

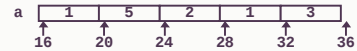
Pointers & Memory Allocation Wrap-Up

Topics

- Arrays
- Association between pointers and arrays
- Memory-related perils and pitfalls

Array Example

```
//Automatic allocation and init
int a[] = { 1, 5, 2, 1, 3 };
```



Notes

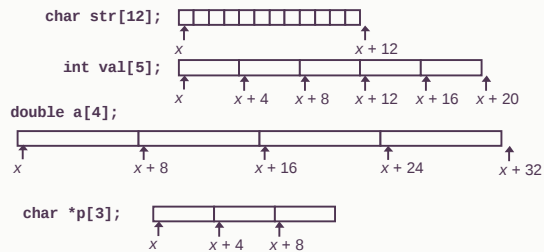
- Declaration above equivalent to:
#define N 5
int a[N]; a[0] = 1; a[1] = 5; ... a[4] = 3;
- Code Does Not Do Any Bounds Checking !
Valid range 0 – (N –1)

Array Allocation

Basic Principle

$T \ A[L];$

- Array of data type T and length L
- Contiguously allocated region of $L * \text{sizeof}(T)$ bytes

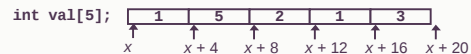


Array Access

Basic Principle

$T \ A[L];$

- Array of data type T and length L
- Identifier A can be used as a pointer to array element 0



Reference	Type	Value
val[4]	int	3
val	int *	x
val+1	int *	x + 4 //!!! Adds sizeof(int)
&val[2]	int *	x + 8
val[5]	int	??
*(val+1)	int	5
val + i	int *	x + 4 i

Multi-Dimensional Array Allocation

Declaration

T A[R][C];

- Array of data type T
- R rows, C columns
- Type T element requires K bytes

A[0][0]	...	A[0][C-1]
⋮		⋮
A[R-1][0]	...	A[R-1][C-1]

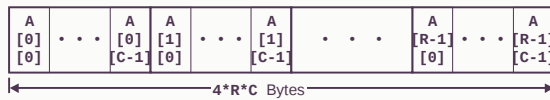
Array Size

- $R * C * K$ bytes

Arrangement

- Row-Major Ordering

int A[R][C];

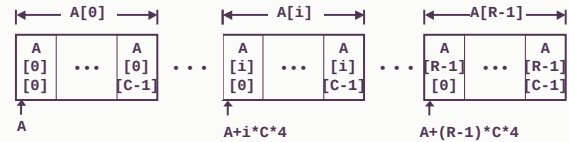


Nested Array Allocation

Row Vectors

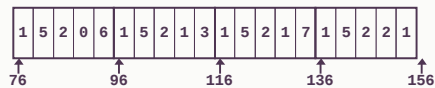
- A[i] is array of C elements
- Each element of type T

int A[R][C];



Array Init/Allocation Example

```
#define PCOUNT 4
int p[PCOUNT][5] =
{{1, 5, 2, 0, 6},
 {1, 5, 2, 1, 3 },
 {1, 5, 2, 1, 7 },
 {1, 5, 2, 2, 1 }};
```



- Allocated contiguously
- Each element is an array of 5 int's

Dynamic Allocation of Array

- int *a;
- (the same as a[N]
- except needs to be allocated dynamically)
- cin >> n;
- a = new int[n];

Feeling of Array Space

- `int *a;`
- `cin >> n;`
- `a = new int[n];`
- `delete a[ ];` //Otherwise if say "delete a"
//only first element is
//freed

Array Loop Using Pointers

Computations

- `zbegin = pointer to begin,`
`zend = pointer to end`
- `z++` increments by 4

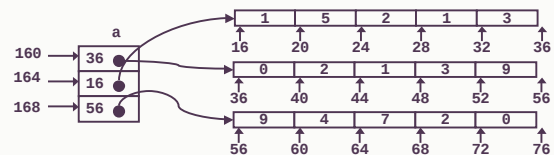
```
int A[10];
int i = 0;
int *zbegin = A;
int *zptr = zbegin;
int *zendptr = A + 10;
while(zptr < zendptr) {
    i++;
    zptr++;
    cout << A[i] << *zptr;
}
```

Dyn. Allocation of Multi-Dim Array

- `int **a;`
(the same as `a[N][M]`
except needs to be allocated dynamically)
- `cin >> n; cin >> m; //dimensions can be set at run-time`
- `a = new int[n];`
for each `i`
 `a[i] = new int[m];`

Dyn. Allocation of Multi-Dim Array

- `int **a;`
- Each pointer points to an array of `int`'s



Summary

Arrays in C++

- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Memory-Related Bugs

Dereferencing “dangling” or NULL pointers

Reading uninitialized memory

Overwriting memory

Referencing nonexistent variables

Freeing blocks multiple times

Referencing freed blocks

Failing to free blocks

Next – examples of each – BAD ! please AVOID !

Dereferencing NULL Pointers

The classic bug:

```
int i;  
int *pi = NULL; //pointer pi contains a bad address  
cout << *i; //this will likely result in a SEGFault
```

Dereferencing “dangling” Pointers

Another classic bug:

```
int i;  
int *pi = new int; // pi contains a valid address addr  
...  
delete pi; //pi is now a “dangling” pointer  
           //still contains addr, but the object at  
           //           addr has just been freed  
...  
*pi = 5;    //dereferencing a dangling pointer can  
           //cause arbitrary memory corruption
```

Reading Uninitialized Memory

Another classic bug (already shown):

```
/* return y = Ax */
int *matvec(int **A, int *x) {
    int *y = new int[N];
    int i, j;

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            y[i] += A[i][j]*x[j];
    return y;
}
```

Overwriting Memory

Off-by-one error (all arrays start at 0, go to N-1)

```
int *p;
p = new int[N];

for (i=0; i<=N; i++) {
    p[i] = i;
}
```

```
int **p;
p = new int*[N];

for (i=0; i<=N; i++) {
    p[i] = new int[M];
}
```

Overwriting Memory

Not checking the max string size

```
char s[8];
int i;

cin >> s; /* reads "123456789" from keyboard */
```

Basis for buffer overflow

Overwriting Memory

Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {
    while (*p && *p != val)
        p += sizeof(int);
    return p;
}
```

Referencing Nonexistent Variables

Forgetting that local variables disappear when a function returns

```
int *foo () {
    int val;
    return &val;
}

int main () {

    int *p = foo ();
    cout << *p;
    //p is a "dangling" pointer
    //the variable it was pointing to
    //has been freed.
```

Freeing Blocks Multiple Times

Nasty!

```
px = new int;
<manipulate x>
delete px;

py = new int;
<manipulate y>
delete px;
```

Referencing Freed Blocks

Evil!

```
px = new int[N];
<manipulate x>
delete px[];
...
py = new int[M];
for (i=0; i<M; i++)
    y[i] = x[i]++;
```

Failing to Free Blocks (Memory Leaks)

Slow, long-term killer!

```
void foo() {
    int *p = new int[128];
    return; /* p block is now garbage */
}
```

Failing to Free Blocks (Memory Leaks)

Freeing only part of a data structure

```
struct list {
    int val;
    struct list *next;
};

foo() {
    struct list *head =
        new struct list;
    head->val = 0;
    head->next = NULL;
    <create and manipulate the rest of the list>
    ...
    delete head;
    return;
}
```

Dealing With Memory Bugs

Conventional debugger (gdb)

- Good for finding bad pointer dereferences
- Hard to detect the other memory bugs
- Self-guards such as making pointers = NULL when uninitialized or “dangling” (after delete), may make bugs easier to track