CS 106X, Lecture 18
Arrays and Trees

reading:

Programming Abstractions in C++, Chapters 11.3, 16.1

Arrays (11.3)

```
type name[length];
```

- A statically allocated *(stack-allocated)* array; can never be resized.
- Memory does not need to be freed; will be automatically released.
- Example: `int homeworkGrades[7];`

```
type* name = new type[length];
```

- A **dynamically allocated** *(heap-allocated)* array.
- The variable that refers to the array is a **pointer**.
- The memory allocated for the array must be manually released, or else the program will have a **memory leak**. (>_<)
- Example: `int* homeworkGrades = new int[7];`

uh i saw that hand up here um what is the type of the stack allocated

Initialized?

```
type* name = new type[length];   // uninitialized
type* name = new type[length](); // initialize to 0
```

int* heapArray = new int[10]() 加()初始化代0，zero out；也可以带初始值int*a = new int[10]{1, 2, 3…}加{}

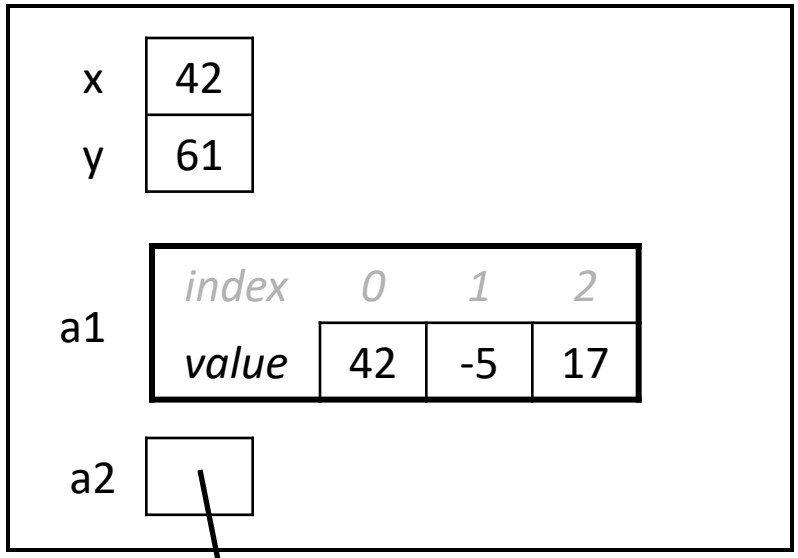
- If ( ) are written after the array [ ], it will set all array elements to their default zero-equivalent value for the data type. *(slower)*
- If no ( ) are written, the elements are uninitialized, so whatever garbage values were stored in that memory beforehand will be your elements.

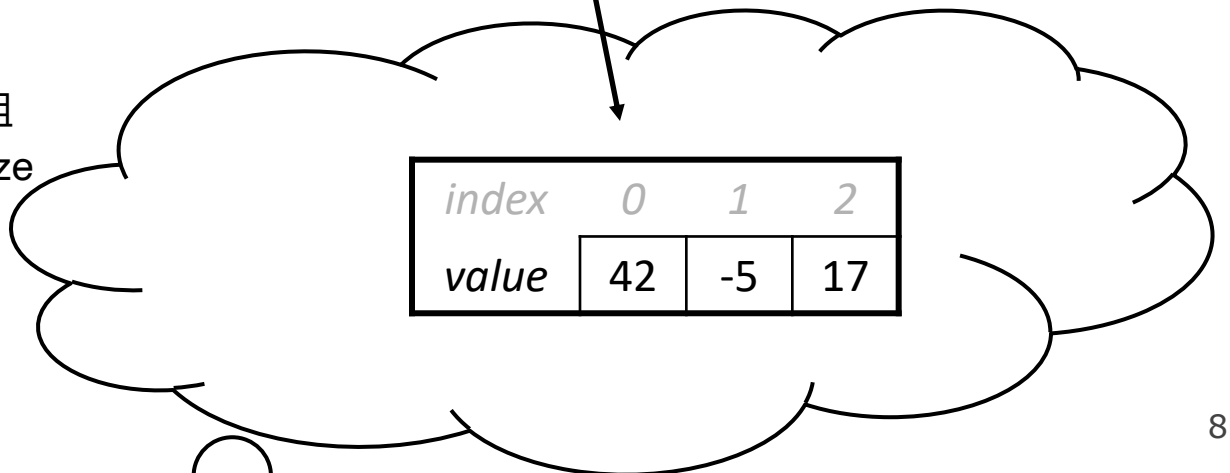
```
int* a = new int[3];
cout << a[0];              // 2395876
cout << a[1];              // -197630894

int* a2 = new int[3]();
cout << a[0];              // 0
cout << a[1];              // 0
```

this leads to bugs to have garbage values and so we're going to

Stack and Heap

```
void makeArrays() {
    int x = 42;
    int y = 61;
    int a1[3];
    int* a2 = new int[3];
    delete[] a2;
    ...
}
```





c++: 数组尽量不用pointer，用index

stack array和heap array的对比：

- 语法：int a1[10]：stack

int* a2 = new int[10]: heap

- 特点：stack长度定死了；heap数组长度可以根据情况修改指针进行resize

Arrays As Parameters

```
void myFunction(int arr[]) {
      ...
}

int main() {
     int myArray[6];
     myFunction(myArray);
     return 0;
}
```

Array Length

- C++ arrays have no members!
  - No **`size`** field
  - No helpful methods like **`indexOf`**

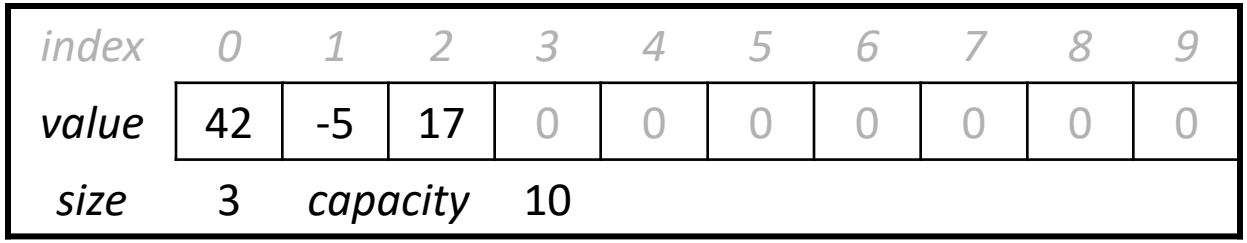
数组传参数：array+size

```
void myFunction(int arr[], int size) {
      ...
}

int main() {
    int arraySize = ...
    int myArray[arraySize];
    myFunction(myArray,  arraySize);
}
```

How Vector/Stack works

- Vectors and Stacks contain internal **arrays** to store elements.
- The array is created with some extra space (it is an "unfilled array"), and is replaced with a larger array when space runs out.

```
Vector<int> v;
v.add(42);
v.add(-5);
v.add(17);
```



0为底；右边为顶

```
Stack<int> s;
s.push(42);
s.push(-5);
s.push(17);
```

Outgrowing Our Shell

可变数组长度一般以x的整数倍增长

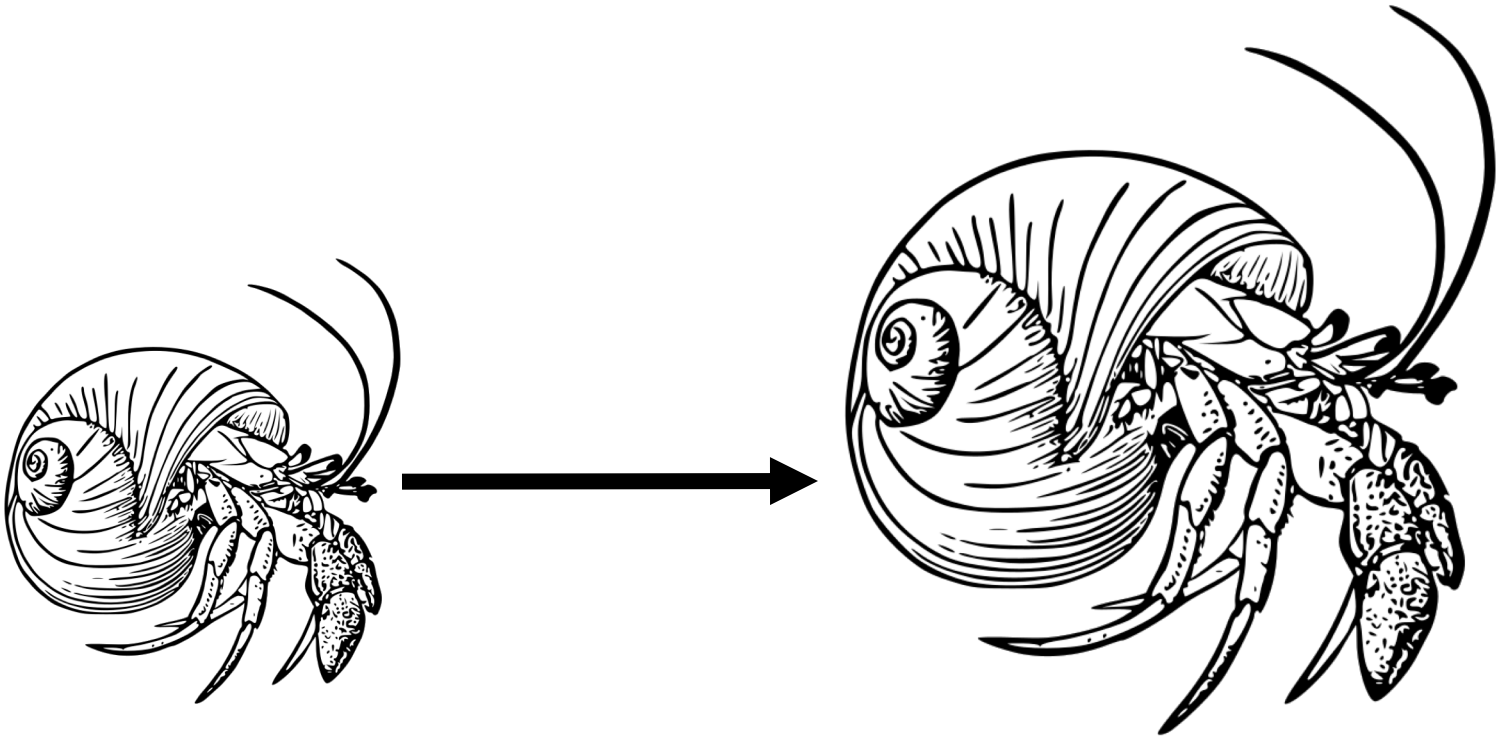


Image from Freshwater and Marine Image Bank: https://commons.wikimedia.org/wiki/File:FMIB_52669_Pagurus_bernhardus,_the_hermit-crab-retouched.svg

Exercise

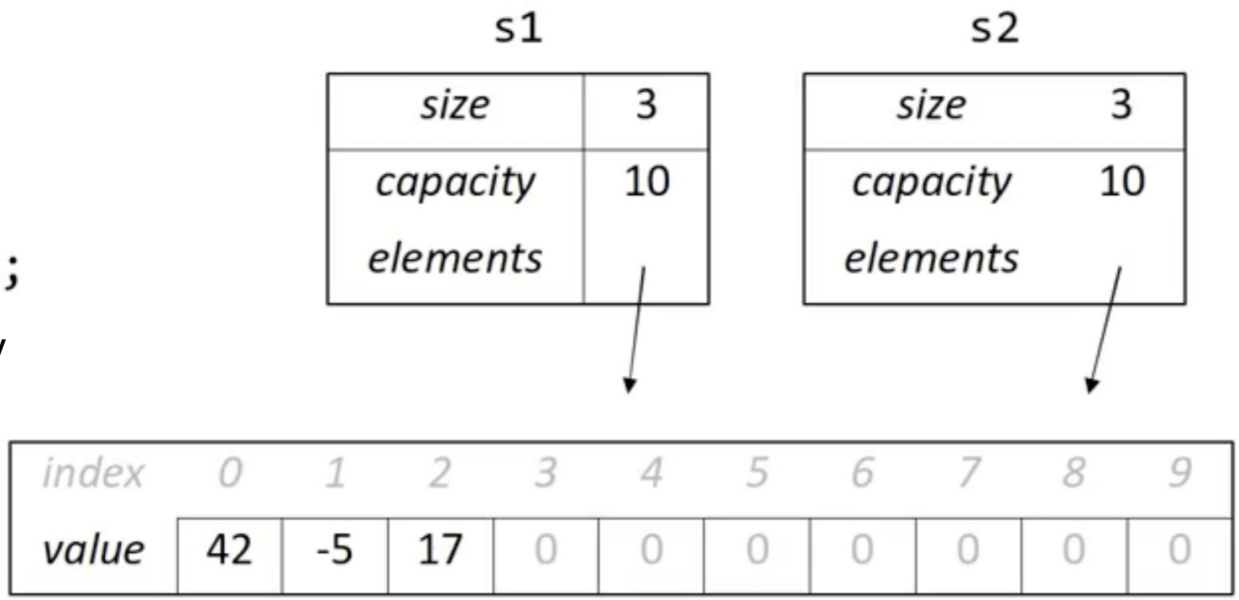
- Let's implement a **stack** using an unfilled array.
  - We'll call it **`ArrayStack`**. It will be very similar to the C++ `Stack`.
  - its behavior:
    - `push(`***`value`***`)`
    - `pop()`
    - `peek()`
    - `isEmpty()`
    - `toString()`
    - `operator <<`
  - The stack's *size* will be the number of elements added to it so far.
    - The actual array length ("capacity") in the object may be larger.
      We'll start with an array of **length 10** by default.

Shallow copy bug (12.7)

- Assigning one list to another causes them to share an array:

```
ArrayStack s1;
s1.push(42);
s1.push(-5);
s1.push(17);
ArrayStack s2 = s1;
```

直接用=赋值是shallow copy



- A change to one will affect the other. *(That's bad!)*

```
s2.push(88);
s1.pop();
```

  - Also: When the objects are deleted, the memory is freed twice. *(bad)*

Deep copy

- To correct the shallow copy bug, we must define:

  再创建一个copy constructor或者overload = operator

  - a **copy constructor** (constructor that takes a list as a parameter)

```
ArrayStack(const ArrayStack& stack);
```

  - an **assignment operator** (overloaded = op between two lists)

```
ArrayStack& operator =(const ArrayStack& stack);
```

    return a reference：返回等式左边的应用

    - in both of these, we will make a *deep copy* of the array of elements.

Rule of Three: 当类里面出现了用new分配的内存时，要同时考虑三点：1. copy constructor 2.= 3. destruclor清内存

- *Rule of Three*: In C++, when you define one of these three items in your class, you probably should define all three:

  1) copy constructor 2) assignment operator 3) destructor

it's basically like if you need any of these three

Forbid copying

- One quick fix is to just forbid your objects from being copied.
  - Declare a *private* copy constructor and = operator in the `.h` file.
  - Don't give them any actual definition/body in the `.cpp` file.

直接把两个东西放在private，也不给定义，禁用copy

```
// ArrayStack.h
private:
    ArrayStack(const ArrayStack& stack);
    ArrayStack& operator =(const ArrayList& stack);
```

  - Now if the client tries `s2 = s1;` it will not compile.
  - Avoids the shallow copy bug, but restrictive and less usable.

Our class is simple enough that we've defined all but the assignment operator in the class body. The first constructor takes an (optional) `string` argument. That constructor dynamically allocates its own copy of that `string` and stores a pointer to that `string` in `ps`. The copy constructor also allocates its own, separate copy of the `string`. The destructor frees the memory allocated in its constructors by executing `delete` on the pointer member, `ps`.

Valuelike Copy-Assignment Operator

Assignment operators typically combine the actions of the destructor and the copy constructor. Like the destructor, assignment destroys the left-hand operand's resources. Like the copy constructor, assignment copies data from the right-hand operand. However, it is crucially important that these actions be done in a sequence that is correct even if an object is assigned to itself. Moreover, when possible, we should also write our assignment operators so that they will leave the left-hand operand in a sensible state should an exception occur (§ 5.6.2, p. 196).

In this case, we can handle self-assignment—and make our code safe should an exception happen—by first copying the right-hand side. After the copy is made, we'll free the left-hand side and update the pointer to point to the newly allocated `string`:

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    auto newp = new string(*rhs.ps);   // copy the underlying string
    delete ps;         // free the old memory
    ps = newp;         // copy data from rhs into this object
    i = rhs.i;
    return *this;      // return this object
}
```

In this assignment operator, we quite clearly first do the work of the constructor: The initializer of `newp` is identical to the initializer of `ps` in `HasPtr`'s copy constructor. As in the destructor, we next `delete` the `string` to which `ps` currently points. What remains is to copy the pointer to the newly allocated `string` and the `int` value from `rhs` into this object.

KEY CONCEPT: ASSIGNMENT OPERATORS

There are two points to keep in mind when you write an assignment operator:

- **Assignment operators must work correctly if an object is assigned to itself.**
- **Most assignment operators share work with the destructor and copy constructor.**

A good pattern to use when you write an assignment operator is to first copy the right-hand operand into a local temporary. *After* the copy is done, it is safe to destroy the existing members of the left-hand operand. Once the left-hand operand is destroyed, copy the data from the temporary into the members of the left-hand operand.

To illustrate the importance of guarding against self-assignment, consider what would happen if we wrote the assignment operator as

`=`赋值operator写法：要确保self- assignment 安全： 先建立一个等式右边的副本；再把原有左边的地址删了；再把**副本**的元素给左边的元素。

```
// WRONG way to write an assignment operator!
HasPtr&
HasPtr::operator=(const HasPtr &rhs)
{
    delete ps;  // frees the string to which this object points
    // if rhs and *this are the same object, we're copying from deleted memory!
    ps = new string(*(rhs.ps));
    i = rhs.i;
    return *this;
}
```

If `rhs` and this object are the same object, deleting `ps` frees the `string` to which both `*this` and `rhs` point. When we attempt to copy `*(rhs.ps)` in the `new` expression, that pointer points to invalid memory. What happens is undefined.

It is crucially important for assignment operators to work correctly, even when an object is assigned to itself. A good way to do so is to copy the right-hand operand before destroying the left-hand operand.

Exercises Section 13.2.1

Exercise 13.23: Compare the copy-control members that you wrote for the solutions to the previous section's exercises to the code presented here. Be sure you understand the differences, if any, between your code and ours.

Exercise 13.24: What would happen if the version of `HasPtr` in this section didn't define a destructor? What if `HasPtr` didn't define the copy constructor?

Exercise 13.25: Assume we want to define a version of `StrBlob` that acts like a value. Also assume that we want to continue to use a `shared_ptr` so that our `StrBlobPtr` class can still use a `weak_ptr` to the `vector`. Your revised class will need a copy constructor and copy-assignment operator but will not need a destructor. Explain what the copy constructor and copy-assignment operators must do. Explain why the class does not need a destructor.

Exercise 13.26: Write your own version of the `StrBlob` class described in the previous exercise.

13.2.2 Defining Classes That Act Like Pointers

For our `HasPtr` class to act like a pointer, we need the copy constructor and copy-assignment operator to copy the pointer member, not the `string` to which that pointer points. Our class will still need its own destructor to free the memory allocated by the constructor that takes a `string` (§ 13.1.4, p. 504). In this case, though, the destructor cannot unilaterally free its associated `string`. It can do so only when the last `HasPtr` pointing to that `string` goes away.

The easiest way to make a class act like a pointer is to use `shared_ptrs` to manage the resources in the class. Copying (or assigning) a `shared_ptr` copies