POINTER

PROGRAMMING TECHNIQUES

ADVISOR: Trương Toàn Thịnh

CONTENTS

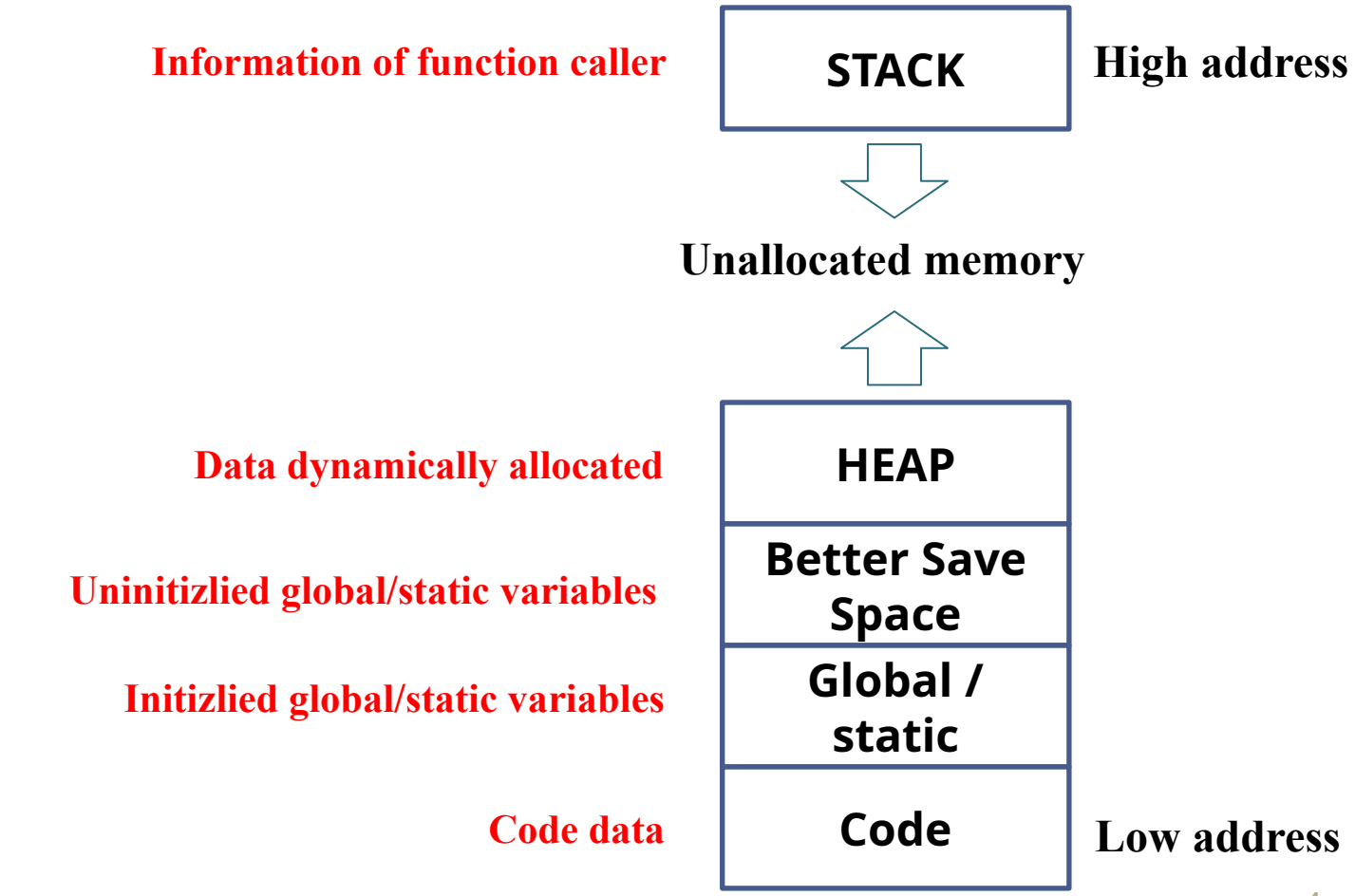
- Heap & stack
- Memory support functions
- Memory manipulation functions
- Allocate & use dynamic array
- Pointer-checking techniques
- Exercise

HEAP & STACK

- The stack is allocated for each function when they are called
- Stack contains the parameters (input / output) and local variables
- When function is finished, the memory of stack is returned to OS
- Terminology ‘stack’ implies it operates with ‘last in – first out’ rule
- The heap is global (does not depend on caller)
- Must explicitly free the memory of heap

HEAP & STACK

- Memory model of C/C++ program



MEMORY SUPPORT FUNCTIONS

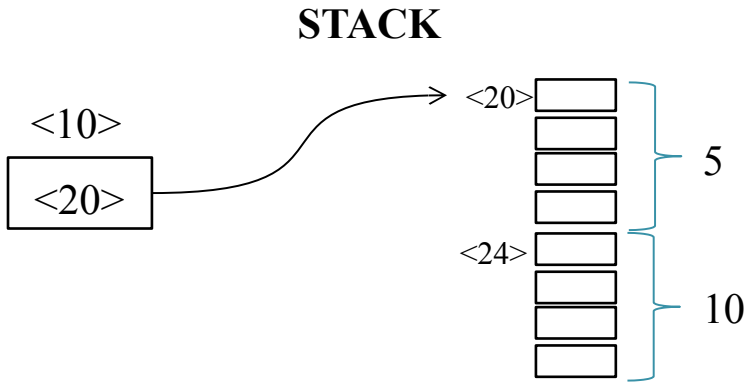
- Functions work with the memory heap
- Need to declare "#include <malloc.h>"
- constant NULL (zero) in "<stdio.h>"
- void* malloc(int nSz)
  - Allocate the memory with nSz bytes
  - Return the address of this memory
- Example of malloc:
  - int* a = (int*)malloc(3*sizeof(int));
  - for(int i = 0; i < 3; i++){*(a + i) = i;}
  - for(int i = 0; i < 3; i++){cout << *(a + i);}
  - free((int*)a)

MEMORY SUPPORT FUNCTIONS

- Functions work with the memory heap
- void* realloc(void* p, int nSz)
  - Re-allocate the old memory with **new** nSz bytes
  - **Returning** the address of **old memory or new memory depends** on the compiler's optimization
- Example of realloc:
  - int* a = (int*)malloc(3*sizeof(int));
  - cout << hex << a << endl;
  - for(int i = 0; i < 3; i++){*(a + i) = i;}
  - for(int i = 0; i < 3; i++){cout << *(a + i) << endl;}
  - realloc(a, 4);
  - cout << hex << a << endl;
  - for(int i = 0; i < 4; i++){cout << *(a + i) << endl;}
  - free((int*)a)

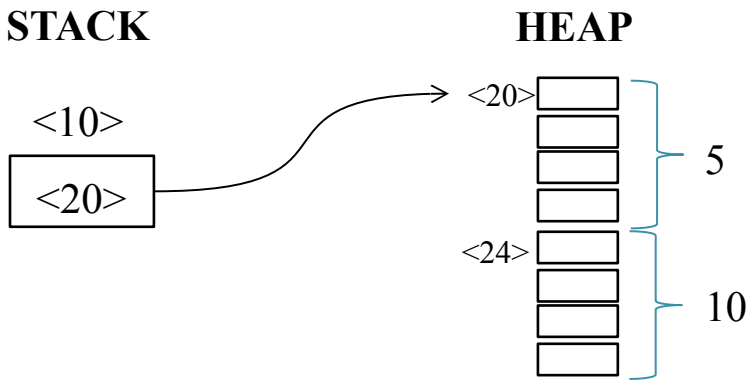
MEMORY SUPPORT FUNCTIONS

- Functions work with the memory stack
- Can allocate in the memory stack
- **Not allowing** to free after using
- Data in this memory are **automatically destroyed** when finishing calling function
- void* _alloca(int nSz): similar to malloc, but allocated in stack
- Example:
  - int* a = (int*)_alloca(8);
  - *a = 5;
  - *(a+1) = 10;
  - cout << *a << endl;
  - cout << *(a+1) << endl;



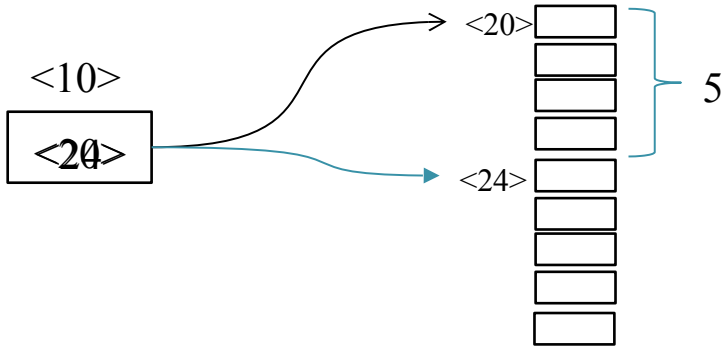
MEMORY SUPPORT FUNCTIONS

- Functions work with the memory stack
- void* _expand(void* p, int nSz): similar to realloc, but only supported in C++
- If you want to know a number of bytes allocated, we can use some functions
  - int _msize(void* p): return a number of bytes p manages, only used in Windows
  - int malloc_size(void* p): similarly but used in Mac OS
  - int malloc_usable_size(void* p): similarly but used in Linux
- Note: _msize only returns a number of bytes allocated by malloc, realloc or calloc
- Ex: int _msize(void*):
  - int* a = (int*)malloc(8);
  - *a = 5;
  - *(a+1) = 10;
  - cout << *a << " " << *(a+1)<<endl;
  - cout << _msize(a) << endl;



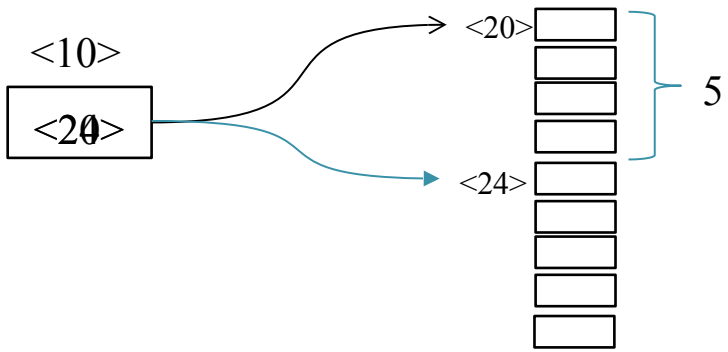
MEMORY SUPPORT FUNCTIONS

- Functions work with the memory heap
- Build three support functions
  - Allocate memory: re-use malloc function
  - Free memory: re-use free function
  - Return the bytes allocated
- void* mAlloc(int size){//example size is 5 bytes
  - void* p = malloc(size + sizeof(int));
  - *(int*)p = size;
  - return ((int*)p) + 1;
- }



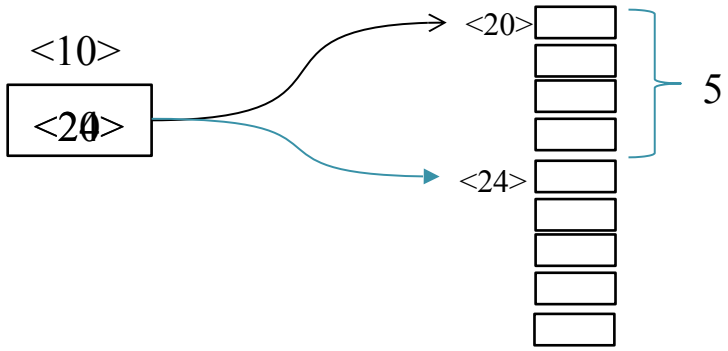
MEMORY SUPPORT FUNCTIONS

- Functions work with the memory heap
- void* mAlloc(int size){//example size is 5 bytes
  - void* p = malloc(size + sizeof(int));
  - *(int*)p = size;
  - return ((int*)p) + 1;
- }
- void mFree(void* p){
  - p = ((int*)p) – 1;
  - free(p);
- }



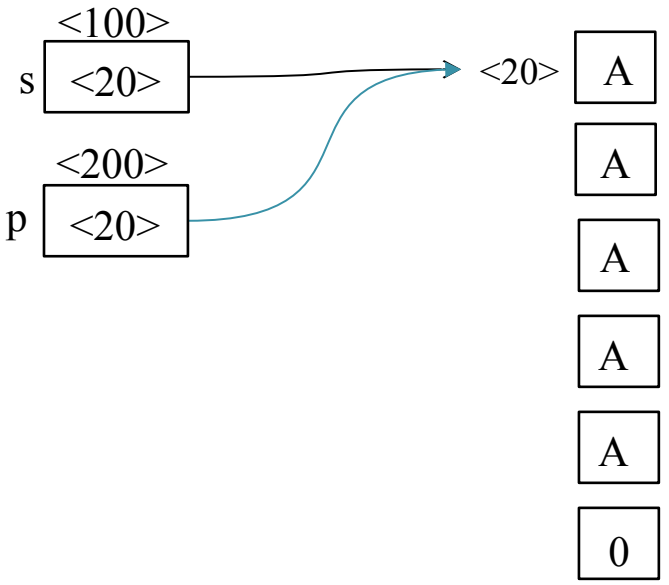
MEMORY SUPPORT FUNCTIONS

- Functions work with the memory heap
- void* mAlloc(int size){//example size is 5 bytes
  - void* p = malloc(size + sizeof(int));
  - *(int*)p = size;
  - return ((int*)p) + 1;
- }
- int mSize(void* p){
  - return *(((int*)p) – 1);
- }
- void main(){
  - char* p = (char*)mAlloc(5);
  - gets(p);
  - cout<<p<<endl;
  - cout<<mSize(p)<<endl;
  - mFree(p);
- }



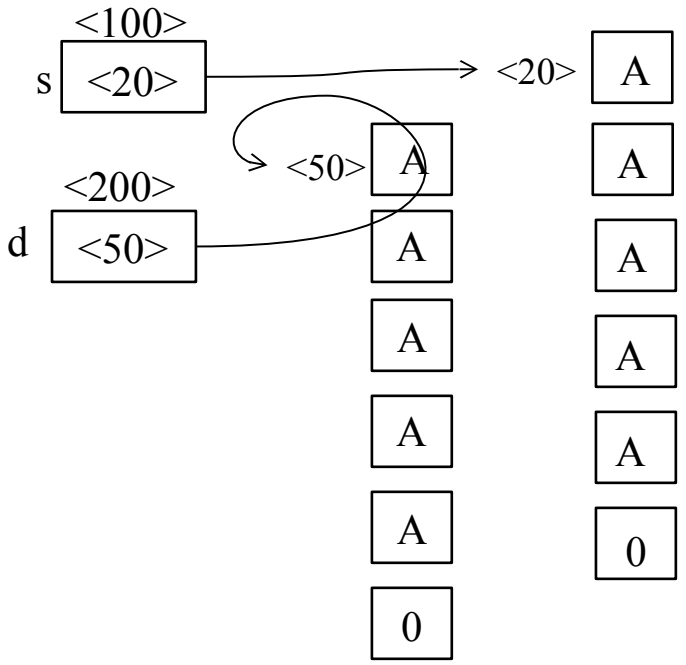
MEMORY MANIPULATION FUNCTIONS

- Library <memory.h> contain memory manipulation functions
- void* memset(void* des, int ch, int n): Initialize n bytes with value of ch for the memory des
- void main(){
  - char* s = (char*)malloc(6);
  - s[5] = 0;
  - char* p = memset(s, 65, 5);
  - cout << s << endl;
  - cout << (int*)s << endl;
  - cout << (int*)p << endl;
  - free(s);
- }



MEMORY MANIPULATION FUNCTIONS

- Library <memory.h> contain memory manipulation functions
- void* memmove(void* des, const void* src, int n): Copy n bytes from memory src to memory des. This function returns the address of des pointer
- void main(){
  - char* s = (char*)malloc(6);
  - s[5] = 0;
  - memset(s, 65, 5);
  - char* d = (char*)malloc(6);
  - d = (char*)memmove(d, s, 6);
  - cout << d << endl;
  - cout << (int*)d << endl;
  - free(s);
  - free(d);
- }



MEMORY MANIPULATION FUNCTIONS

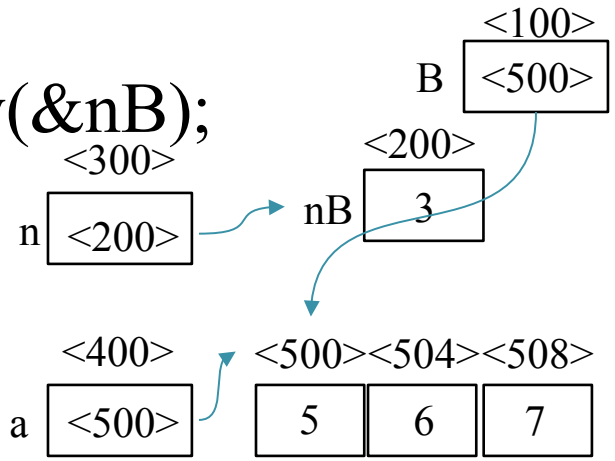
- Library <memory.h> contain memory manipulation functions
- int memcmp(const void* buf1, const void* buf2, int n): Compare n bytes of two memories buf1 and buf2. Function returns 1 if buf1 > buf2, -1 if buf1 < buf2 and 0 if equality. Note: This function only compares **value of each byte**
- void main(){
  - char* s = (char*)malloc(6);
  - s[5] = 0;
  - memset(s, 65, 5);
  - char* d = (char*)malloc(6);
  - d = (char*)memmove(d, s, 6);
  - cout << memcmp(s, d, 6) << endl;
  - free(s);
  - free(d);
- }

MEMORY MANIPULATION FUNCTIONS

- Library <memory.h> contain memory manipulation functions
- int _memicmp(const void* buf1, const void* buf2, unsigned int n): Similar to memcmp, but case insensitive
- void main(){
  - char* s = (char*)malloc(6);
  - s[5] = 0;
  - memset(s, 65, 5);
  - char* d = (char*)malloc(6);
  - d = (char*)memmove(d, s, 6);
  - d[0] = **'a'**;
  - cout << memcmp(s, d, 6) << endl;
  - cout << _memicmp(d, s, 6) << endl;
  - free(s);
  - free(d);
- }

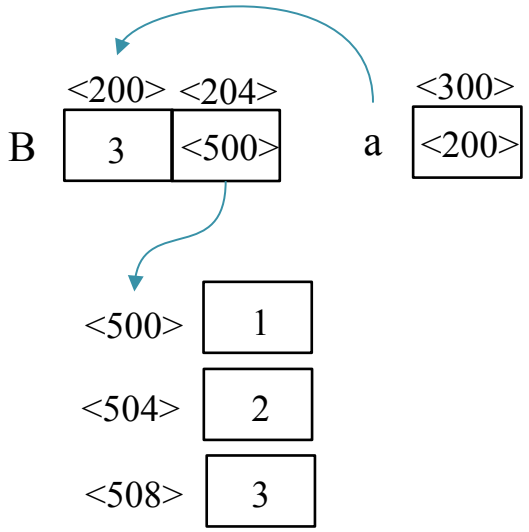
ALLOCATE & USE DYNAMIC ARRAY (WITH PRIOR SIZE)

- Separate functions of input/output
- **void main()**
  - int nB; float* B = inputArray(&nB);
  - if(B != NULL)
    - outputArray(B, nB); free(B);
- **void inputArray(int* n)**
  - cin>>(*n);
  - float* a = (float*)calloc((*n), sizeof(float));
  - if(a != NULL)
    - for(int i = 0; i < (*n); i++){cin>>a[i];}
  - return a;



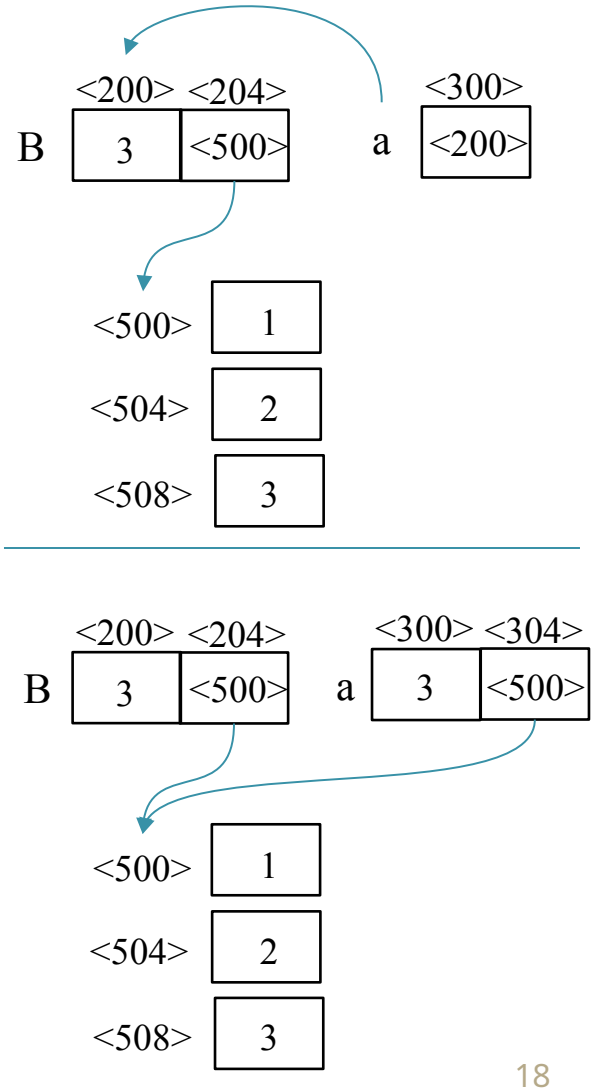
ALLOCATE & USE DYNAMIC ARRAY (WITH PRIOR SIZE)

- Build struct intArray
- Example
  - struct intArray{ int n, *arr;};
  - void main(){
    - intArray B;
    - nhap(&B);
    - xuat(&B);
    - huy(&B);
  - }
  - void nhap(intArray* a){
    - cin >> a->n;
    - a->arr = (int*)malloc(a->n * sizeof(int));
    - for(int i = 0; i < a->n; i++) cin >> a->arr[i];
  - }



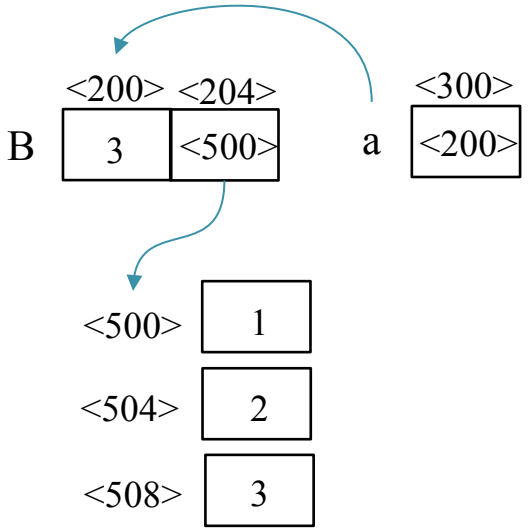
ALLOCATE & USE DYNAMIC ARRAY (WITH PRIOR SIZE)

- Build struct intArray
- Example
  - struct intArray{ int n, *arr;};
  - void main(){
    - intArray B;
    - nhap(&B);
    - xuat(&B); //xuat(B);
    - huy(&B);
  - }
  - void xuat(const intArray* a){
    - for(int i = 0; i < a->n; i++) cout << a->arr[i];
  - }
  - void xuat(intArray a){
    - for(int i = 0; i < a.n; i++) cout << a.arr[i];
  - }



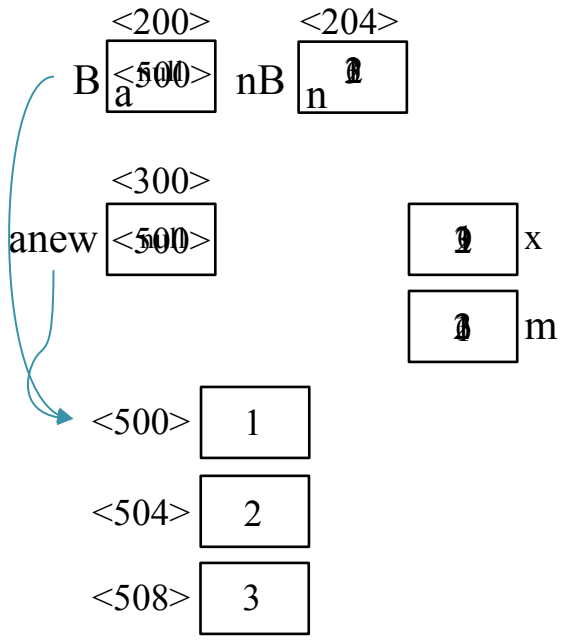
ALLOCATE & USE DYNAMIC ARRAY (WITH PRIOR SIZE)

- Build struct intArray
- Example
  - struct intArray{ int n, *arr;};
  - void main(){
    - intArray B;
    - nhap(&B); xuat(&B); huy(&B);
  - }
  - void huy(intArray* a){
    - if(a != NULL)
      - if(a->arr != NULL)
        - free(a->arr);
  - }



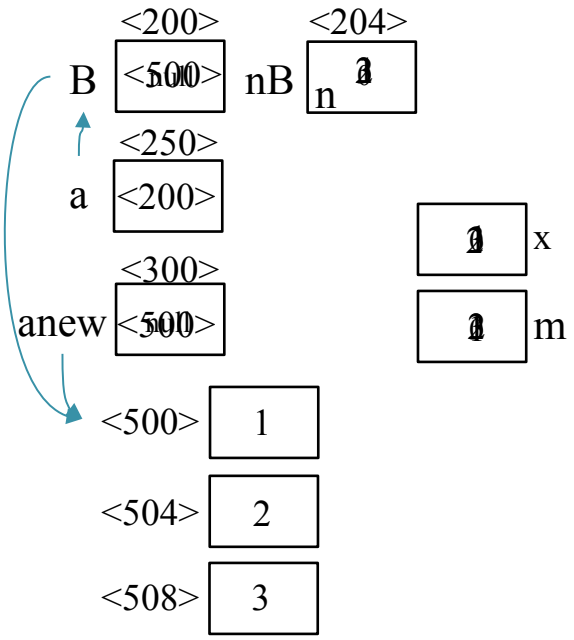
ALLOCATE & USE DYNAMIC ARRAY (WITH UNKNOWN SIZE)

- User determines when to stop input
- Ex: Use version of 'nhap' function with parameter of **reference pointer**
  - void main(){
    - int* B, nB;
    - nhap(B, nB); xuat(B, nB); free(B);
  - }
  - void nhap(int*& a, int &n){
    - int x, m, *anew;
    - anew = a = NULL;
    - m = x = n = 0;
    - while(cin >> x){
      - m = n + 1;
      - anew = (int*)realloc(a, m * sizeof(int));
      - if(anew != NULL){ anew[n++] = x; a = anew;}
    - }
    - cin.clear();
  - }



ALLOCATE & USE DYNAMIC ARRAY (WITH UNKNOWN SIZE)

- User determines when to stop input
- Ex: Use version of 'nhap' function with parameter of **level 2 pointer**
  - void main(){
    - int* B, nB;
    - nhap(B, nB); xuat(B, nB); free(B);
  - }
  - void nhap(int** a, int &n){
    - int x, m, *anew;
    - anew = *a = NULL;
    - m = x = n = 0;
    - while(cin >> x){
      - m = n + 1;
      - anew = (int*)realloc(*a, m * sizeof(int));
      - if(anew != NULL){ anew[n++] = x; *a = anew;}
    - }
    - cin.clear();
  - }



ALLOCATE & USE DYNAMIC ARRAY (WITH UNKNOWN SIZE)

- Template function allows us to use with various datatypes
- Ex: Use version of 'nhap' function with parameter of **level 2 pointer**
  - template <class T>
  - void nhap(T** a, int &n){
    - T x, *anew; int m;
    - anew = *a = NULL;
    - m = n = 0;
    - while(cin >> x){
      - m = n + 1;
      - anew = (T*)realloc(*a, m * sizeof(T));
      - if(anew != NULL){ anew[n++] = x; *a = anew;}
    - }
    - cin.clear();
  - }

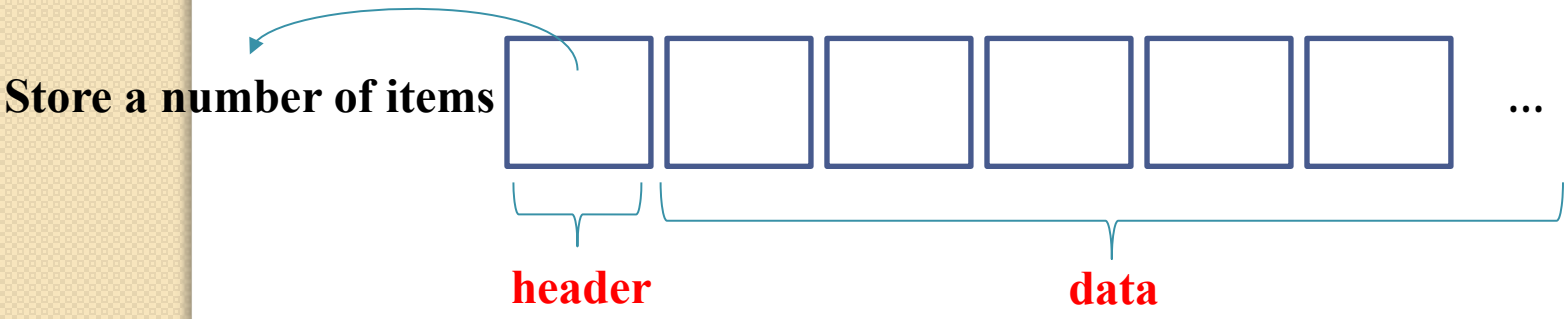
Need overload operator '>>' for user-defined type

ALLOCATE & USE DYNAMIC ARRAY (WITH UNKNOWN SIZE)

- Template function allows us to use with various datatypes
- Ex: Use version of 'nhap' function with parameter of **level 2 pointer**
  - struct PhanSo{int tu, mau;};
  - void nhap(T** a, int &n){
    - //...
  - }
  - istream& operator>>(istream& inDev, PhanSo& p){
    - inDev >> p.tu >> p.mau;
    - return inDev;
  - }
  - void main(){
    - PhanSo* B; int nB;
    - nhap(B, nB); xuat(B, nB); free(B);
  - }

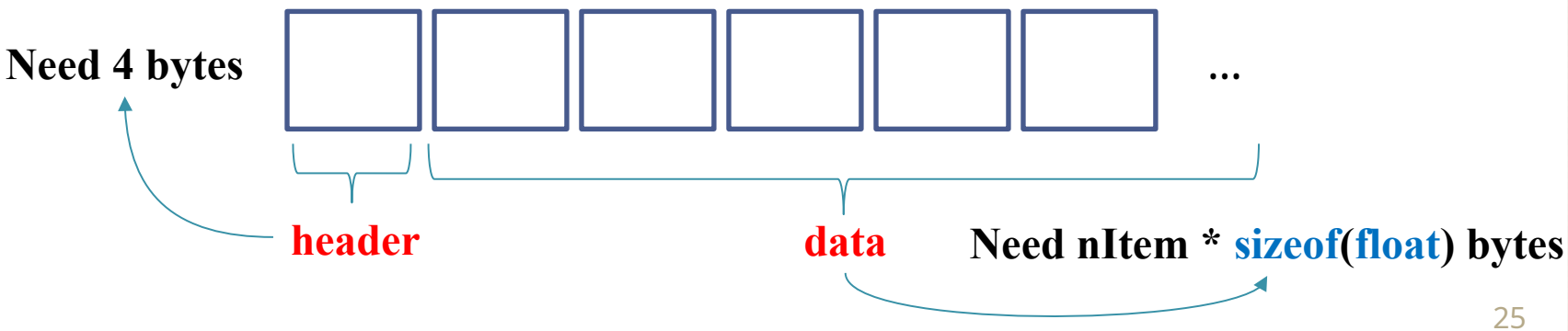
ALLOCATE & USE DYNAMIC ARRAY

- Goals: build support functions processing 1D array
- The functions' advantage:
  - Process with GENERAL TYPE
  - Need not vector<T>
- Need firstly checking array with **float**



ALLOCATE & USE DYNAMIC ARRAY

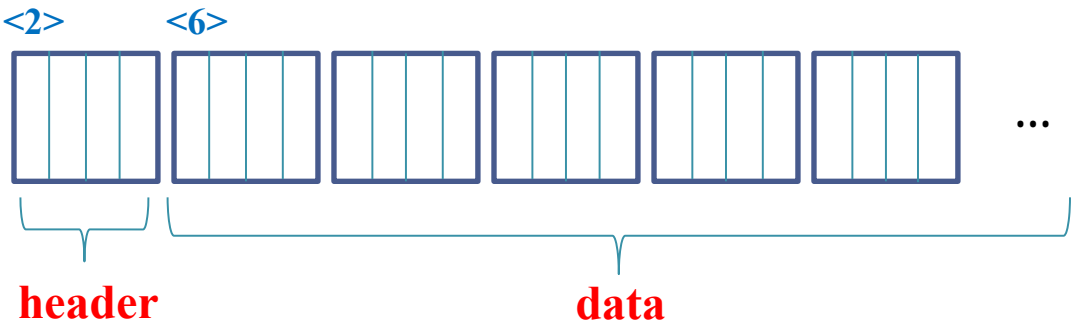
- Check array with **float**
  - The size of header: int headSize = sizeof(int)
  - int memSize(int nItem): calculate needed size (bytes) to store nItem
    - int memSize(int nItem){
      - return headSize + nItem * sizeof(float);
    - }



ALLOCATE & USE DYNAMIC ARRAY

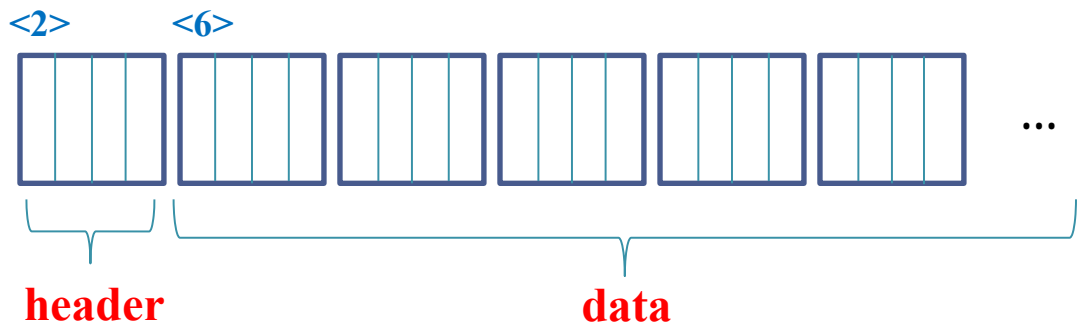
- Check array with **float**
  - void* origin_addr(float* aData): take the head's address from the data's address
    - void* origin_addr(void* aData){
      - return (char*)aData – headSize;
    - }

<6> - 4*sizeof(char)



ALLOCATE & USE DYNAMIC ARRAY

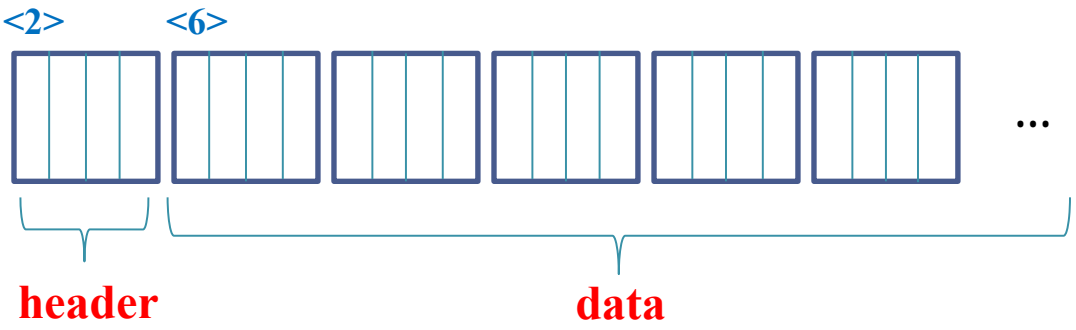
- Check array with **float**
  - float* data_addr(void* origin): take the data's address from the head's address (pointer origin points to the head's address)
    - float* data_addr(void* origin){
      - return (float*)((char*)origin + headSize); → <2> + 4*sizeof(char)
    - }



ALLOCATE & USE DYNAMIC ARRAY

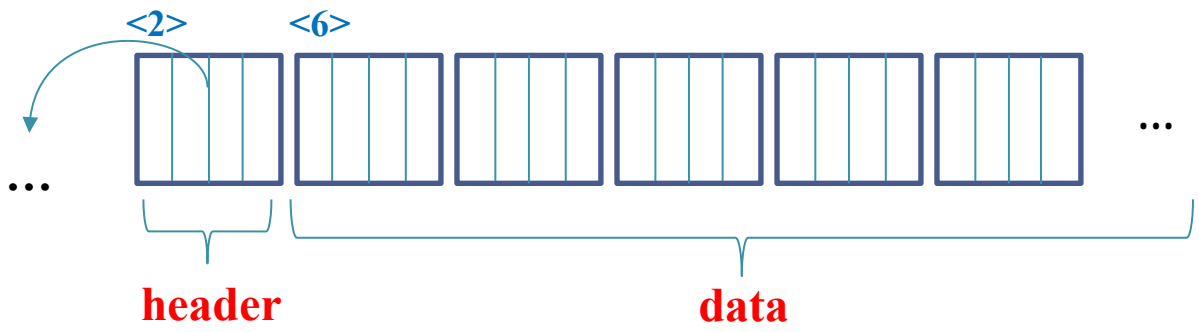
- Check array with **float**
  - void set_nItem(float* aData, int nItem): assign value nItem into the head's memory
    - void set_nItem(float* aData, int nItem){
      - *((int*)origin_addr(aData)) = nItem;
    - }

*(<2>)



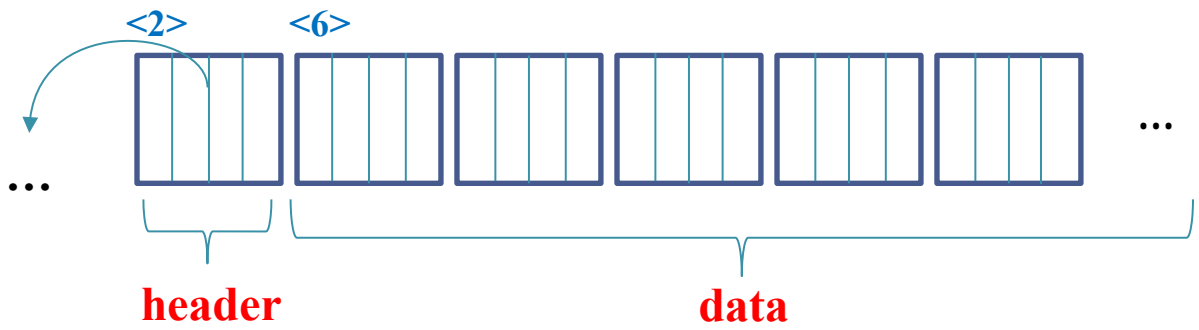
ALLOCATE & USE DYNAMIC ARRAY

- Check array with **float**
  - int get_nItem(float* aData): take the value nItem of the head's memory
    - int get_nItem(float* aData){
      - return *((int*)origin_addr(aData));
    - }



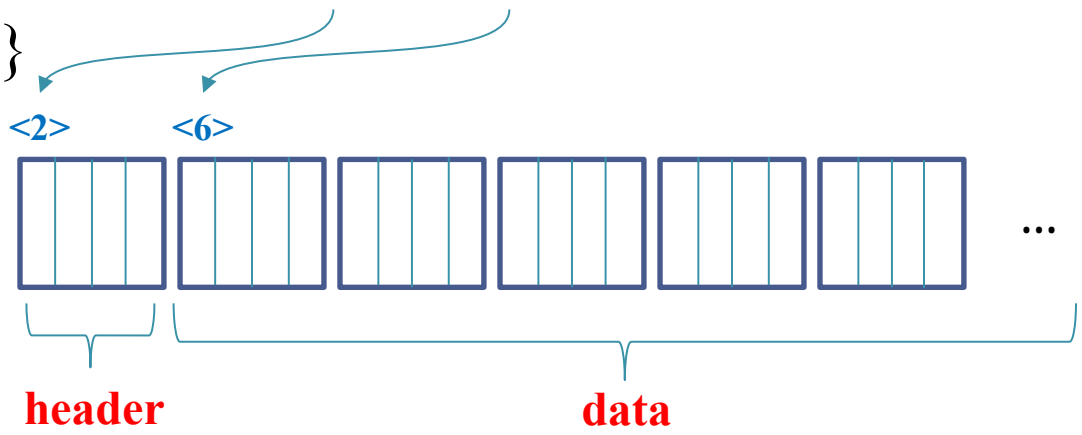
ALLOCATE & USE DYNAMIC ARRAY

- Check array with **float**
  - int floatArrSize (float* aData): take **a number of items** of 1D array aData
    - int floatArrSize(float* aData){
      - if(aData != NULL) return get_nItem(aData);
      - return 0;
    - }



ALLOCATE & USE DYNAMIC ARRAY

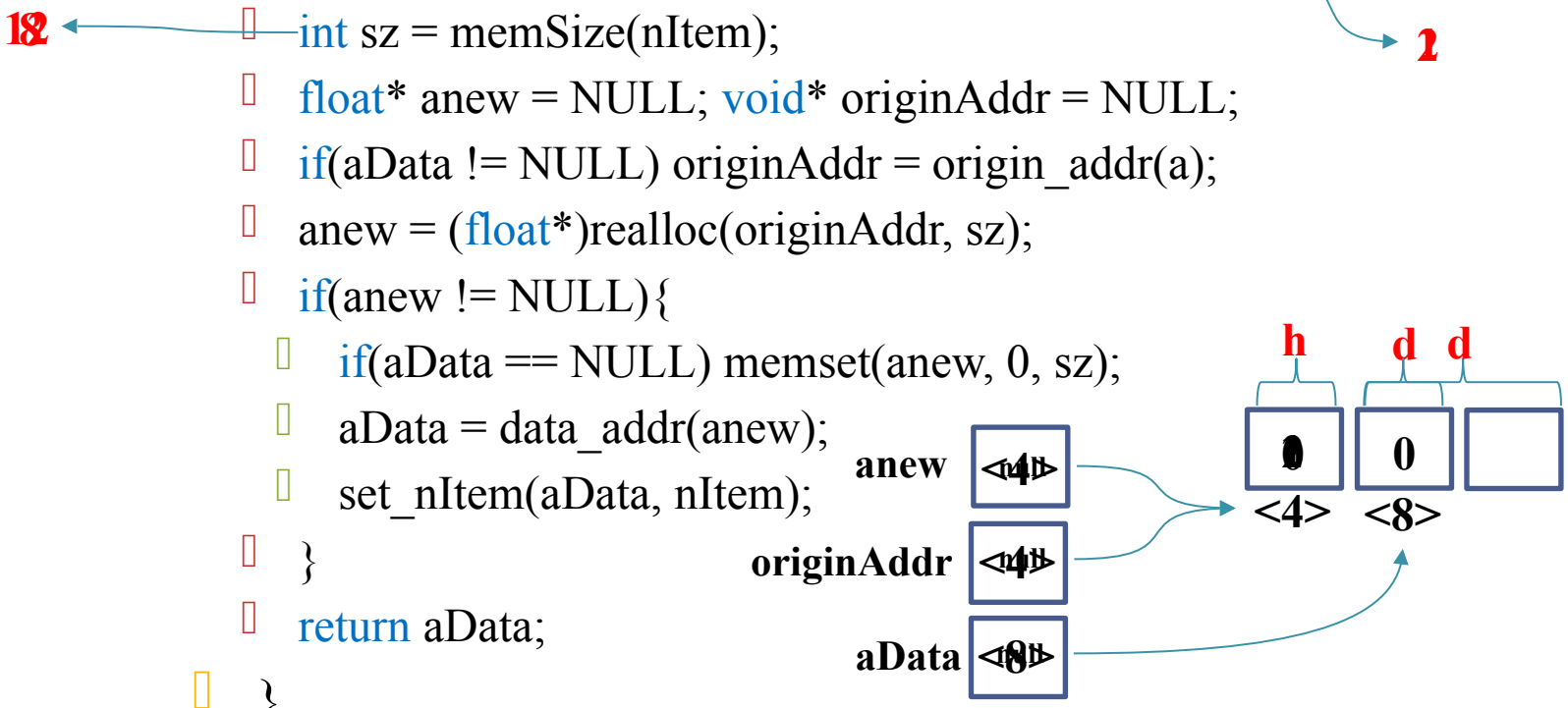
- Check array with **float**
  - void floatArrFree (float* aData): destroy all memory allocated (from address of head portion)
    - void floatArrFree (void* aData){
      - if(aData != NULL)
        - free(origin_addr(aData))
    - }



ALLOCATE & USE DYNAMIC ARRAY

- Check array with **float**
  - float* floatArrResize(float* aData, int nItem): **resize to new size** (nItem: a new number of items)

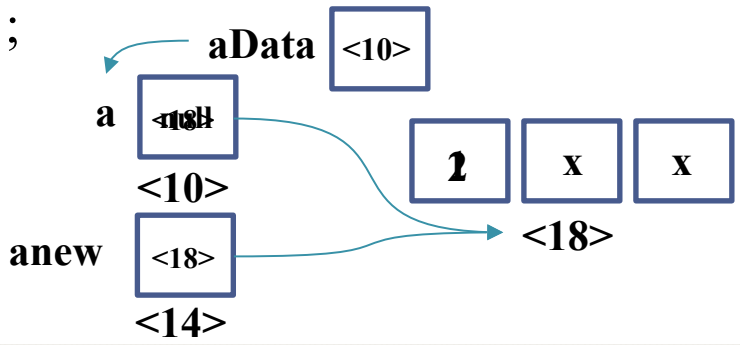
```
float* floatArrResize (float* aData, int nItem){
    int sz = memSize(nItem);
    float* anew = NULL; void* originAddr = NULL;
    if(aData != NULL) originAddr = origin_addr(a);
    anew = (float*)realloc(originAddr, sz);
    if(anew != NULL){
        if(aData == NULL) memset(anew, 0, sz);
        aData = data_addr(anew);
        set_nItem(aData, nItem);
    }
    return aData;
}
```



ALLOCATE & USE DYNAMIC ARRAY

- Check array with **float**
  - int floatArrPushback(float** aData, float x): **add *x* into 1D** array aData

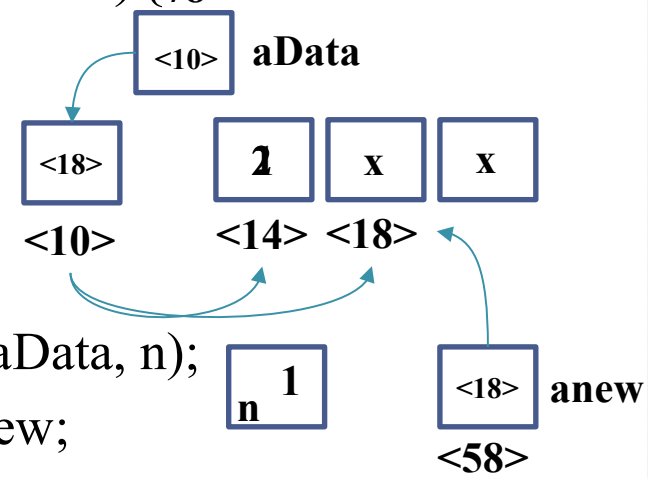
```
int floatArrPushback(float** aData, float x){
    int n = floatArrSize(*aData);
    float* anew = floatArrResize(*aData, n + 1);
    if(anew != NULL){
        anew[n] = x;
        *aData = anew;
        return 1;
    }
    return 0;
}
```



ALLOCATE & USE DYNAMIC ARRAY

- Check array with **float**
  - float floatArrPopback(float** aData): **take the last item out of** 1D array aData

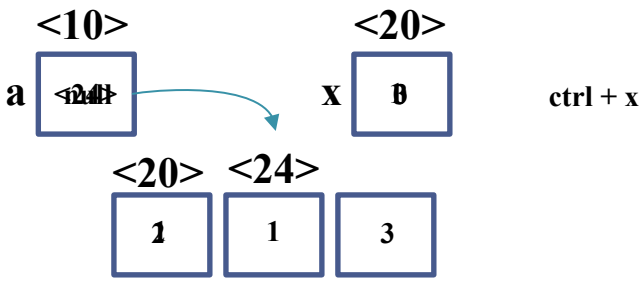
```
int floatArrPopback(float** aData){
    int n = floatArrSize(*aData);
    float x = 0;
    if(*aData != NULL && n > 0){
        n--; x = (*aData)[n];
        float* anew = floatArrResize(*aData, n);
        if(anew != NULL) *aData = anew;
    }
    return x;
}
```



ALLOCATE & USE DYNAMIC ARRAY

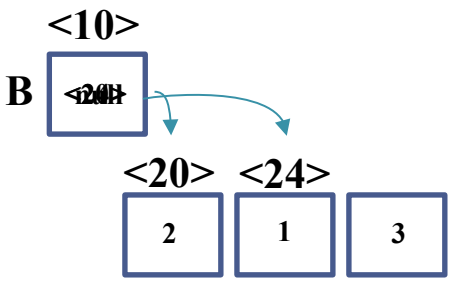
- Check array with **float**
  - float* floatArrInput(): **return 1D** array of **float** items

```
float* floatArrInput(){
    float* a = NULL, x = 0;
    while(cin >> x) {
        floatArrPushback(&a, x);
    }
    cin.clear();
    return a;
}
```



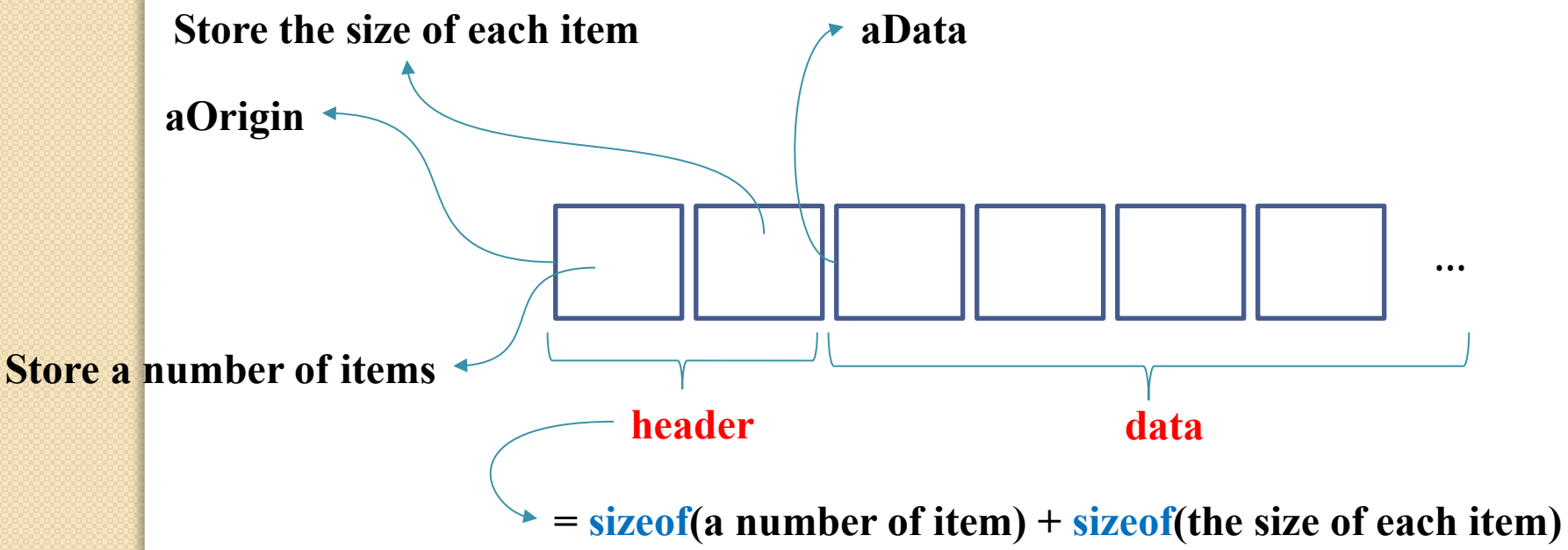
ALLOCATE & USE DYNAMIC ARRAY

- Check array with **float**
  - Demonstration of main()
    - void main(){
      - float* B = NULL;
      - B = floatArrInput();
      - floatArrFree(B);
    - }



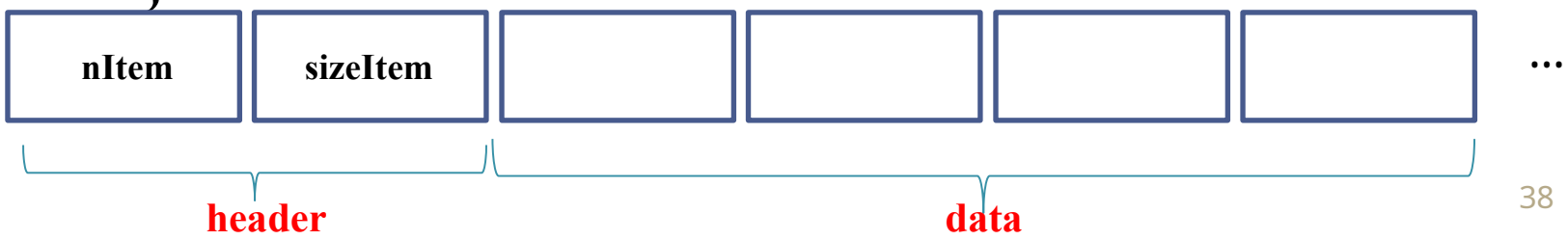
ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE



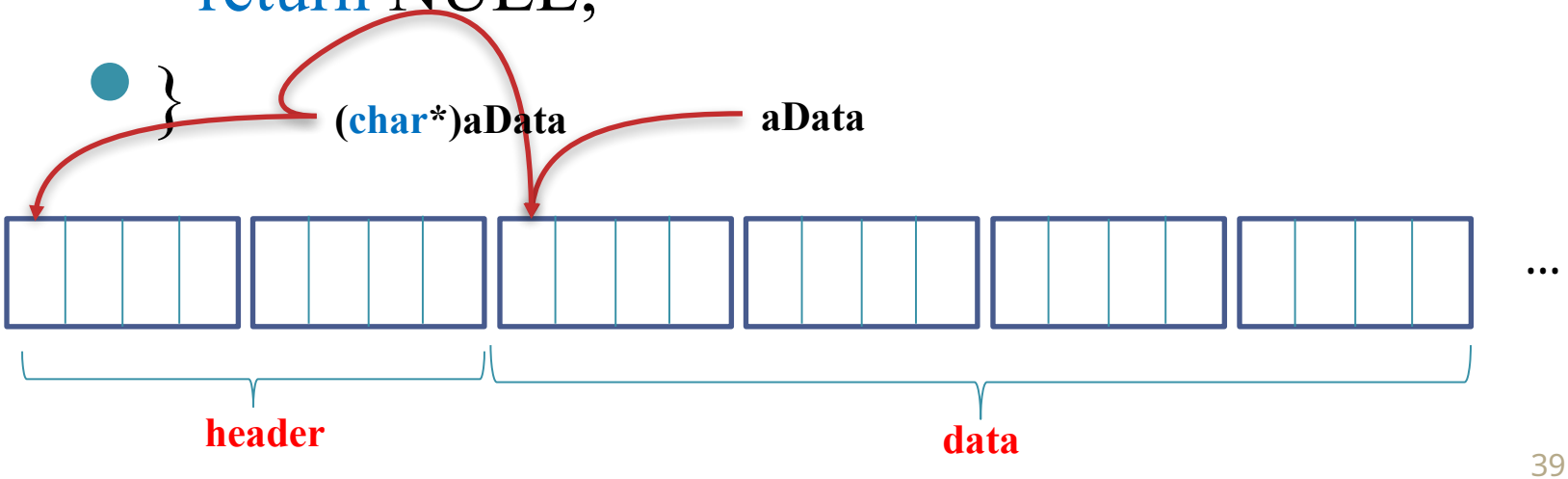
ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- Size of header: int headSize = sizeof(int) + sizeof(int)
- memSize(): calculate **sum of byte** needed
- int memSize(int nItem, int szItem){
  - return headSize + nItem * szItem;
- }



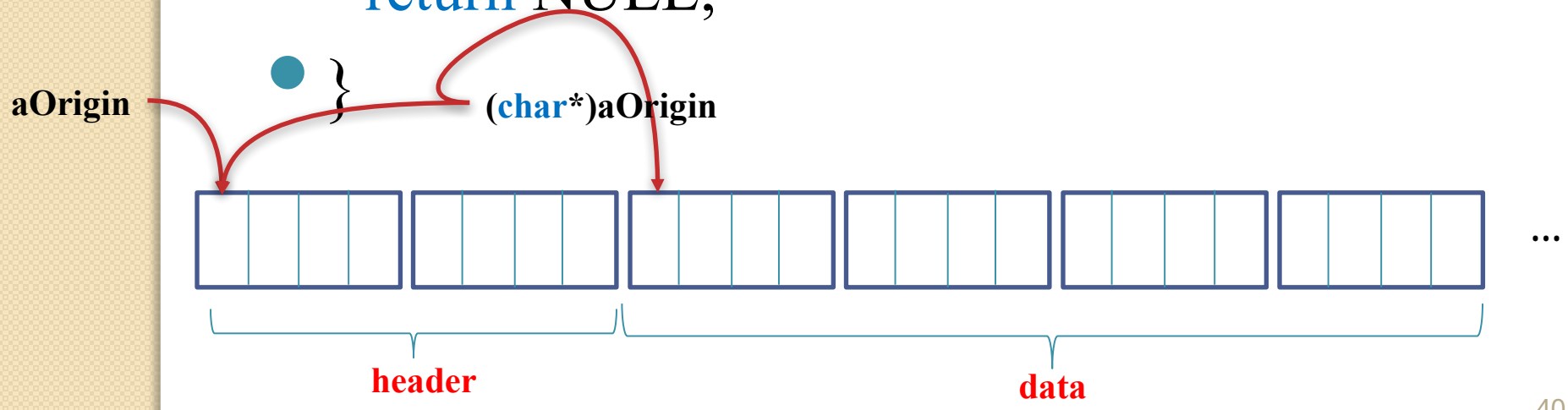
ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- void* origin_addr (void* aData){
  - if(aData != NULL)
    - return (char*)aData – headSize;
  - return NULL;
- }



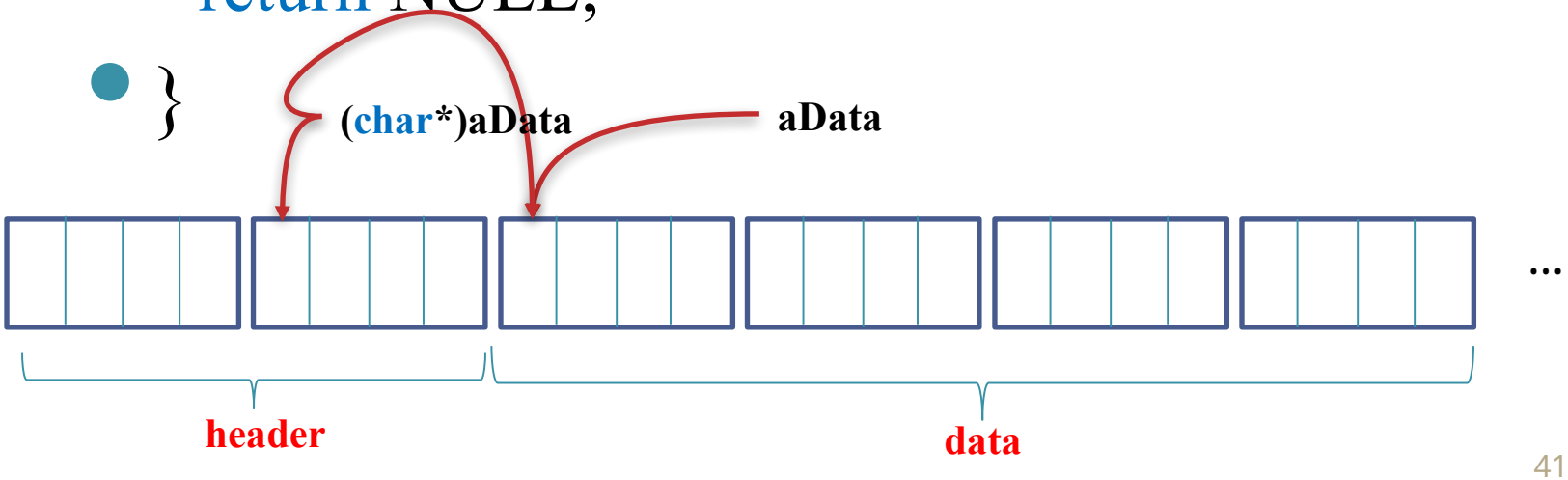
ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- void* data_addr (void* aOrigin){
  - if(aOrigin != NULL)
    - return (char*)aOrigin + headSize;
  - return NULL;
- }



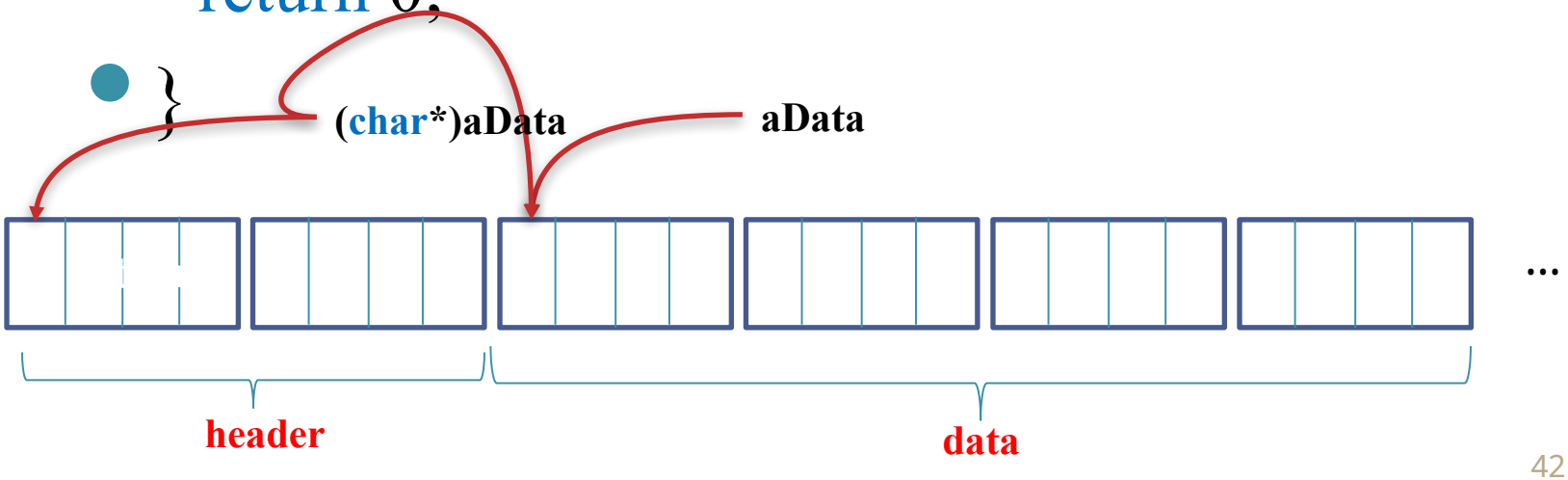
ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- void* sizeItem_addr (void* aData){
  - if(aData != NULL)
    - return (char*)aData – sizeof(int);
  - return NULL;
- }



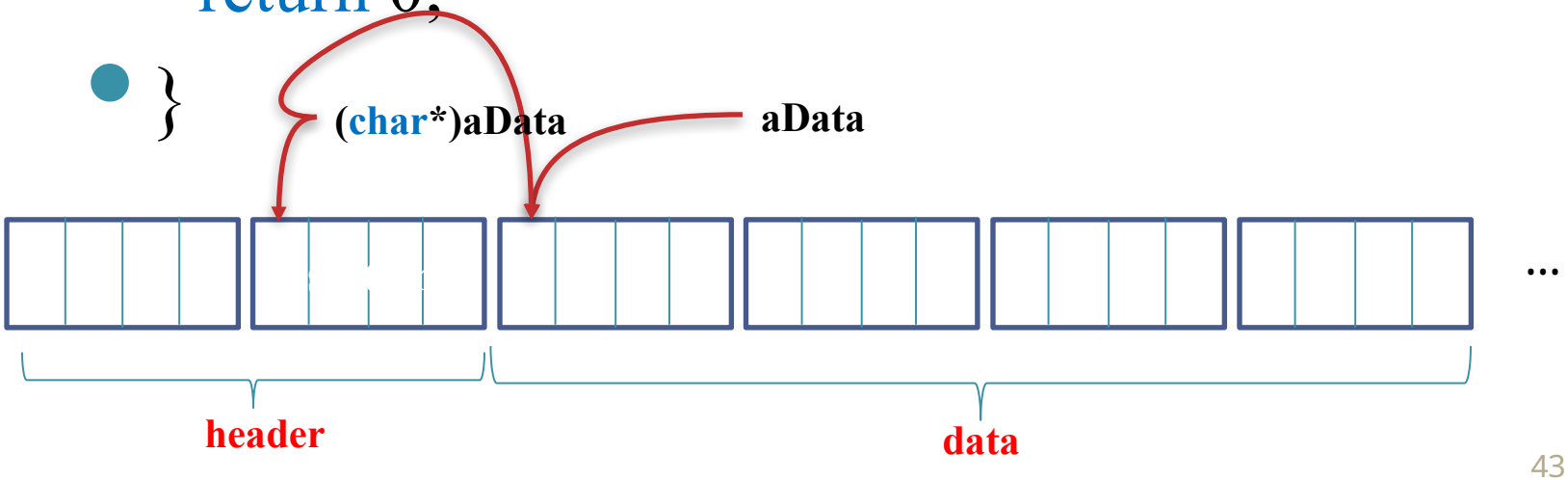
ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- int arrSize(void* aData){
  - if(aData != NULL)
    - return *(int*)origin_addr(aData);
  - return 0;
- }



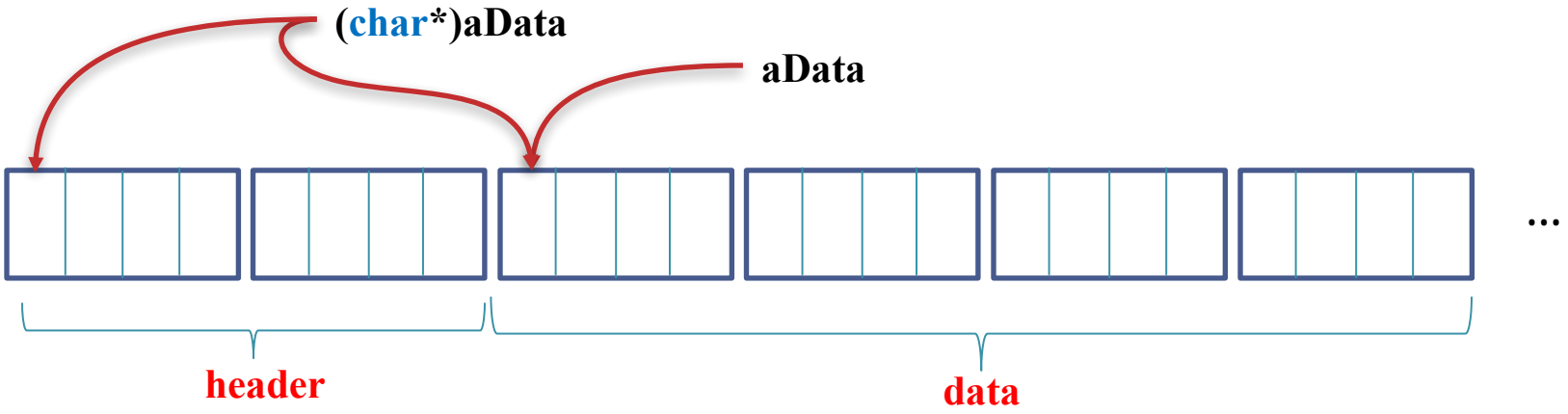
ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- int arrItemSize(void* aData){
  - if(aData != NULL)
    - return *(int*)sizeItem_addr(aData);
  - return 0;
- }



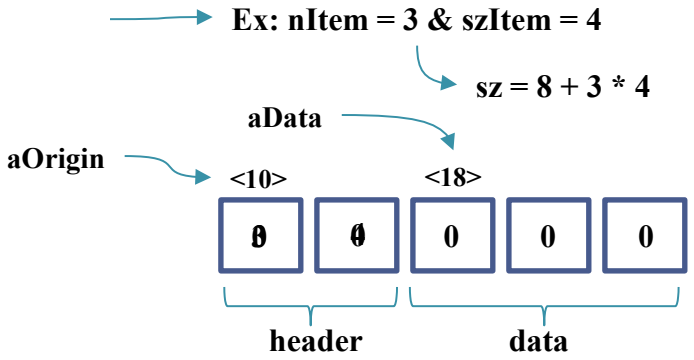
ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- void arrFree(void* aData){
  - if(aData != NULL) free(origin_addr(aData));
- }



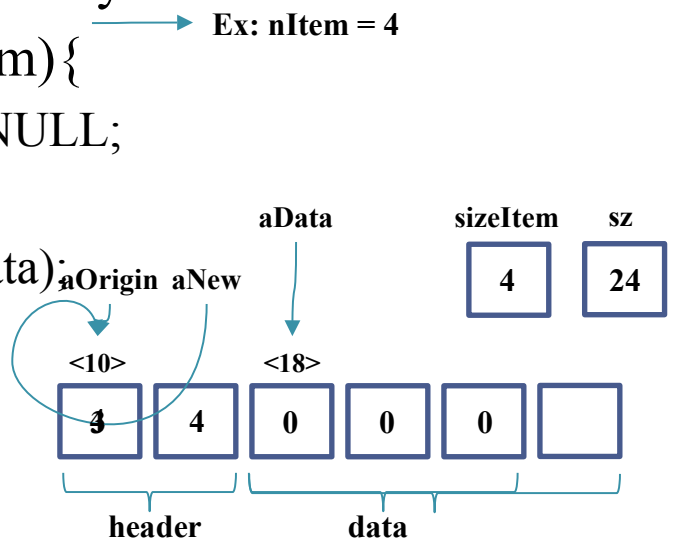
ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- arrInit(): Allocate the sum of byte
- void* arrInit(int nItem, int szItem){
  - int sz = memSize(nItem, szItem);
  - void* aOrigin = malloc(sz);
  - if(aOrigin != NULL){
    - memset(aOrigin, 0, sz);
    - void *aData = data_addr(aOrigin);
    - *(int*)origin_addr(aData) = nItem;
    - *(int*)sizeItem_addr(aData) = szItem;
    - return aData;
  - }
  - return NULL;
- }



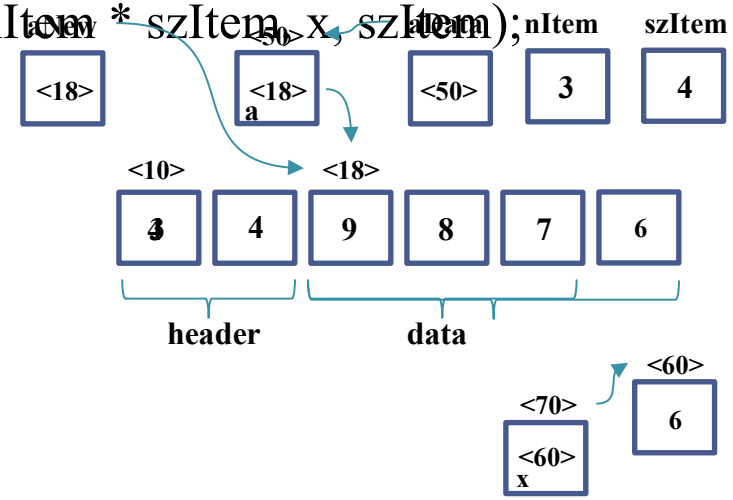
ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- arrResize(): resize the memory of 1D array aData
- void* arrResize(void* aData, int nItem){
  - if(aData == NULL || nItem < 0) return NULL;
  - void* aOrigin = origin_addr(aData);
  - int sizeItem = *(int*)sizeItem_addr(aData);
  - int sz = memSize(nItem, sizeItem);
  - void *aNew = realloc(aOrigin, sz);
  - if(aNew != NULL){
    - aData = data_addr(aNew);
    - *(int*)origin_addr(aData) = nItem;
    - return aData;
  - }
  - return NULL;
- }



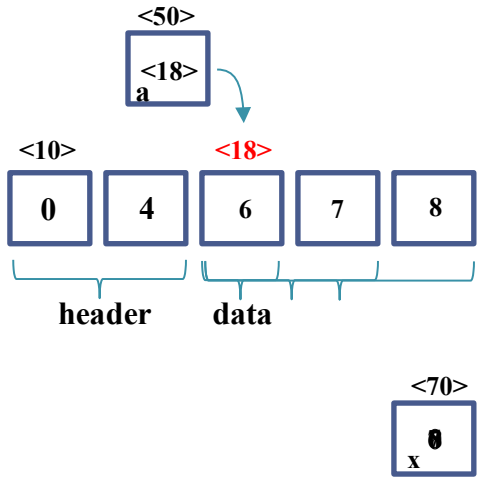
ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- Function arrPushback(): Add *x* into 1D array aData
- int arrPushback(void** aData, void* x){
  - int nItem = arrSize(*aData), szItem = arrItemSize(*aData);
  - void* aNew = arrResize(*aData, 1 + nItem);
  - if(aNew != NULL){
    - memmove((char*)aNew + nItem * szItem, x, szItem);
    - *aData = aNew;
    - return 1;
  - }
  - return 0;
- }



ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- Example: rebuild floatArrIn(): create and return a 1D array of items with float
- float* floatArrIn(){
  - float* a = (float*)arrInit(0, sizeof(float)), x = 0;
  - while(cin >> x){
    - arrPushback((void**)&a, &x);
  - }
  - cin.clear();
  - return a;
- }

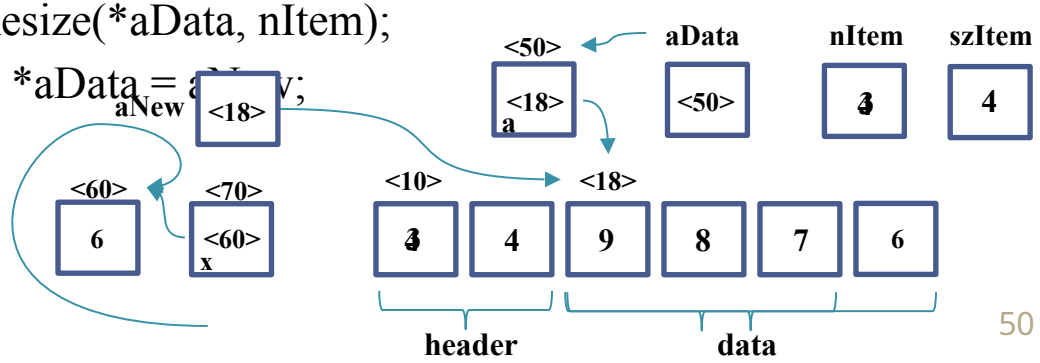


ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- Example: rebuild PhanSoArrIn(): create and return a 1D array of items with type of PhanSo
- PhanSo* PhanSoArrIn(){
  - PhanSo* a = (PhanSo*)arrInit(0, sizeof(PhanSo));
  - PhanSo x = {0, 1};
  - while(cin >> x){
    - arrPushback((void**)&a, &x);
  - }
  - cin.clear();
  - return a;
- }

ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- arrPopback: Take the last item of 1D array aData, and decrease it size by one
- void* arrPopback(void** aData){
  - int nItem = arrSize(*aData), szItem = arrItemSize(*aData);
  - void* x = malloc (szItemSize);
  - if(*aData != NULL && nItem > 0){
    - nItem--;
    - memmove(x, (char*)(*aData) + nItem * szItem, szItem);
    - void* aNew = arrResize(*aData, nItem);
    - if(aNew != NULL) *aData = aNew;
  - }
  - return x;
- }



ALLOCATE & USE DYNAMIC ARRAY

- Goal: build support functions processing 1D array with GENERAL TYPE
- Example: main() with float
- void main(){
  - cout << "Input items:\n";
  - float* B = floatArrIn();
  - float* x = (float*)arrPopback((void**)&B);
  - cout << "After pop: \n";
  - floatArrOut(B);
  - cout << "\nPopped element: " << *x << endl;
  - free(x);
  - arrFree((float*)B)
- }

ALLOCATE & USE DYNAMIC ARRAY (OPERATOR '[]')

- Goal: help code be more natural when using struct to create array
- Need to return reference when using []

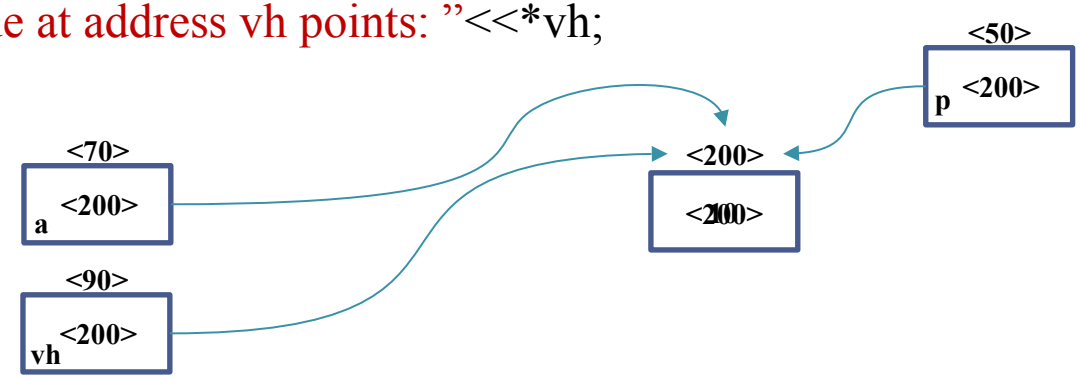
//File.h	//File.cpp
struct floatArray{	#include "File.h"
int n;	float&floatArray::operator[](int i){
float* arr;	if(arr != NULL && i >= 0 && i < n) return arr[i];
float& operator[](int);	return dummy;
};	}
void floatArrayOutput(floatArray& a){	
if(a.arr == NULL) return;	
for(int i = 0; i < a.n; i++) cout << a[i] << " "; // a.operator[](i)	
}	

POINTER-CHECKING TECHNIQUES

- Can convert normal variable = pointer
- Need to carefully control

```
void markMem(void* a){
  cout<<"Address of a: "<<&a;
  cout<<"Address a points: "<<a;
  void** vh = (void**)a;
  cout<<"Address of vh: "<<&vh;
  cout<<"Address vh points: "<<vh;
  cout<<"Value at address vh points: "<<*vh;
  *vh = a;
  cout<<"Value at address vh points: "<<*vh;
}
```

```
void main(){
  int* p = new int; *p = 10;
  cout<<"Address of p: "<<&p;
  cout<<"Address p points: "<<p;
  cout<<"Value at address p points: "<<*p;
  markMem(p);
  free(p);
}
```

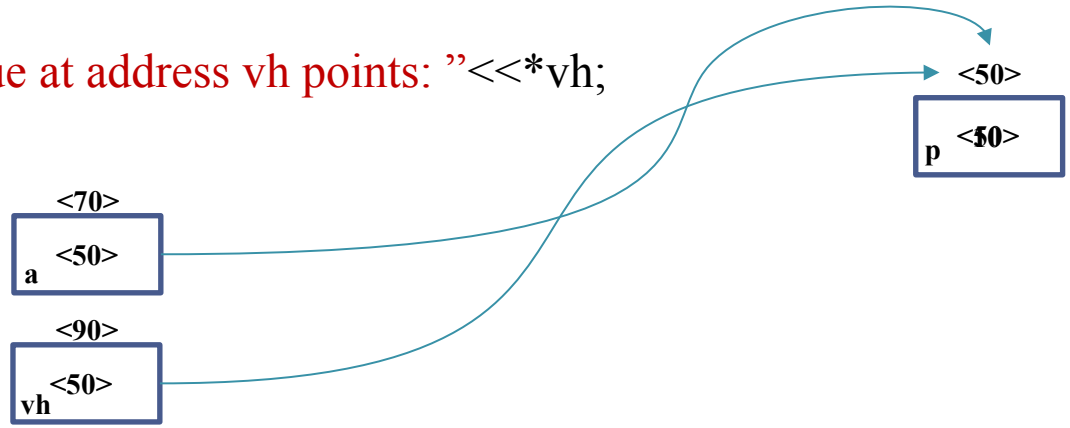


POINTER-CHECKING TECHNIQUES

- Can convert normal variable $=$ pointer
- Need to carefully control

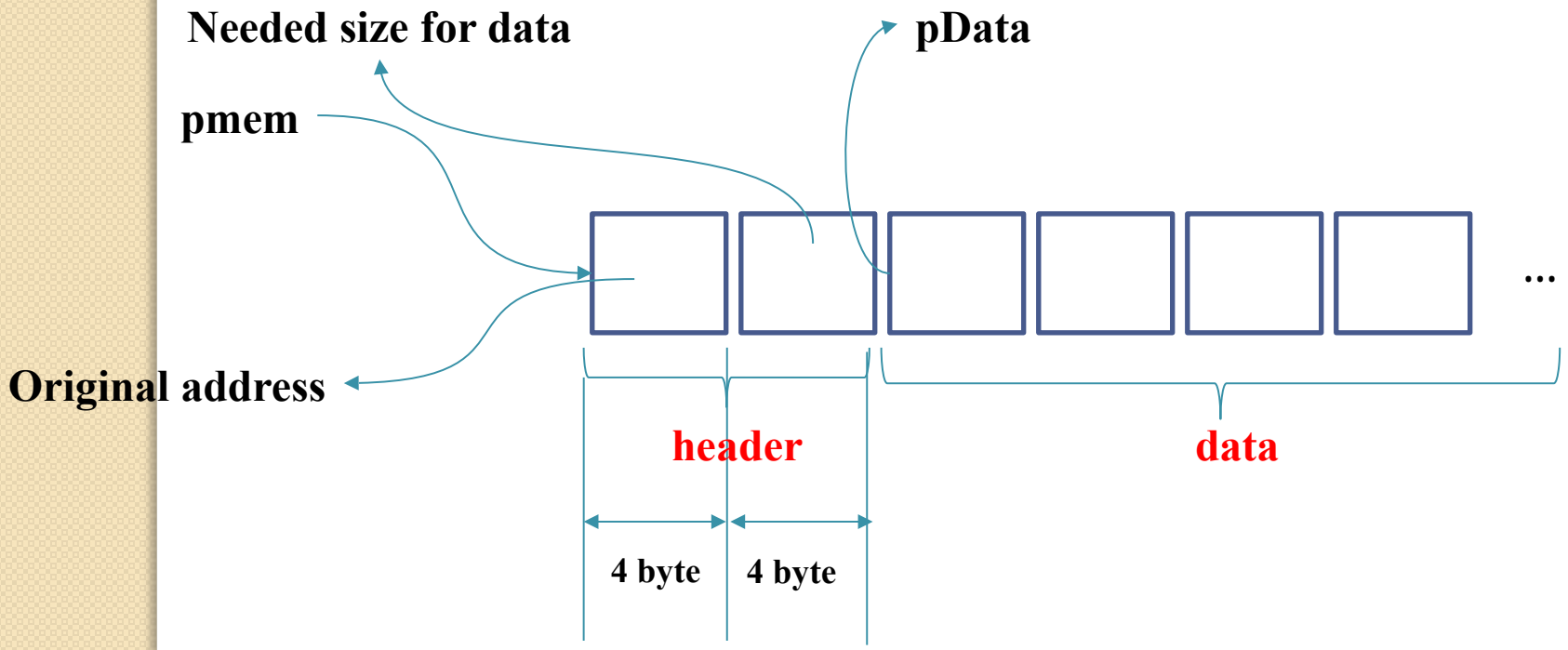
```
void markMem(void* a){
  cout<<"Address of a: "<<&a;
  cout<<"Address a points: "<<a;
  void** vh = (void**)a;
  cout<<"Address of vh: "<<&vh;
  cout<<"Address vh points: "<<vh;
  cout<<"Value at address vh points: "<<*vh;
  *vh = a;
  cout<<"Value at address vh points: "<<*vh;
}
```

```
void main(){
  int p = 10;
  cout<<"Address of p: "<<&p;
  cout<<"Value of p: "<<p;
  markMem(&p);
}
```



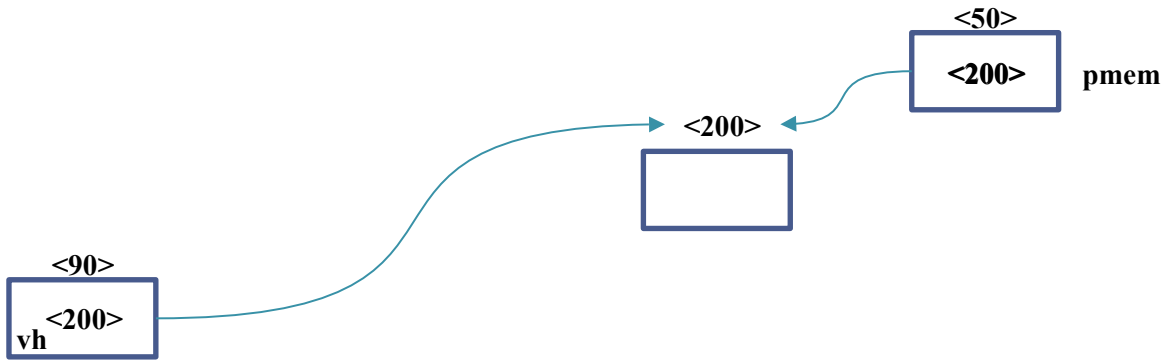
POINTER-CHECKING TECHNIQUES

- Divide memory into 2 parts:
  - Header: contains information of **original pointer and needed size** for data
  - Data: contains data



POINTER-CHECKING TECHNIQUES

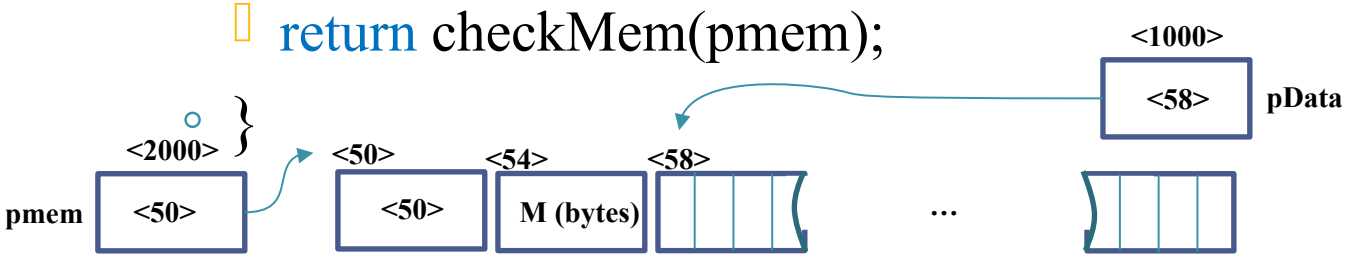
- Build markMem and checkMem
  - static void markMem(void* pmem){
    - *(void**)pmem = pmem;
  - }



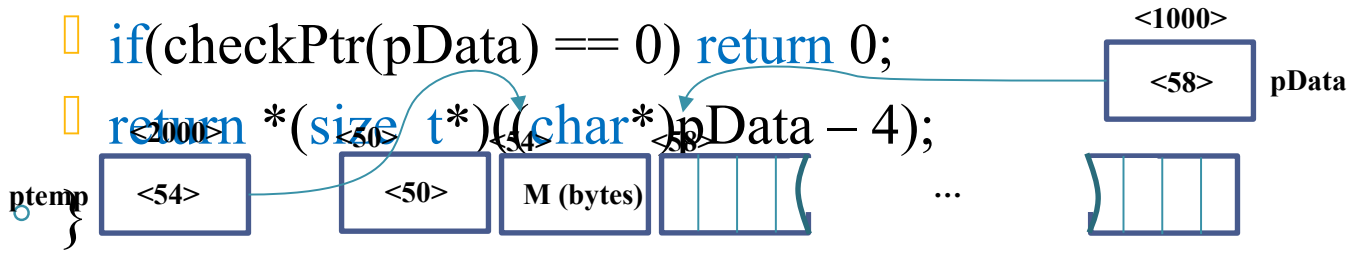
  - static int checkMem(void* pmem){
    - return *(void**)pmem == pmem;
  - }

POINTER-CHECKING TECHNIQUES

- Build checkPtr and safeSize
  - int checkPtr(void* pData){
    - if(pData == NULL) return 0;
    - void* pmem = (char*)pData – 8;
    - return checkMem(pmem);
  - }

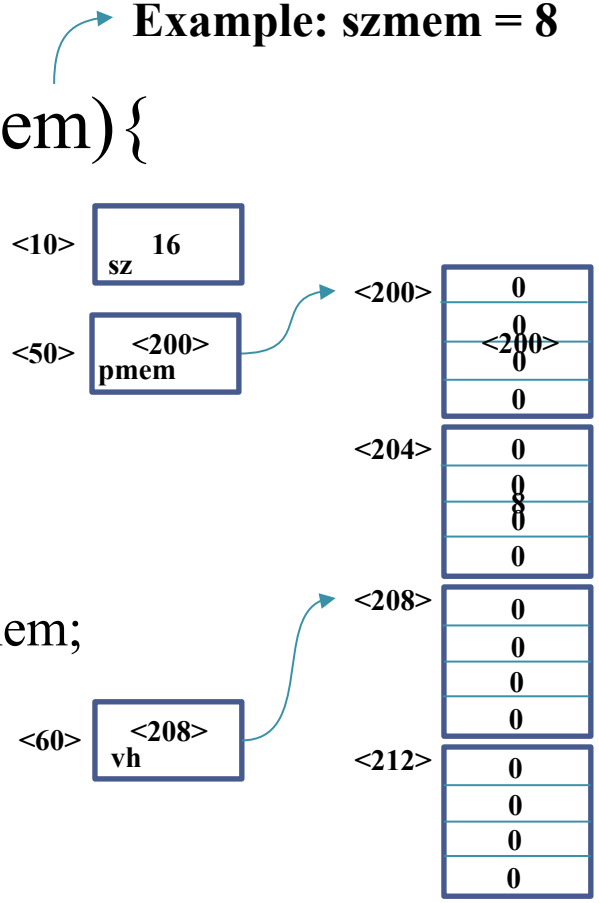


  - size_t safeSize(void* pData){
    - if(checkPtr(pData) == 0) return 0;
    - return *(size_t*)((char*)pData – 4);
  - }



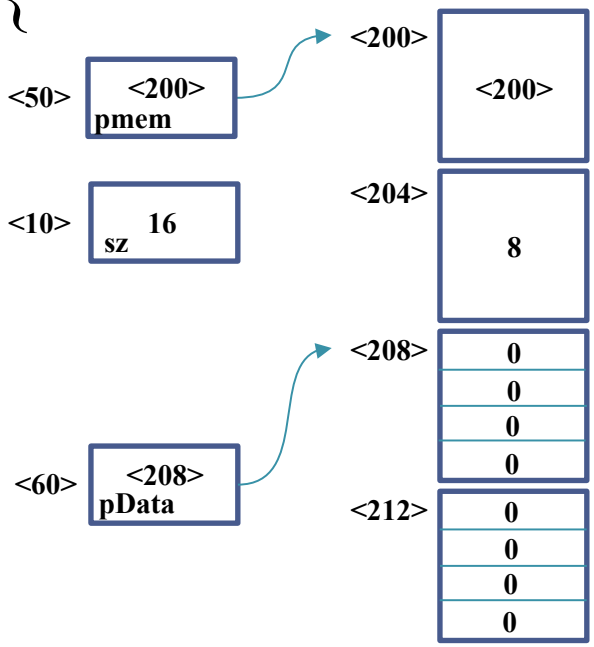
POINTER-CHECKING TECHNIQUES

- Build safeMalloc
  - void* safeMalloc(size_t szmem){
    - size_t sz = szmem + 8;
    - void* pmem = malloc(sz);
    - if(pmem != NULL){
      - memset(pmem, 0, sz);
      - markMem(pmem);
      - *(size_t*)((char*)pmem + 4) = szmem;
      - return (char*)pmem + 8;
    - }
    - return NULL;
  - }



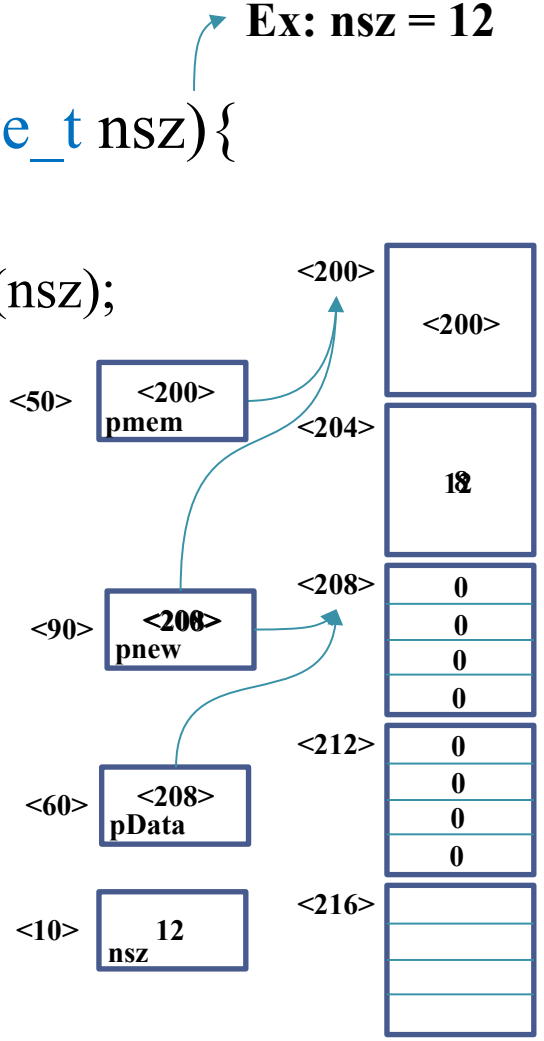
POINTER-CHECKING TECHNIQUES

- Build safeFree
  - void* safeFree(void* pData){
    - if(pData != NULL){
      - void* pmem = (char*)pData - 8;
      - if(checkMem(pmem)){
        - size_t sz = safeSize(pData) + 8;
        - memset(pmem, 0 ,sz);
        - free(pmem);
      - }
    - }
  - }



POINTER-CHECKING TECHNIQUES

- Buid safeRealloc
  - void* safeRealloc(void* pData, size_t nsz){
    - if(nsz <= 0) return NULL;
    - if(pData == NULL) return safeMalloc(nsz);
    - void* pmem = (char*)pData – 8;
    - if(!checkMem(pmem)) return NULL;
    - void* pnew = realloc(pmem, nsz + 8);
    - if(pnew != NULL){
      - markMem(pnew);
      - *(size_t*)((char*)pnew + 4) = nsz;
      - pnew = (char*)pnew + 8;
    - }
    - return pnew;
  - }



POINTER-CHECKING TECHNIQUES

- How to use
  - void main(){
    - int* arr = (int*)safeMalloc(sizeof(int) * 4);
    - if(checkPtr(arr) != 0){
      - arr[0] = 1; arr[1] = 2;
      - arr[2] = 3; arr[3] = 4;
      - for(int i = 0; i < 4; i++) cout << arr[i];
    - }
    - safeFree(arr);
  - }