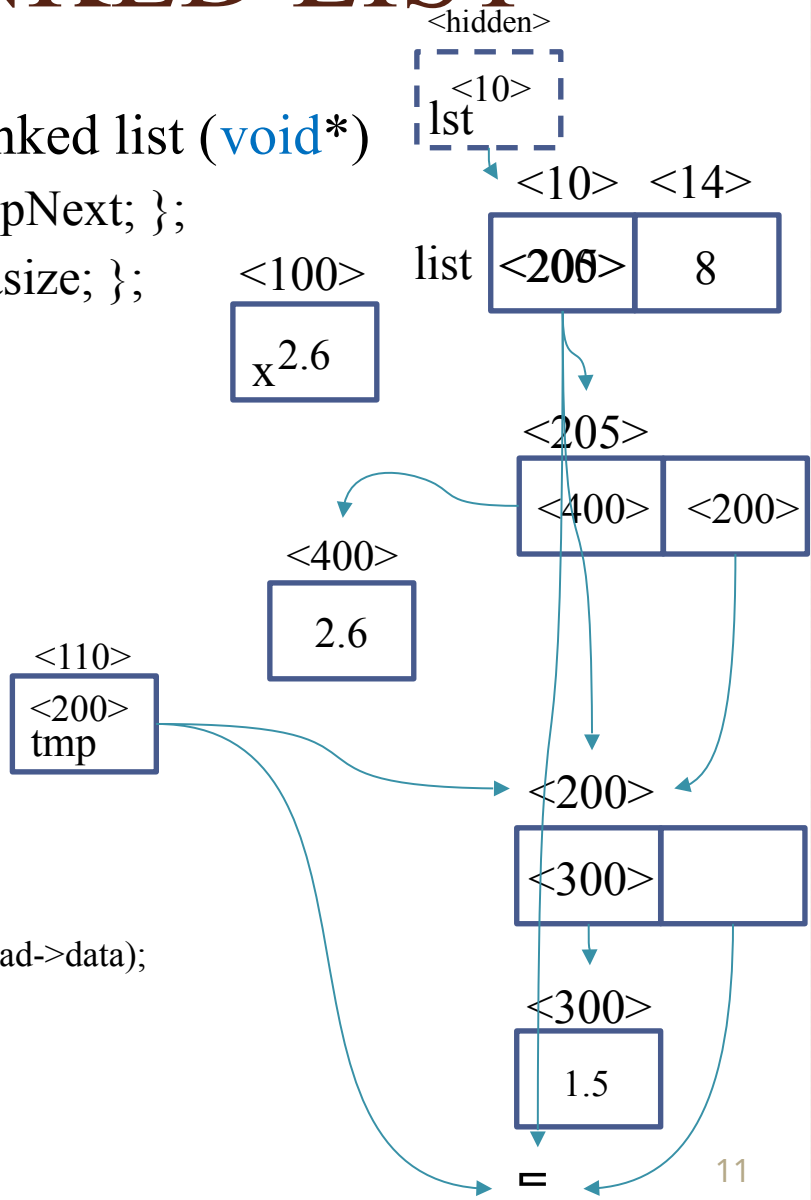
ABSTRACT LINKED LIST

- Ex: add a node at the start of linked list (void*)
  - struct Node { void* data; Node* pNext; };
  - struct List { Node* head; int datasize; };
  - void main(){
    - //…
    - x = 1.5; insertFirst(list, &x);
    - x = 2.6; **insertFirst**(list, &x);
    - freeList(list);
  - }
  - void insertFirst(List& lst, void* dat){
    - if(!dat) return;
    - Node* tmp = lst.head, *newnode = new Node;
    - if(newnod){
      - lst.head = newnod; newnod->pNext = tmp;
      - newnod->data = new char[lst.datasize];
      - if(newnod->data) memmove(newnod->data, dat, lst.datasize);
    - }
  - }



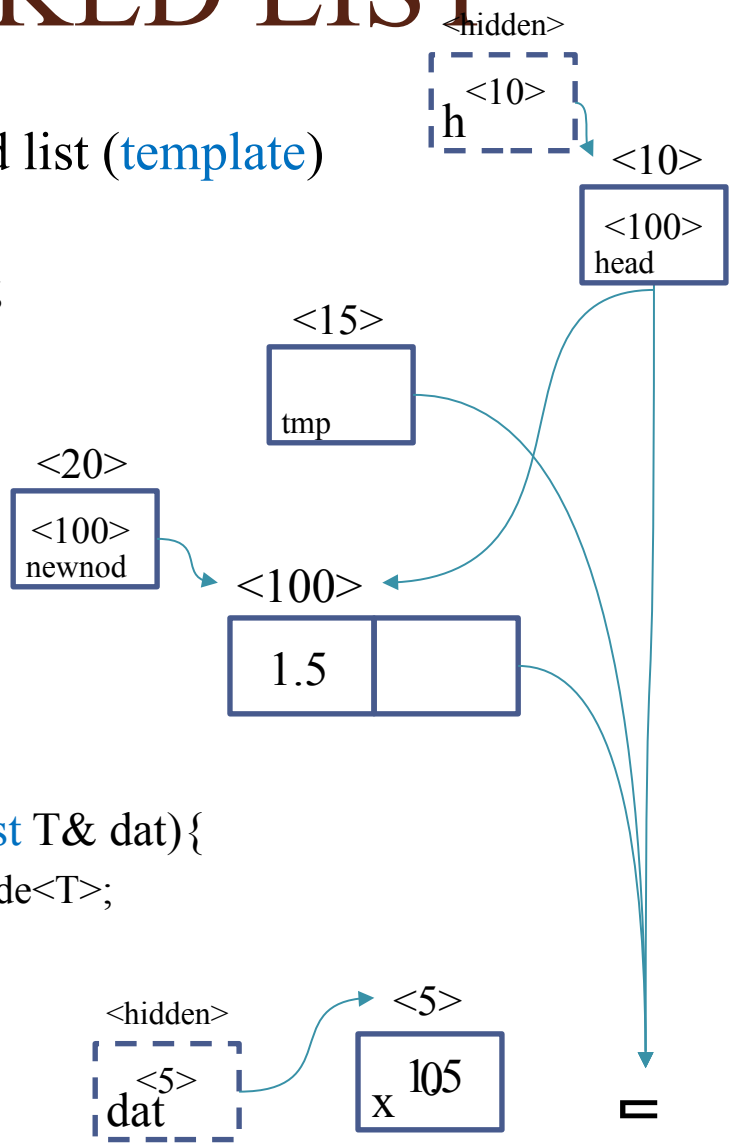
ABSTRACT LINKED LIST

- Ex: add a node at the start of linked list (void*)
  - struct Node { void* data; Node* pNext; };
  - struct List { Node* head; int datasize; };
  - void main(){
    - //…
    - x = 1.5; insertFirst(list, &x);
    - x = 2.6; insertFirst(list, &x);
    - **freeList**(list);
  - }
  - void freeList(List& lst){
    - Node* tmp = NULL;
    - while(lst.head){
      - tmp = lst.head->pNext;
      - if(lst.head->data) delete[] (char*)(lst.head->data);
      - delete lst.head;
      - lst.head = tmp;
    - }
  - }



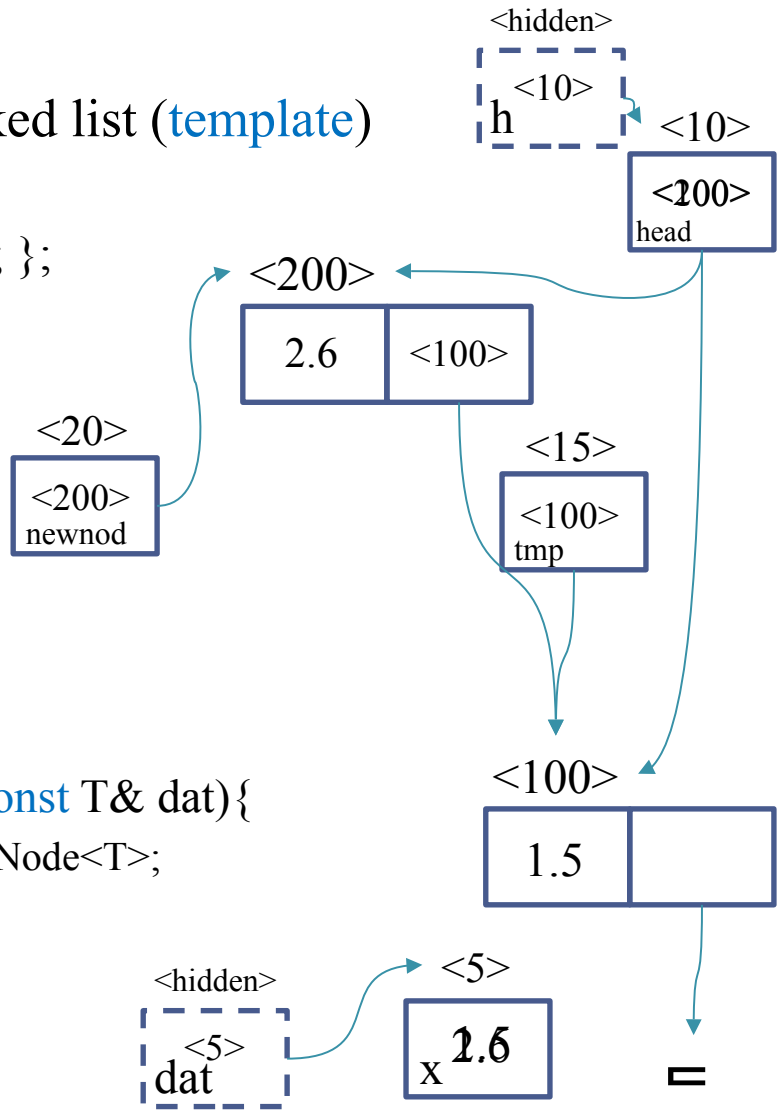
ABSTRACT LINKED LIST

- Ex: add a node at the start of linked list (template)
  - template <class T>
  - struct Node {T data; Node* pNext; };
  - void main(){
    - double x = 0;
    - Node<double> *head = NULL;
    - x = 1.5; **insertFirst**(head, x);
    - x = 2.6; insertFirst(head, x);
    - freeList(head);
  - }
  - template <class T>
  - void **insertFirst**(Node<T>* &h, const T& dat){
    - Node<T> *tmp = h, *newnod = new Node<T>;
    - if(newnod){
      - h = newnod;
      - newnod->pNext = tmp;
      - newnod->data = dat;
    - }
  - }



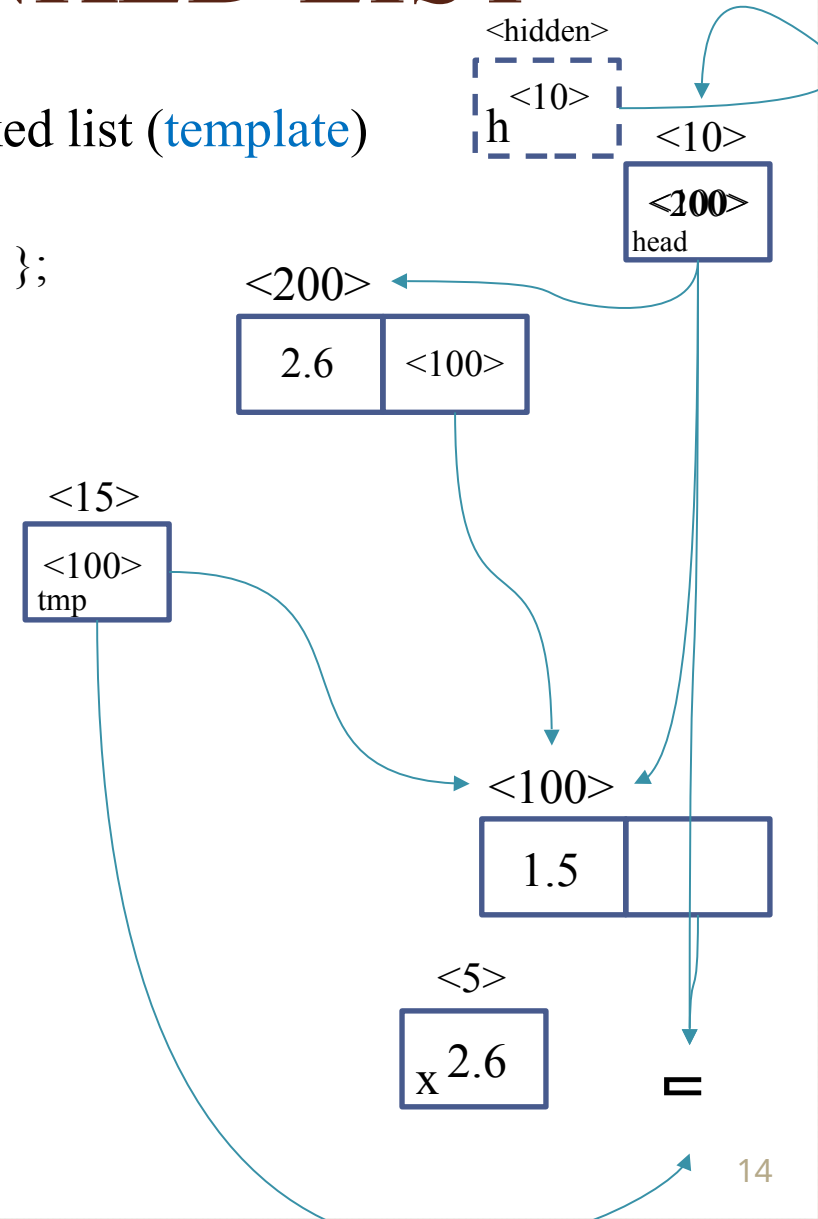
ABSTRACT LINKED LIST

- Ex: add a node at the start of linked list (template)
  - template <class T>
  - struct Node {T data; Node* pNext; };
  - void main(){
    - double x = 0;
    - Node<double> *head = NULL;
    - x = 1.5; insertFirst(head, x);
    - x = 2.6; **insertFirst**(head, x);
    - freeList(head);
  - }
  - template <class T>
  - void **insertFirst**(Node<T>* &h, const T& dat){
    - Node<T> *tmp = h, *newnod = new Node<T>;
    - if(newnod){
      - h = newnod;
      - newnod->pNext = tmp;
      - newnod->data = dat;
    - }
  - }



ABSTRACT LINKED LIST

- Ex: add a node at the start of linked list (template)
  - template <class T>
  - struct Node {T data; Node* pNext; };
  - void main(){
    - double x = 0;
    - Node<double> *head = NULL;
    - x = 1.5; insertFirst(head, x);
    - x = 2.6; insertFirst(head, x);
    - **freeList**(head);
  - }
  - template <class T>
  - void **freeList**(Node<T>* &h){
    - Node<T>* tmp = NULL;
    - while(h){
      - tmp = h->pNext;
      - delete h;
      - h = tmp;
    - }
  - }

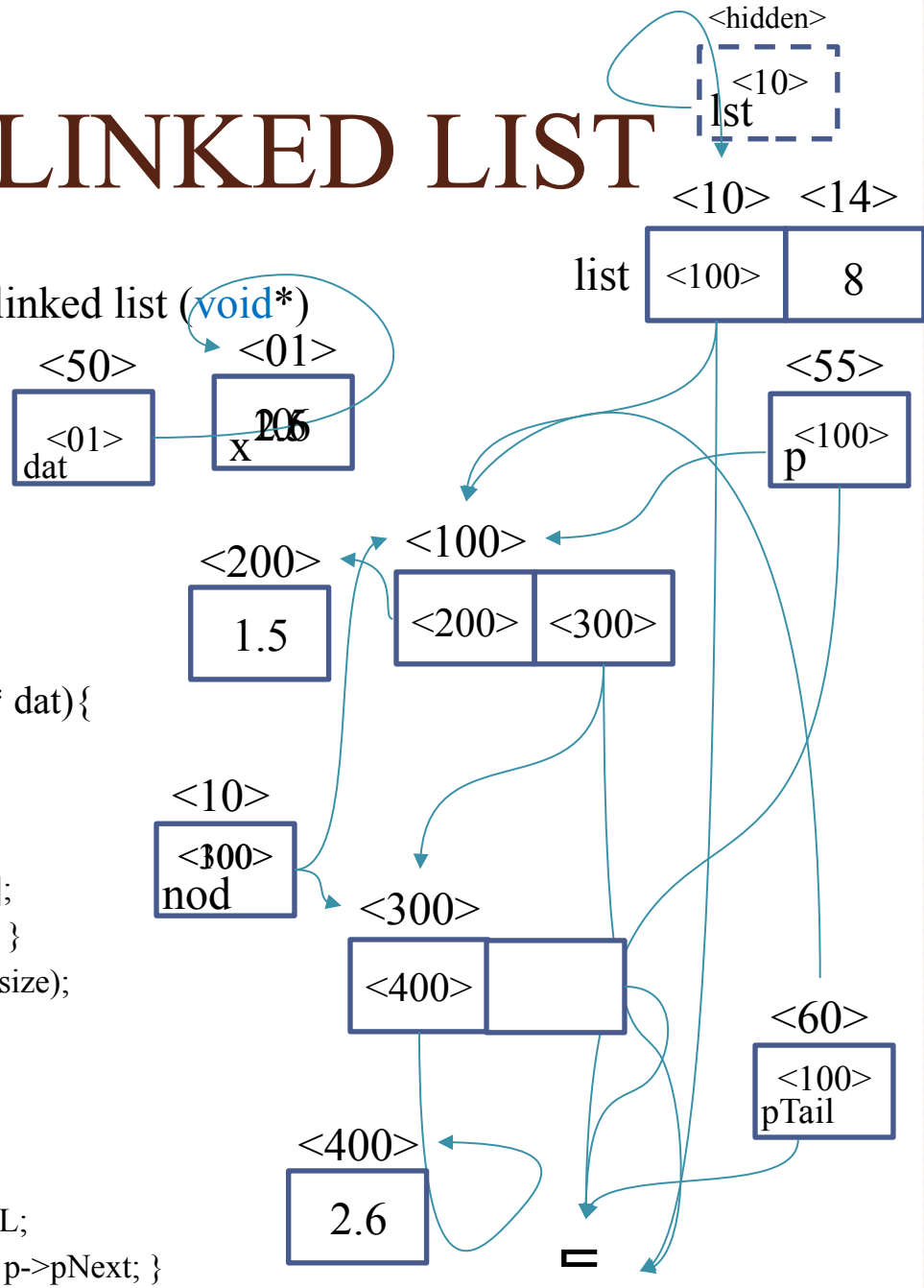


ABSTRACT LINKED LIST

- There are 2 ways creating a function of adding node at the end of the linked list
  - Using template of C++: simple, natural code
  - Using void* of C:
    - Complex code but quick speed
    - Note to free the memory of data before deleting the node itself
- With void*, we need to build struct List to manage the information of sizeof(<datatype>)

ABSTRACT LINKED LIST

- Ex: add the node at the end of linked list (void*)
  - void main(){
    - double x = 0; List list;
    - initList(list, sizeof(double));
    - x = 1.5; insertLast(list, &x);
    - x = 2.6; insertLast(list, &x);
    - freeList(list);
  - }
  - void **insertLast**(List& lst, void* dat){
    - Node* nod = new Node;
    - if(!nod) return;
    - nod->pNext = NULL;
    - nod->data = new char[lst.datasize];
    - if(!nod->data){ delete nod; return; }
    - memmove(nod->data, dat, lst.datasize);
    - Node* pTail = **getTail**(lst);
    - if(pTail == NULL) lst.head = nod;
    - else pTail->pNext = nod; }
  - Node* **getTail**(List& lst){
    - Node *p = lst.head, *pTail = NULL;
    - while(p != NULL){ pTail = p; p = p->pNext; }
    - return pTail; }



ABSTRACT LINKED LIST

- Ex: add the node at the end of linked list (template)

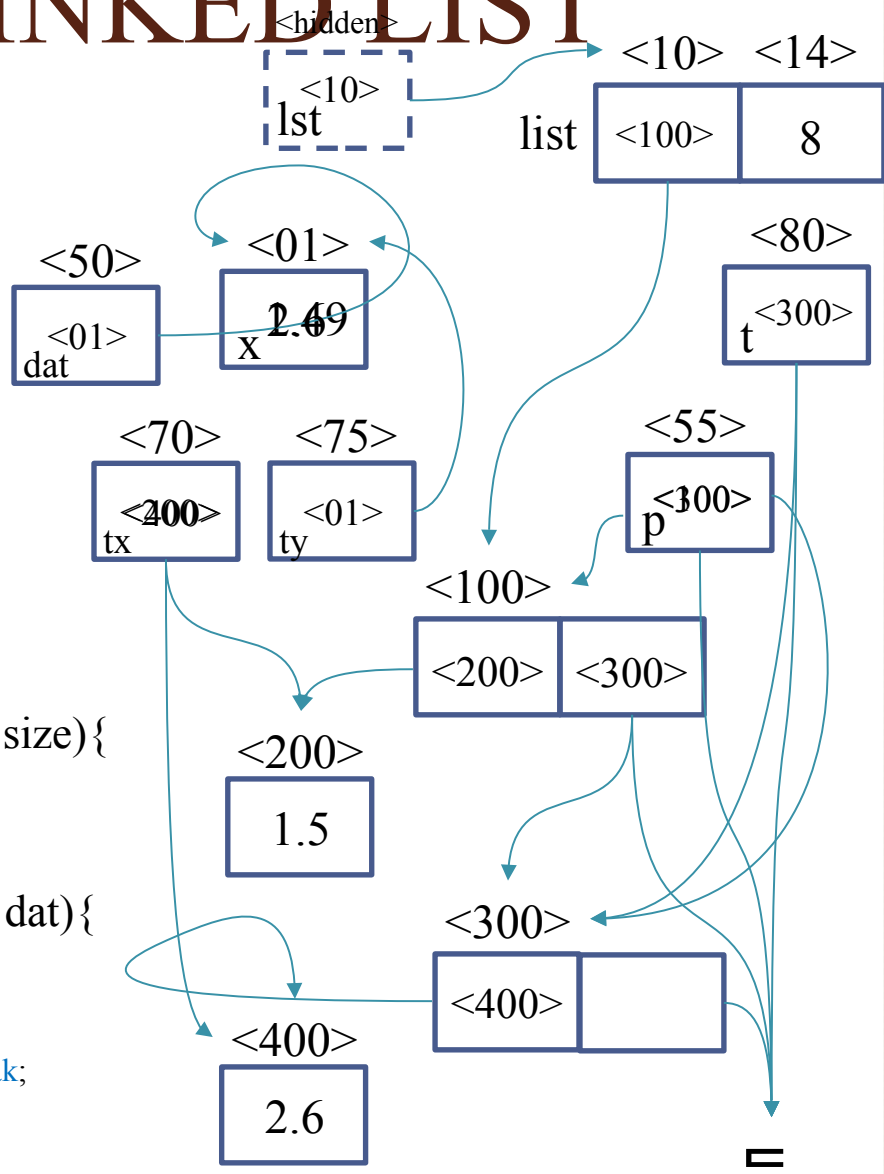
	template <class T>
Node* **getTail**(List& lst){	Node<T>* **getTail**(Node<T>* h){
Node *p = lst.head, *pTail = NULL;	Node<T> *p = h, *pTail = NULL;
while(p != NULL){pTail = p; p = p->pNext; }	while(p != NULL){pTail = p; p = p->pNext; }
return pTail; }	return pTail; }
	template <class T>
void **insertLast**(List& lst, void* dat){	void **insertLast**(Node<T>* &h, const T& dat) {
Node* nod = new Node; if(!nod) return;	Node<T>* nod = new Node<T>;if(!nod) return;
nod->pNext = NULL;	nod->pNext = NULL;
nod->data = new char[lst.datasize];	
if(!nod->data){ delete nod; return; }	
memmove(nod->data, dat, lst.datasize);	nod->data = dat;
Node* pTail = **getTail**(lst);	Node<T>* pTail = **getTail**(h);
pTail == NULL? lst.head = nod : pTail->pNext = nod; }	pTail == NULL? h = nod : pTail->pNext = nod;}

ABSTRACT LINKED LIST

- Finding the element satisfying the customized condition in linked list
  - Build the highly re-used code
  - Handle the finding in ordered/unordered linked list
  - Handle the problem of “equivalence” of two elements when comparing to finding value
- May use void* of C or template of C++

ABSTRACT LINKED LIST

- Finding problem (using void*)
  - void main(){
    - double x; List list; Node* t;
    - initList(list, sizeof(double));
    - x = 1.5; insertLast(list, &x);
    - x = 2.6; insertLast(list, &x);
    - x = 1.49; t = **findList**(list, &x);
    - x = 2.6; t = **findList**(list, &x);
    - freeList(lst);
  - }
  - bool Cmp(void* tx, void* ty, int size){
    - return memcmp(tx, ty, size) == 0;
  - }
  - Node* **findList**(List& lst, void* dat){
    - Node* p = lst.head;
    - while(p){
      - if(Cmp(p->data, dat, lst.datasize)) break;
      - p = p->pNext;
    - } return p;
  - }



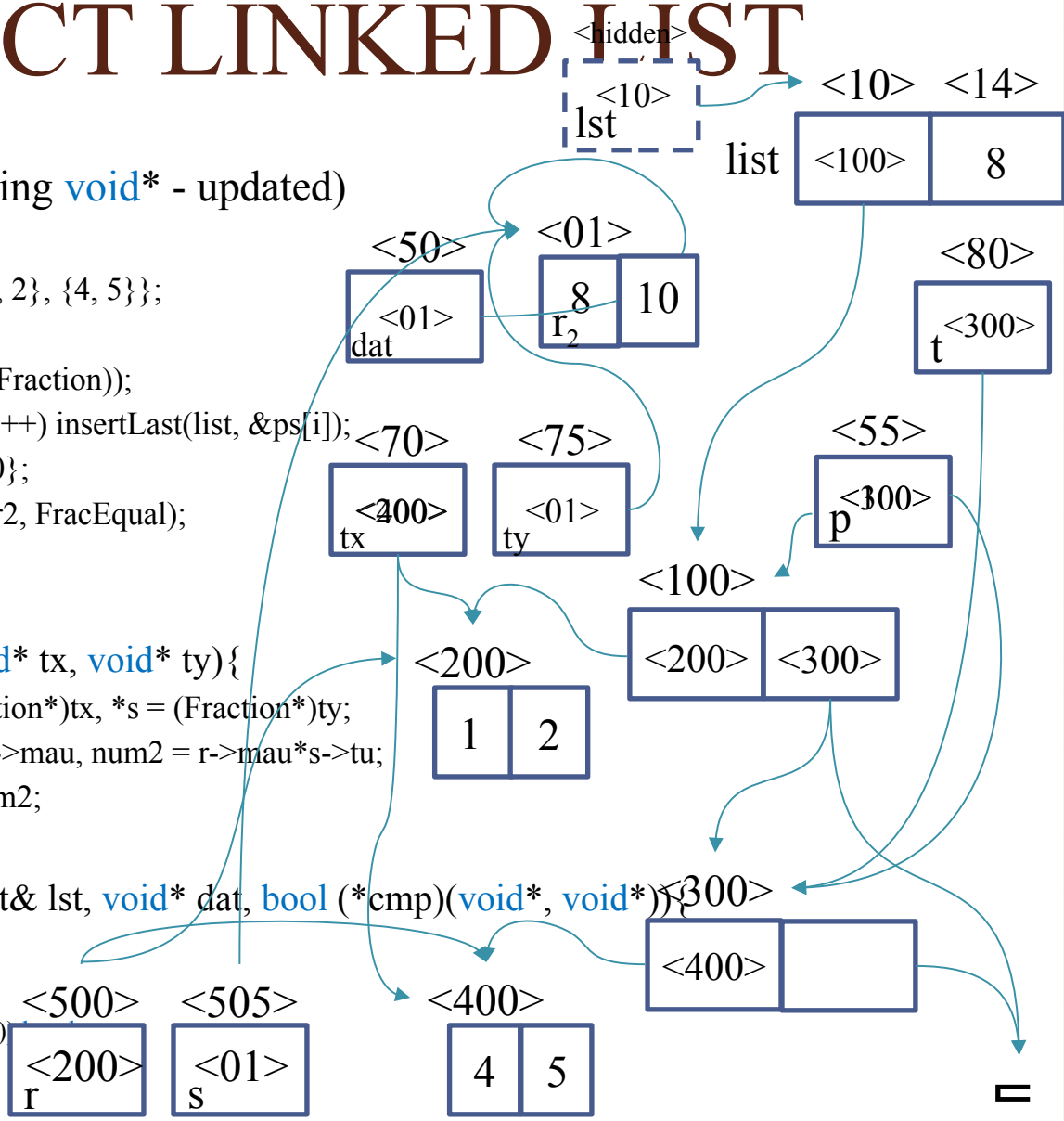
ABSTRACT LINKED LIST

- Finding problem (using void*)
  - Problem arose when linked list of fractions
  - Example:
    - 4/5 = 8/10 mathematically
    - 4/5 ㄴㅎ 8/10 bit-compare
  - Function **memcmp** only compares two string of bits
  - Solution:
    - Reducing fraction (not general)
    - Add a parameter of “function pointer” into function findList()

ABSTRACT LINKED LIST

- Finding problem (using void* - updated)

```
void main(){
    Fraction ps[] = {{1, 2}, {4, 5}};
    List list; Node* t;
    initList(list, sizeof(Fraction));
    for(int i = 0; i < 2; i++) insertLast(list, &ps[i]);
    Fraction r2 = {8, 10};
    t = findList(list, &r2, FracEqual);
    freeList(lst);
}
bool FracEqual(void* tx, void* ty){
    Fraction *r = (Fraction*)tx, *s = (Fraction*)ty;
    int num1 = r->tu*s->mau, num2 = r->mau*s->tu;
    return num1 == num2;
}
Node* findList(List& lst, void* dat, bool (*cmp)(void*, void*)){
    Node* p = lst.head;
    while(p){
        if(cmp(p->data, dat)
        p = p->pNext;
    } return p;
}
```



ABSTRACT LINKED LIST

- Finding problem (using template): findList() similar to void*

```
void main(){
    Fraction ps[] = {{1, 2}, {-4, 5}}, r2 = {8, 10};
    Node<Fraction>* head = NULL;
    for(int i = 0; i < 2; i++) insertLast(head, ps[i]);
    Node<Fraction>* p = findList(head, r2, fracAbsEqual);
    p = findList(head, r2, fracEqual);
    freeList(head);
}
template <class T>
Node<T>* findList(Node<T>* head, const T& dat, bool (*cmp)(T*, T*)){
    Node<T>* p = head;
    while(p){ if(cmp(&p->data, (T*)(&dat))) break; p = p->pNext; } return p;
}
bool FracAbsEqual(Fraction* r, Fraction* s){
    return abs(r->tu*s->mau) == abs(r->mau*s->tu);
}
bool FracEqual(Fraction* r, Fraction* s){
    return r->tu*s->mau == r->mau*s->tu;
}
```

ABSTRACT LINKED LIST

- Finding problem (using template): findList() similar to void*
  - Build a default function findList(), always compares with “normal” sense
  - When there are a need of context, we transmit that comparison function to findList()
  - Example: fracEqual is a normal sense, and fracAbsEqual is a concrete sense
  - Solution
    - Using default parameter at the position of function pointer
    - With new datatype, redefine operator ‘==’

ABSTRACT LINKED LIST

- Finding problem (use template): findList() with default parameter

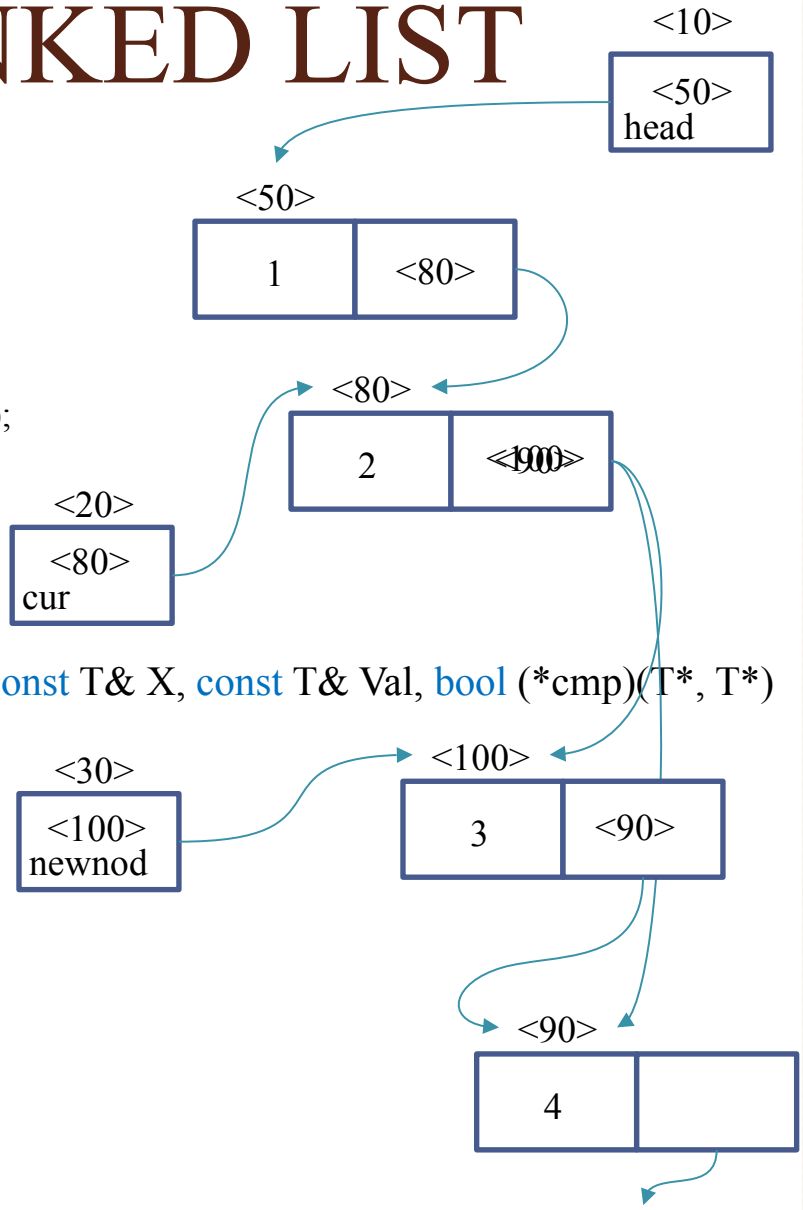
```
void main(){
    Fraction ps[] = {{1, 2}, {-4, 5}}, r2 = {8, 10};
    Node<Fraction>* head = NULL;
    for(int i = 0; i < 2; i++) insertLast(head, ps[i]);
    Node<Fraction>* p = findList(head, r2, fracAbsEqual);
    p = findList(head, r2); // Call isEqual default
    freeList(head);
}
bool operator==(const Fraction& r, const Fraction& s){
    return r.tu*s.mau == r.mau*s.tu;
}
template <class T>
bool isEqual(T* r, T* s){ return *r == *s; }
template <class T>
Node<T>* findList(Node<T>* head, const T& dat, bool (*cmp)(T*, T*) = isEqual){
    Node<T>* p = head;
    while(p){ if(cmp(&p->data, (T*)(&dat))) break; p = p->pNext; } return p;
}
bool fracAbsEqual(Fraction* r, Fraction* s){
    return abs(r->tu*s->mau) == abs(r->mau*s->tu);
}
```

ABSTRACT LINKED LIST

- Problem: insert a node after another "given" node into linked list
  - "given" means giving data
  - Need determining which node with data "equivalent to" data "given"
  - Reuse the solution of problem of "equivalence" when comparing with the finding value
- May use void* of C or template of C++

ABSTRACT LINKED LIST

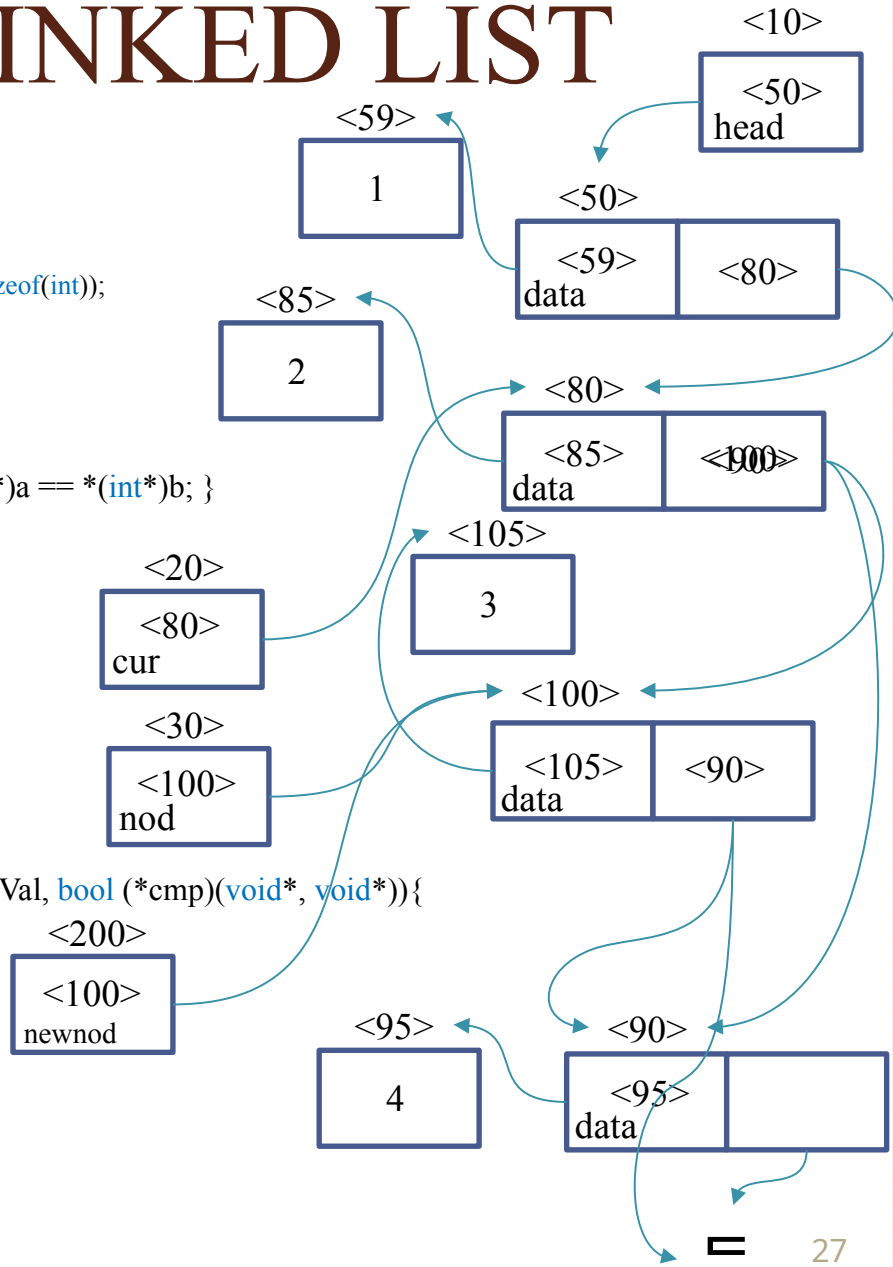
- Problem: Insert after (use template)
  - void main(){
    - int a[] = {1, 2, 4}, b = 3;
    - Node<int>* head = NULL;
    - for(int i = 0; i < 3; i++) insertLast(head, a[i]);
    - **insertAfter**(head, a[1], b);
    - freeList(head);
  - }
  - template <class T>
    - Node<T>* **insertAfter**(Node<T>* &h, const T& X, const T& Val, bool (*cmp)(T*, T*) = isEqual){
      - Node<T>* cur = findList(h, X, cmp);
      - if(cur == NULL) return NULL;
      - Node<T>* newnod = new Node<T>;
      - if(newnod){
        - newnod->data = Val;
        - newnod->pNext = cur->pNext;
        - cur->pNext = newnod;
      - }
      - return newnod;
    - }



ABSTRACT LINKED LIST

- Problem: Insert after (use void*)

```
void main(){
    int a[] = {1, 2, 4}, b = 3; List list; initList(list, sizeof(int));
    for(int i = 0; i < 3; i++) insertLast(list, &a[i]);
    insertAfter(list, &a[1], &b, intComp);
    freeList(head);
}
bool intComp(void* a, void* b){ return *(int*)a == *(int*)b; }
Node* makeNode(List& l, void* d){
    Node* nod = new Node;
    if(!nod) return NULL;
    nod->pNext = NULL;
    nod->data = new char[l.datasize];
    if(!nod->data){ delete nod; return NULL; }
    memmove(nod->data, d, l.datasize);
    return nod;
}
Node* insertAfter(List &lst, void* X, void* Val, bool (*cmp)(void*, void*)){
    Node* cur = findList(lst, X, cmp);
    if(cur == NULL) return NULL;
    Node* newnod = makeNode(lst, Val);
        if(newnod){
        newnod->pNext = cur->pNext;
        cur->pNext = newnod;
    }
    return newnod;
}
```



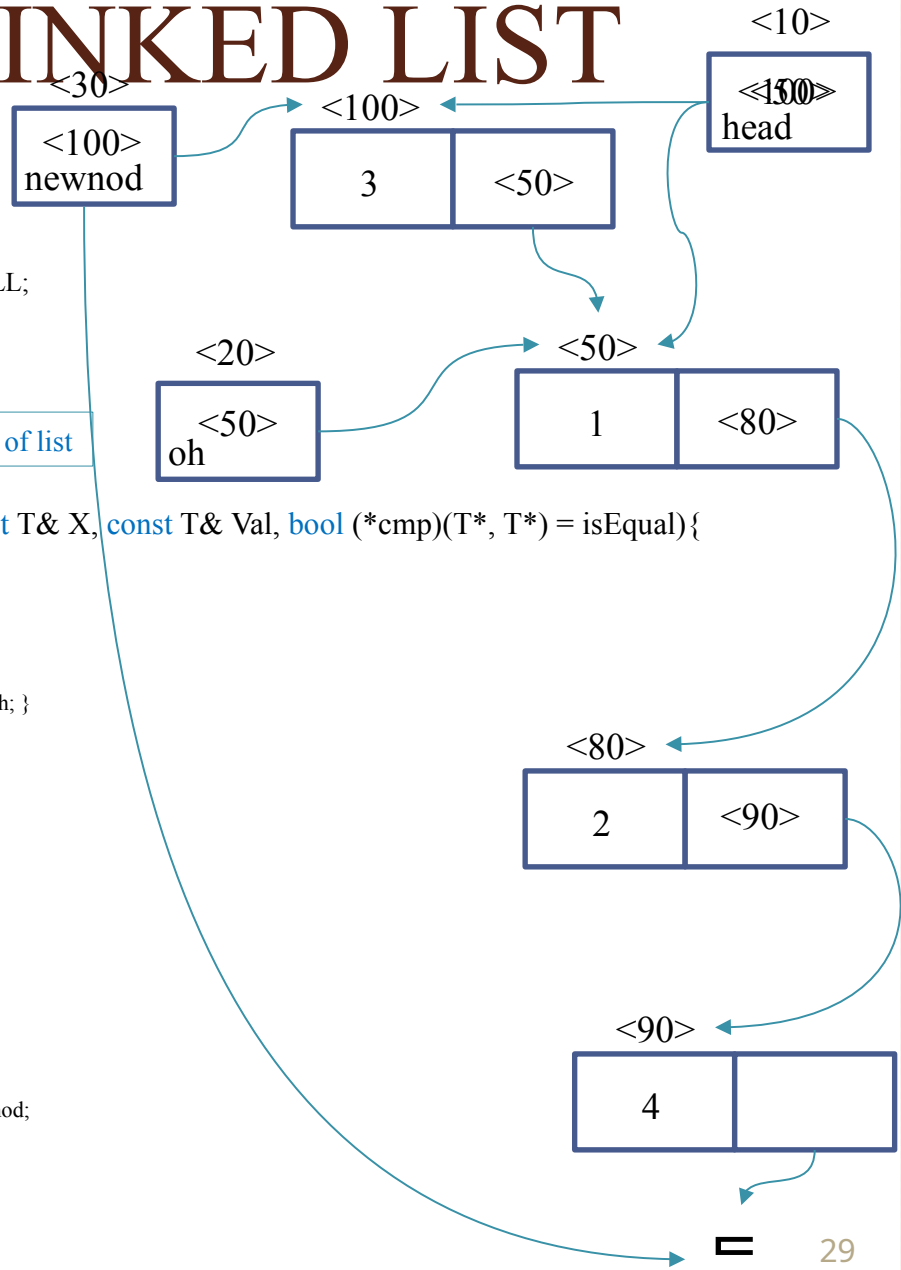
ABSTRACT LINKED LIST

- Problem: insert a node before another "given" node into linked list
  - "given" means giving data
  - Need determining which node with data "equivalent to" data "given"
  - Reuse the solution of problem of "equivalence" when comparing with the finding value
  - Should not use "data swap" in insert operation
- May use void* of C or template of C++

ABSTRACT LINKED LIST

- Problem: insert before (use template)
  - void main(){
    - int a[] = {1, 2, 4}, b = 3; Node<int>* head = NULL;
    - for(int i = 0; i < 3; i++) insertLast(head, a[i]);
    - **insertBefore**(head, **a[0]**, b);
    - freeList(head);
  - }
  - template <class T>
  - Node<T>* **insertBefore**(Node<T>* &h, const T& X, const T& Val, bool (*cmp)(T*, T*) = isEqual){
    - if(h == NULL) return NULL;
    - Node<T>* newnod = NULL;
    - if(cmp(&(h->data), (T*)&X)){
      - Node<T>* oh = h; newnod = new Node<T>;
      - if(newnod) { h = newnod; h->data = Val; h->pNext = oh; }
      - return h;
    - }
    - Node<T>* cur = h, *pNext = cur->pNext;
    - while(pNext){
      - if(cmp(&(pNext->data), (T*)&X)) break;
      - cur = cur->pNext; pNext = cur->pNext;
    - }
    - if(pNext){
      - newnod = new Node<T>;
      - if(newnod){
        - newnod->data = Val;
        - newnod->pNext = cur->pNext; cur->pNext = newnod;
      - }
    - }
    - return newnod;
  - }

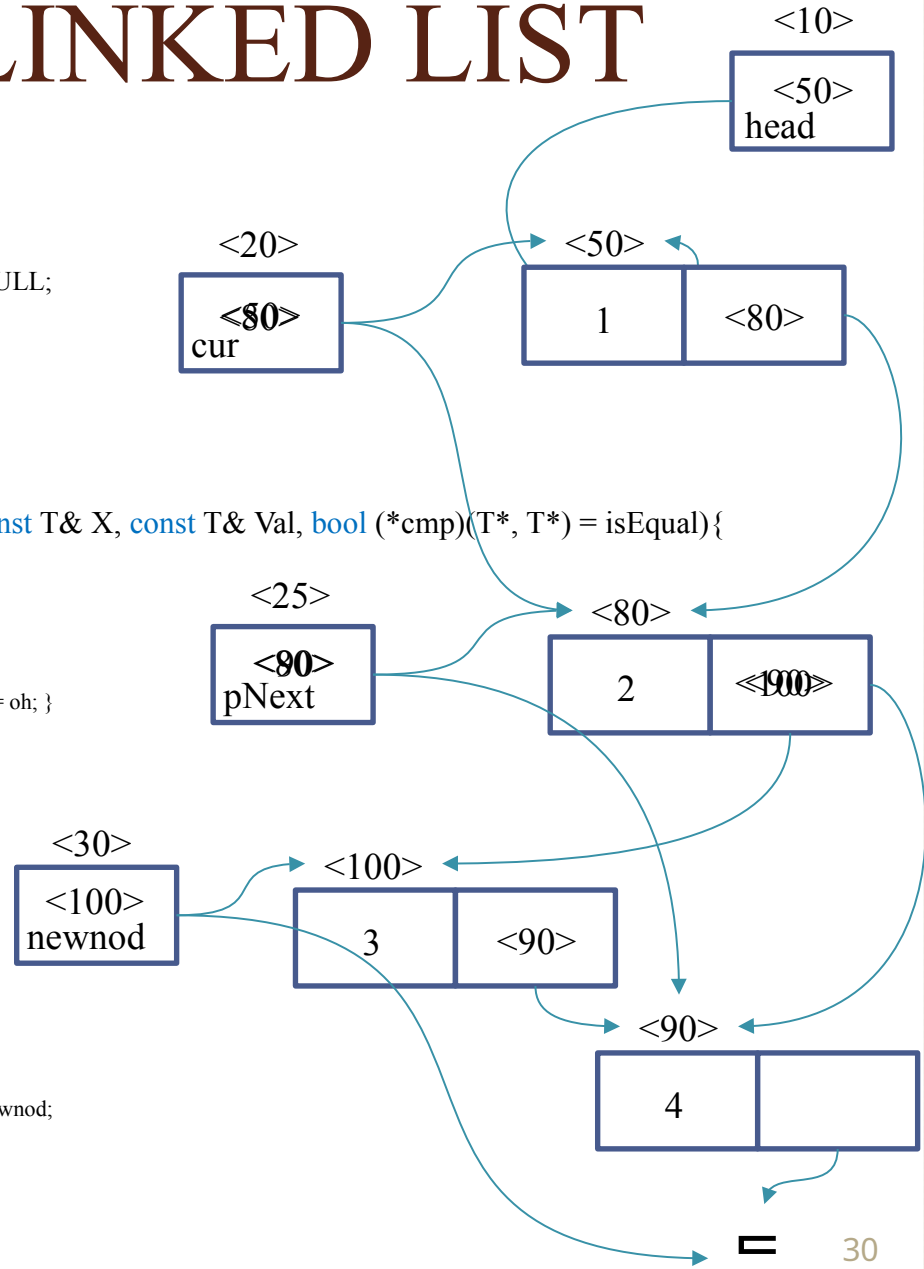
Insert at the beginning of list



ABSTRACT LINKED LIST

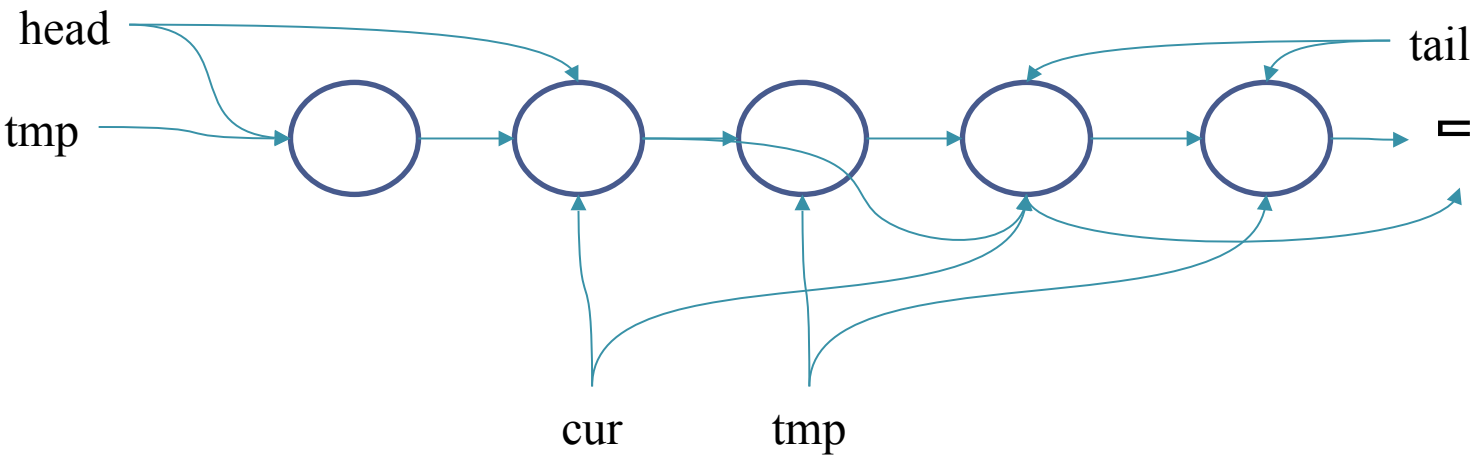
- Problem: insert before (use template)

```
void main(){
    int a[] = {1, 2, 4}, b = 3; Node<int>* head = NULL;
    for(int i = 0; i < 3; i++) insertLast(head, a[i]);
    insertBefore(head, a[2], b);
    freeList(head);
}
template <class T>
Node<T>* insertBefore(Node<T>* &h, const T& X, const T& Val, bool (*cmp)(T*, T*) = isEqual){
    if(h == NULL) return NULL;
    Node<T>* newnod = NULL;
    if(cmp(&(h->data), (T*)&X)){
        Node<T>* oh = h; newnod = new Node<T>;
        if(newnod) { h = newnod; h->data = Val; h->pNext = oh; }
        return h;
    }
    Node<T>* cur = h, *pNext = cur->pNext;
    while(pNext){
        if(cmp(&(pNext->data), (T*)&X)) break;
        cur = cur->pNext; pNext = cur->pNext;
    }
    if(pNext){
        newnod = new Node<T>;
        if(newnod){
            newnod->data = Val;
            newnod->pNext = cur->pNext; cur->pNext = newnod;
        }
    }
    return newnod;
}
```



ABSTRACT LINKED LIST

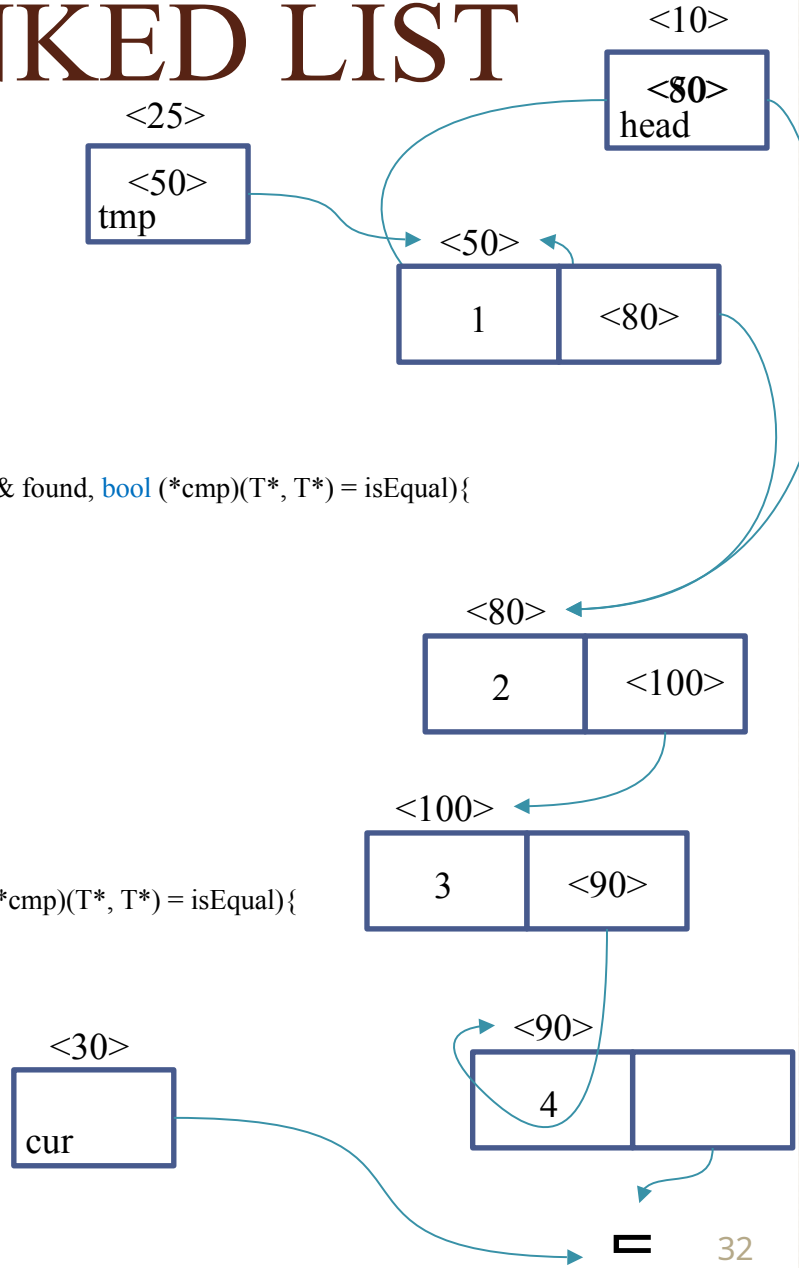
- Problem: deleting a “given” node in linked list
  - “given” means that giving data
  - Need determining which node with data “equivalent to” “given” data
  - Reuse the solution of problem of “equivalence” when comparing with the finding value
  - Need to find a node preceding a node needed to delete
- May use void* of C or template of C++



ABSTRACT LINKED LIST

- Problem: delete the first node (use template)

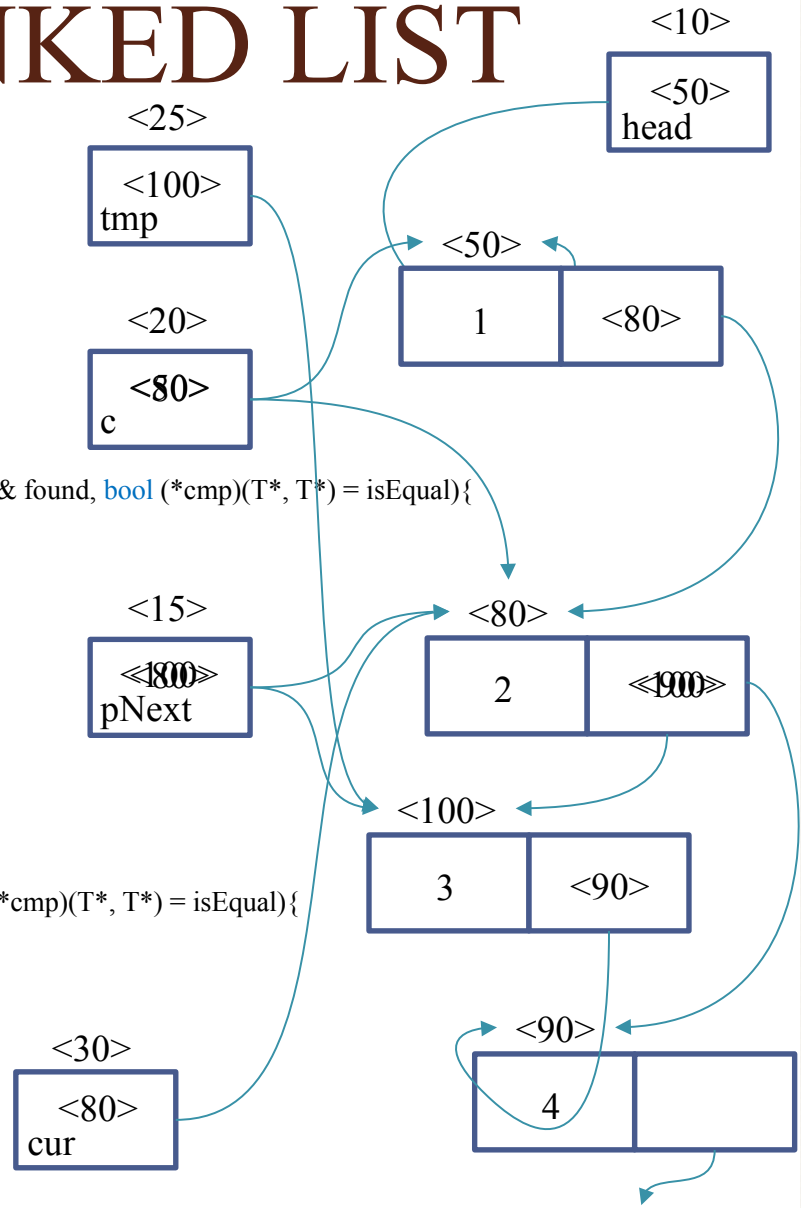
```
void main(){
    int a[] = {1, 2, 3, 4}, b = 1; Node<int>* head = NULL;
    for(int i = 0; i < 4; i++) insertLast(head, a[i]);
    deleteNode(head, b);
    freeList(head);
}
template <class T>
Node<T>* findBeforeX(Node<T>* &h, const T& X, bool& found, bool (*cmp)(T*, T*) = isEqual){
    if(!h) return NULL;
    if(cmp(&(h->data), (T*)&X)){ found = true; return NULL; }
    Node<T>* cur = h, *pNext = cur->pNext;
    while(pNext){
        if(cmp(&(pNext->data), (T*)&X)) break;
        cur = cur->pNext; pNext = cur->pNext;
    }
    if(pNext) { found = true; return cur; }
    return NULL;
}
template <class T>
Node<T>* deleteNode(Node<T>* &h, const T& X, bool (*cmp)(T*, T*) = isEqual){
    bool found = false;
    Node<T>* cur = findBeforeX(h, X, found, cmp), *tmp;
    if(found == false) return found;
    if(cur == NULL) { tmp = h; h = h->pNext; }
    else {
        tmp = cur->pNext;
        cur->pNext = tmp->pNext;
    }
    delete tmp;
    return found;
}
```



ABSTRACT LINKED LIST

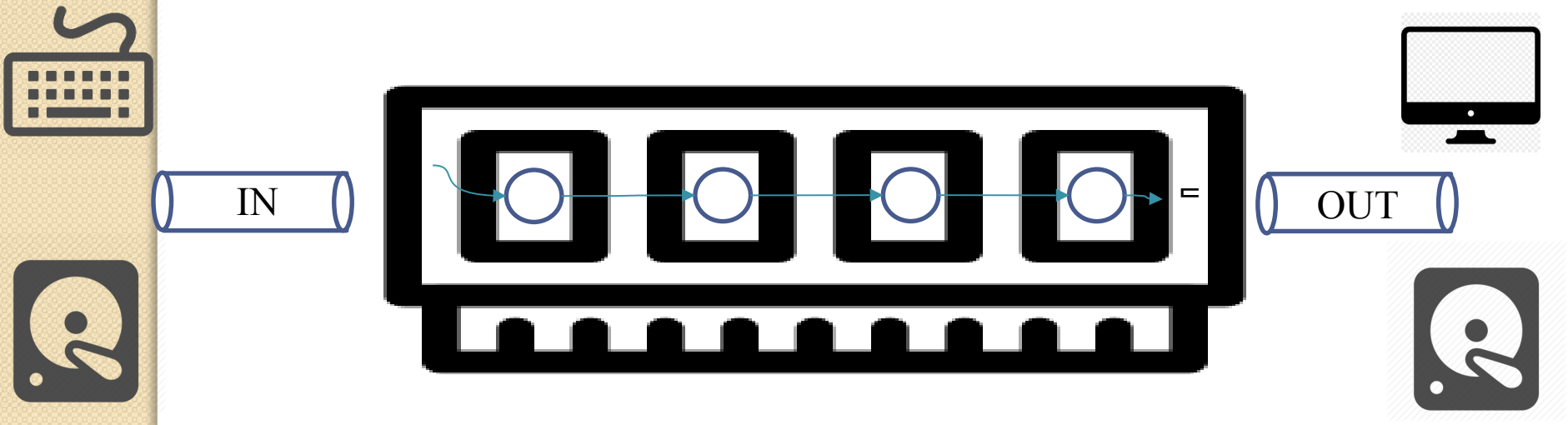
- Problem: delete a node 는 the first node (use template)

```
void main(){
    int a[] = {1, 2, 3, 4}, b = 3; Node<int>* head = NULL;
    for(int i = 0; i < 4; i++) insertLast(head, a[i]);
    deleteNode(head, b);
    freeList(head);
}
template <class T>
Node<T>* findBeforeX(Node<T>* &h, const T& X, bool& found, bool (*cmp)(T*, T*) = isEqual){
    if(!h) return NULL;
    if(cmp(&(h->data), (T*)&X)){ found = true; return NULL; }
    Node<T>* c = h, *pNext = c->pNext;
    while(pNext){
        if(cmp(&(pNext->data), (T*)&X)) break;
        c = c->pNext; pNext = c->pNext;
    }
    if(pNext) { found = true; return c; }
    return NULL;
}
template <class T>
Node<T>* deleteNode(Node<T>* &h, const T& X, bool (*cmp)(T*, T*) = isEqual){
    bool found = false;
    Node<T>* cur = findBeforeX(h, X, found, cmp), *tmp;
    if(found == false) return found;
    if(cur == NULL) { tmp = h; h = h->pNext; }
    else {
        tmp = cur->pNext;
        cur->pNext = tmp->pNext;
    }
    delete tmp;
    return found;
}
```



ABSTRACT LINKED LIST

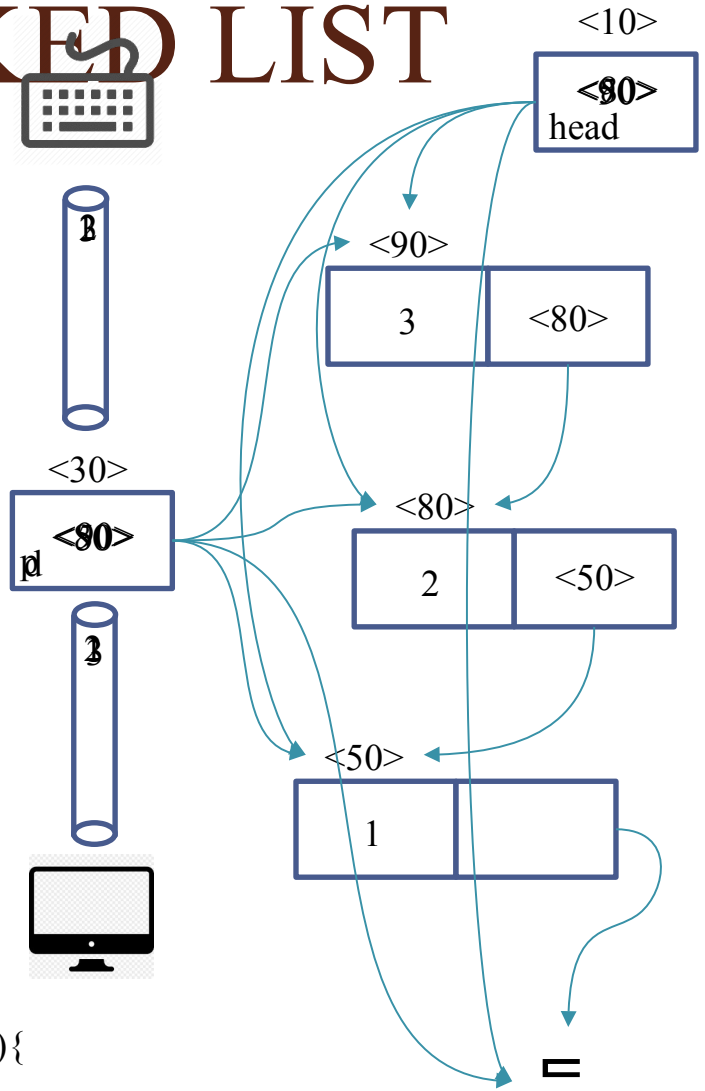
- Problem: input/output in abstract linked list
  - Input from input-device (keyboard, file, scanner…)
  - Output data to output-device (screen, file, printer…)
  - In C++, using stream for input/output
  - Need the corresponding objects standing for input/output device, for example **cin** –| keyboard or **cout** –| screen



ABSTRACT LINKED LIST

- Input/output **reversed order** (use template)

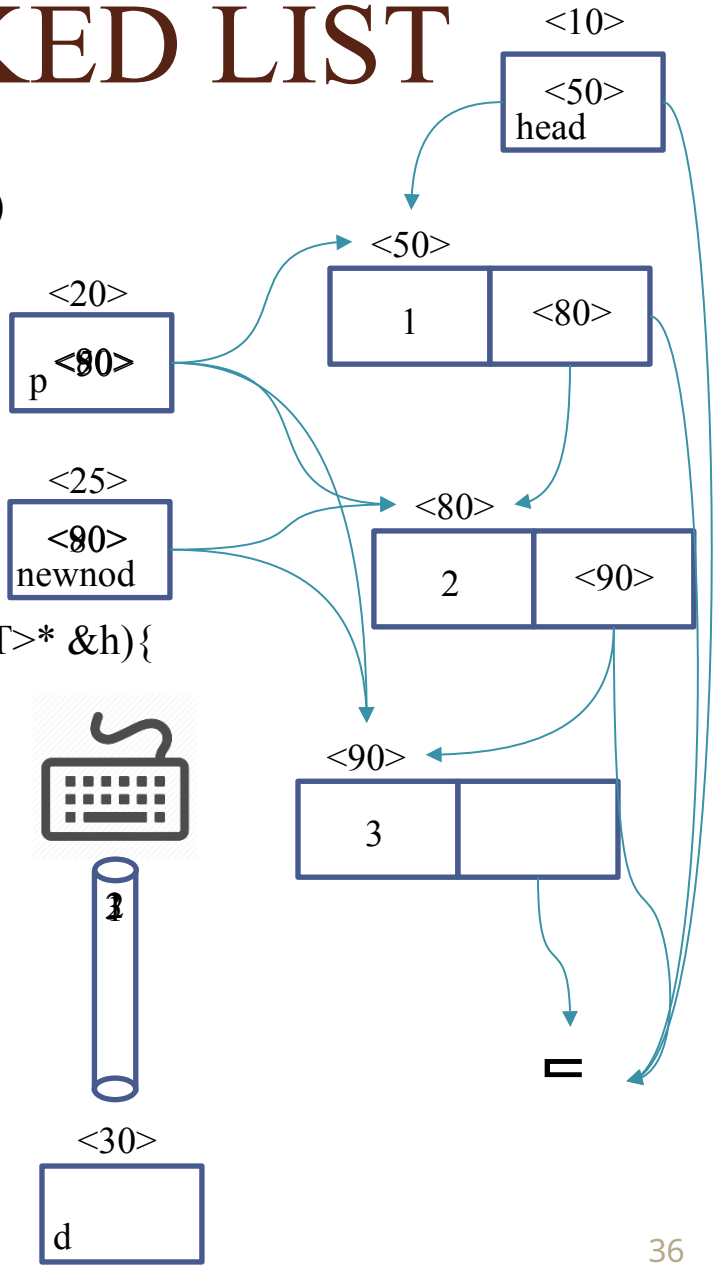
```
void main(){
    ofstream outFile("InData.txt");
        Node<int>* head = NULL;
        readListFrom(cin, head);
        writeListTo(cout, head); // print to screen
        writeListTo(outFile, head); // print to file
        freeList(head);
}
template <class T>
void writeListTo(ostream& oDev, Node<T>* h){
    Node<T>* p = h;
    while(p){
        oDev << p->data << " ";
        p = p->pNext;
    }
}
template <class T>
void readListFrom(istream& iDev, Node<T>* &h){
    T d;
    while(iDev){ // still remain data
        if(iDev >> d) insertFirst(h, d);
    }
}
```



ABSTRACT LINKED LIST

- Input/output **normal order** (use template)

```
void main(){
        Node<int>* head = NULL;
        ReadListFrom(cin, head);
        writeListTo(cout, head); // print to screen
        freeList(head);
}
template <class T>
void ReadListFrom(istream& iDev, Node<T>* &h){
    T d; iDev >> d;
    h = new Node<T>; h->data = d;
    h->pNext = NULL;
    Node<T>* p = h;
    while(iDev){ // still remain data
        if(!(iDev >> d)) break;
        Node<T>* newnod = new Node<T>;
        if(newnod){
            newnod->data = d; newnod->pNext = NULL;
            p->pNext = newnod; p = newnod;
        }
    }
}
```



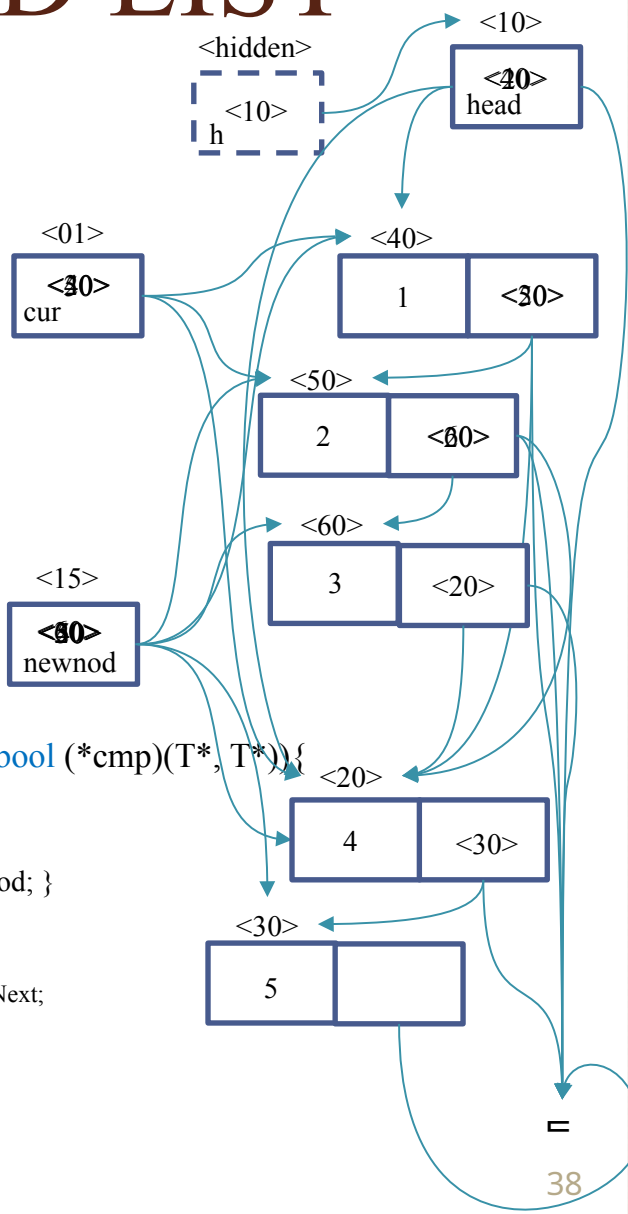
ABSTRACT LINKED LIST

- Problem: input/output for ordered linked list
  - Need to find the right position to insert a new node
  - Insert a new node into just found position
  - Need to write code such that:
    - Generalize for all datatype
    - Generalize for all order-standard
- May use void* of C or template of C++ to generalize for datatype
- Use function-pointer to generalize order-standard

ABSTRACT LINKED LIST

- Problem: input/output for ordered linked list

```
void main(){
    int a[] = {4, 5, 1, 2}, b = 3; Node<int>* head = NULL;
    for(int i = 0; i < 4; i++) insertOrderedList(head, a[i], IntGreater);
    insertOrderedList(head, b, IntGreater);
    freeList(head);
}
bool IntGreater(int* x, int* y){ return *x < *y; }
template <class T>
Node<T>* makeNode(const T& dat){
    Node<T>* newnod = new Node<T>;
    if(newnod){ newnod->data = dat; newnod->pNext = NULL; }
    return newnod; }
template <class T>
Node<T>* insertOrderedList(Node<T>* &h, const T& X, bool (*cmp)(T*, T*)){
    Node<T>* newnod = makeNode(X);
    if(!newnod) return NULL;
    if(!h || cmp((T*)&X, &(h->data))){ newnod->pNext = h; h = newnod; }
    else{
        Node<T>* cur = h;
        while(cur->pNext && !cmp((T*)&X, &(cur->pNext->data))) cur = cur->pNext;
        newnod->pNext = cur->pNext;
        cur->pNext = newnod;
    }
    return newnod;
}
```



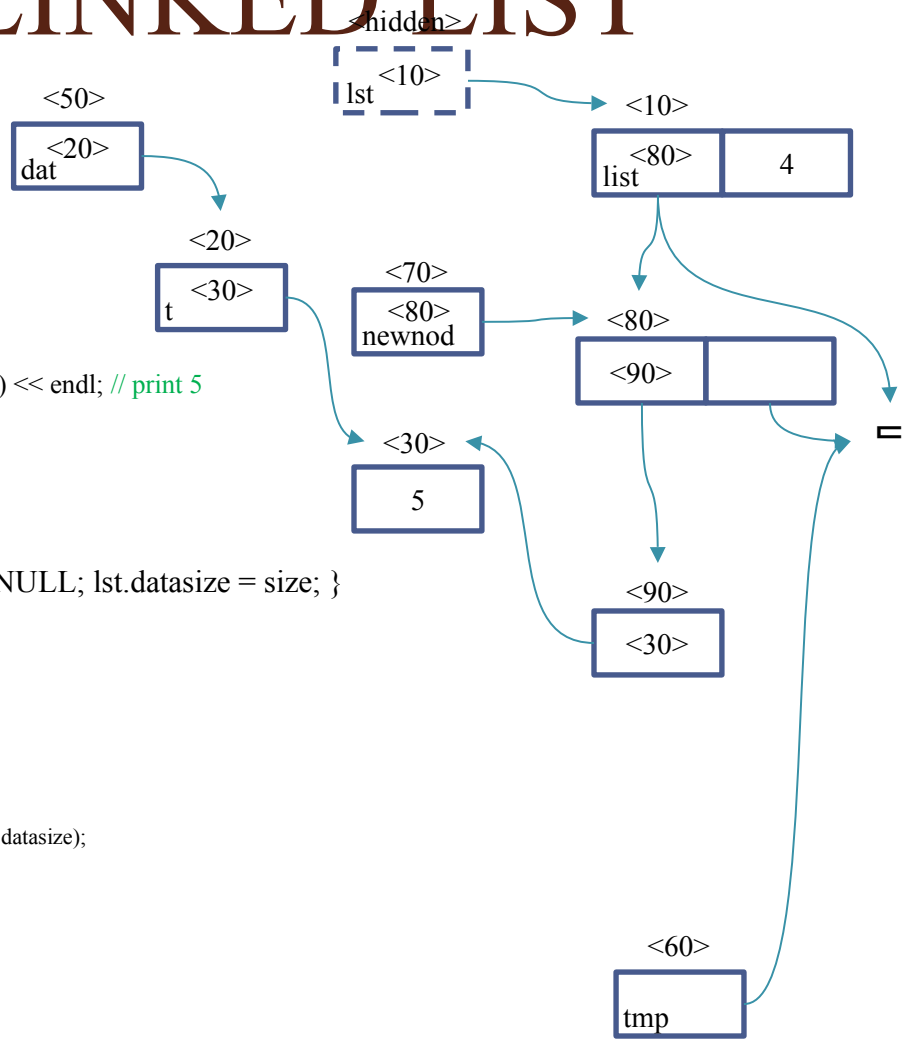
ABSTRACT LINKED LIST

- Some note of single linked list
  - Check if allocation is successful or not
  - Stop before the target node to perform the operations
  - Avoid moving the head pointer if not necessary
- Problem of organize memory in each node: because data-field is void*, when freeing a linked list, a node need to:
  - Free the memory of data pointing to
  - Free the memory of the node
  - Free the linked list

ABSTRACT LINKED LIST

- Example of **leaking memory**

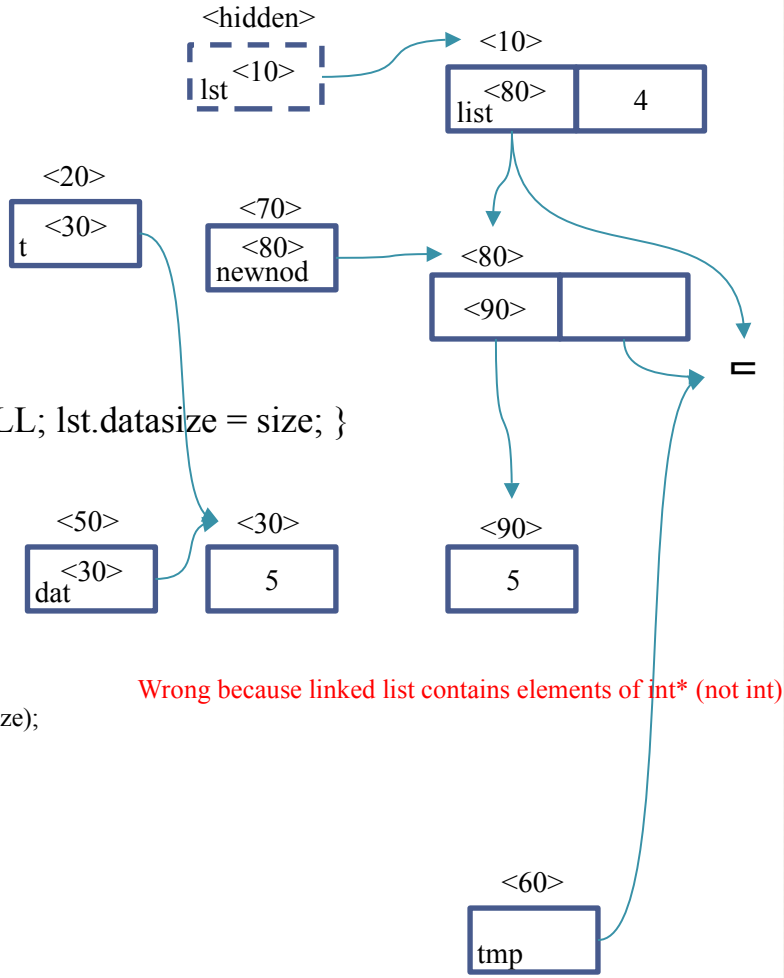
```
void main(){
    List list; initList(list, sizeof(int*));
    int* t = new int; *t = 5;
    insertFirst(list, &t);
    //*(int*)(list.head)->data equal to t
    cout << hex << *(int*)(*(int*)(list.head)->data) << endl; // print 5
    freeList(list);
    cout << *t << endl;
    delete t;
}
void initList(List& lst, int size){ lst.head = NULL; lst.datasize = size; }
void insertFirst(List& lst, void* dat){
    if(!dat) return;
    Node* tmp = lst.head, *newnod = new Node;
    if(newnod){
        lst.head = newnod; newnod->pNext = tmp;
        newnod->data = new char[lst.datasize];
        if(newnod->data) memmove(newnod->data, dat, lst.datasize);
    }
}
void freeList(List& lst){
    Node* tmp = NULL;
    while(lst.head){
        tmp = lst.head->pNext;
        if(lst.head->data) delete[] (char*)(lst.head->data);
        delete lst.head; lst.head = tmp;
    }
}
```



ABSTRACT LINKED LIST

- Example of **leaking memory**

```
void main(){
    List list; initList(list, sizeof(int*));
    int* t = new int; *t = 5;
    insertFirst(list, t);
    //…
}
void initList(List& lst, int size){ lst.head = NULL; lst.datasize = size; }
void insertFirst(List& lst, void* dat){
    if(!dat) return;
    Node* tmp = lst.head, *newnod = new Node;
    if(newnod){
        lst.head = newnod; newnod->pNext = tmp;
        newnod->data = new char[lst.datasize];
        if(newnod->data) memmove(newnod->data, dat, lst.datasize);
    }
}
void freeList(List& lst){
    Node* tmp = NULL;
    while(lst.head){
        tmp = lst.head->pNext;
        if(lst.head->data) delete[] (char*)(lst.head->data);
        delete lst.head; lst.head = tmp;
    }
}
```



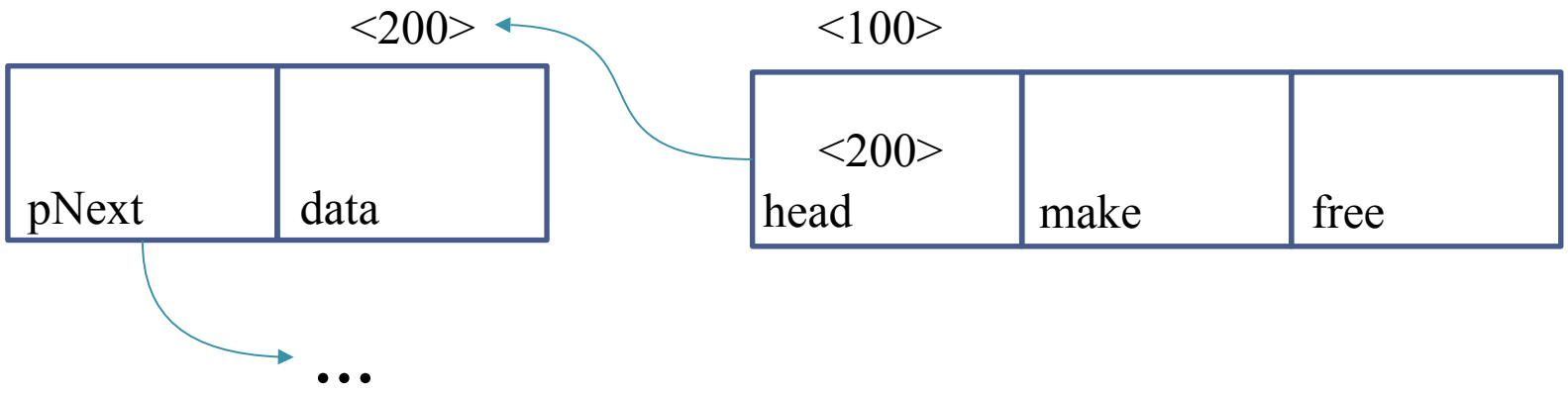
Wrong because linked list contains elements of int* (not int)

ABSTRACT LINKED LIST

- Idea of avoiding leaking memory
  - Delegate a creation of node->data to an external function
  - Delegate a deletion of node->data to an external function
- Solution: using function pointer
- Advantage:
  - Cut back on a number of functions linked list has
  - Flexibly create/delete the memory of node->data
- Drawback: need a countermeasure when external function performs unreasonably

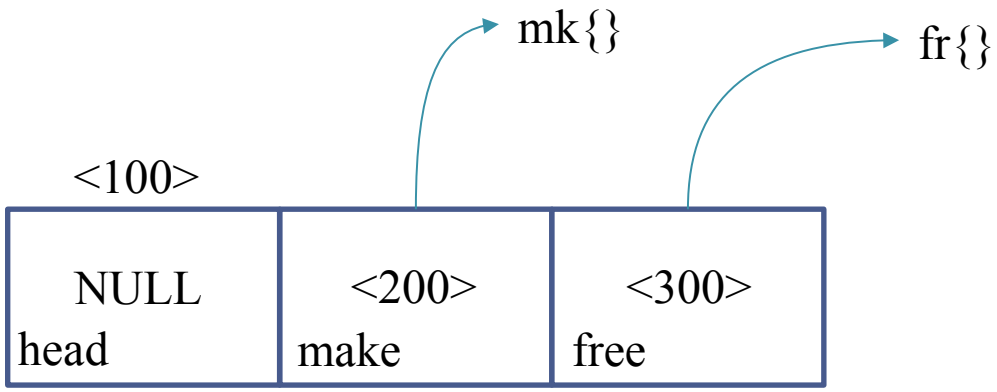
ABSTRACT LINKED LIST

- Re-declare the structure of Node and List
  - struct Node { void* data; Node* pNext; };
  - struct List {
    - Node* head;
    - void* (*make)(void*); void (*free)(void*);
  - };



ABSTRACT LINKED LIST

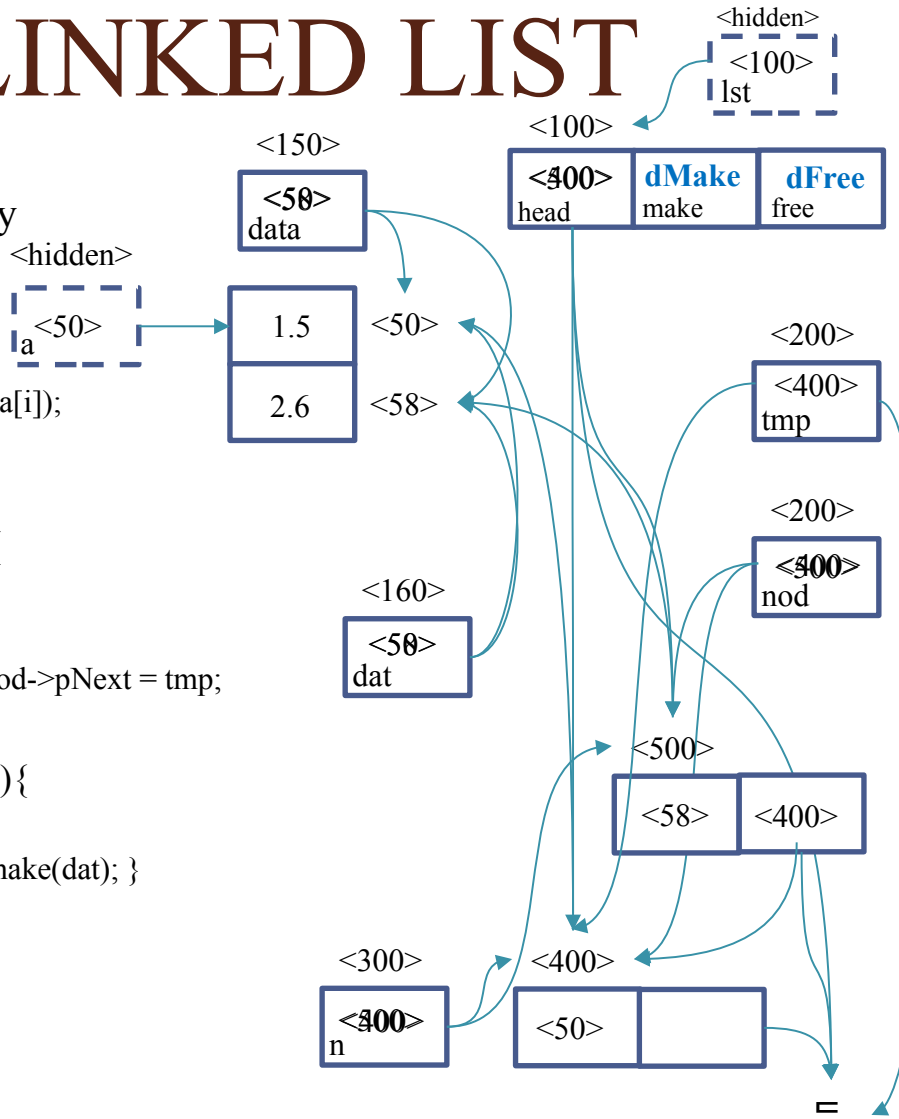
- Build two default functions make & free
  - void* defaultMake(void* dat) { return dat; }: create data for node->data
  - void defaultFree(void* dat) {}: delete data of node->data
- Build initList
  - void initList(List& lst, void* (*mk)(void*), void (*fr)(void*)){
    - lst.head = NULL;
    - mk != NULL ? lst.make = mk : lst.make = defaultMake;
    - fr != NULL ? lst.free = fr : lst.free = defaultFree;
  - }



ABSTRACT LINKED LIST

- Ex: linked list connects to static array

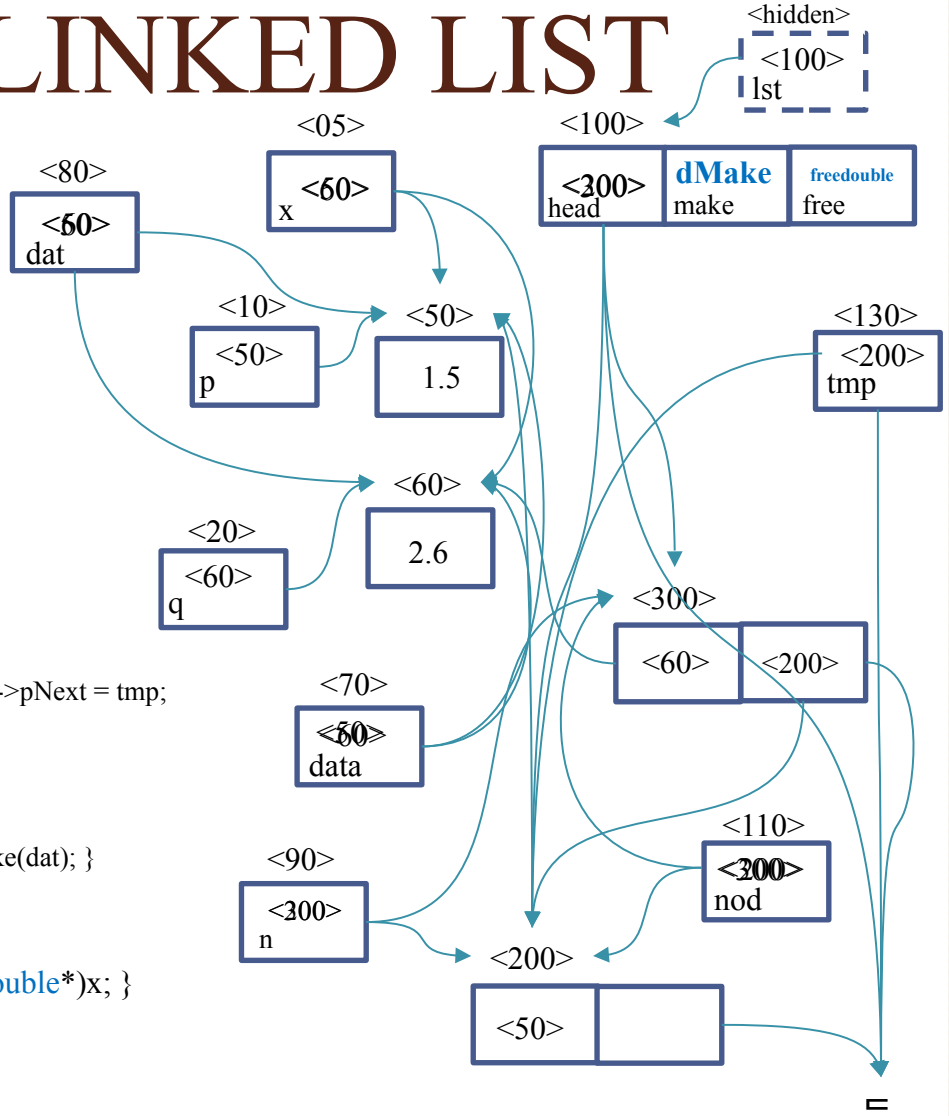
```
void main(){
    List list; double a[] = {1.5, 2.6};
    initList(list, NULL, NULL);
    for(int i = 0; i < 2; i++) insertFirst(list, &a[i]);
            freeList(list);
}
void insertFirst(List& lst, void* data){
    Node* nod = makeNode(lst, data);
    if(!nod || !data) return;
    Node* tmp = lst.head;  lst.head = nod;  nod->pNext = tmp;
}
Node* makeNode(List& lst, void* dat){
    Node* n = new Node;
    if(n){ n->pNext = NULL; n->data = lst.make(dat); }
    return n;
}
void freeList(List& lst){
    while(lst.head){
        Node* tmp = lst.head->pNext;
        if(lst.head->data) lst.free(lst.head->data);
        delete lst.head;
        lst.head = tmp;
    }
}
```



ABSTRACT LINKED LIST

- Ex: linked list connects to heap

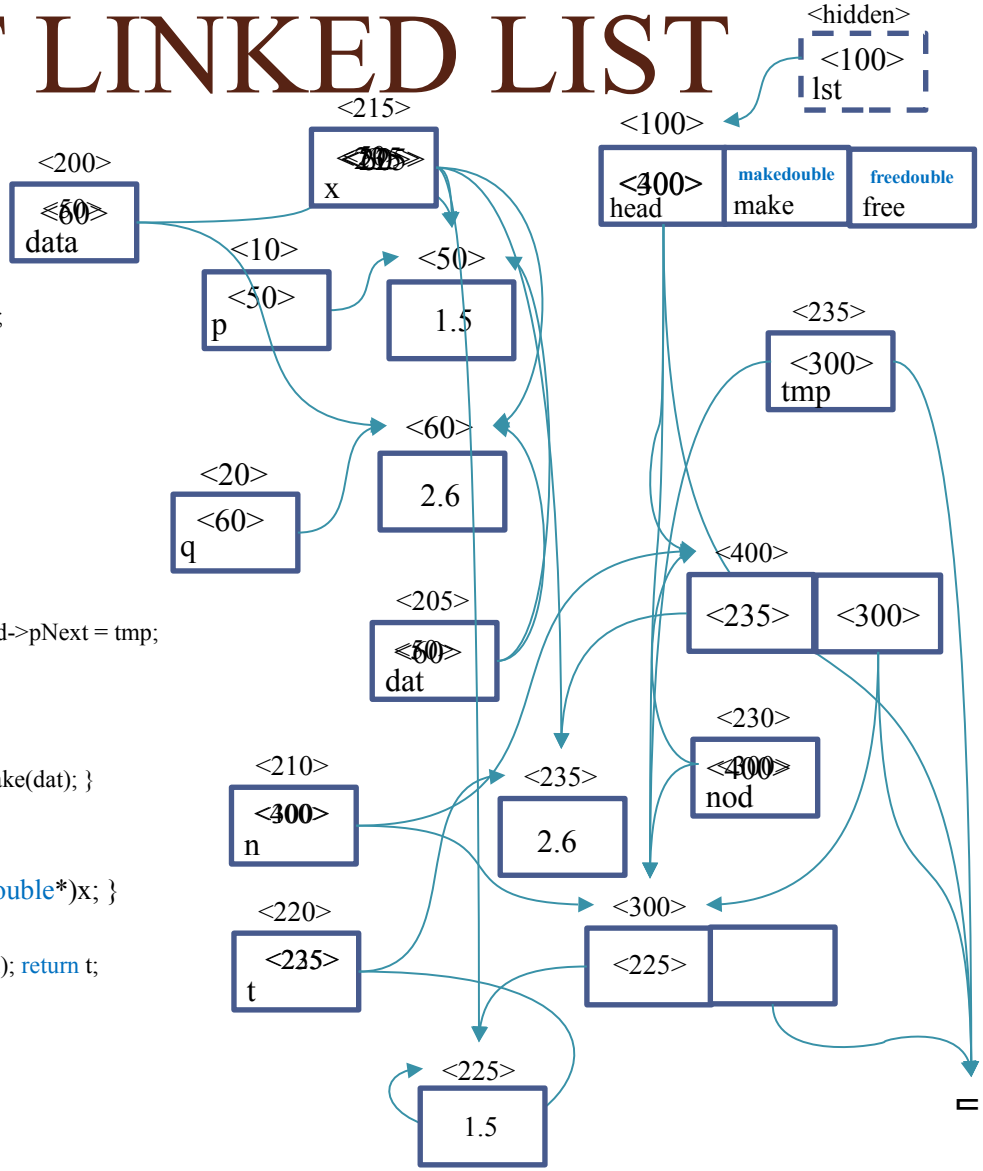
```
void main(){
    List list;
    double* p = new double, *q = new double;
    *p = 1.5; *q = 2.6;
    initList(list, NULL, freedouble);
    insertFirst(list, p); insertFirst(list, q);
        freeList(list);
}
void insertFirst(List& lst, void* data){
    Node* nod = makeNode(lst, data);
    if(!nod || !data) return;
    Node* tmp = lst.head;  lst.head = nod;  nod->pNext = tmp;
}
Node* makeNode(List& lst, void* dat){
    Node* n = new Node;
    if(n){ n->pNext = NULL; n->data = lst.make(dat); }
    return n;
}
void freedouble(void* x){ if(x) delete (double*)x; }
void freeList(List& lst){
    while(lst.head){
        Node* tmp = lst.head->pNext;
        if(lst.head->data) lst.free(lst.head->data);
        delete lst.head;
        lst.head = tmp;
    }
}
```



ABSTRACT LINKED LIST

- Ex: linked list connects to heap

```
void main(){
    List list;
    double* p = new double, *q = new double;
    *p = 1.5; *q = 2.6;
    initList(list, makedouble, freedouble);
    insertFirst(list, p); insertFirst(list, q);
        freeList(list);
}
void insertFirst(List& lst, void* data){
    Node* nod = makeNode(lst, data);
    if(!nod || !data) return;
    Node* tmp = lst.head;  lst.head = nod;  nod->pNext = tmp;
}
Node* makeNode(List& lst, void* dat){
    Node* n = new Node;
    if(n){ n->pNext = NULL; n->data = lst.make(dat); }
    return n;
}
void freedouble(void* x){ if(x) delete (double*)x; }
void* makedouble(void* x){
    double* t = new double; *t = *((double*)x); return t;
}
void freeList(List& lst){
    while(lst.head){
        Node* tmp = lst.head->pNext;
        if(lst.head->data) lst.free(lst.head->data);
        delete lst.head;
        lst.head = tmp;
    }
}
```



ABSTRACT LINKED LIST

- Build a linked list of polynomials
  - struct Polynomial { int n; double* a; };
- Need overloading operator "<<" to print a variable with Polynomial type
  - ostream& operator<<(ostream& oDev, const Polynomial& P){

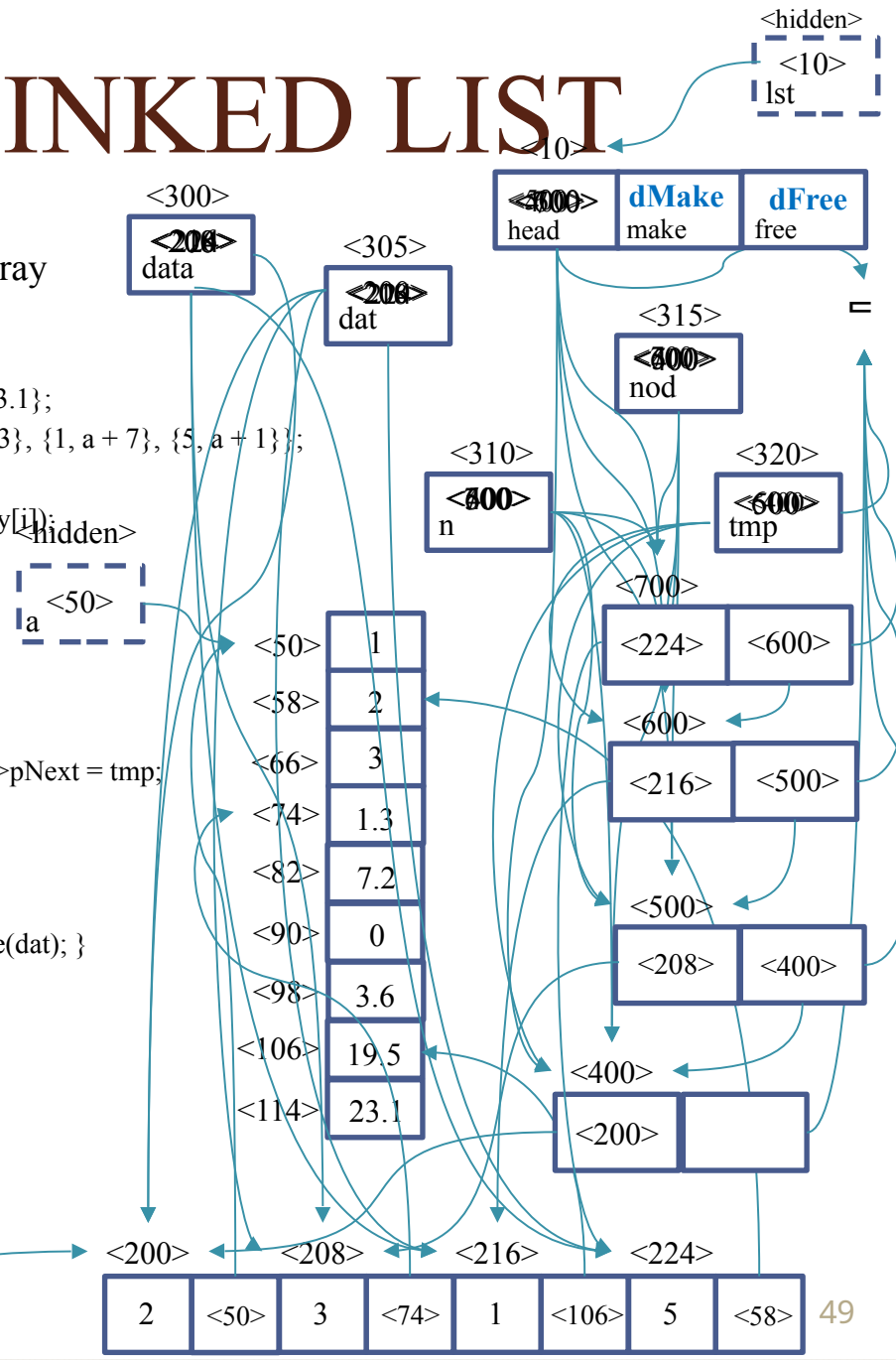
```
for(int i = P.n; i > 0; i--){
    if(P.a[i] == 0) continue;
    if(i > 1) oDev << P.a[i] << "x^" << i << " + ";
    else oDev << P.a[i] << "x" << " + ";
}
oDev << P.a[0];
return oDev;
```

  - }

ABSTRACT LINKED LIST

- Linked list of polynomials from static array

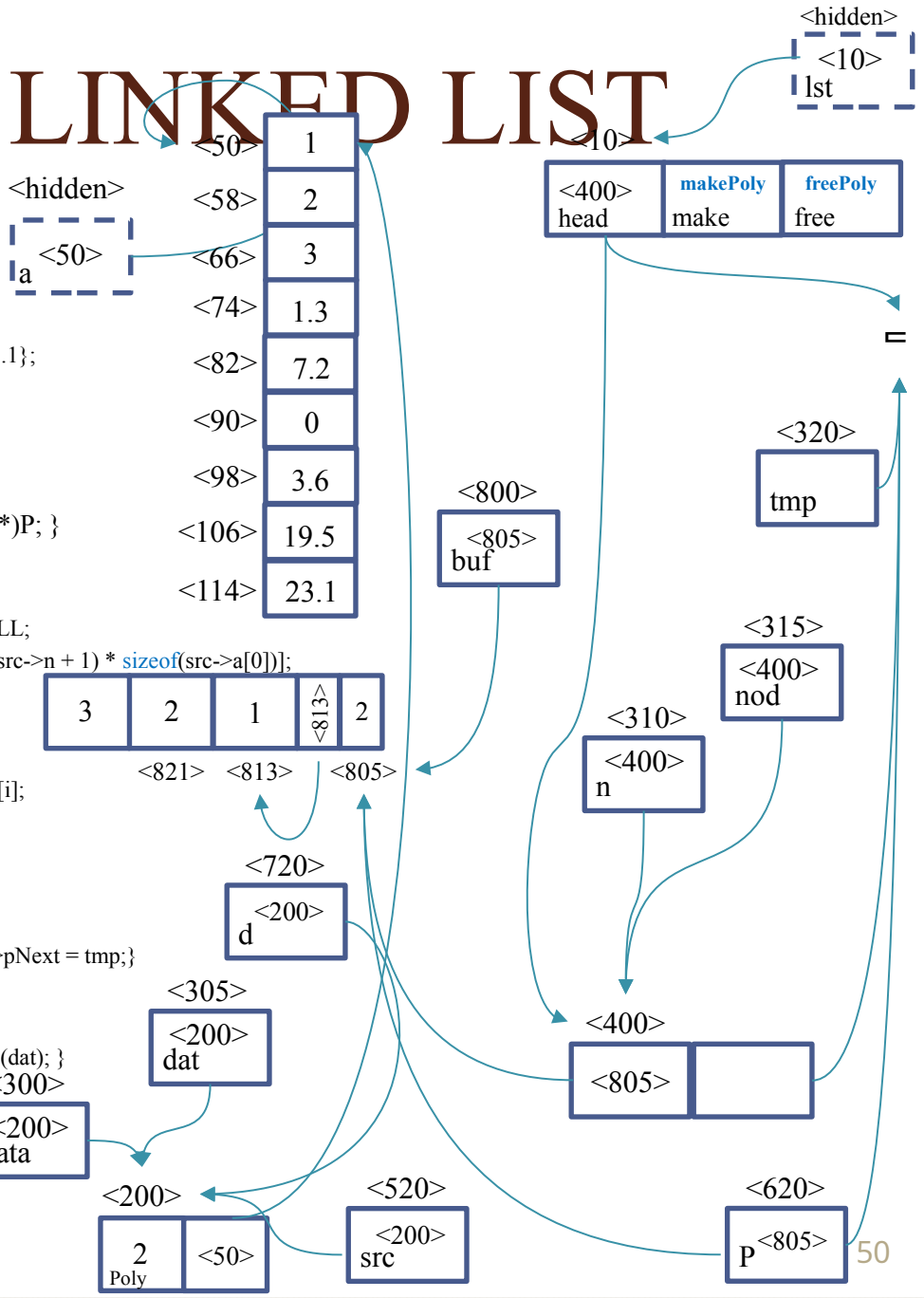
```
void main(){
    List list;
    double a[] = {1, 2, 3, 1.3, 7.2, 0, 3.6, 19.5, 23.1};
        Polynomial Poly[] = {{2, a}, {3, a + 3}, {1, a + 7}, {5, a + 1}};
    initList(list, NULL, NULL);
    for(int i = 0; i < 4; i++) insertFirst(list, &Poly[i]);
        freeList(list);
}
void insertFirst(List& lst, void* data){
    Node* nod = makeNode(lst, data);
    if(!nod || !data) return;
    Node* tmp = lst.head;  lst.head = nod;  nod->pNext = tmp;
}
Node* makeNode(List& lst, void* dat){
    Node* n = new Node;
    if(n){ n->pNext = NULL; n->data = lst.make(dat); }
    return n;
}
void freeList(List& lst){
    while(lst.head){
        Node* tmp = lst.head->pNext;
        if(lst.head->data) lst.free(lst.head->data);
        delete lst.head;
        lst.head = tmp;
    }
}
```



ABSTRACT LINKED LIST

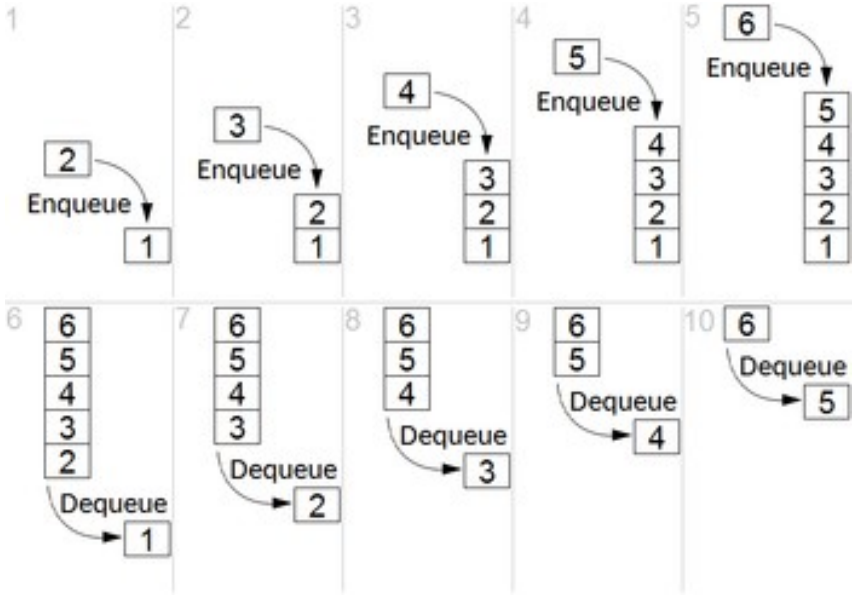
- Linked list of polynomials in heap

```
void main(){
    List list;
    double a[] = {1, 2, 3, 1.3, 7.2, 0, 3.6, 19.5, 23.1};
        Polynomial Poly = {2, a};
    initList(list, makePoly, freePoly);
    insertFirst(list, &Poly); freeList(list);
}
void freePoly(void* P){ if(P) delete[] (char*)P; }
void* makePoly(void* d){
    if(!d) return NULL;
    Polynomial* src = (Polynomial*)d, *P = NULL;
    char* buf = new char[sizeof(Polynomial) + (src->n + 1) * sizeof(src->a[0])];
    if(!buf) return NULL;
    P = (Polynomial*)buf; P->n = src->n;
    P->a = (double*)(buf + sizeof(Polynomial));
    for(int i = 0; i <= P->n; i++) P->a[i] = src->a[i];
    return P; }
void insertFirst(List& lst, void* data){
    Node* nod = makeNode(lst, data);
    if(!nod || !data) return;
    Node* tmp = lst.head;  lst.head = nod;  nod->pNext = tmp;}
Node* makeNode(List& lst, void* dat){
    Node* n = new Node;
    if(n){ n->pNext = NULL; n->data = lst.make(dat); }
    return n; }
void freeList(List& lst){
    while(lst.head){
        Node* tmp = lst.head->pNext;
        if(lst.head->data) lst.free(lst.head->data);
        delete lst.head; lst.head = tmp;
    }}
```



ABSTRACT QUEUE

- Abstract Queue
  - Follow first-in-first-out – FIFO paradigm
  - Implement queue with array or linked list



Source: *https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)*

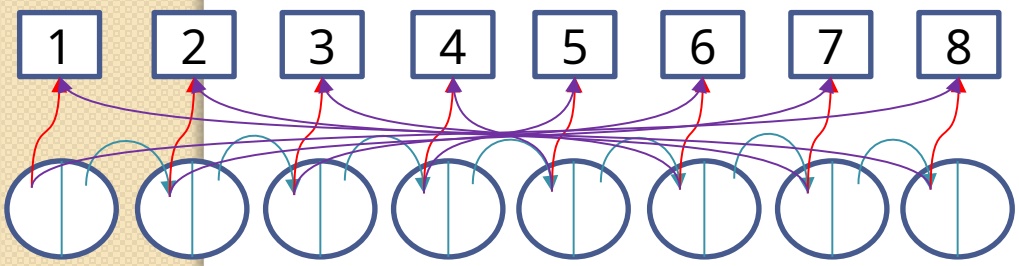
ABSTRACT QUEUE

- Programming interface of abstract queue (using void*)
  - void* initQueue(): initialize queue and return a pointer at the place of **caller**.
  - void closeQueue(void* q): destroy queue at the address q pointing to
  - void enQueue(void* q, void* item): **add** item into queue q (only add to q what item contains)
  - void* deQueue(void* q): **take** the first item out of the queue q and return to the place of caller
  - void* firstQueue(void* q): **read** the first item of queue q and return the place of caller
  - int isEmpty(void* q): check if queue q is empty

ABSTRACT QUEUE

- Programming interface of abstract queue (using void*)

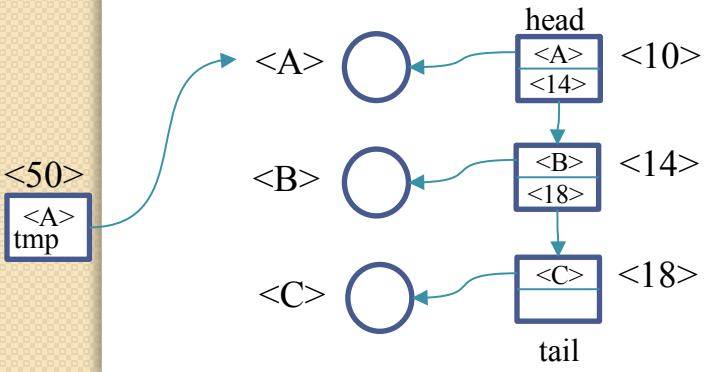
```
//QueueC.h
#ifndef _MY_QUEUE_H_
#define _MY_QUEUE_H_
void* initQueue();
void closeQueue(void* q);
void enQueue(void* q, void* item);
void* deQueue(void* q);
void* firstQueue(void* q);
int isEmpty(void* q);
#endif
```

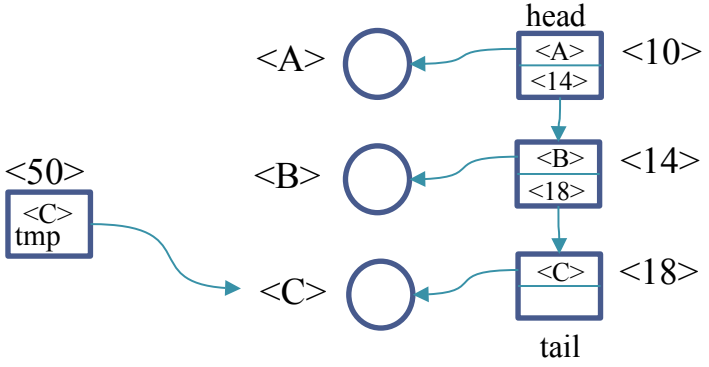



```
#include "QueueC.h"
void takeAllQueue(void* q){
  while(!isEmpty(Q)){
    int x = *(int*)deQueue(Q); cout<<x<<" ";
  }}
void main(){
  int a[] = {1, 2, 3, 4, 5, 6, 7, 8}, i, x, y;
  void* q = initQueue(); i = -1;
  while(++i<8) enQueue(q, &a[i]);
  x = *(int*)deQueue(q);
  takeAllQueue(q); i = 8;
  while(--i) enQueue(q, &a[i]);
  x = *(int*)deQueue(q);
  y = *(int*)deQueue(q);
  takeAllQueue(q); closeQueue(q);
}
```

ABSTRACT QUEUE

- Some conventions of programming interface (using void*)
  - void* firstQueue(void* q)
    - Take out the address the first item pointing to and don't do anything
  - void enQueue(void* q, void* item)
    - Only add address which item pointing to and don't do anything





ABSTRACT QUEUE

- Programming interface of abstract queue (using template)
  - struct Queue {void* data;};
  - void initQueue(Queue<T>& q): initialize queue using reference
  - void closeQueue(Queue<T>& q): destroy queue q contains the address
  - void enQueue(Queue<T>& q, const T& item): **add** item into queue q (only add into q what item contains)
  - T deQueue(Queue<T>& q): **take** the first item out of queue q and return to the place of caller
  - T firstQueue(Queue<T>& q): **read** the first item of queue q and return to the place of caller
  - bool isEmpty(Queue<T>& q): check if queue q is empty

ABSTRACT QUEUE

- Programming interface of abstract queue (using template)

```
//QueueCPP_T.h
#ifndef _MY_QUEUE_CPPT_H_
#define _MY_QUEUE_CPPT_H_
template <class T>
struct Queue {void* data;};
template <class T>
void initQueue(Queue<T>&);
template <class T>
void closeQueue(Queue<T>&);
template <class T>
void enQueue(Queue<T>&, const T&);
template <class T>
T deQueue(Queue<T>&);
template <class T>
T firstQueue(Queue<T>&);
template <class T>
bool isEmpty(Queue<T>&);
#endif
```

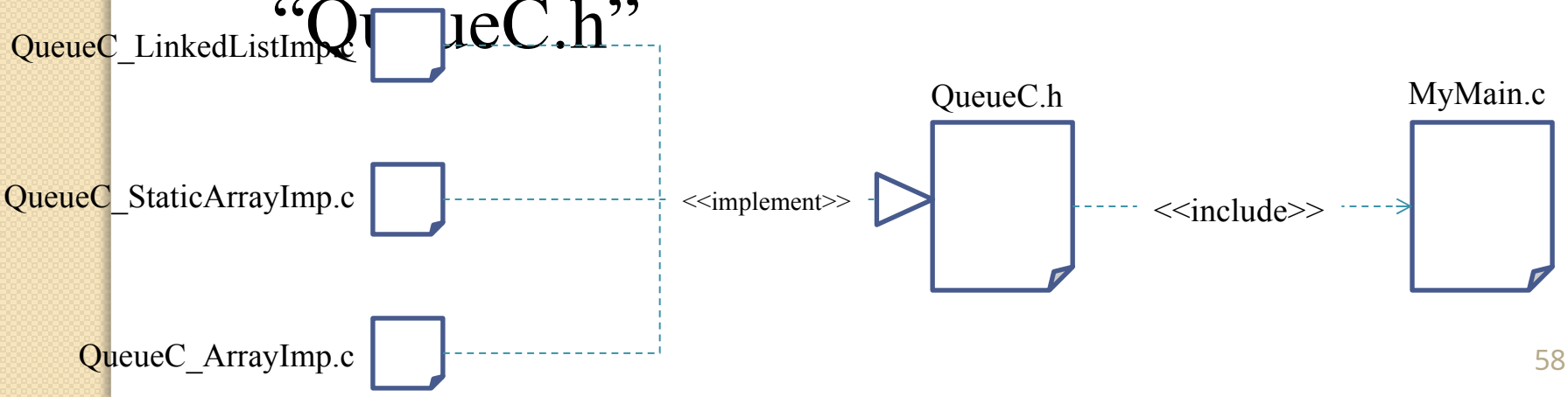
```
#include "QueueCPP_T.h"
#include "QueueCPP_T_Imp.h" // talk later
template <class T>
void takeAllQueue(ostream& o, Queue<T>& q){
  while(!isEmpty(Q)) o << deQueue(Q) << " ";
}
void main(){
  int a[] = {1, 2, 3, 4, 5, 6, 7, 8}, i, x, y;
  Queue<int> q; initQueue(q); i = -1;
  while(++i<8) enQueue(q, a[i]);
  x = deQueue(q);
  takeAllQueue(cout, q); i = 8;
  while(--i) enQueue(q, a[i]);
  x = deQueue(q); y = deQueue(q);
  takeAllQueue(cout, q); closeQueue(q);
}
```

ABSTRACT QUEUE

- Some conventions of programming interface of abstract queue (using template)
  - Two operations firstQueue & enQueue are similar to those of void*
  - There are 2 cases of parameter T:
    - T is pointer type: similar to void*, only receive address passed and don't do anything
    - T is normal type: there are 2 small cases
      - T is a base type, such as int, float, or struct without inner pointer such as struct FRACTION, COMPLEX: need not redefine **operator** '='
      - T is a struct with inner pointer such as struct POLYNOMIAL: need redefine **operator** '='

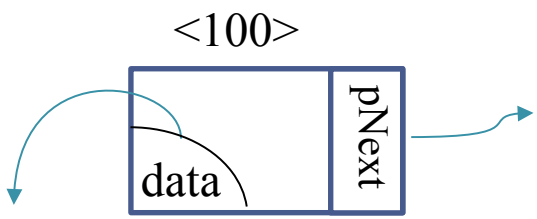
ABSTRACT QUEUE

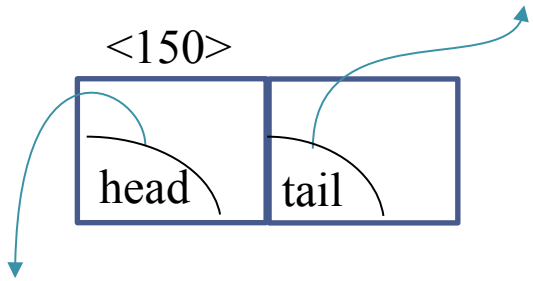
- There are 3 solutions of implementation of abstract queue with void*
  - Linked list: in QueueC_LinkedListImp.c
  - Static array: in QueueC_StaticArrayImp.c
  - Dynamic array: in QueueC_ArrayImp.c
- Note: all files need to include "QueueC.h"



ABSTRACT QUEUE

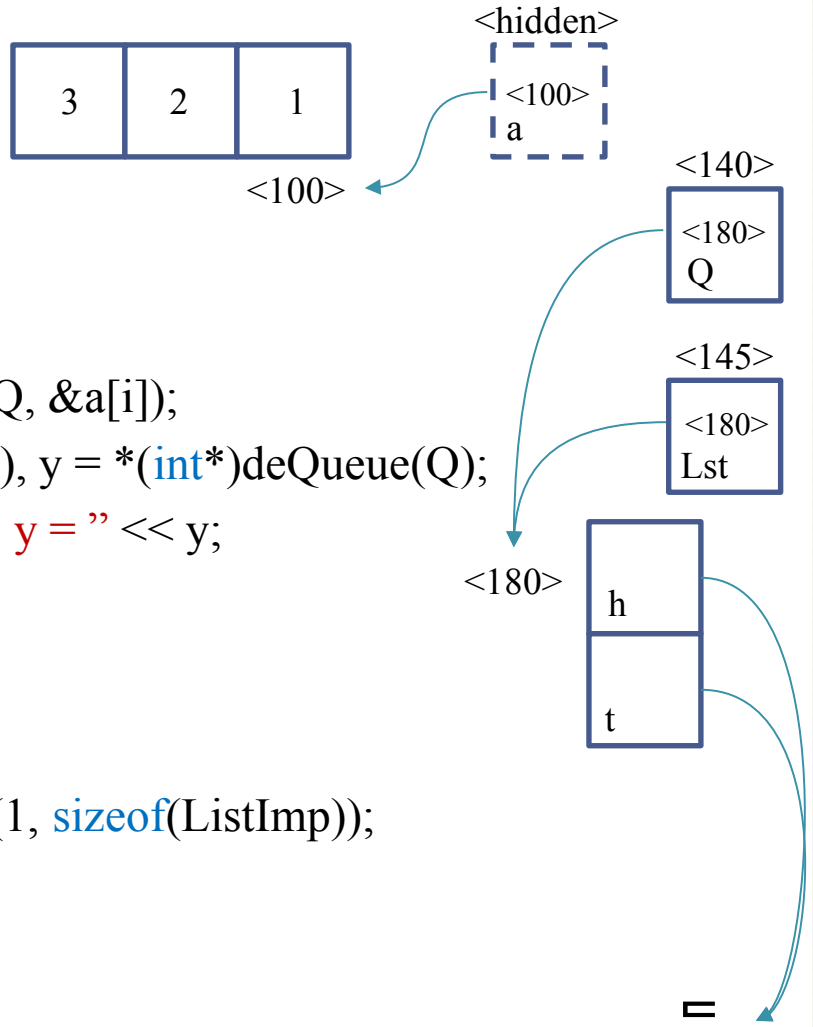
- Ex: using **linked list**
  - struct tagNode{
    - void* data; struct tagNode* pNext;
  - };
  - typedef struct tagNode Node;
  - typedef struct { Node *head, *tail; }ListImp;





ABSTRACT QUEUE

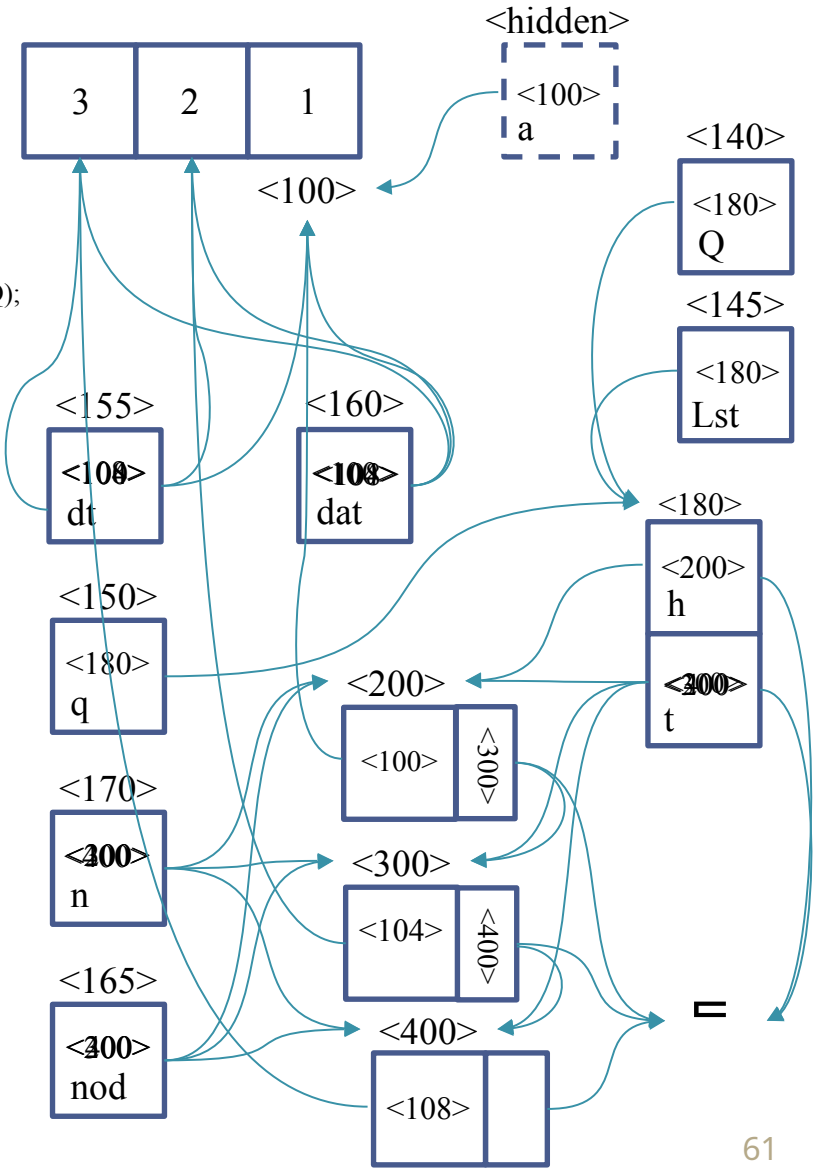
- Ex: using **linked list**
  - void main(){
    - int a[] = {1, 2, 3}, i = -1;
    - void* Q = initQueue();
    - while(++i < 3) enQueue(Q, &a[i]);
    - int x = *(int*)deQueue(Q), y = *(int*)deQueue(Q);
    - cout << "x = " << x << ", y = " << y;
    - closeQueue(Q);
  - }
  - void* initQueue(){
    - ListImp* Lst = (ListImp*)calloc(1, sizeof(ListImp));
    - if(!Lst) return NULL;
    - Lst->head = Lst->tail = NULL;
    - return Lst;
  - }



ABSTRACT QUEUE

- Ex: using **linked list**

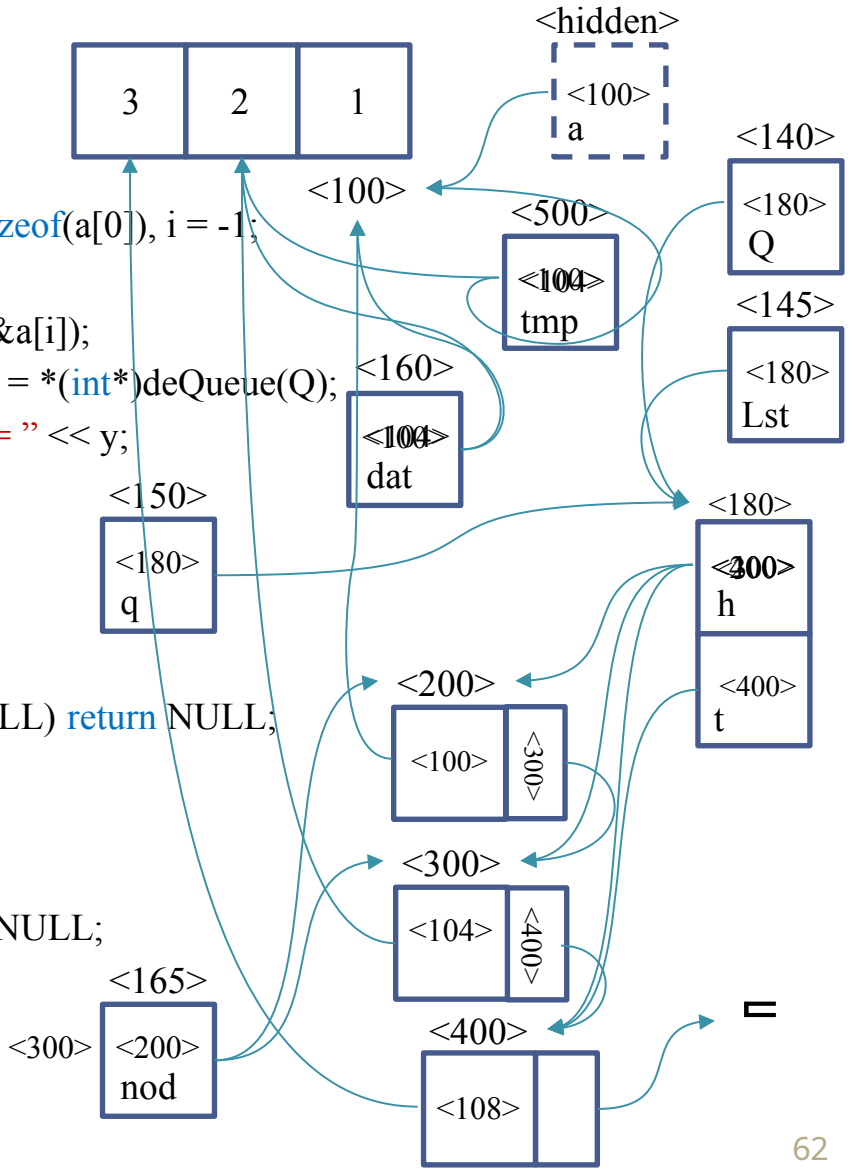
```
void main(){
    int a[] = {1, 2, 3}, i = -1;
        void* Q = initQueue();
        while(++i < 3) enQueue(Q, &a[i]);
        int x = *(int*)deQueue(Q), y = *(int*)deQueue(Q);
        cout << "x = " << x << ", y = " << y;
        closeQueue(Q);
}
Node* makeNode(void* dat){
    Node* nod = (Node*)malloc(sizeof(Node));
    if(nod == NULL) return NULL;
    nod->pNext = NULL;
    nod->data = dat;
    return nod;
}
void enQueue(void* q, void* dt){
    ListImp *Lst = (ListImp*)q;
    if(Lst == NULL) return;
    Node* n = makeNode(dt);
    if(n == NULL) return;
    if(Lst->head == NULL) Lst->head = Lst->tail = n;
    else{
        Lst->tail->pNext = n;
        Lst->tail = Lst->tail->pNext;
    }
}
```



ABSTRACT QUEUE

- Ex: using **linked list**

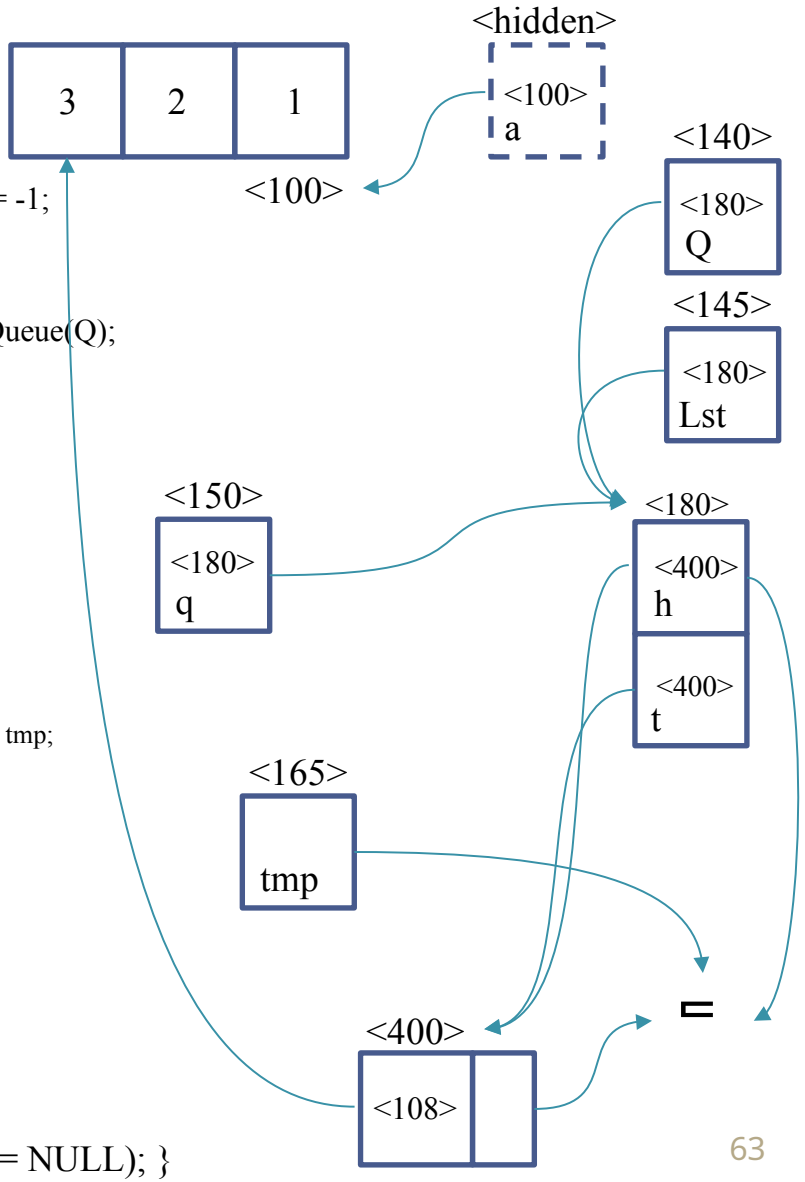
```
void main(){
    int a[] = {1, 2, 3, 4}, n = sizeof(a)/sizeof(a[0]), i = -1;
        void* Q = initQueue();
        while(++i < n) enQueue(Q, &a[i]);
        int x = *(int*)deQueue(Q), y = *(int*)deQueue(Q);
        cout << "x = " << x << ", y = " << y;
        closeQueue(Q);
}
void* deQueue(void* q){
    ListImp* Lst = (ListImp*)q;
    if(Lst == NULL || Lst->head == NULL) return NULL;
    Node* nod = Lst->head;
    void* dat = nod->data;
    Lst->head = Lst->head->pNext;
    if(Lst->head == NULL) Lst->tail = NULL;
    free(nod);
    return dat;
}
```



ABSTRACT QUEUE

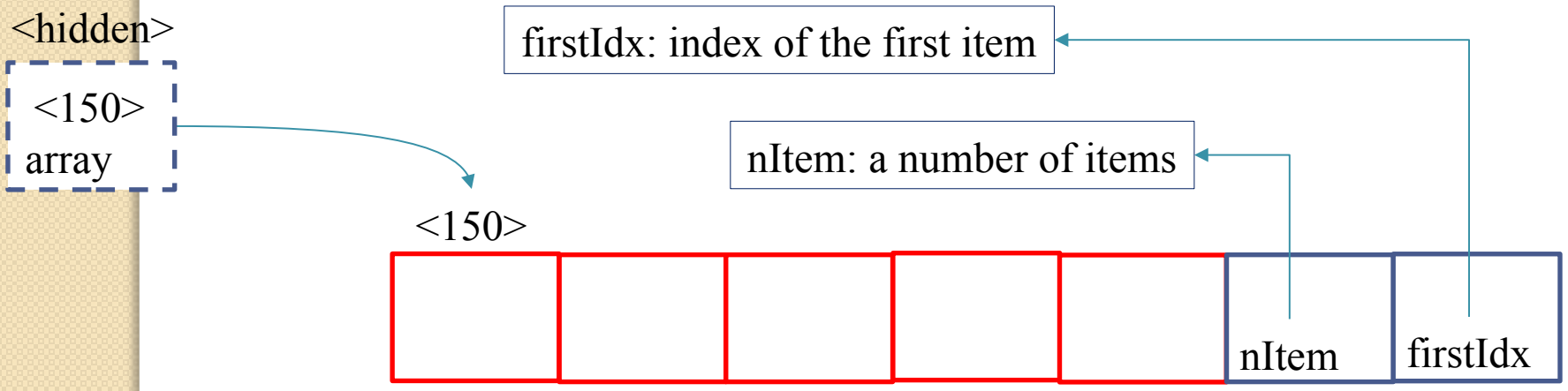
- Ex: using **linked list**

```
void main(){
    int a[] = {1, 2, 3, 4}, n = sizeof(a)/sizeof(a[0]), i = -1;
        void* Q = initQueue();
        while(++i < n) enQueue(Q, &a[i]);
        int x = *(int*)deQueue(Q), y = *(int*)deQueue(Q);
        cout << "x = " << x << ", y = " << y;
        closeQueue(Q);
}
void closeQueue(void* q){
    ListImp* Lst = (ListImp*)q;
    Node* tmp = NULL;
    if(Lst == NULL) return;
    while(Lst->head){
        tmp = Lst->head->pNext; free(Lst->head); Lst->head = tmp;
    }
}
void* firstQueue(void* q){
    ListImp* Lst = (ListImp*)q;
    if(Lst == NULL) return NULL;
    if(Lst->head) return Lst->head->data;
    return NULL;
}
int isEmpty(void* q){ return (firstQueue(q) == NULL); }
```



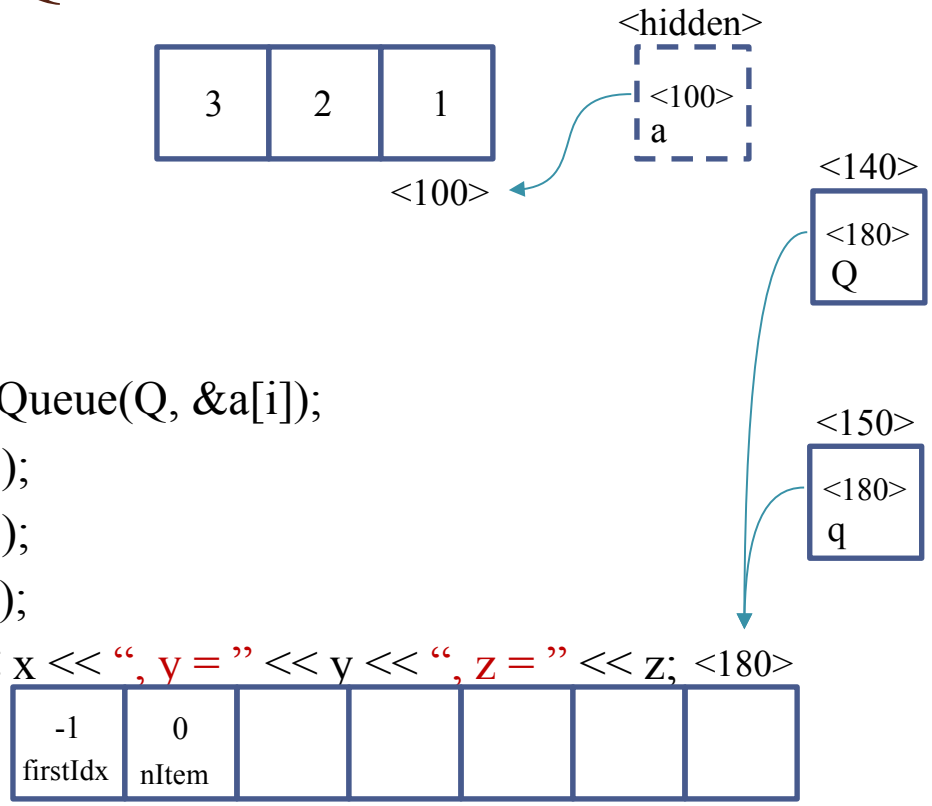
ABSTRACT QUEUE

- Ex: using **static array**
  - #define MAXSZ 5//maximum size of queue
  - typedef struct{
    - void* array[MAXSZ];
    - int nItem, fisrtIdx;
  - } ArrayImp;



ABSTRACT QUEUE

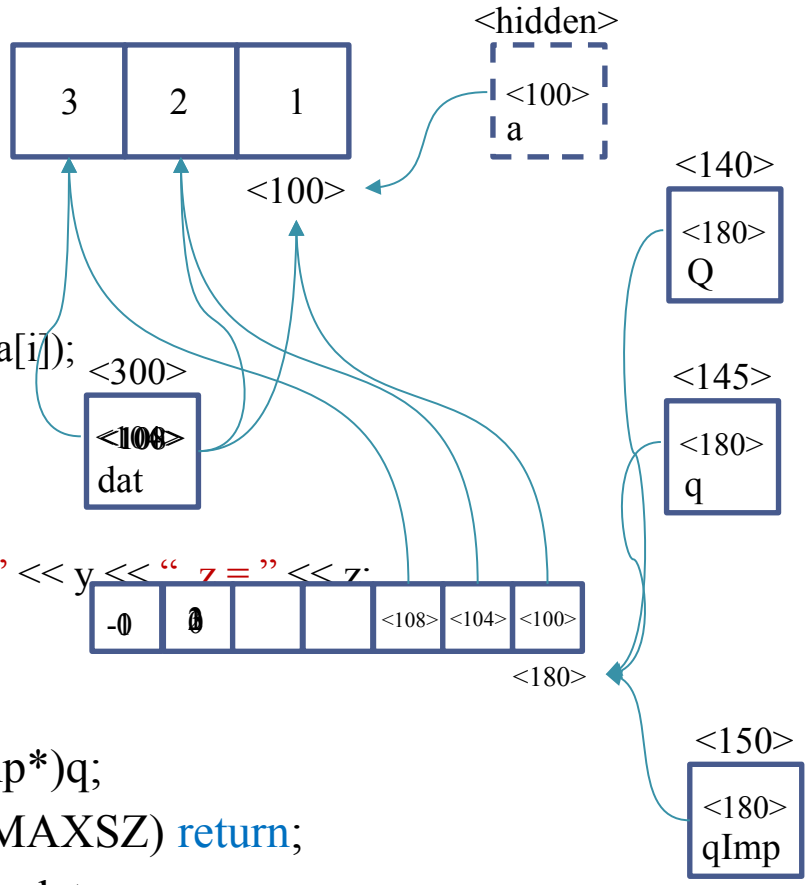
- Ex: using **static array**
  - void main(){
    - int a[] = {1, 2, 3}, i = -1;
    - void* Q = **initQueue**();
    - while(++i < 3) enQueue(Q, &a[i]);
    - int x = *(int*)deQueue(Q);
    - int y = *(int*)deQueue(Q);
    - int z = *(int*)deQueue(Q);
    - cout << "x = " << x << ", y = " << y << ", z = " << z;
  - }
  - void* initQueue(){
    - ArrayImp *q = (ArrayImp*)malloc(sizeof(ArrayImp));
    - if(q){ q->nItem = 0; q->fisrtIdx = -1; }
    - return q;
  - }



ABSTRACT QUEUE

- Ex: using **static array**

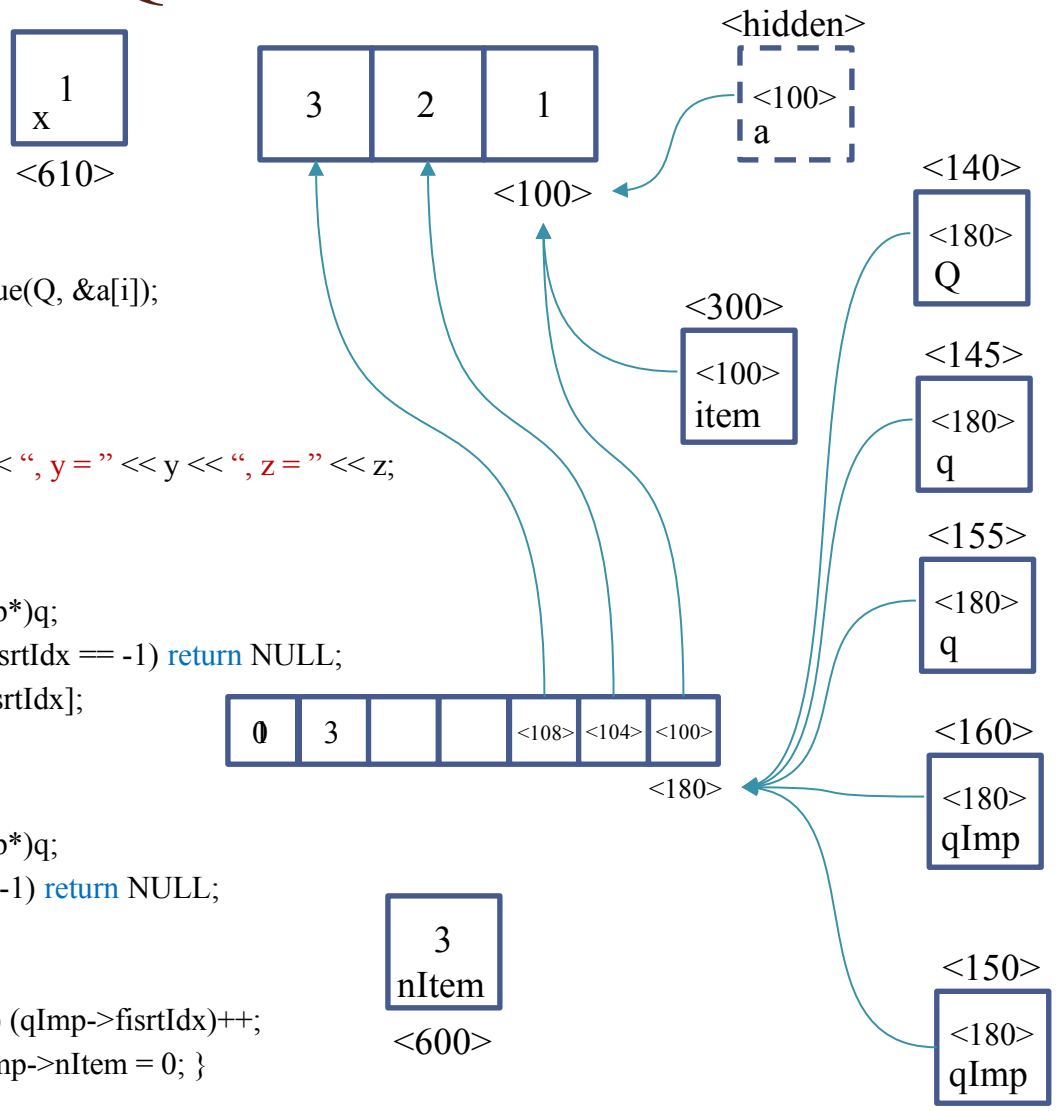
```
void main(){
    int a[] = {1, 2, 3}, i = -1;
    void* Q = initQueue();
        while(++i < 3) enQueue(Q, &a[i]);
    int x = *(int*)deQueue(Q);
    int y = *(int*)deQueue(Q);
    int z = *(int*)deQueue(Q);
        cout << "x = " << x << ", y = " << y << ", z = " << z;
}
void enQueue(void* q, void* dat){
        ArrayImp *qImp = (ArrayImp*)q;
        if(!qImp || qImp->nItem >= MAXSZ) return;
        qImp->array[qImp->nItem] = dat;
        (qImp->nItem)++;
        if(qImp->fisrtIdx == -1) qImp->fisrtIdx = 0;
}
```



ABSTRACT QUEUE

- Ex: using **static array**

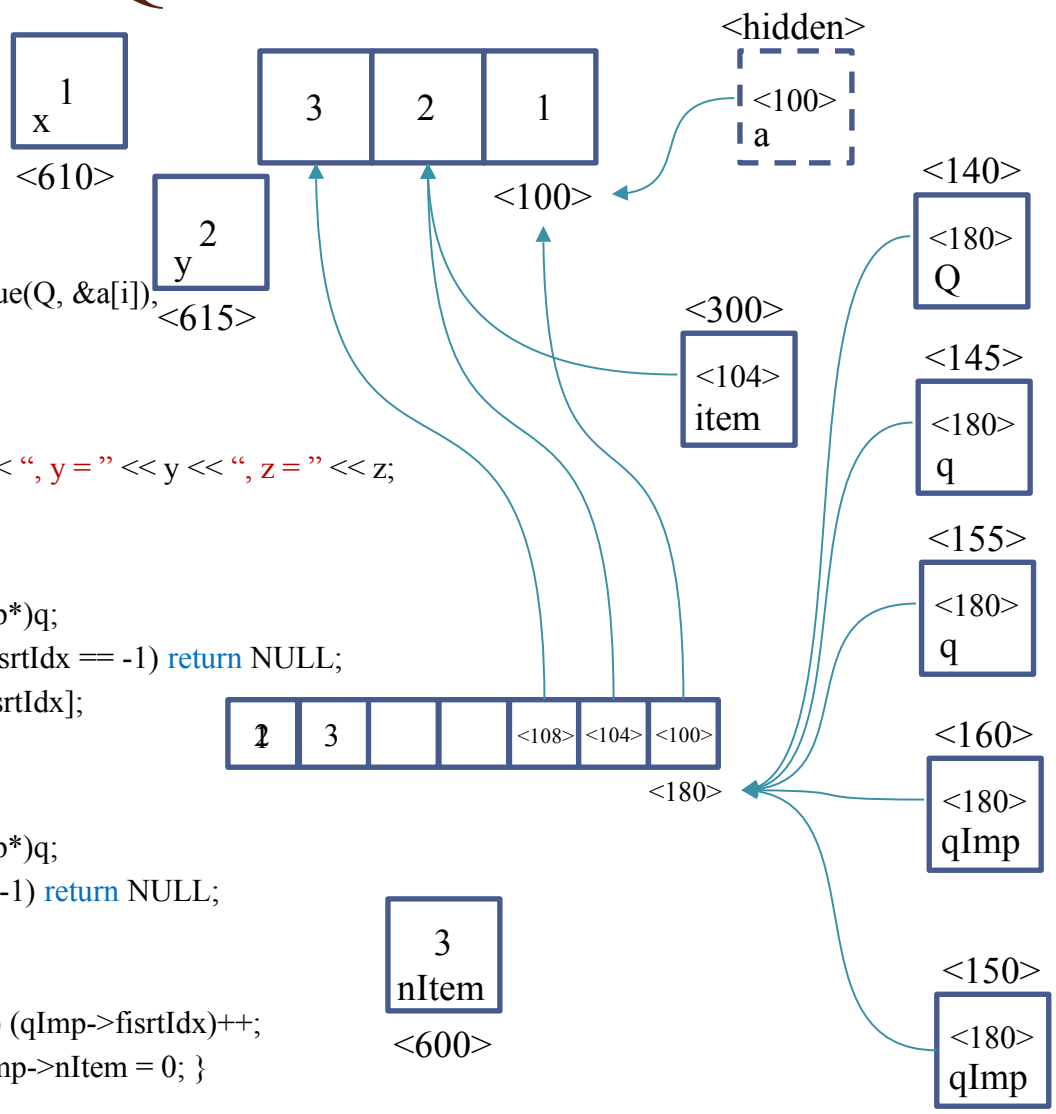
```
void main(){
    int a[] = {1, 2, 3}, i = -1;
    void* Q = initQueue();
        while(++i < 3) enQueue(Q, &a[i]);
    int x = *(int*)deQueue(Q);
    int y = *(int*)deQueue(Q);
    int z = *(int*)deQueue(Q);
        cout << "x = " << x << ", y = " << y << ", z = " << z;
}
void* firstQueue(void* q){
    ArrayImp *qImp = (ArrayImp*)q;
    if(qImp == NULL || qImp->fisrtIdx == -1) return NULL;
    return qImp->array[qImp->fisrtIdx];
}
void deQueue(void* q){
    ArrayImp *qImp = (ArrayImp*)q;
    if(!qImp || qImp->fisrtIdx == -1) return NULL;
    void* item = firstQueue(q);
    int nItem = qImp->nItem;
    if(qImp->fisrtIdx < nItem - 1) (qImp->fisrtIdx)++;
    else{ qImp->fisrtIdx = -1; qImp->nItem = 0; }
    return item;
}
```



ABSTRACT QUEUE

- Ex: using **static array**

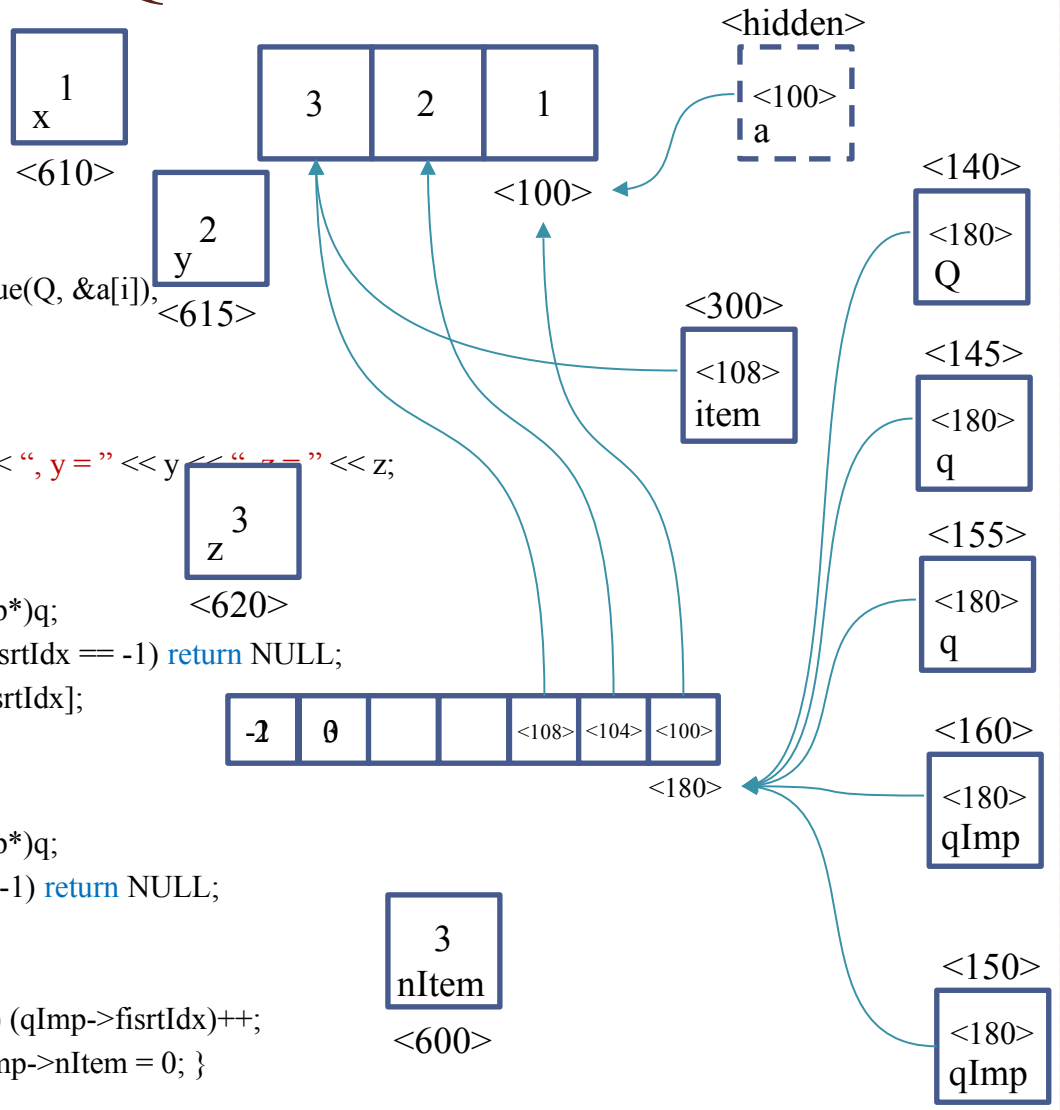
```
void main(){
    int a[] = {1, 2, 3}, i = -1;
    void* Q = initQueue();
        while(++i < 3) enQueue(Q, &a[i]);
    int x = *(int*)deQueue(Q);
    int y = *(int*)deQueue(Q);
    int z = *(int*)deQueue(Q);
        cout << "x = " << x << ", y = " << y << ", z = " << z;
}
void* firstQueue(void* q){
    ArrayImp *qImp = (ArrayImp*)q;
    if(qImp == NULL || qImp->fisrtIdx == -1) return NULL;
    return qImp->array[qImp->fisrtIdx];
}
void deQueue(void* q){
    ArrayImp *qImp = (ArrayImp*)q;
    if(!qImp || qImp->fisrtIdx == -1) return NULL;
    void* item = firstQueue(q);
    int nItem = qImp->nItem;
    if(qImp->fisrtIdx < nItem - 1) (qImp->fisrtIdx)++;
    else{ qImp->fisrtIdx = -1; qImp->nItem = 0; }
    return item;
}
```



ABSTRACT QUEUE

- Ex: using **static array**

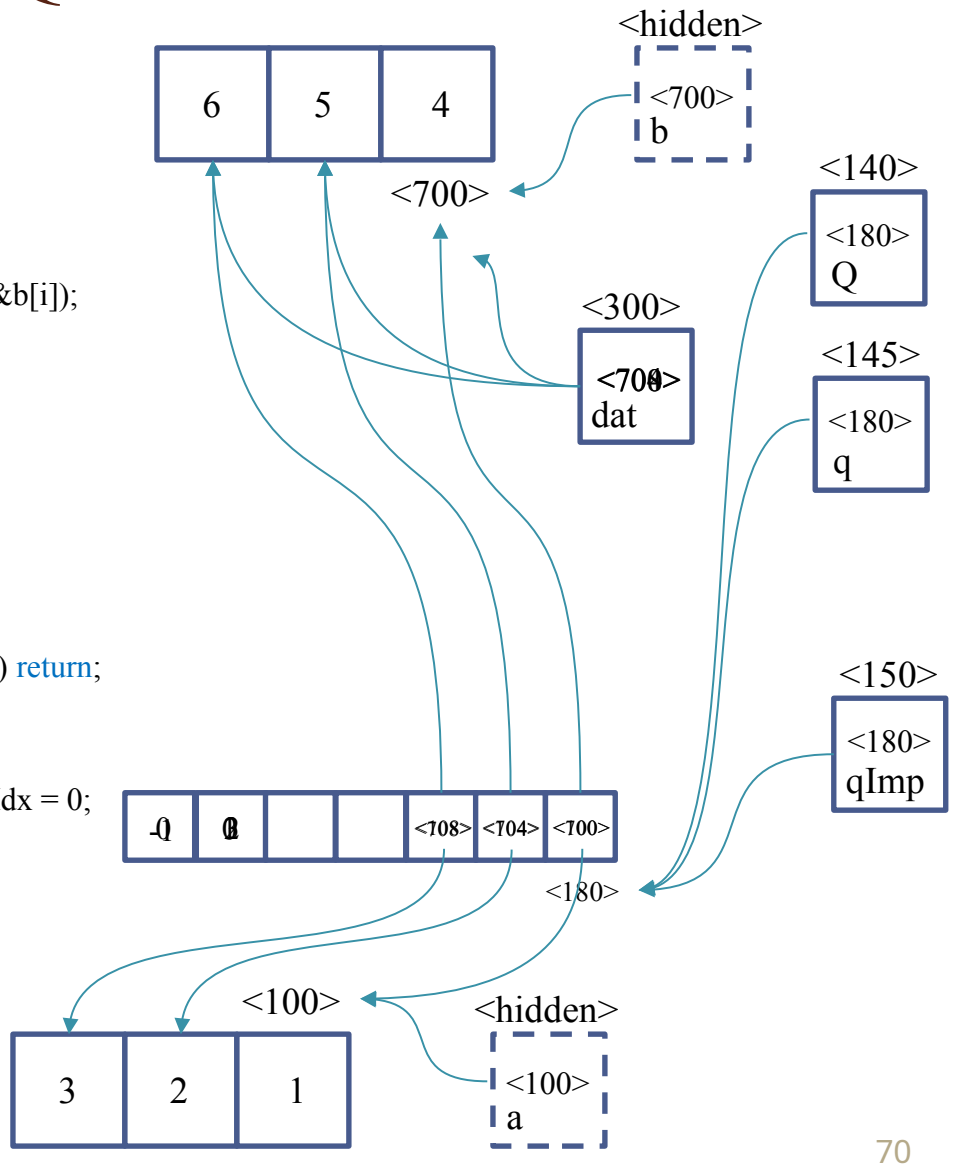
```
void main(){
    int a[] = {1, 2, 3}, i = -1;
    void* Q = initQueue();
        while(++i < 3) enQueue(Q, &a[i]);
    int x = *(int*)deQueue(Q);
    int y = *(int*)deQueue(Q);
    int z = *(int*)deQueue(Q);
        cout << "x = " << x << ", y = " << y << ", z = " << z;
}
void* firstQueue(void* q){
    ArrayImp *qImp = (ArrayImp*)q;
    if(qImp == NULL || qImp->fisrtIdx == -1) return NULL;
    return qImp->array[qImp->fisrtIdx];
}
void deQueue(void* q){
    ArrayImp *qImp = (ArrayImp*)q;
    if(!qImp || qImp->fisrtIdx == -1) return NULL;
    void* item = firstQueue(q);
    int nItem = qImp->nItem;
    if(qImp->fisrtIdx < nItem - 1) (qImp->fisrtIdx)++;
    else{ qImp->fisrtIdx = -1; qImp->nItem = 0; }
    return item;
}
```



ABSTRACT QUEUE

- Ex: using **static array**

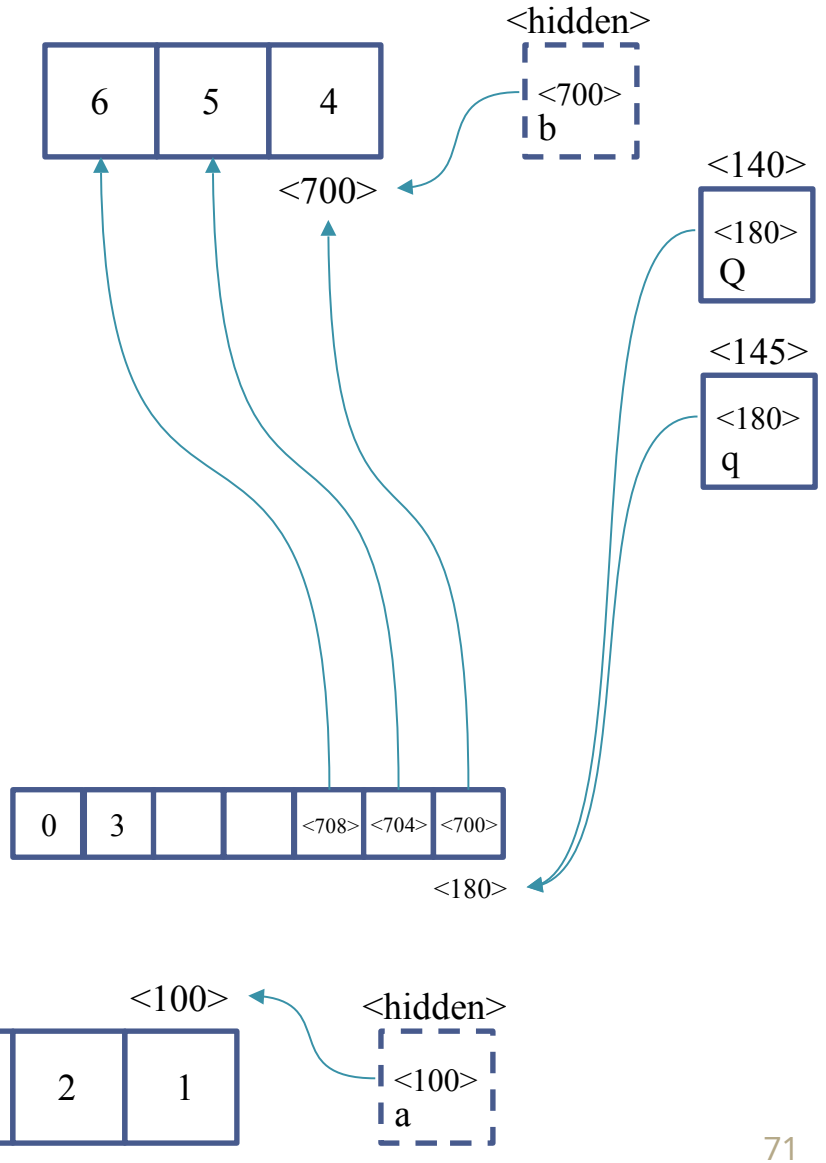
```
void main(){
    //…
    int b[] = {4, 5, 6}; i = -1;
        while(++i < 3) enQueue(Q, &b[i]);
    x = *(int*)deQueue(Q);
    y = *(int*)deQueue(Q);
    z = *(int*)deQueue(Q);
        closeQueue(Q);
}
void enQueue(void* q, void* dat){
    ArrayImp *qImp = (ArrayImp*)q;
    if(!qImp || qImp->nItem >= MAXSZ) return;
    qImp->array[qImp->nItem] = dat;
    (qImp->nItem)++;
    if(qImp->fisrtIdx == -1) qImp->fisrtIdx = 0;
}
int isEmpty(void* q){
    return (firstQueue(q) == NULL);
}
void closeQueue(void* q){
    if(q) free((ArrayImp*)q);
}
```



ABSTRACT QUEUE

- Ex: using **static array**

```
void main(){
    //…
    int b[] = {4, 5, 6}; i = -1;
        while(++i < 3) enQueue(Q, &b[i]);
    x = *(int*)deQueue(Q);
    y = *(int*)deQueue(Q);
    z = *(int*)deQueue(Q);
        closeQueue(Q);
}
void enQueue(void* q, void* dat){
    ArrayImp *qImp = (ArrayImp*)q;
    if(!qImp || qImp->nItem >= MAXSZ) return;
    qImp->array[qImp->nItem] = dat;
    (qImp->nItem)++;
    if(qImp->fisrtIdx == -1) qImp->fisrtIdx = 0;
}
int isEmpty(void* q){
    return (firstQueue(q) == NULL);
}
void closeQueue(void* q){
    if(q) free((ArrayImp*)q);
}
```

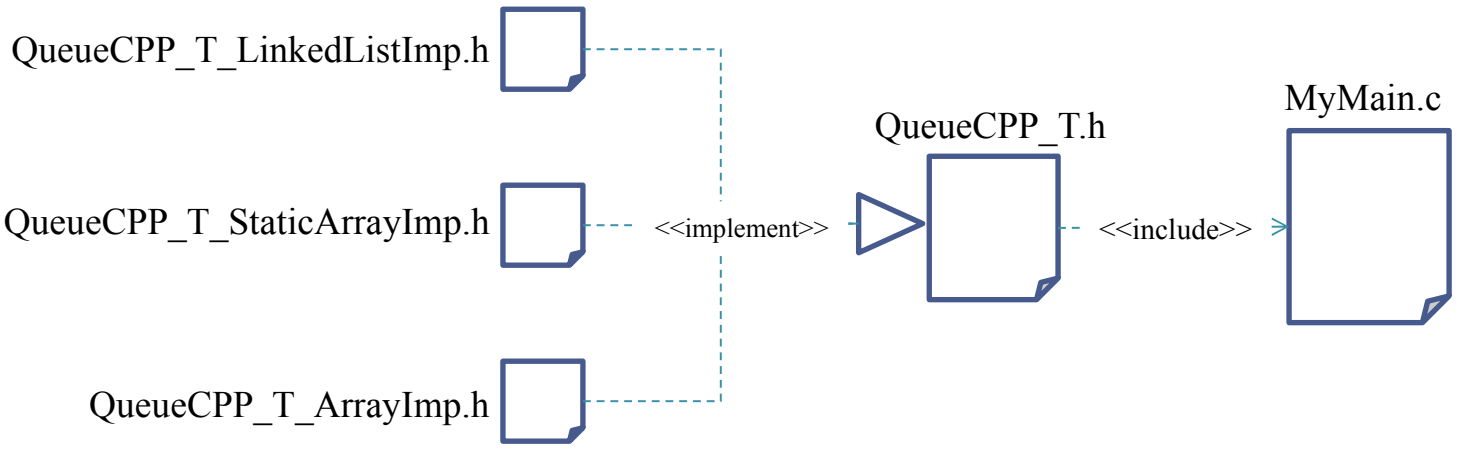


ABSTRACT QUEUE

- Comment on using **static array**
  - Advantage: quick access and easy-to-manage
  - Drawbacks:
    - Waste of memory
    - Due to managing the first item with *firstIdx*, variable *nItem* do not decrease when taking another item out => operation *enQueue*() may not be successful due to out of memory
  - Solution:
    - Using dynamic array

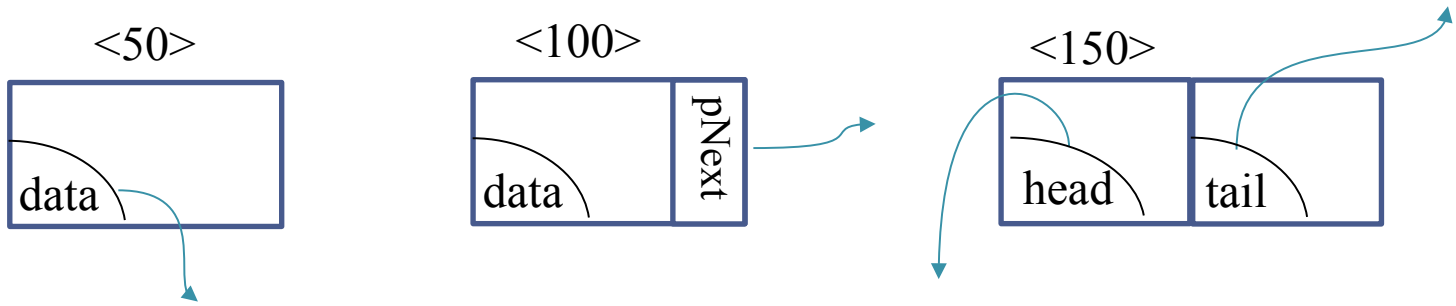
ABSTRACT QUEUE

- There are 3 solutions of implementation of abstract queue with template
  - Linked list: in QueueCPP_T_LinkedListImp.h
  - Static array: in QueueCPP_T_StaticArrayImp.h
  - Dynamic array: in QueueCPP_T_ArrayImp.h
- All files need to include “QueueCPP_T.h”



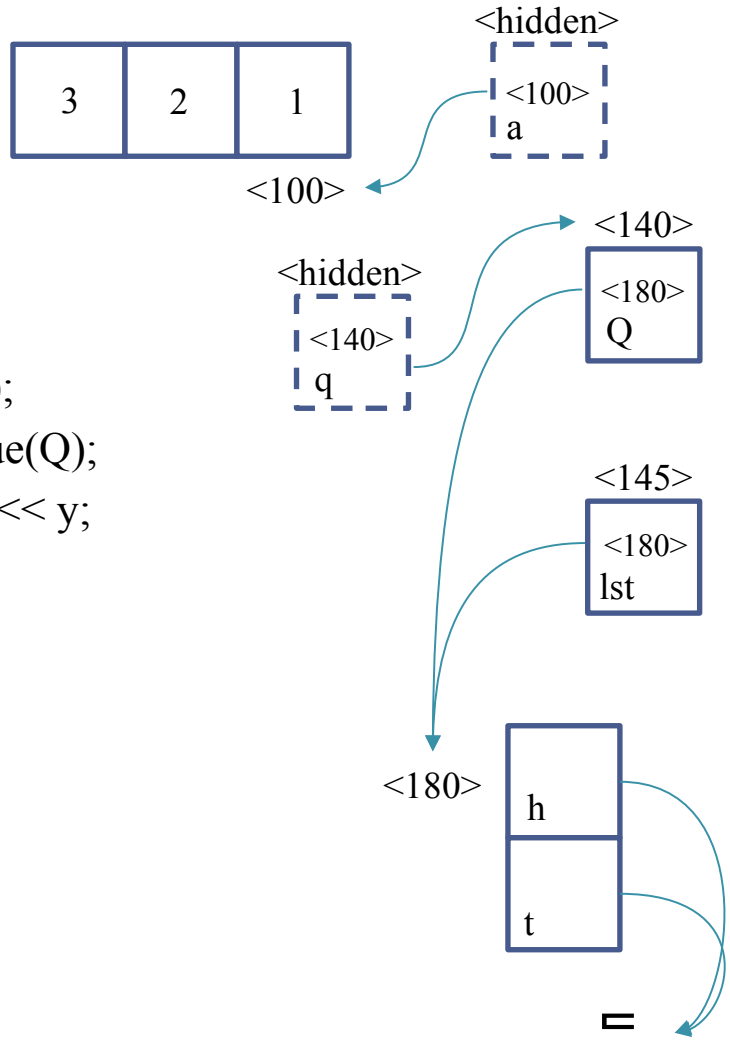
ABSTRACT QUEUE

- Ex: use **linked list**<template>
  - template <class T>
  - struct Queue{ void* data; };
  - template <class T>
  - struct Node { T data; Node<T>* pNext; };
  - template <class T>
  - struct ListImp{ Node<T> *head, *tail; };



ABSTRACT QUEUE

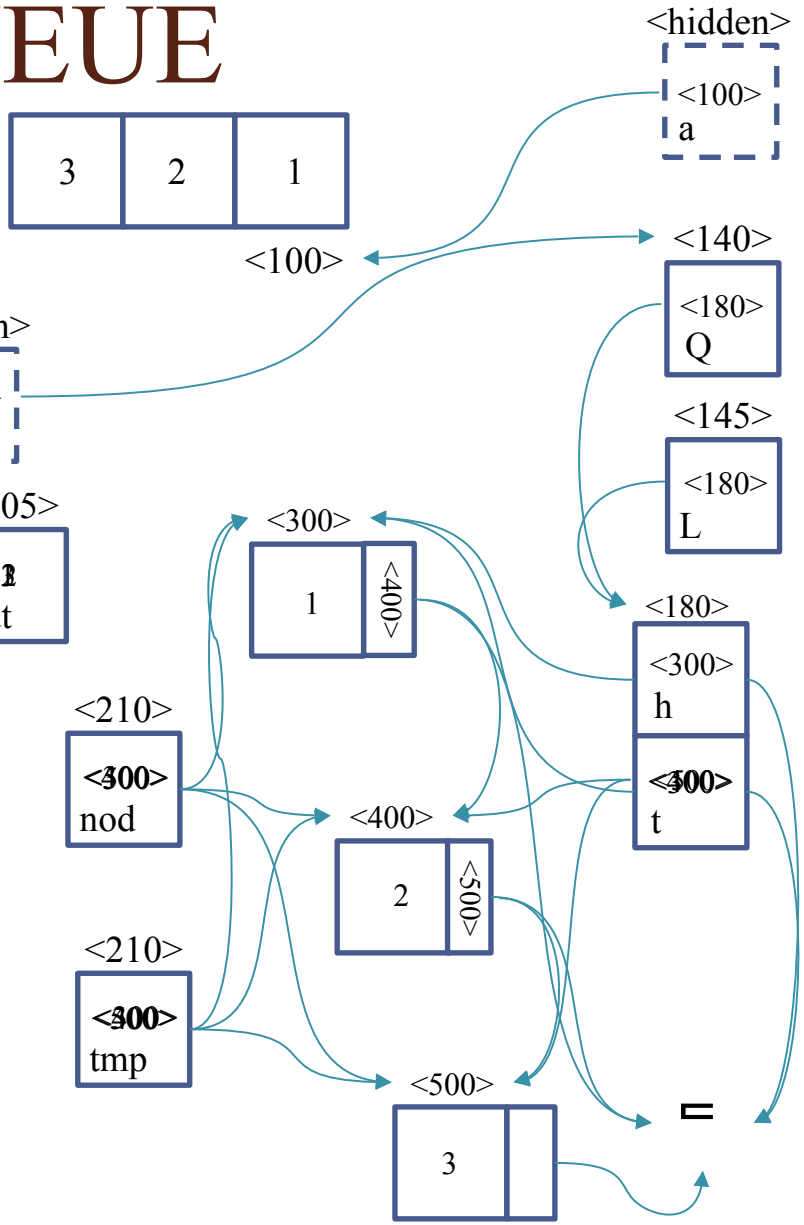
- Ex: use **linked list**<template>
  - void main(){
    - int a[] = {1, 2, 3}, i = -1;
    - Queue<int> Q; **initQueue**(Q);
    - while(++i < 3) enQueue(Q, a[i]);
    - int x = deQueue(Q), y = deQueue(Q);
    - cout << "x = " << x << ", y = " << y;
    - closeQueue(Q);
  - }
  - template <class T>
  - void initQueue(Queue<T>& q){
    - ListImp<T>* lst = new ListImp<T>;
    - if(!lst) return;
    - q.data = lst;
    - if(lst) lst->head = lst->tail = NULL;
  - }



ABSTRACT QUEUE

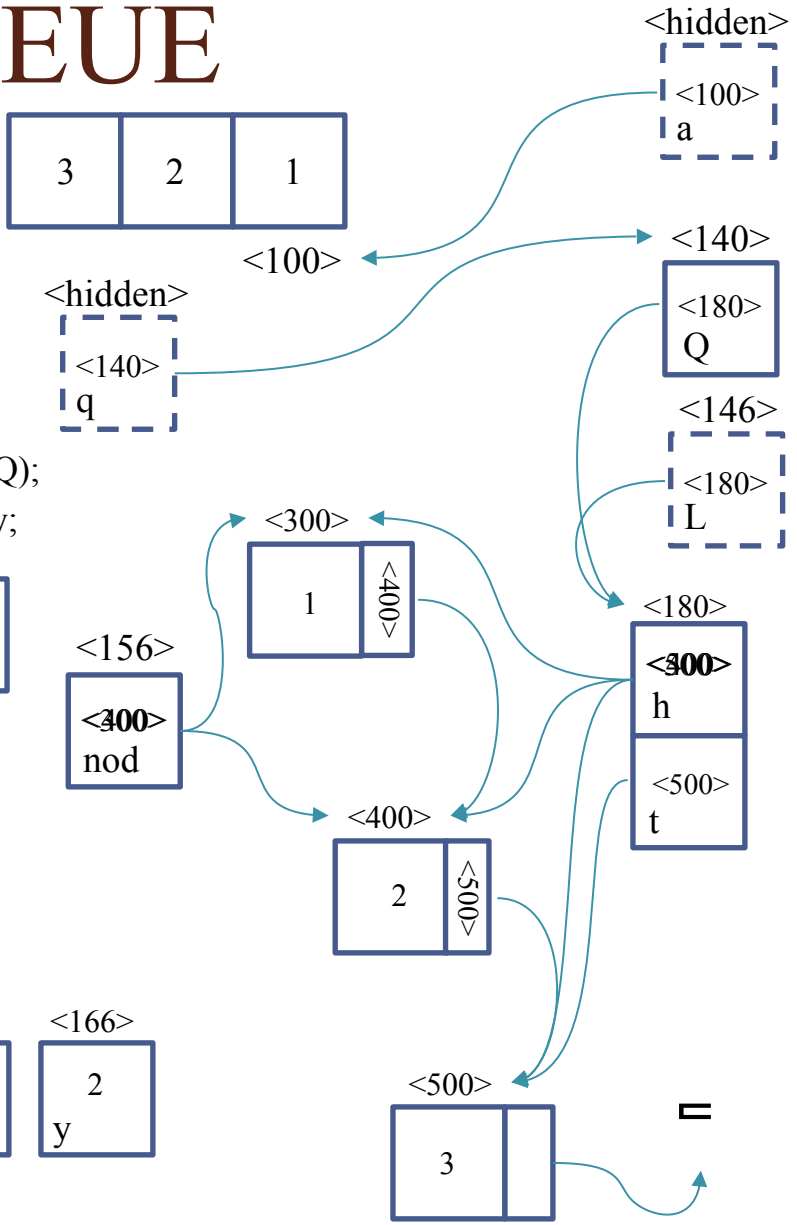
- Ex: use **linked list**<template>

```
void main(){
    int a[] = {1, 2, 3}, i = -1;
        Queue<int> Q; initQueue(Q);
        while(++i < 3) enQueue(Q, a[i]);
        int x = deQueue(Q), y = deQueue(Q);
        cout << "x = " << x << ", y = " << y;
        closeQueue(Q);
}
template <class T>
Node<T>* makeNode(T dat){
    Node<T>* nod = new Node<T>;
    if(nod){ nod->pNext = NULL; nod->data = dat; }
    return nod;
}
template <class T>
void enQueue(Queue<T>& q, const T& item){
    ListImp<T>* L = (ListImp<T>*)q.data;
    if(!L) return;
    Node<T>* tmp = makeNode(item);
    if(L->head == NULL) L->head = L->tail = tmp;
    else {
        L->tail->pNext = tmp;
        L->tail = L->tail->pNext;
        }
}
```



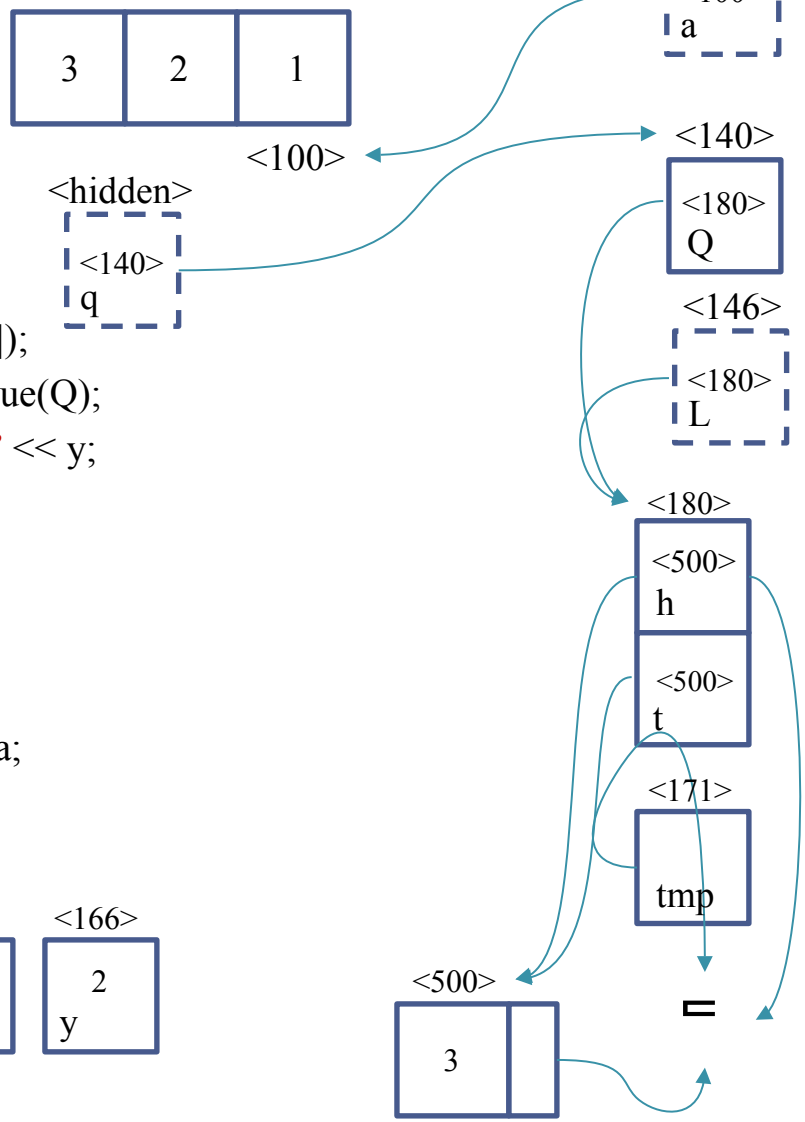
ABSTRACT QUEUE

- Ex: use **linked list**<template>
  - void main(){
    - int a[] = {1, 2, 3}, i = -1;
    - Queue<int> Q; initQueue(Q);
    - while(++i < 3) enQueue(Q, a[i]);
    - int x = **deQueue**(Q), y = **deQueue**(Q);
    - cout << "x = " << x << ", y = " << y;
    - closeQueue(Q);
  - }
  - template <class T>
  - T deQueue(Queue<T>& q){
    - ListImp<T>* L = (ListImp<T>*)q.data;
    - static T dummy;
    - if(!L || !(L->head)) return dummy;
    - Node<T>* nod = L->head;
    - T dat = nod->data;
    - L->head = L->head->pNext;
    - if(!L->head) L->tail = NULL;
    - delete nod;
    - return dat;
  - }



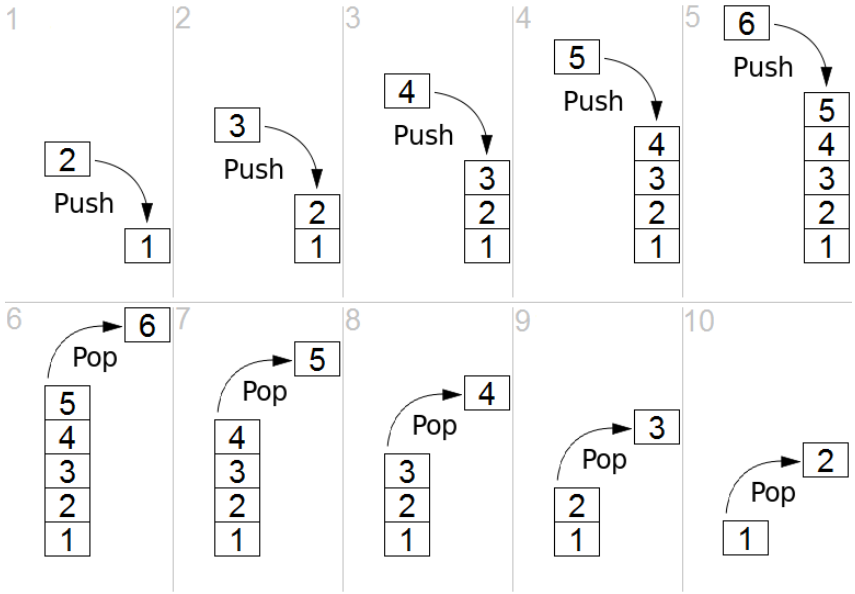
ABSTRACT QUEUE

- Ex: use **linked list**<template>
  - void main(){
    - int a[] = {1, 2, 3}, i = -1;
    - Queue<int> Q; initQueue(Q);
    - while(++i < 3) enQueue(Q, a[i]);
    - int x = deQueue(Q), y = deQueue(Q);
    - cout << "x = " << x << ", y = " << y;
    - **closeQueue**(Q);
  - }
  - template <class T>
  - void closeQueue(Queue<T>& q){
    - ListImp<T>* L = (ListImp<T>*)q.data;
    - if(!L) return;
    - while(L->head){
      - Node<T>* tmp = L->head->pNext;
      - delete L->head;
      - L->head = tmp;
    - }
    - delete L;
  - }



ABSTRACT STACK

- Abstract stack
  - Follow last-in-first-out – LIFO paradigm
  - Implement stack with array or linked list



Source: *https://en.wikipedia.org/wiki/Stack_(abstract_data_type)*

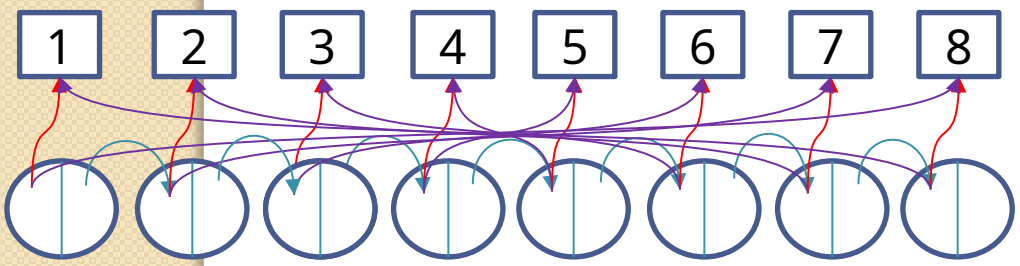
ABSTRACT STACK

- Programming interface of abstract stack (use void*)
  - void* initStack(): initialize stack and return a pointer to the place of **caller**
  - void closeStack(void* s): destroy stack s contains the address
  - void pushStack(void* s, void* item): **add** item into stack s (only add into s what item contains)
  - void* popStack(void* s): **take** the first item out of stack s and return to the place of caller
  - void* topStack(void* s): **read** the first item of stack s and return to the place of caller
  - int isEmpty(void* s): check if the stack s is empty

ABSTRACT STACK

- Programming interface of abstract stack (use void*)

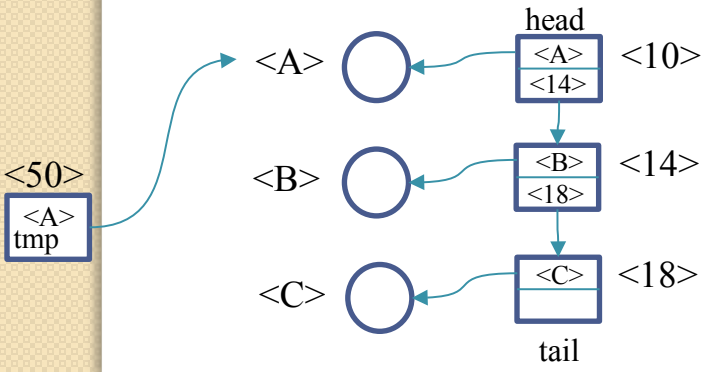
```
//StackC.h
#ifndef _MY_STACK_H_
#define _MY_STACK_H_

void* initStack();
void closeStack(void*);
void pushStack(void*, void*);
void* popStack(void*);
void* topStack(void*);
bool isEmpty(void*);
#endif
```

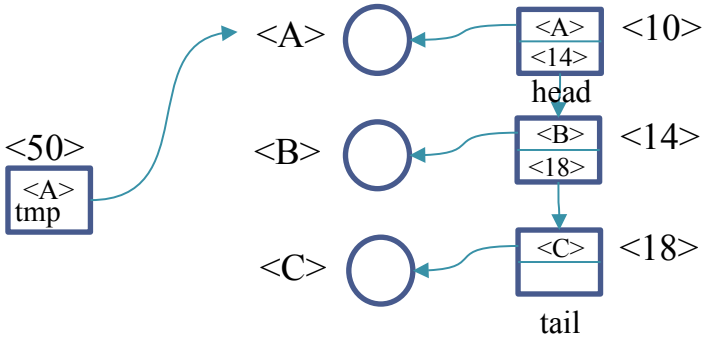



```
#include "StackC.h"
void takeAllStack(void* s){
  while(!isEmpty(s)){
    int x = *(int*)popStack(s); cout<<x<<" ";
  }}
void main(){
  int a[] = {1, 2, 3, 4, 5, 6, 7, 8}, i, x, y;
  void* s = initStack(); i = -1;
  while(++i<8) pushStack(s, &a[i]);
  x = *(int*)popStack(s);
  takeAllStack(s); i = 8;
  while(--i) pushStack(s, &a[i]);
  x = *(int*)popStack(q);
  y = *(int*)popStack(q);
  takeAllStack(s); closeStack(s);
}
```

ABSTRACT STACK

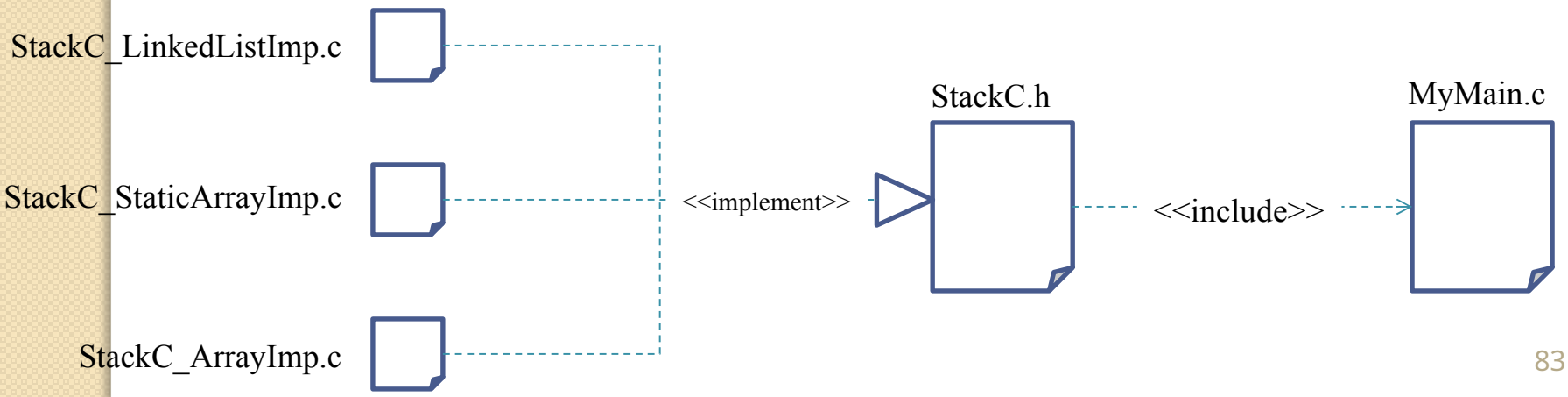
- Some conventions of programming interface of abstract stack (use void*)
  - void* topStack(void* s)
    - Take the address the first item pointing to and don't do anything
  - void pushStack(void* s, void* item)
    - Only add the address which item pointing to and don't do anything





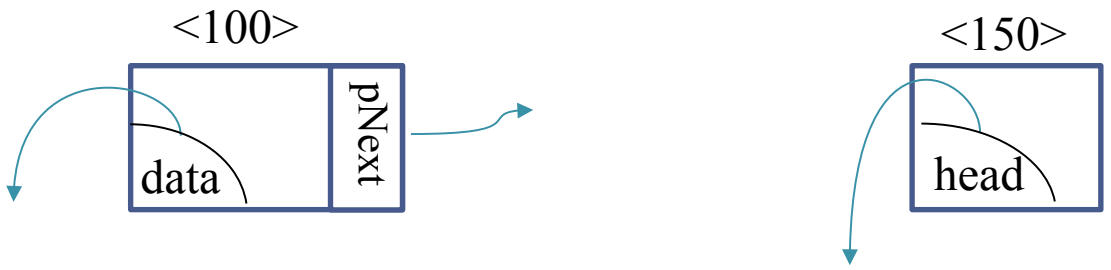
ABSTRACT STACK

- There are 3 solutions of implementation of abstract stack with void*
  - Linked list: in StackC_LinkedListImp.c
  - Static array: in StackC_StaticArrayImp.c
  - Dynamic array: in StackC_ArrayImp.c
- Note: all files need to include “StackC.h”



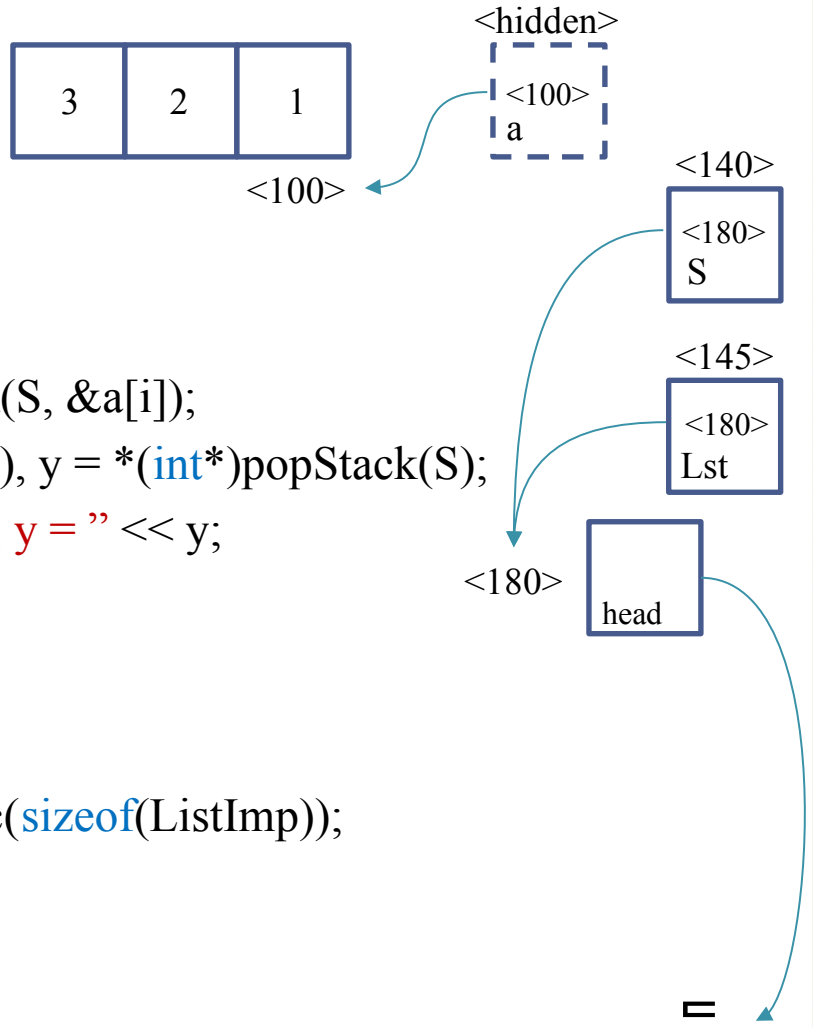
ABSTRACT STACK

- Ex: using **linked list**
  - struct tagNode{
    - void* data; struct tagNode* pNext;
  - };
  - typedef struct tagNode Node;
  - typedef struct { Node *head; }ListImp;



ABSTRACT STACK

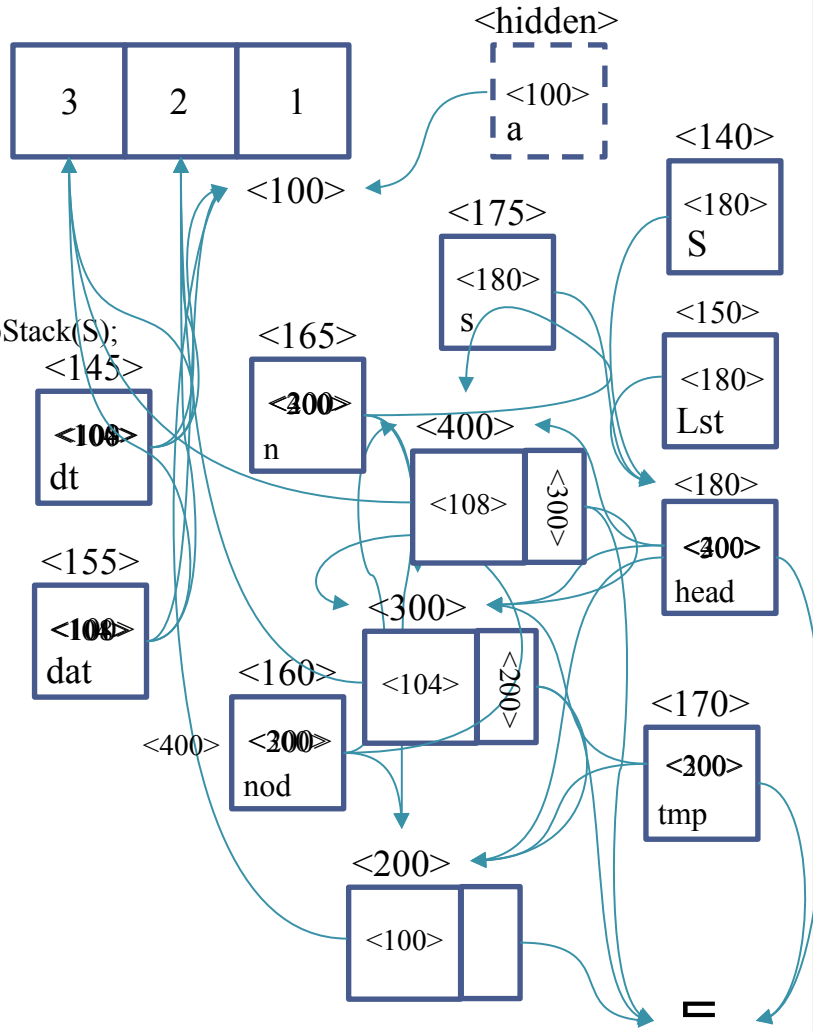
- Ex: using **linked list**
  - void main(){
    - int a[] = {1, 2, 3}, i = -1;
    - void* S = **initStack**();
    - while(++i < 3) pushStack(S, &a[i]);
    - int x = *(int*)popStack(S), y = *(int*)popStack(S);
    - cout << "x = " << x << ", y = " << y;
    - closeStack(Q);
  - }
  - void* initStack(){
    - ListImp* Lst = (ListImp*)malloc(sizeof(ListImp));
    - if(!Lst) return NULL;
    - Lst->head = NULL;
    - return Lst;
  - }



ABSTRACT STACK

- Ex: using **linked list**

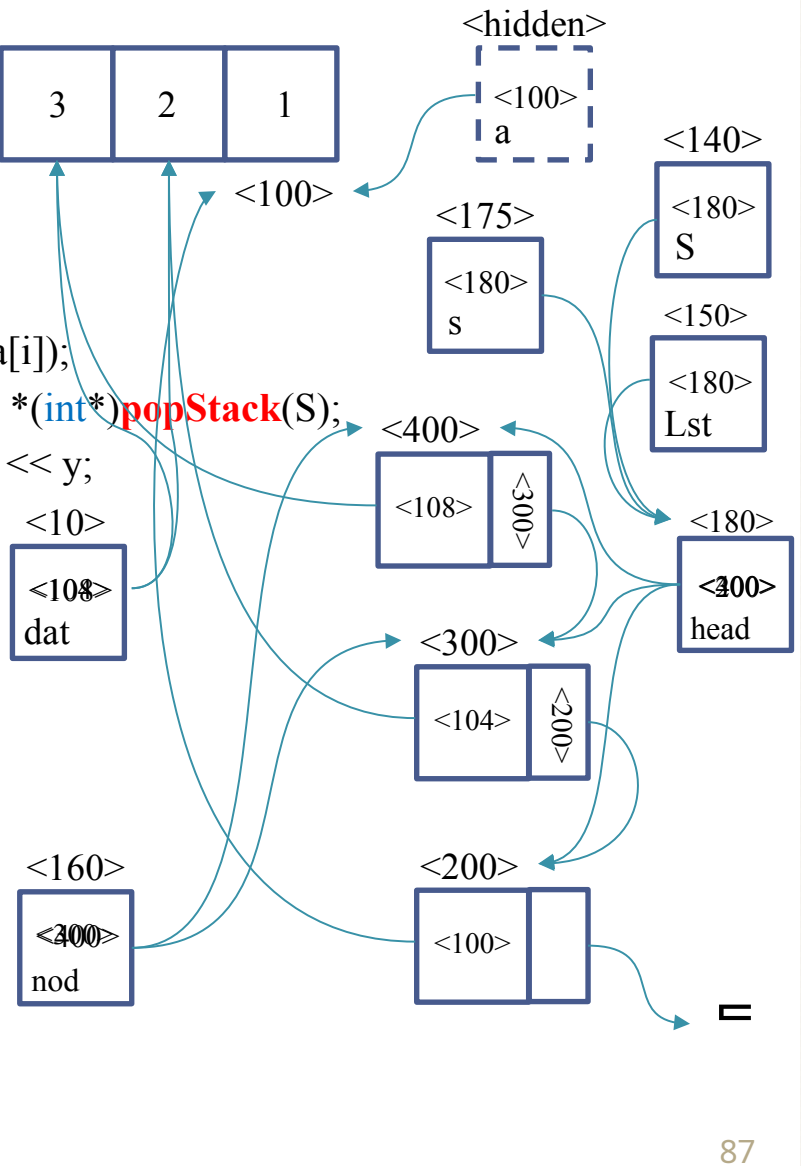
```
void main(){
    int a[] = {1, 2, 3}, i = -1;
        void* S = initStack();
        while(++i < 3) pushStack(S, &a[i]);
        int x = *(int*)popStack(S), y = *(int*)popStack(S);
        cout << "x = " << x << ", y = " << y;
        closeStack(S);
}
Node* makeNode(void* dat){
    Node* nod = (Node*)malloc(sizeof(Node));
    if(nod == NULL) return NULL;
    nod->pNext = NULL; nod->data = dat;
    return nod;
}
void pushStack(void* s, void* dt){
    ListImp *Lst = (ListImp*)s;
    if(Lst == NULL) return;
    Node* n = makeNode(dt), *tmp = Lst->head;
    if(n == NULL) return;
    Lst->head = n;
    n->pNext = tmp;
}
```



ABSTRACT STACK

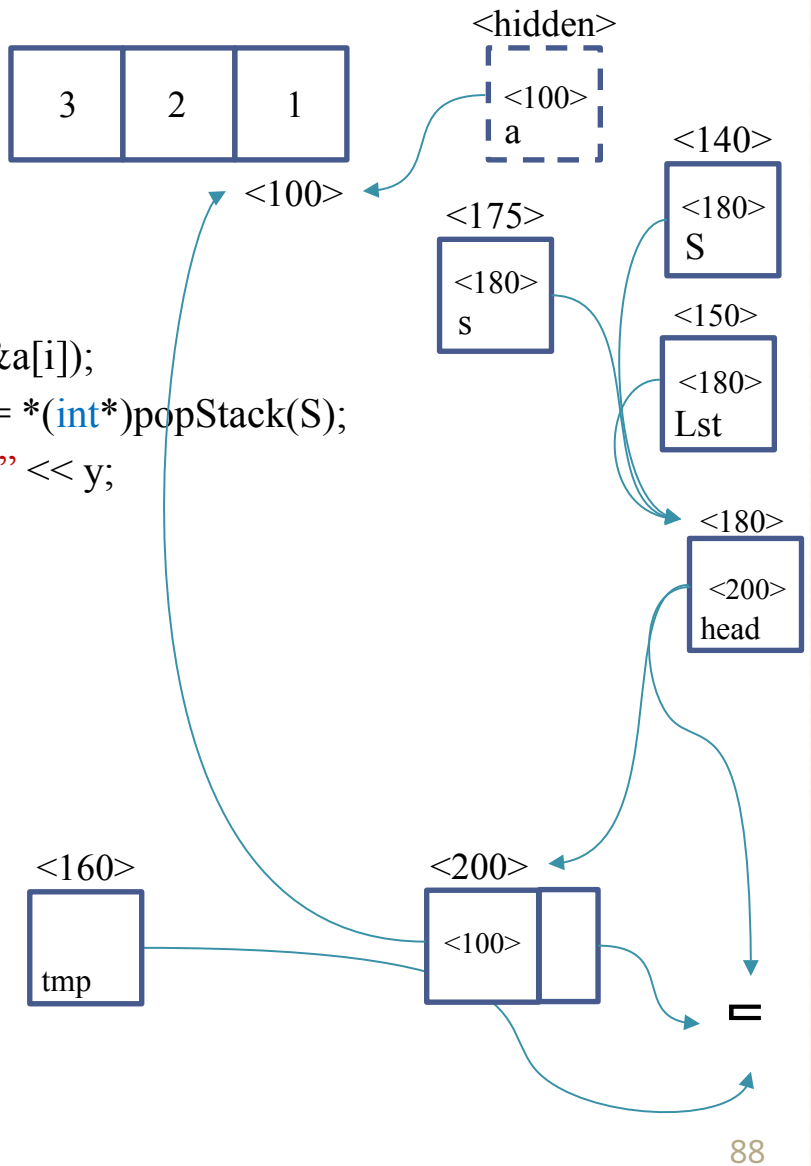
- Ex: using **linked list**

```
void main(){
    int a[] = {1, 2, 3}, i = -1;
        void* S = initStack();
        while(++i < 3) pushStack(S, &a[i]);
        int x = *(int*)popStack(S), y = *(int*)popStack(S);
        cout << "x = " << x << ", y = " << y;
        closeStack(S);
}
void* popStack(void* s){
    ListImp* Lst = (ListImp*)s;
    if(!Lst || !(Lst->head)) return NULL;
    Node* nod = Lst->head;
    void* dat = nod->data;
    Lst->head = Lst->head->pNext;
    free(nod);
    return dat;
}
```



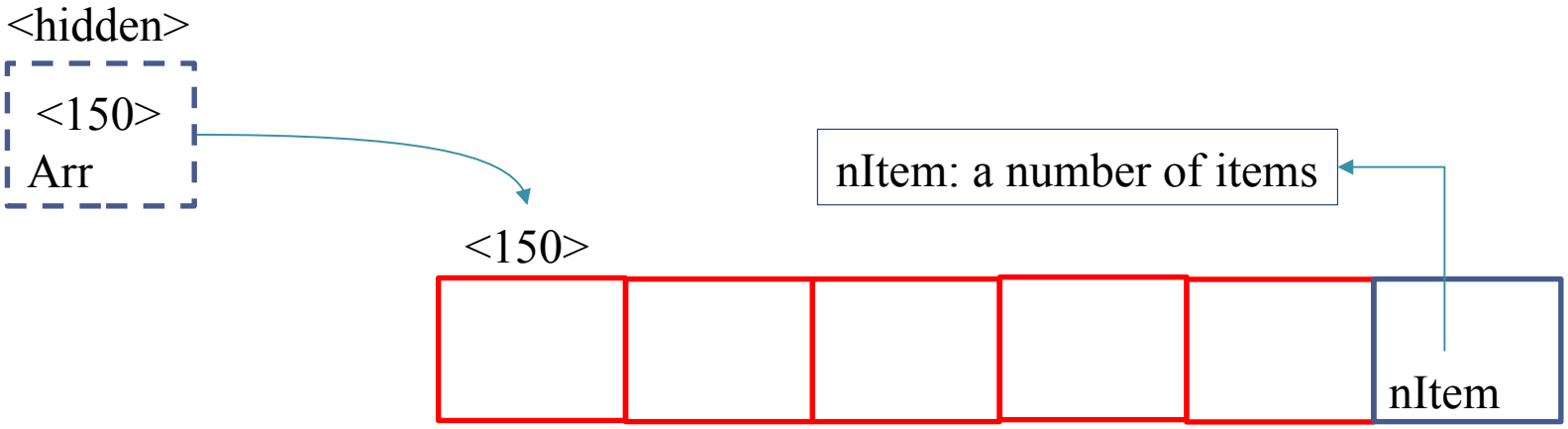
ABSTRACT STACK

- Ex: using **linked list**
  - void main(){
    - int a[] = {1, 2, 3}, i = -1;
    - void* S = initStack();
    - while(++i < 3) pushStack(S, &a[i]);
    - int x = *(int*)popStack(S), y = *(int*)popStack(S);
    - cout << "x = " << x << ", y = " << y;
    - **closeStack**(S);
  - }
  - void closeStack(void* s){
    - ListImp* Lst = (ListImp*)s;
    - if(!Lst) return;
    - while(Lst->head){
      - Node* tmp = Lst->head->pNext;
      - free(Lst->head);
      - Lst->head = tmp;
    - }
  - }



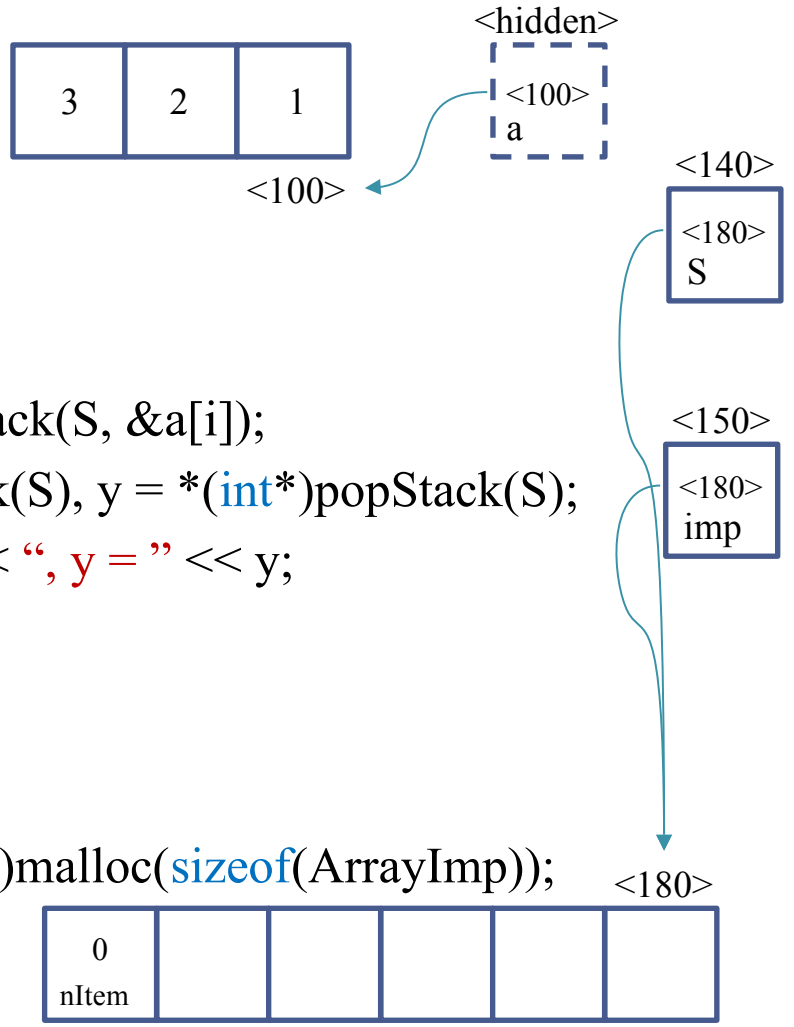
ABSTRACT STACK

- Ex: using **static array**
  - #define MAXSZ 5//maximum size of stack
  - typedef struct{
    - void* Arr[MAXSZ];
    - int nItem;
  - } ArrayImp;



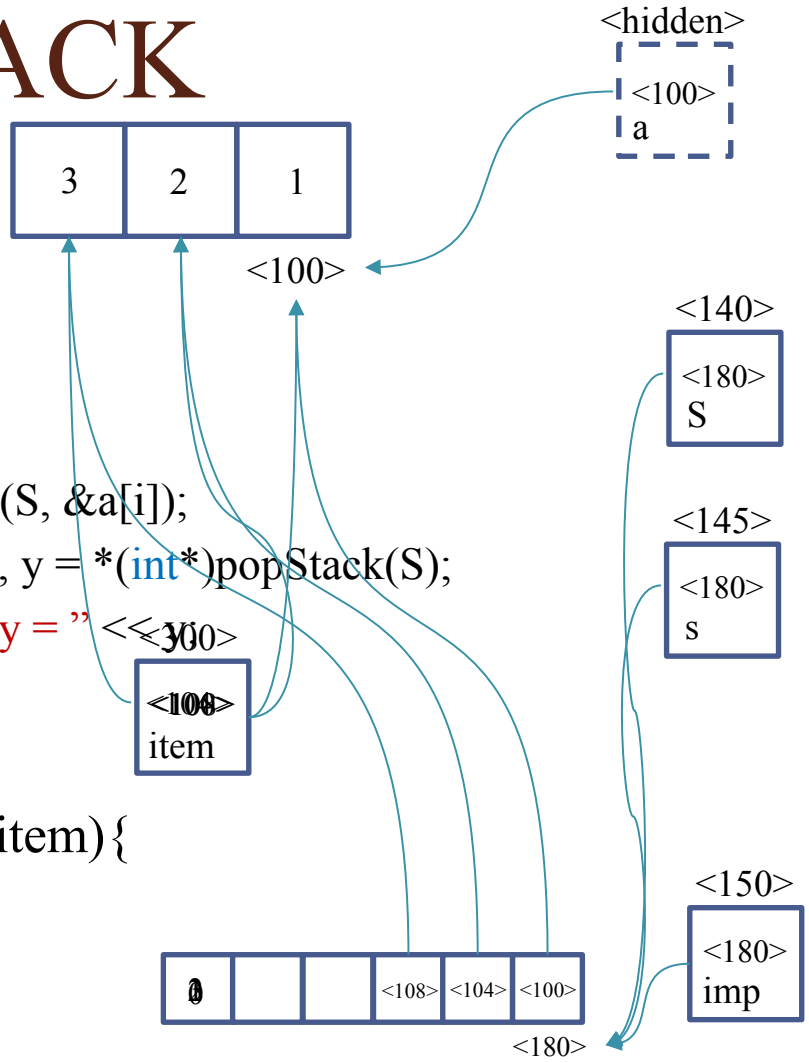
ABSTRACT STACK

- Ex: using **static array**
  - void main(){
    - int a[] = {1, 2, 3}, i = -1;
    - void* S = **initStack**();
    - while(++i < 3) pushStack(S, &a[i]);
    - int x = *(int*)popStack(S), y = *(int*)popStack(S);
    - cout << "x = " << x << ", y = " << y;
    - closeStack(S);
  - }
  - void* initStack(){
    - ArrayImp* imp = (ArrayImp*)malloc(sizeof(ArrayImp));
    - if(!imp) return NULL;
    - imp->nItem = 0;
    - return imp;
  - }



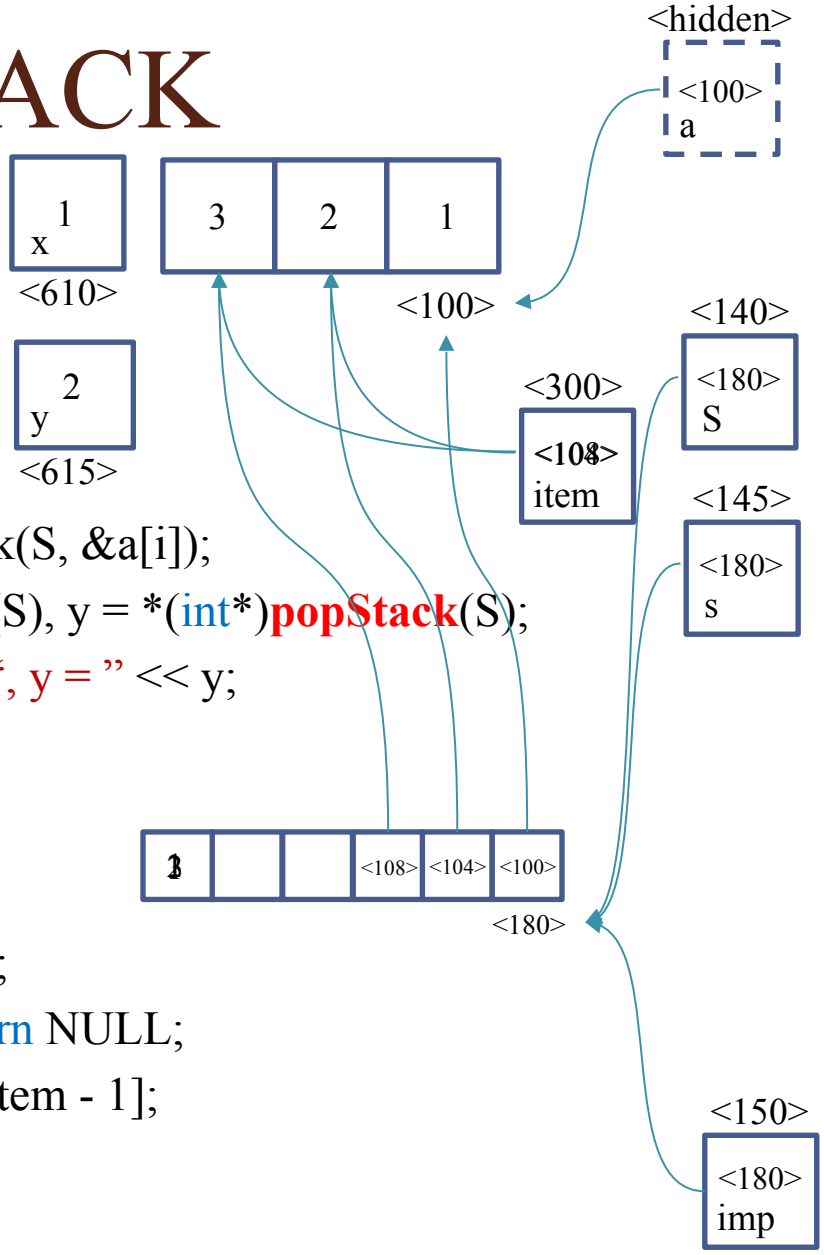
ABSTRACT STACK

- Ex: using **static array**
  - void main(){
    - int a[] = {1, 2, 3}, i = -1;
    - void* S = initStack();
    - while(++i < 3) **pushStack**(S, &a[i]);
    - int x = *(int*)popStack(S), y = *(int*)popStack(S);
    - cout << "x = " << x << ", y = " << y;
    - closeStack(S);
  - }
  - void pushStack(void* s, void* item){
    - ArrayImp* imp = (ArrayImp*)s;
    - if(!imp) return;
    - if(imp->nItem < MAXSZ){
      - imp->Arr[imp->nItem] = item;
      - (imp->nItem)++;
    - }
  - }



ABSTRACT STACK

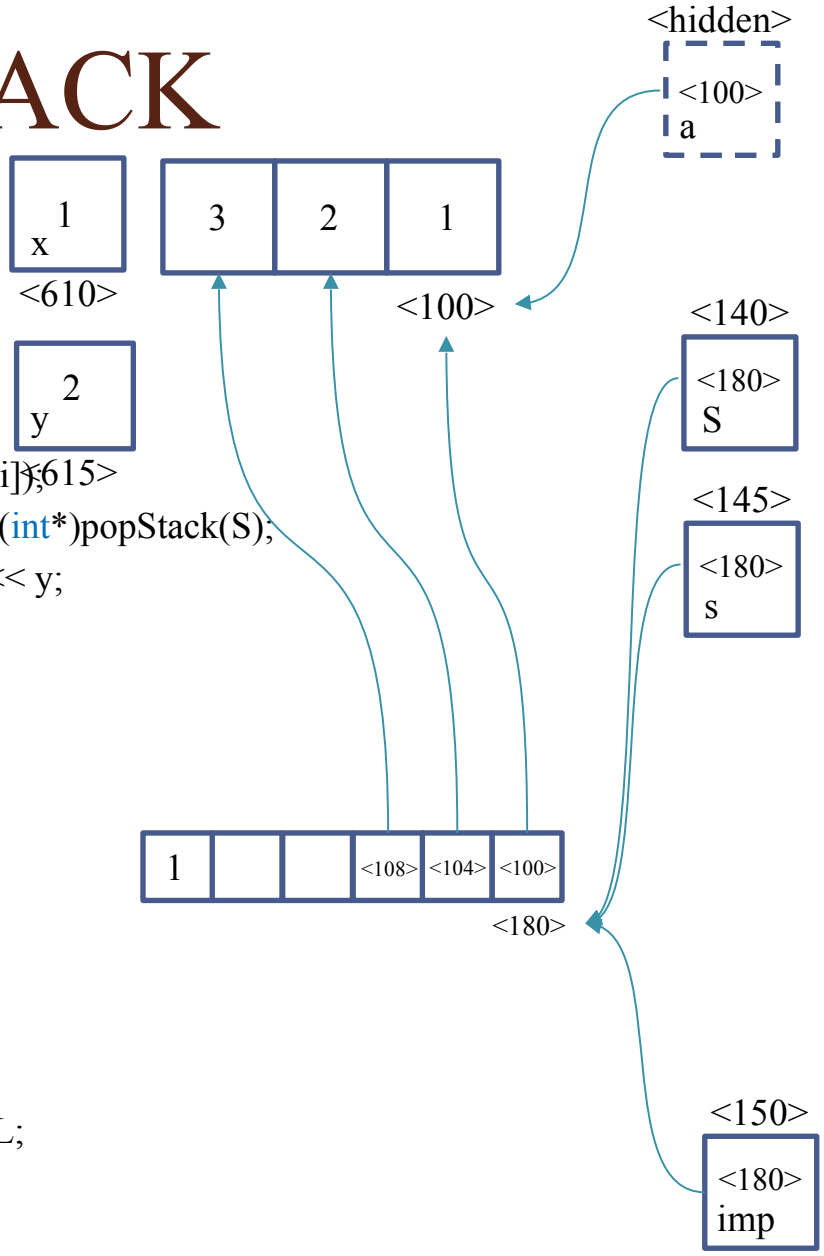
- Ex: using **static array**
  - void main(){
    - int a[] = {1, 2, 3}, i = -1;
    - void* S = initStack();
    - while(++i < 3) pushStack(S, &a[i]);
    - int x = *(int*)**popStack**(S), y = *(int*)**popStack**(S);
    - cout << "x = " << x << ", y = " << y;
    - closeStack(S);
  - }
  - void* popStack(void* s){
    - ArrayImp* imp = (ArrayImp*)s;
    - if(!imp || imp->nItem <= 0) return NULL;
    - void* item = imp->Arr[imp->nItem - 1];
    - (imp->nItem)--;
    - return item;
  - }



ABSTRACT STACK

- Ex: using **static array**

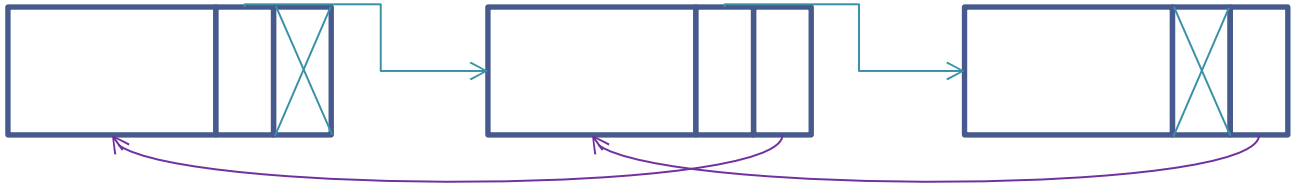
```
void main(){
    int a[] = {1, 2, 3}, i = -1;
        void* S = initStack();
        while(++i < 3) pushStack(S, &a[i]);
        int x = *(int*)popStack(S), y = *(int*)popStack(S);
        cout << "x = " << x << ", y = " << y;
        closeStack(S);
}
void closeStack(void* s){
    ArrayImp* imp = (ArrayImp*)s;
    if(!imp) return;
    free(imp);
}
void* topStack(void* s){
    ArrayImp* imp = (ArrayImp*)s;
    if(!imp || imp->nItem <= 0) return NULL;
    return imp->Arr[imp->nItem - 1];
}
int isEmpty(void* s){ return (topStack(s) == NULL); }
```



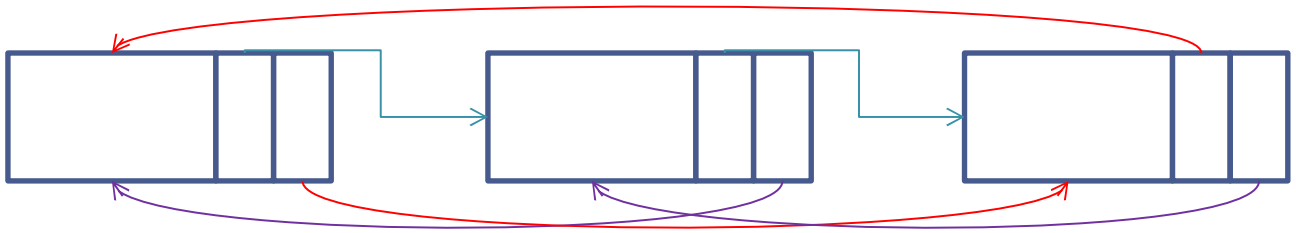
OTHER DATA STRUCTURES

- Circular linked list

- Doubly linked list

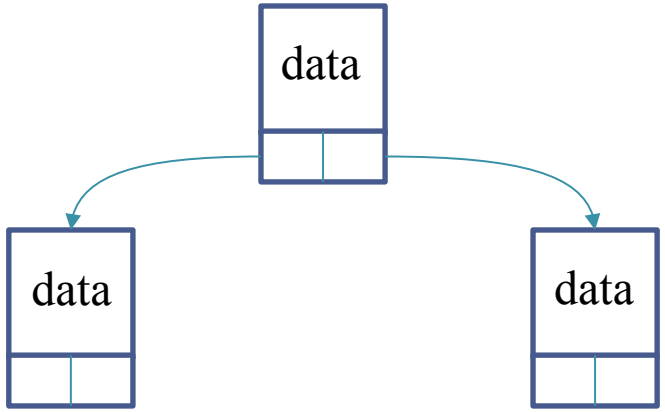


- Circular doubly linked list



OTHER DATA STRUCTURES

- Binary tree



- Graph