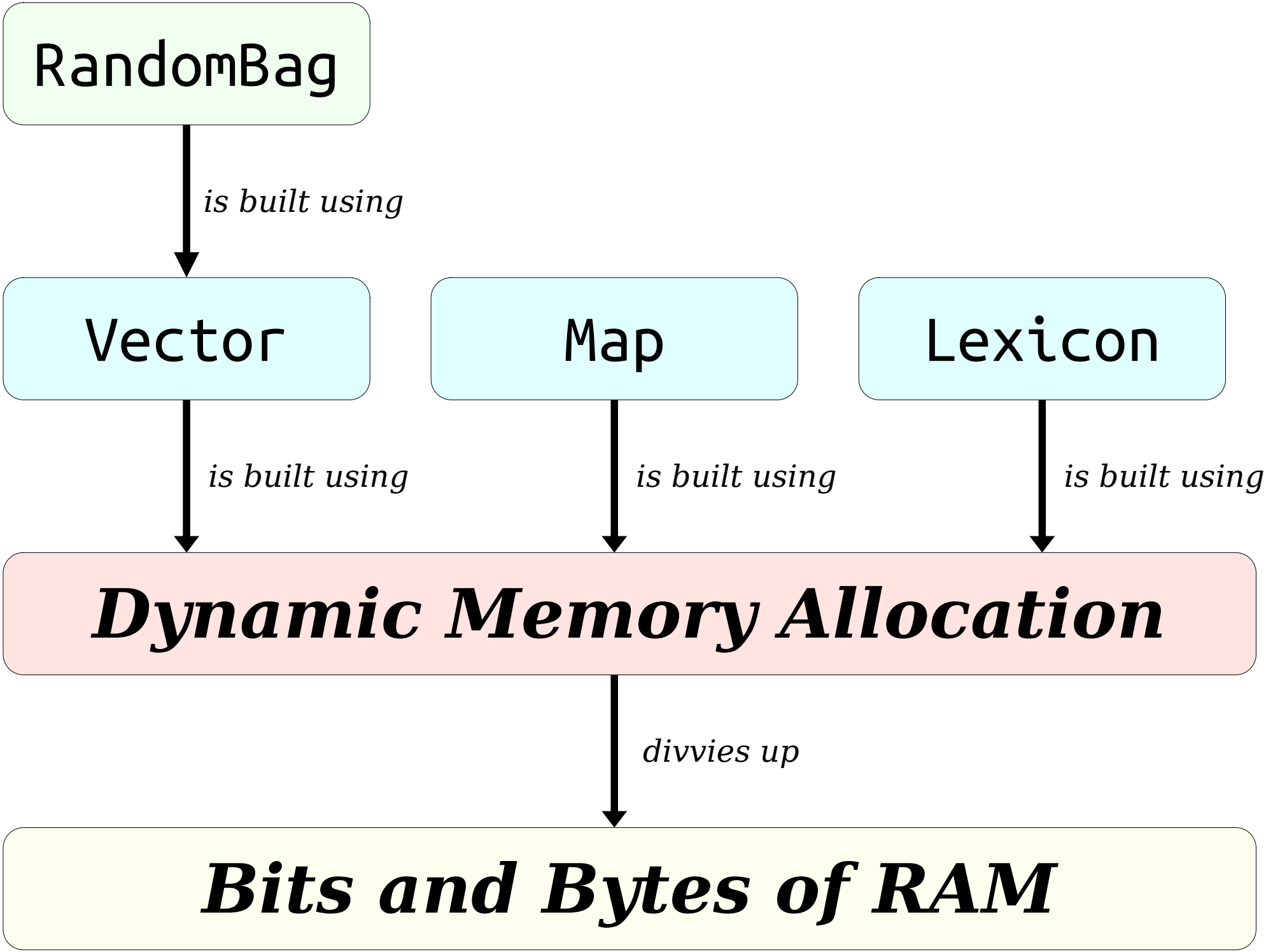
Implementing Abstractions

Part One

Previously on CS106B...

```
class RandomBag {
public:
    void add(int value);
    int  removeRandom();

    int  size() const;
    bool isEmpty() const;

private:
    Vector<int> elems;
};
```

```
class RandomBag {
public:
    void add(int value);
    int  removeRandom();

    int  size() const;
    bool isEmpty() const;

private:
    Vector<int> elems;
};
```


RandomBag
is built using
Vector
Map
Lexicon
is built using
is built using
is built using
Dynamic Memory Allocation
divvies up
Bits and Bytes of RAM

Dynamic Memory Allocation

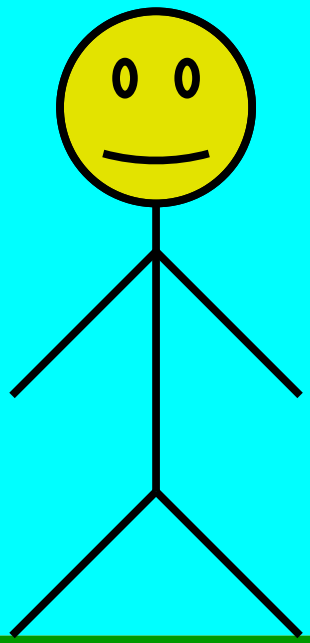
- Types like `Vector`, `Map`, `Set`, etc. that store a variable number of items need space to store those elements.
- When you use those types as a client, they just "work" and somehow figure out where to store things. You as the end user don't see how.
- Internally, those types use a technique called ***dynamic memory allocation*** to get space in RAM where they can put their elements.
- How they do this – and how you can do this in your own code – is our next major topic.

A Change in Perspective

- ***Key Question From Before:*** How do we use the `Map`, `Vector`, etc. to model and solve complex problems?
- ***Key Question For Now:*** How can we use the simple tools afforded by C++ to build things like `Map`, `Vector`, etc.?
- The coding techniques that go into this will subjectively feel very different than what we've seen so far.
  - There will be fewer tools available to you.
  - Those tools require different mental models than what you're used to.
- And yet, by learning how to use them:
  - You'll learn more about how the computer actually works.
  - You'll see how to build complex systems out of simple parts.
  - You'll get an appreciation for just how clever the techniques that power the `Map`, `Set`, and `Vector` are.

Dynamic Allocation: The Basics

```
string* ptr;
```



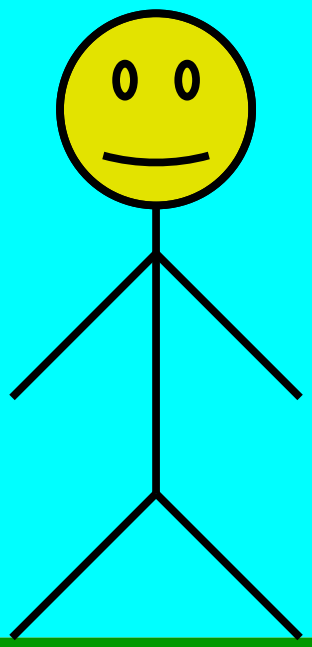
ptr

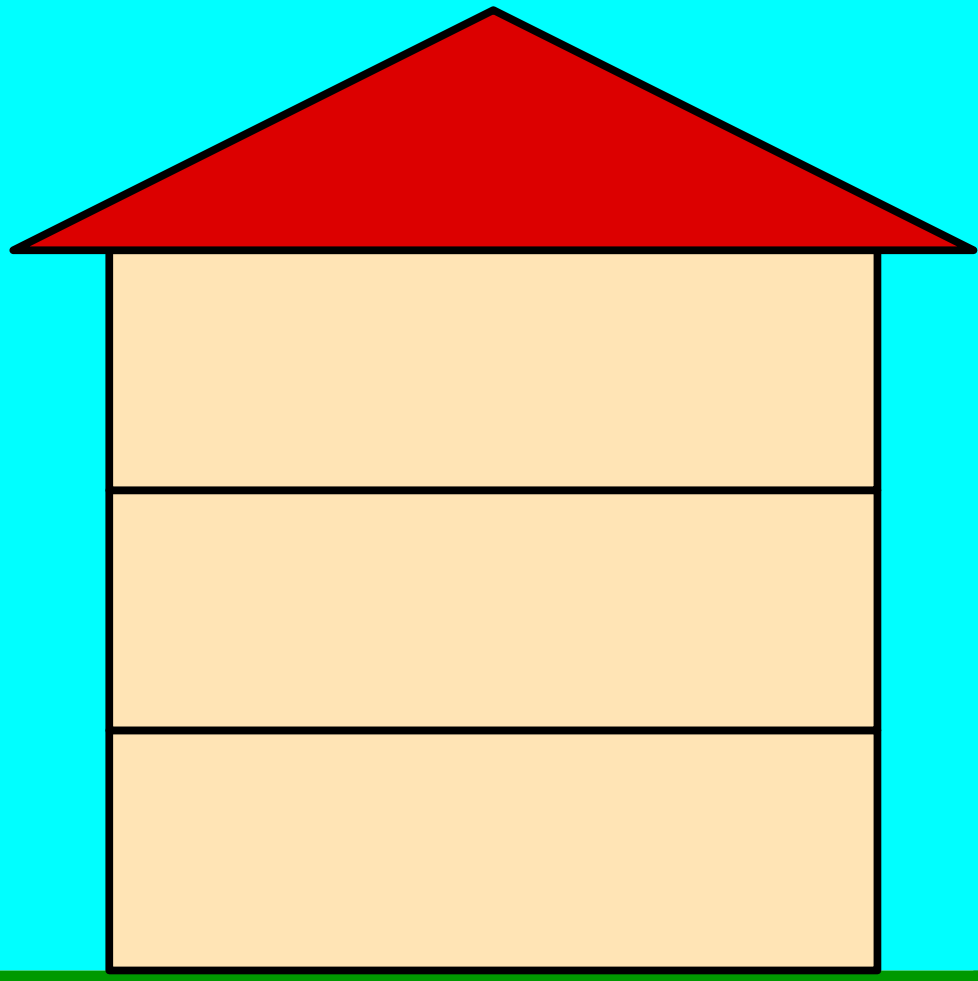
The variable `ptr` has type

`string*`

rather than `string`. We'll explain this in a moment.

```
string* ptr;
ptr = new string[3];
```

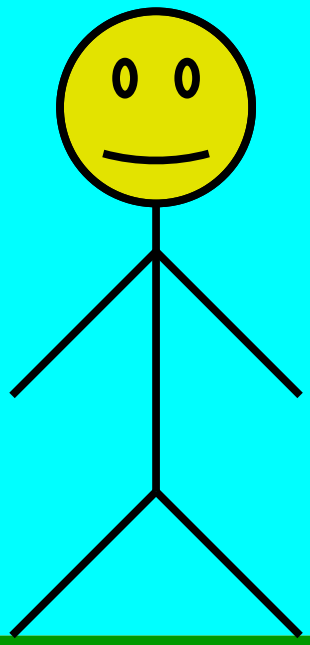


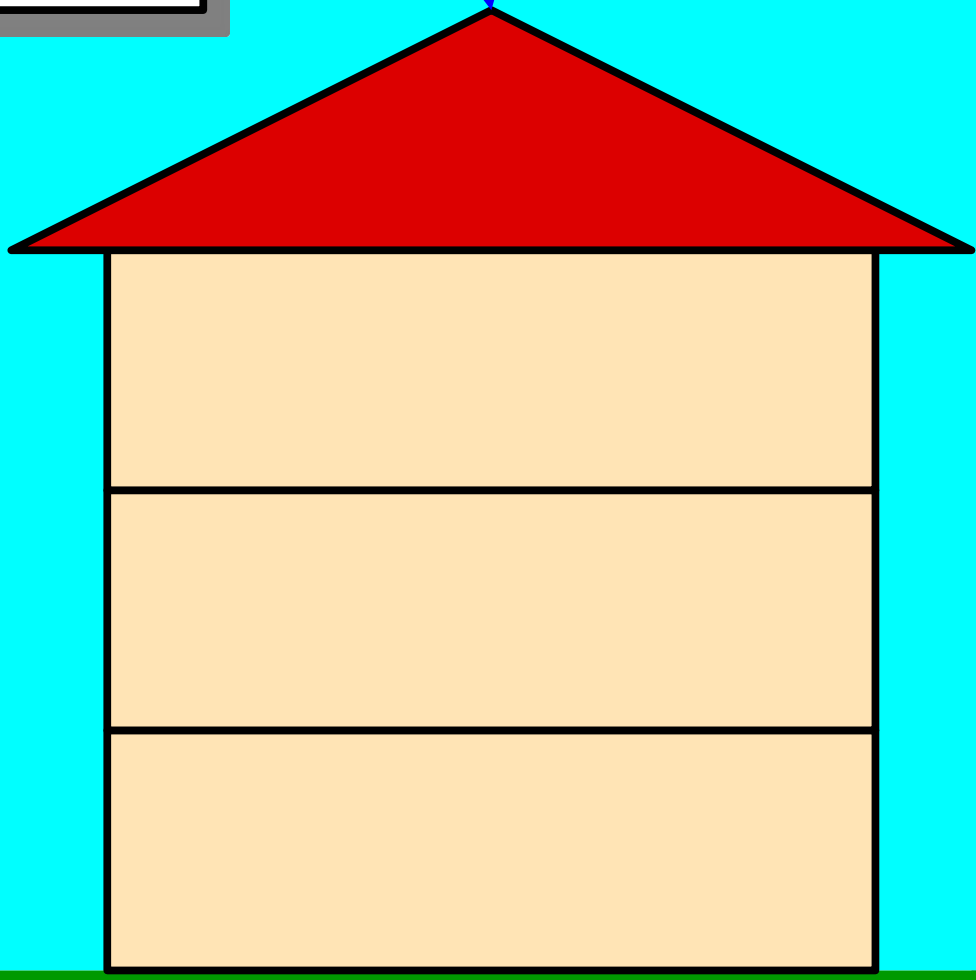


ptr

```
string* ptr;
ptr = new string[3];
```

This is an array of three strings. I'll represent it as a three-story building.

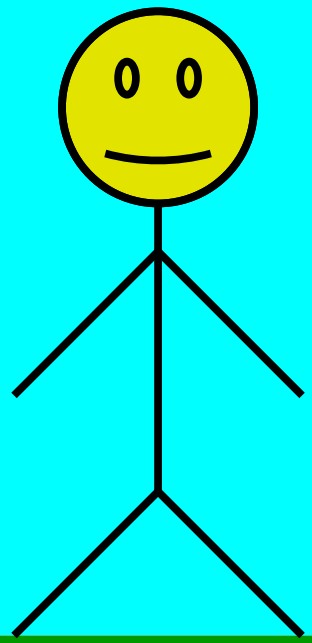


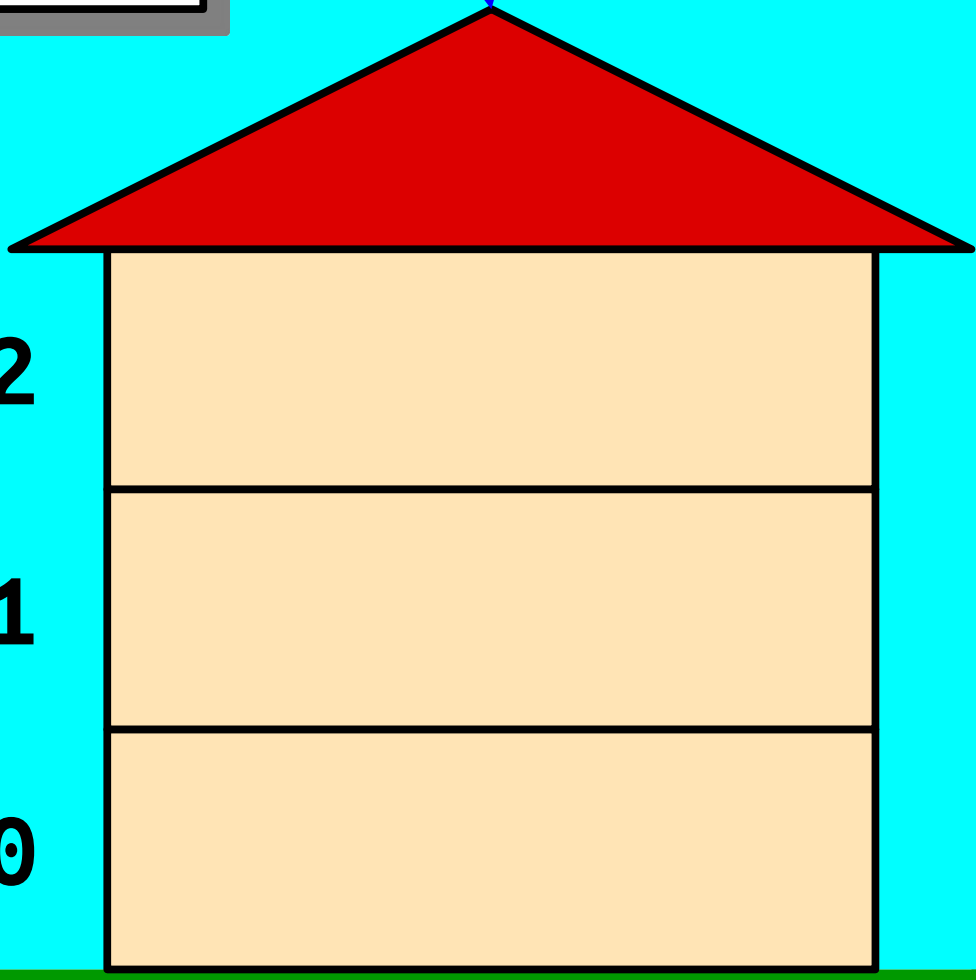


ptr

```
string* ptr;
ptr = new string[3];
```

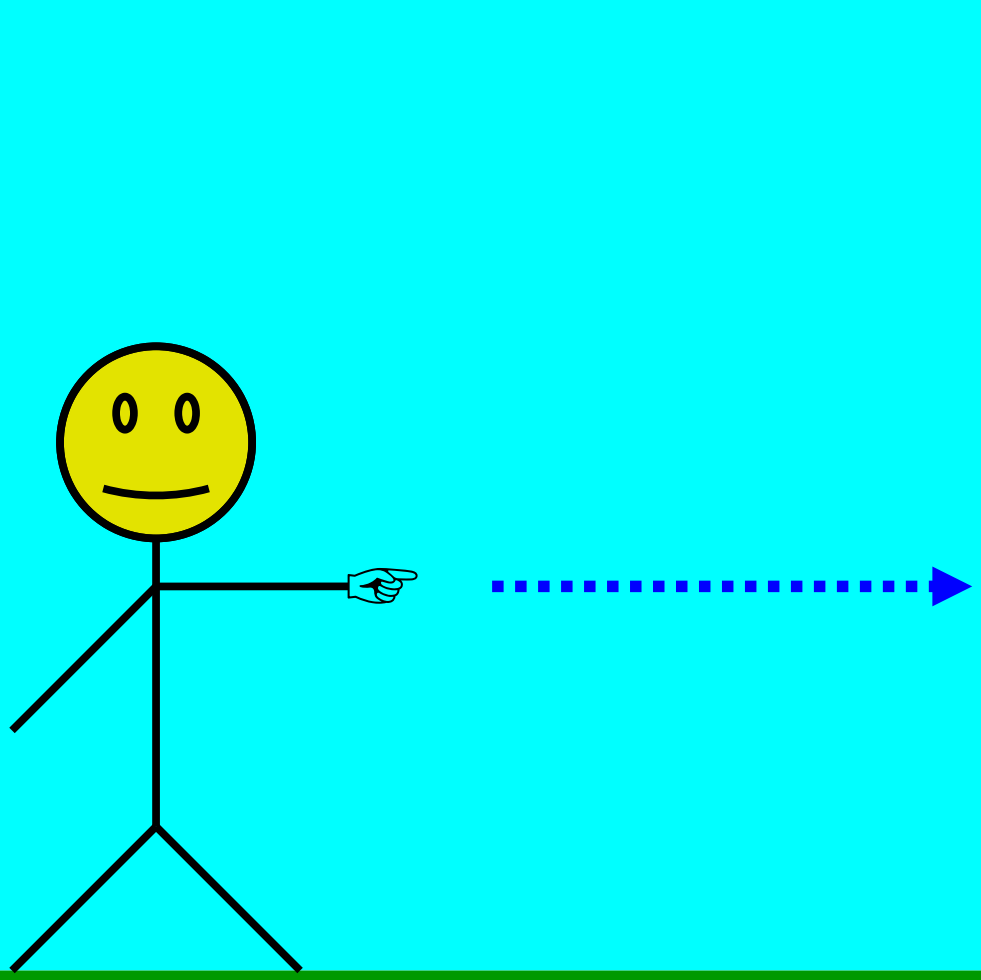
This is an array of three strings. I'll represent it as a three-story building.

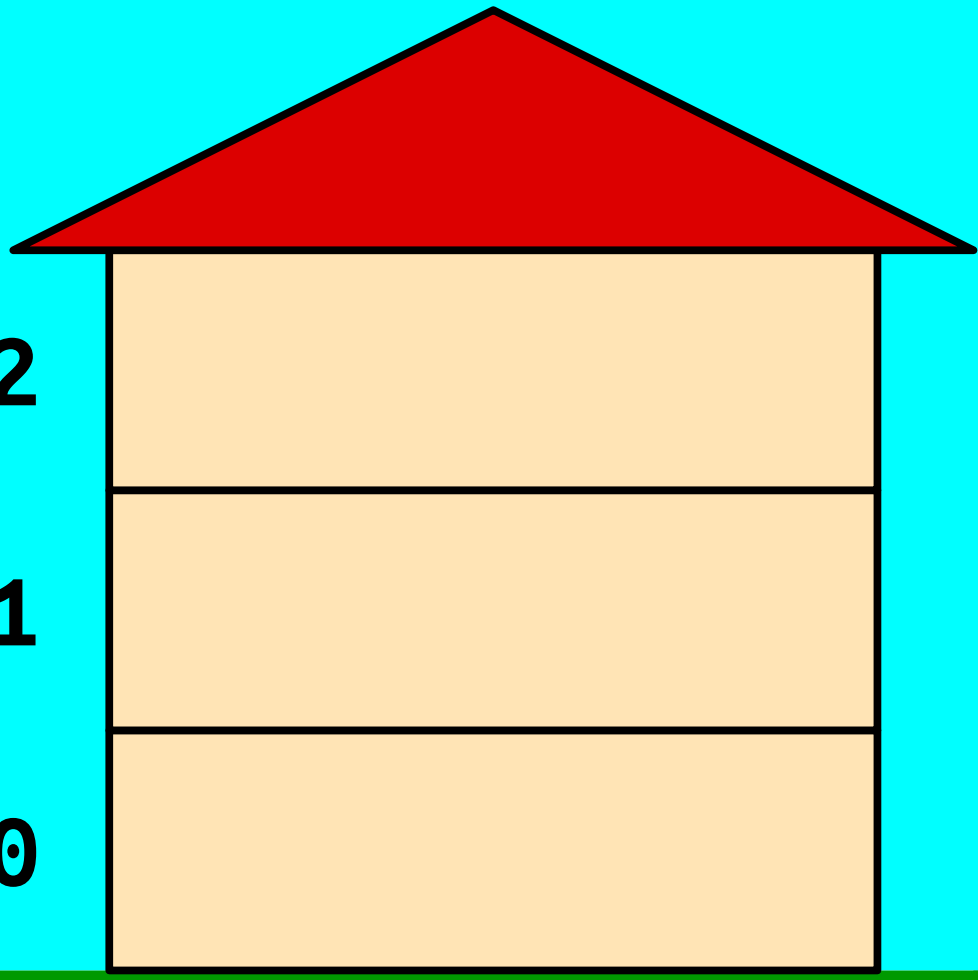




ptr

```
string* ptr;
ptr = new string[3];
```



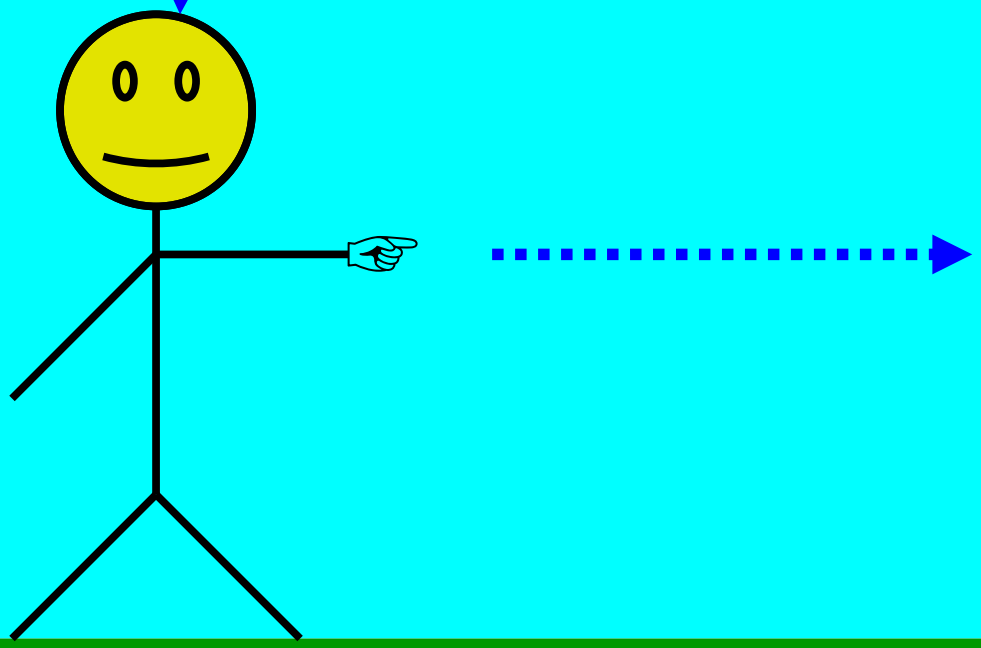


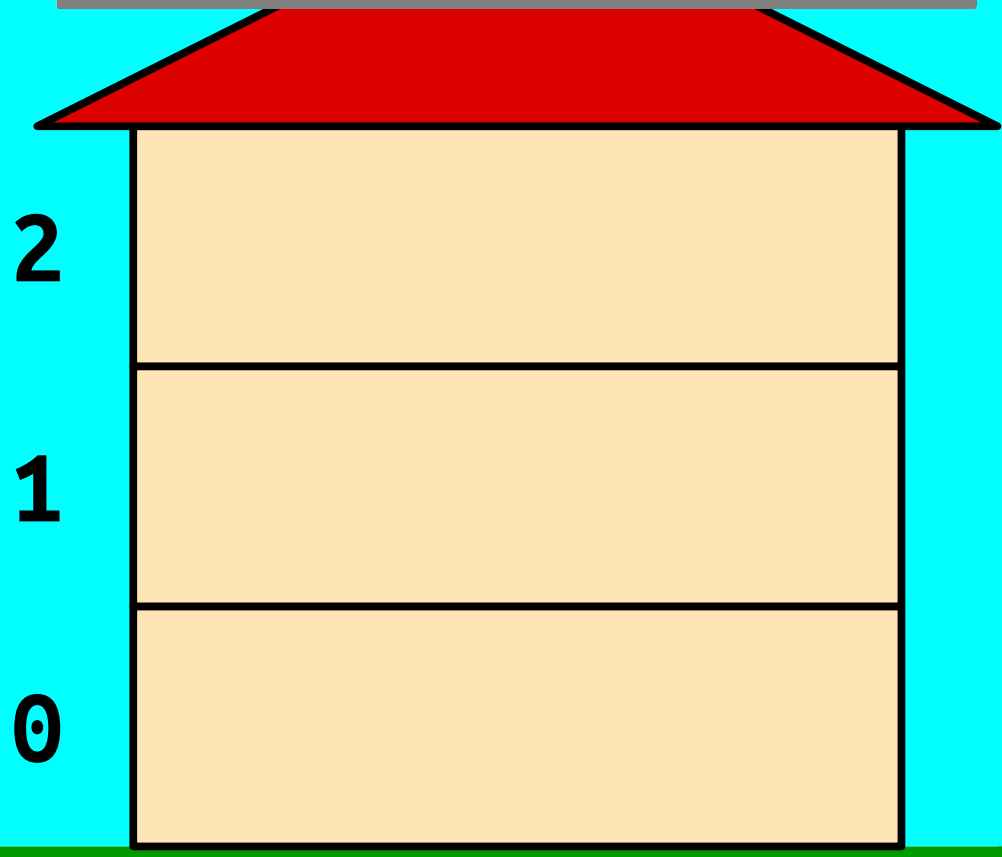
ptr

```
string* ptr;
ptr = new string[3];
```

The variable `ptr` ***points*** to an array of strings. It's therefore called a ***pointer***.

Since `ptr` points to an array of strings, we give it the type `string*`.



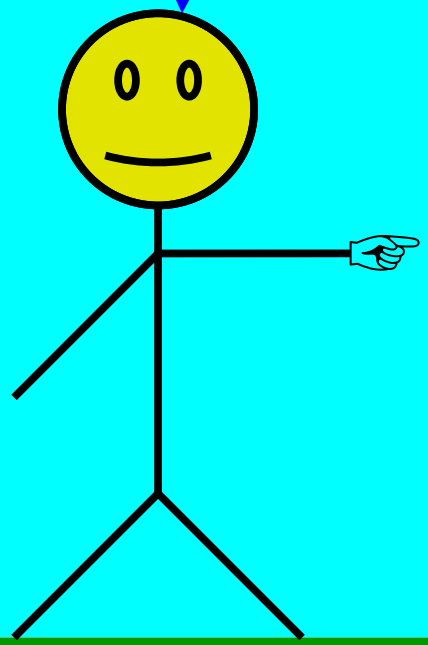


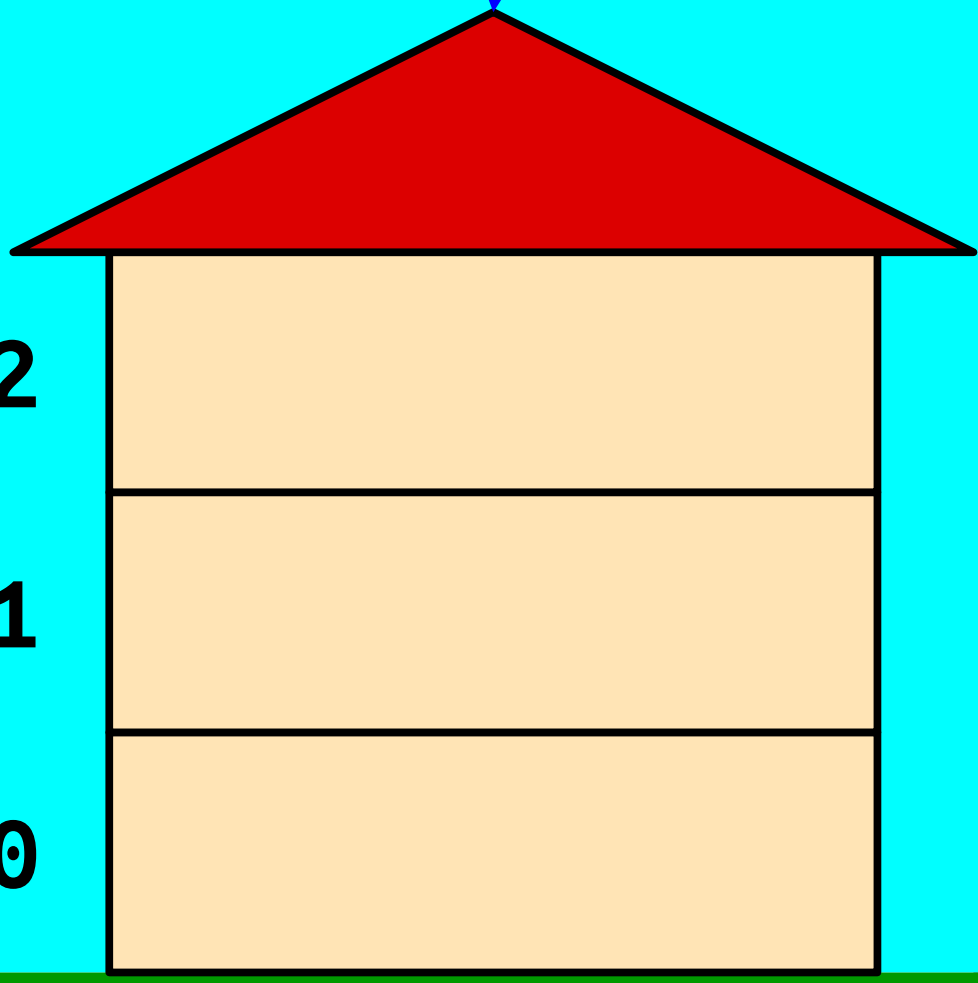
`ptr`

```
string* ptr;
ptr = new string[3];
```

The variable `ptr` just points to where the array is.

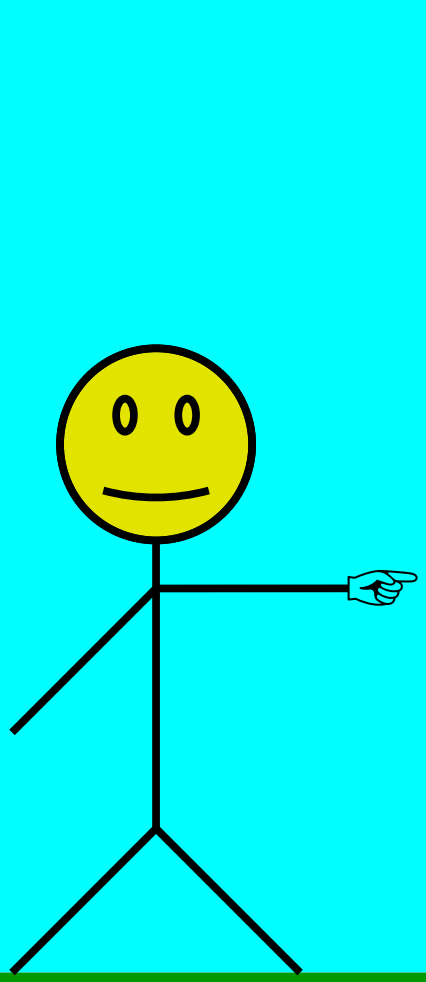
The array itself is this building over here.

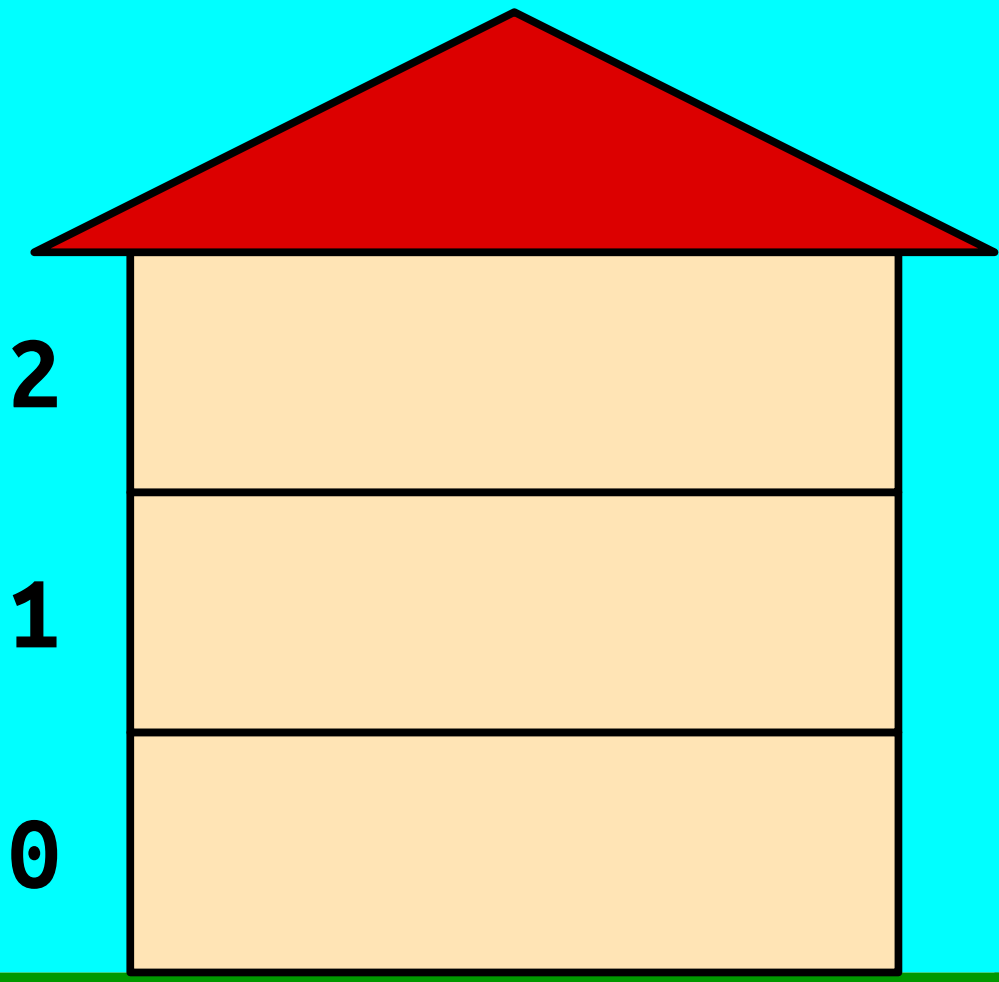




ptr

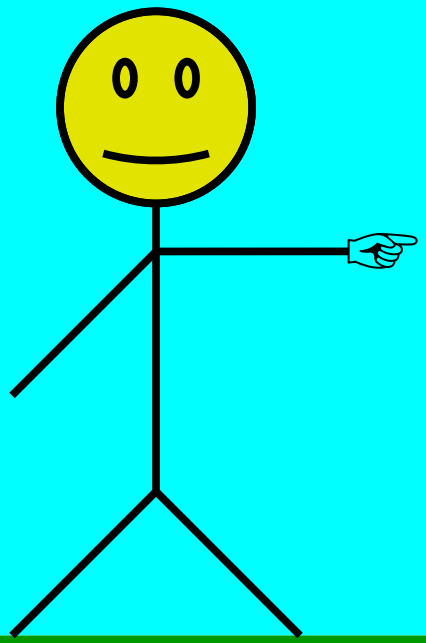
```
string* ptr;
ptr = new string[3];
ptr[0] = "Coffee Shop";
```



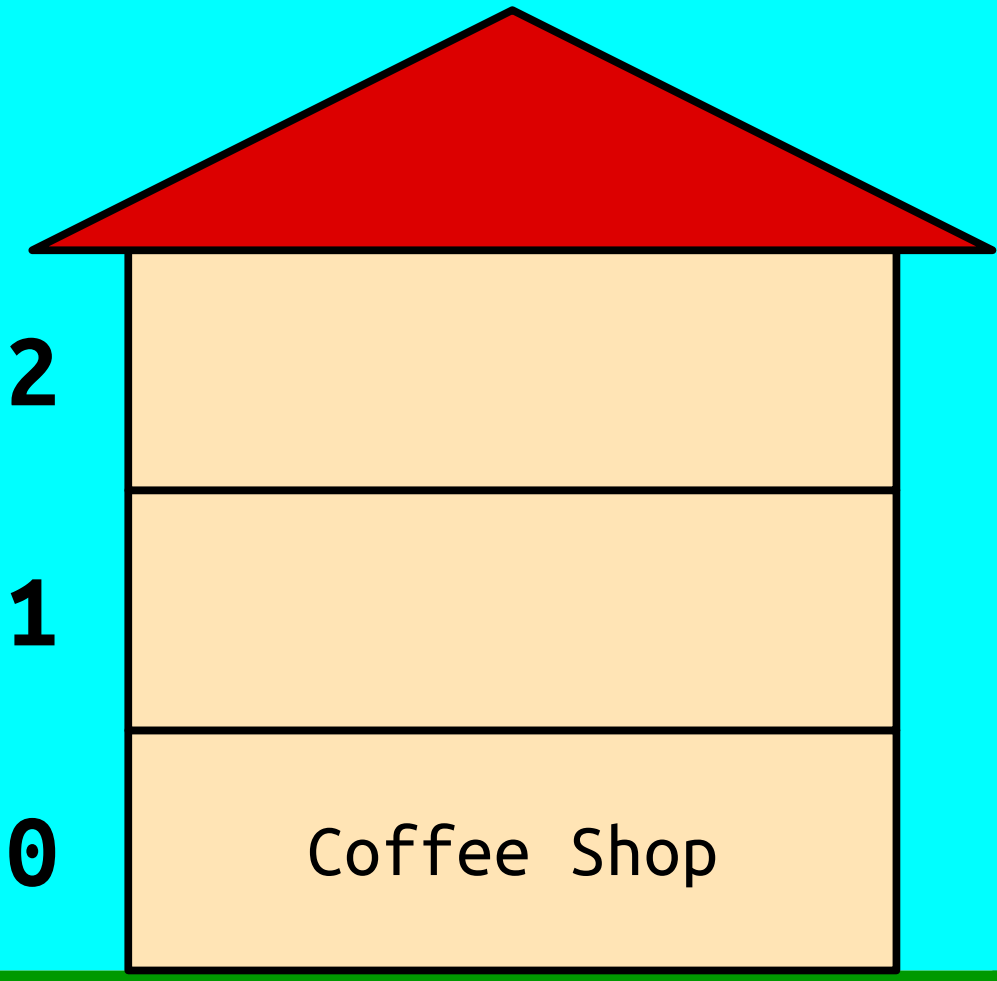


ptr

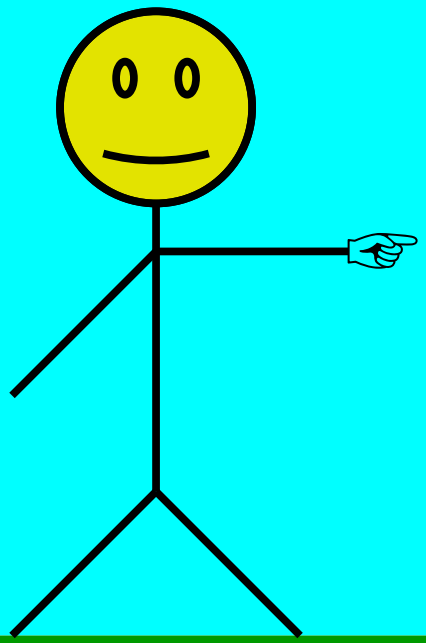
```
string* ptr;
ptr = new string[3];
ptr[0] = "Coffee Shop";
```



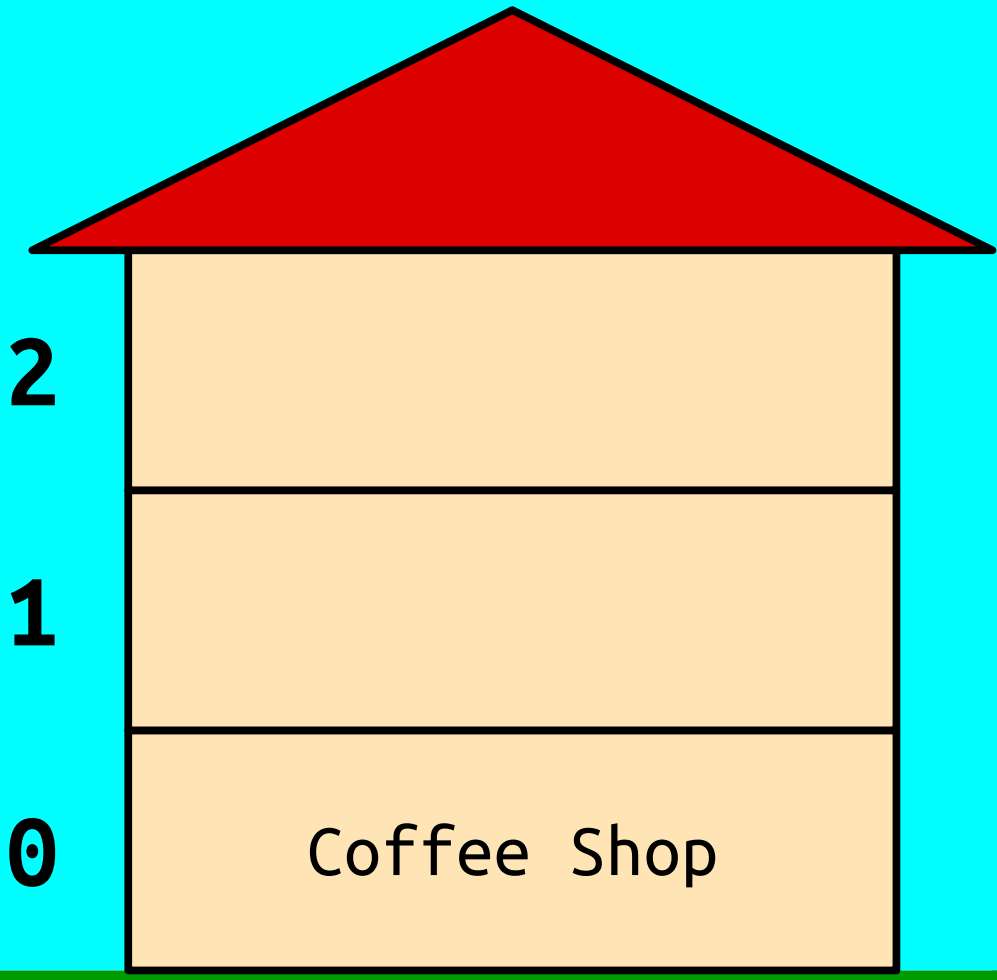
ptr



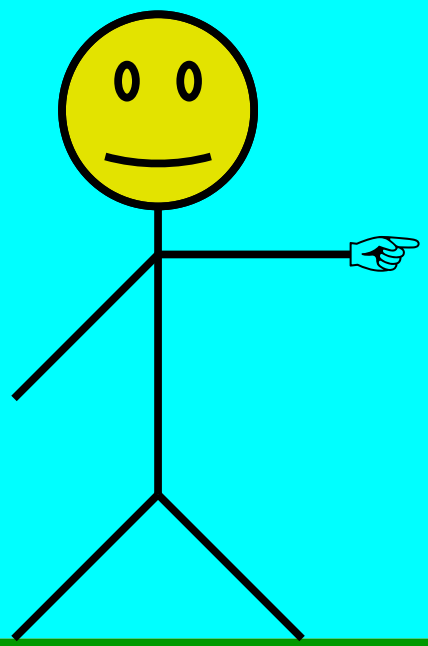
```
string* ptr;
ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
```



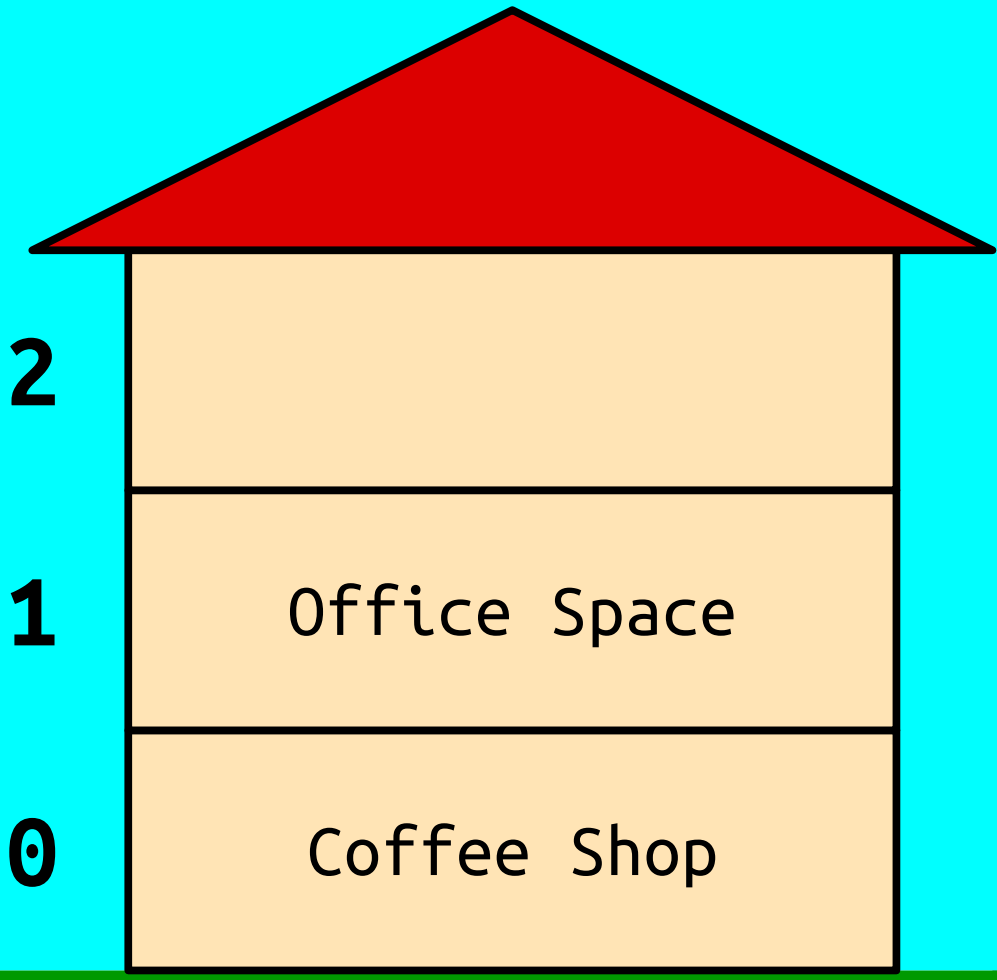
ptr



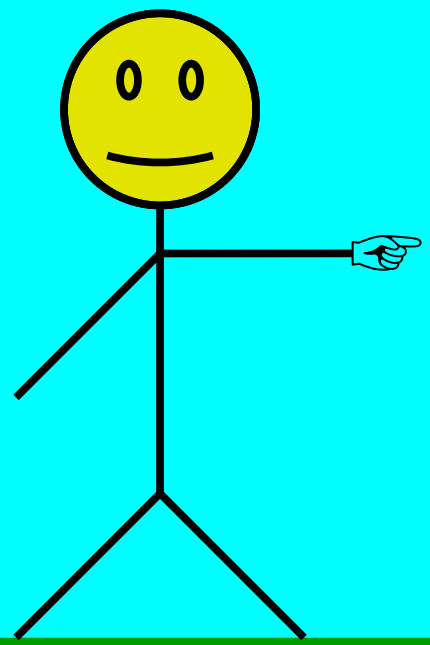
```
string* ptr;
ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
```



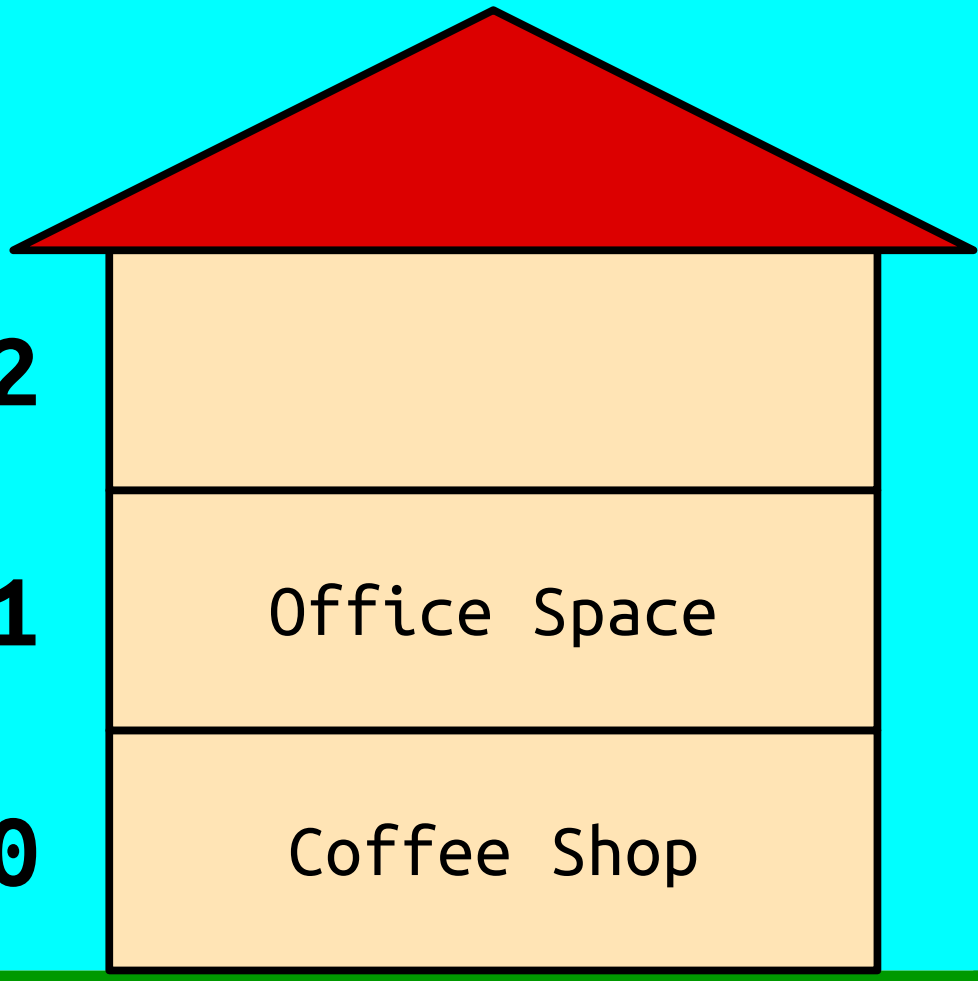
ptr



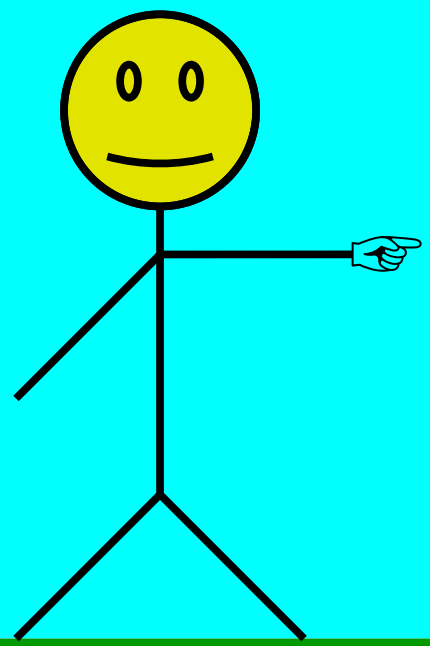
```
string* ptr;
ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";
```



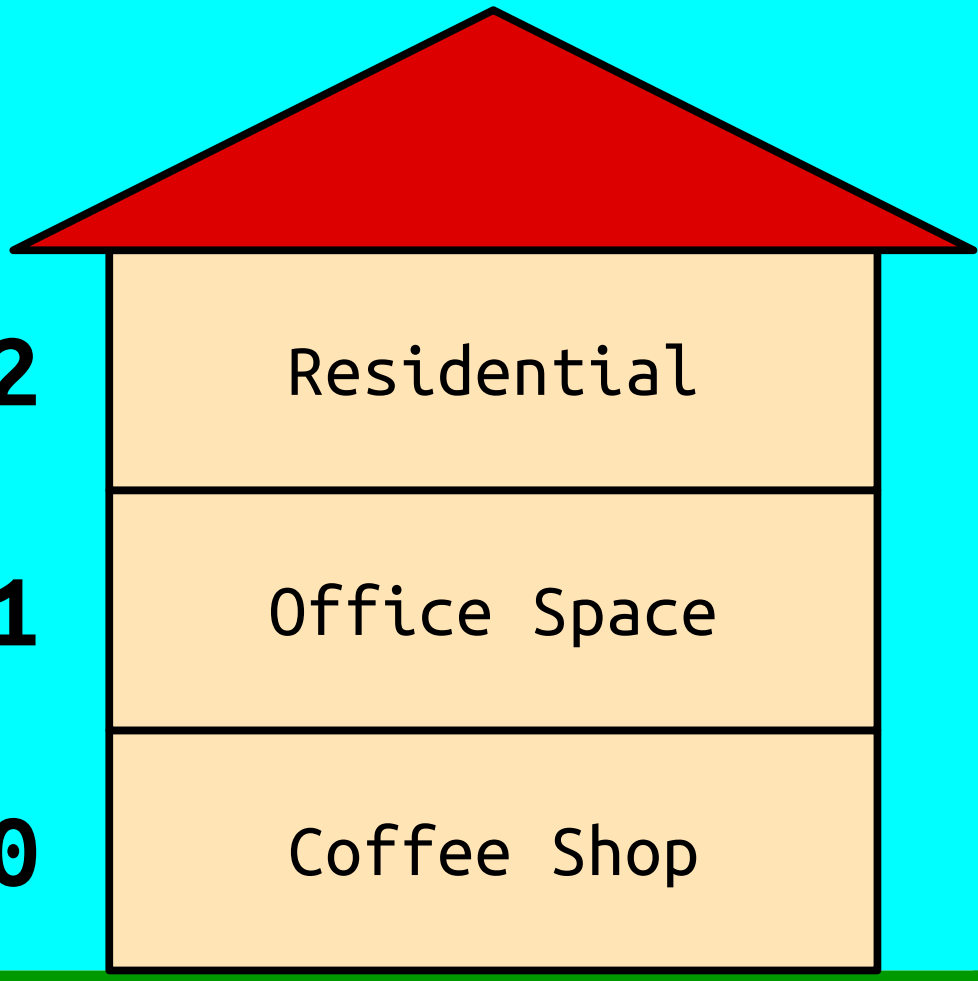
ptr



```
string* ptr;
ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";
```

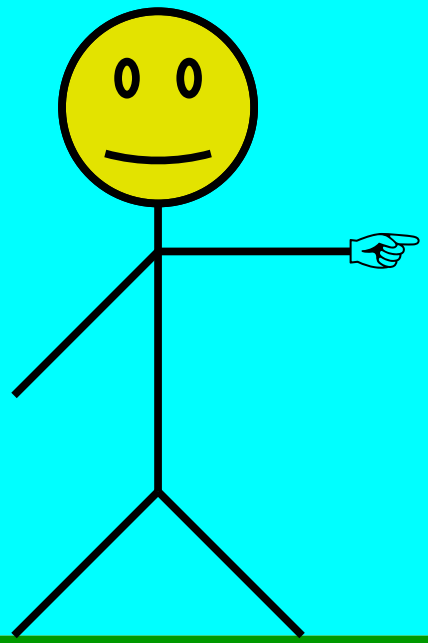


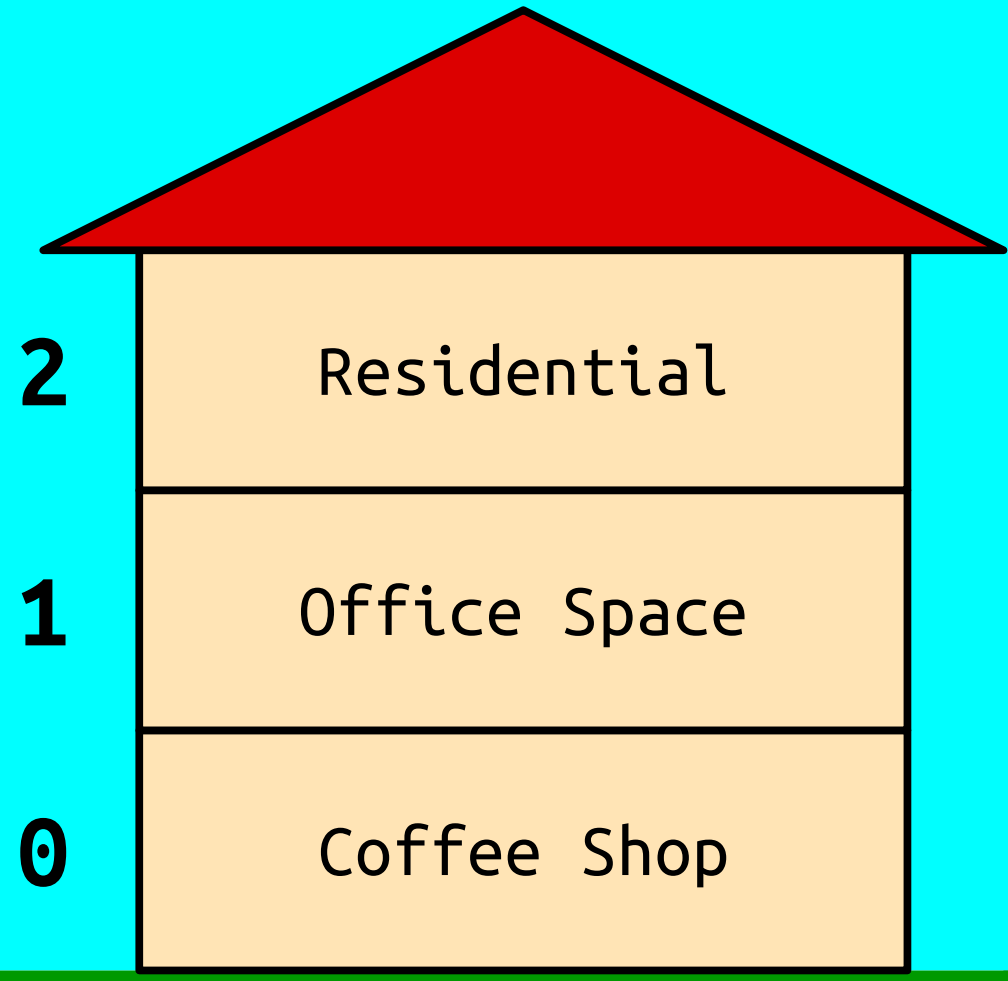
ptr



```
string* ptr;
ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";
```

This building will always be exactly three stories tall. You cannot add or remove floors. (“Conservation of mass.”)

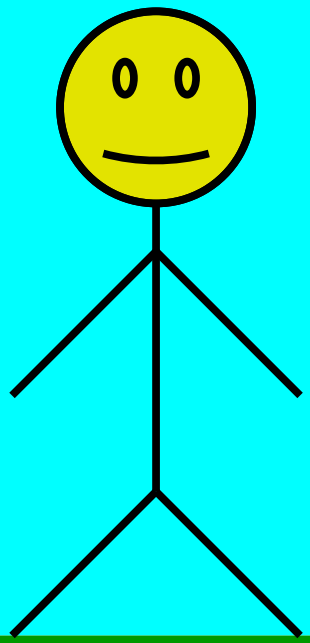


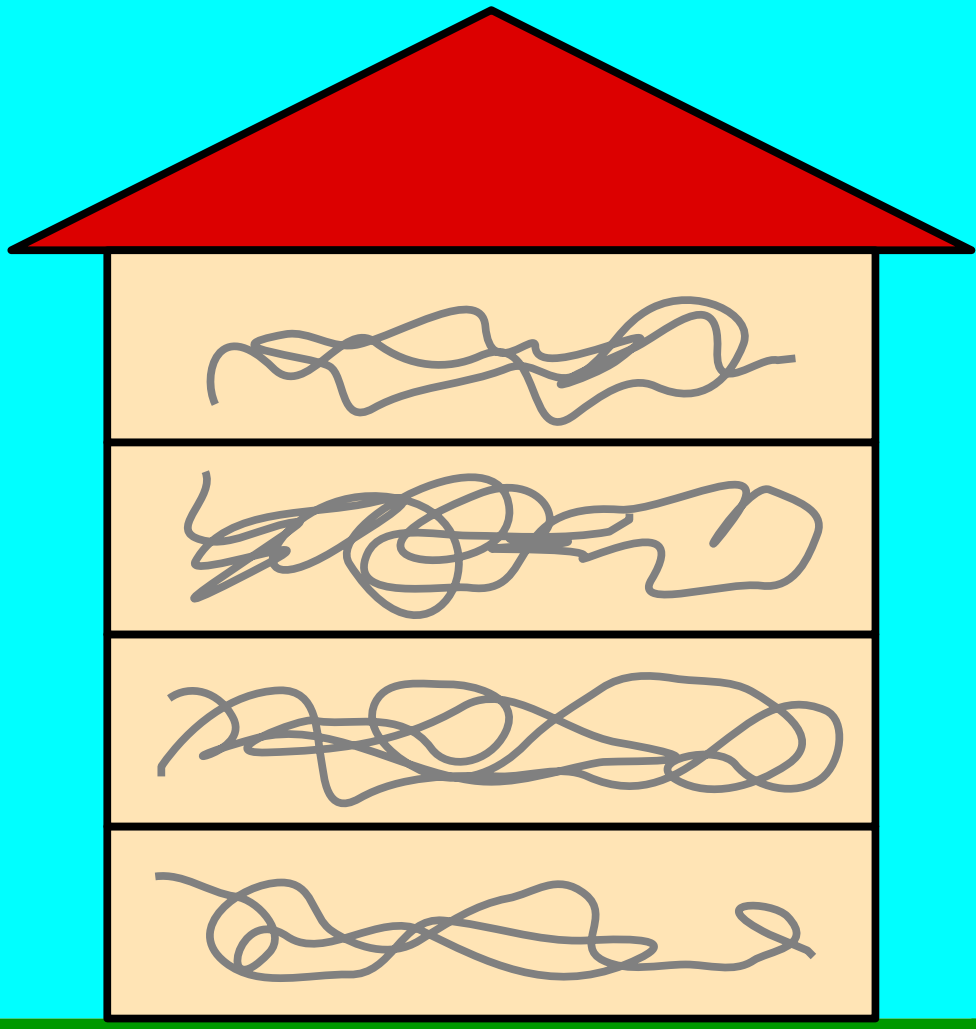


ptr

```
int* ptr = new int[4];
```

Arrays of `int`s, `double`s, `char`s, or `bool`s initially have garbage values. Other types use good defaults.

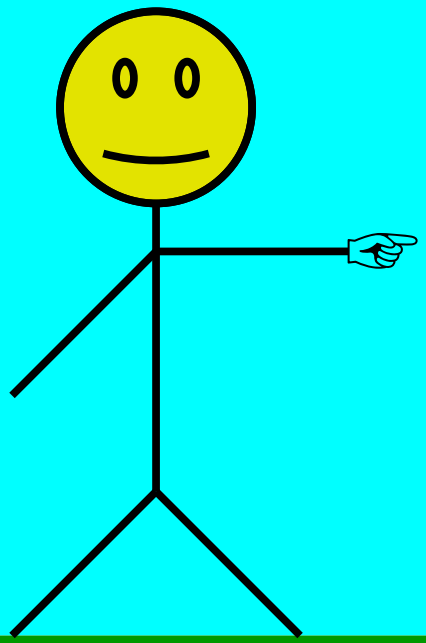


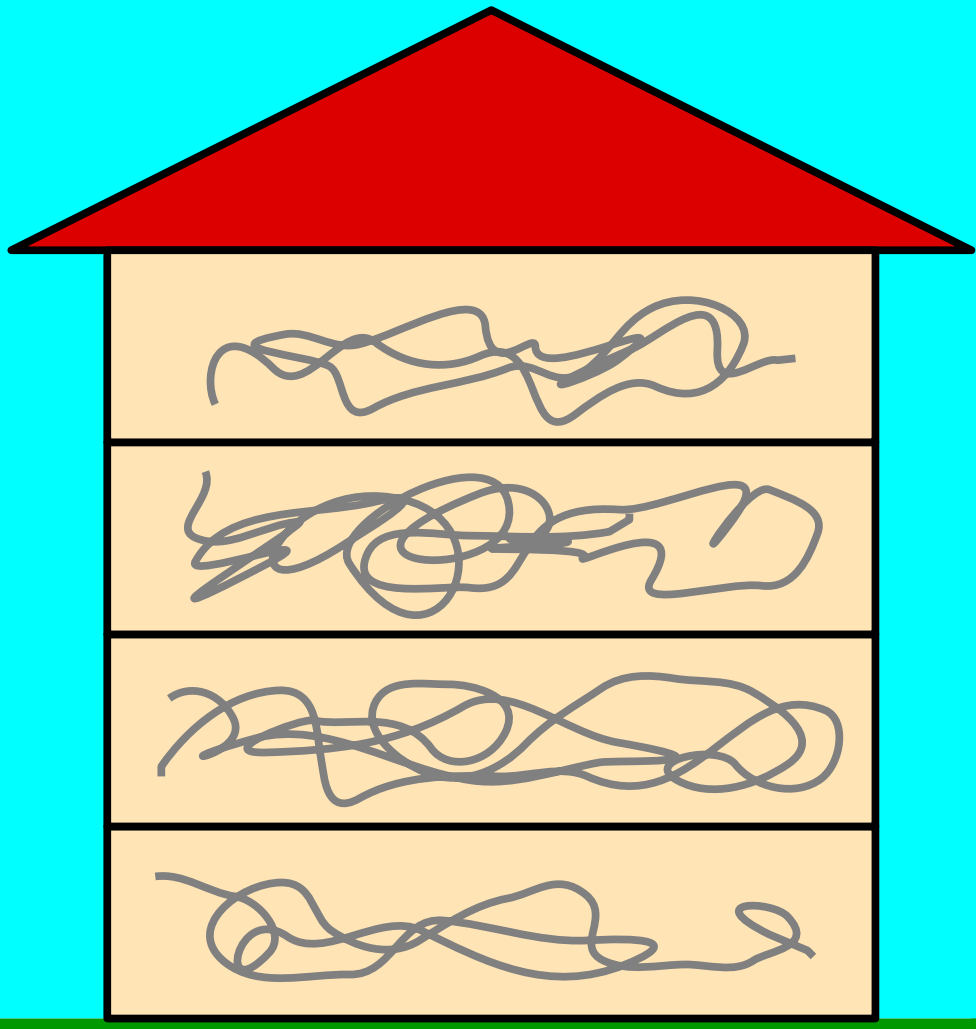


`ptr`

```
int* ptr = new int[4];
```

Arrays of `int`s, `double`s, `char`s, or `bool`s initially have garbage values. Other types use good defaults.

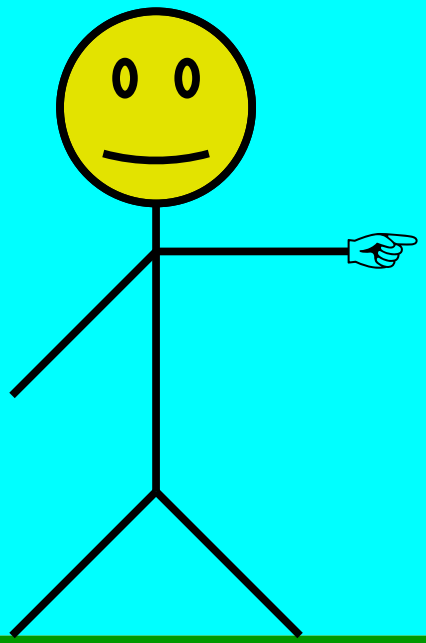


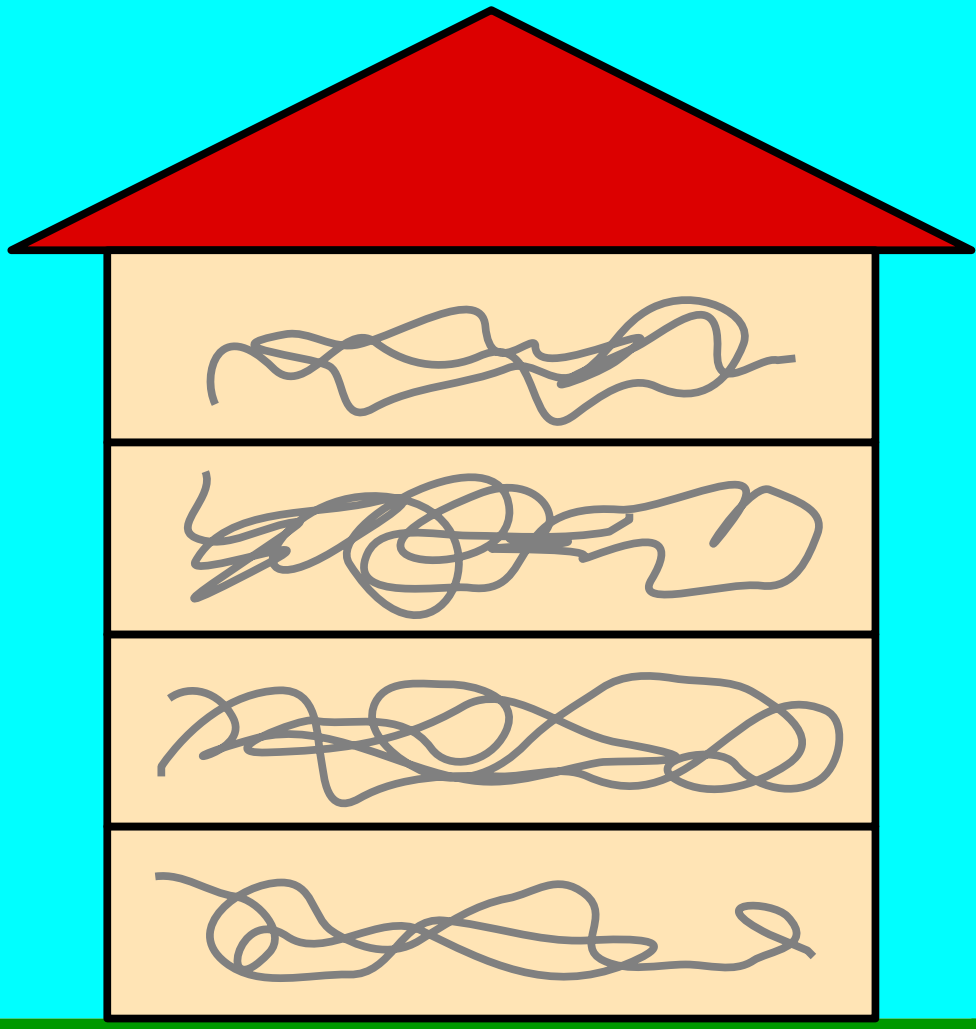


`ptr`

```
int* ptr = new int[4];
cout << ptr[0] << endl;
```

Arrays of `int`s, `double`s, `char`s, or `bool`s initially have garbage values. Other types use good defaults.



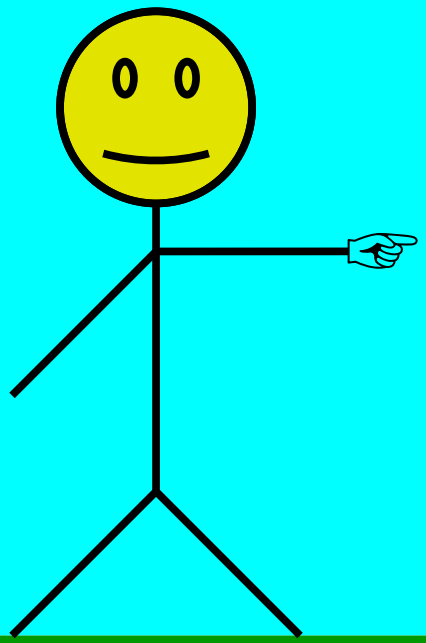


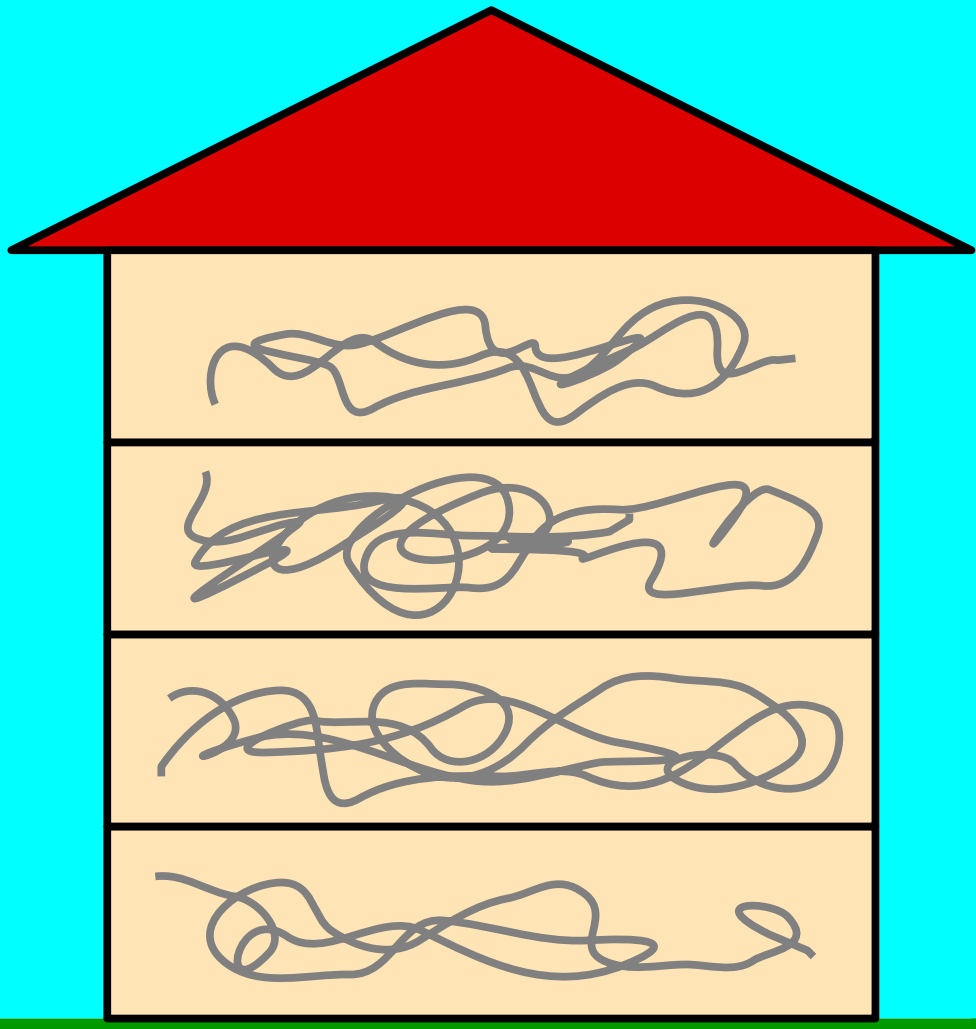
ptr

```
int* ptr = new int[4];
cout << ptr[0] << endl;
```

This might print different numbers from run to run of the same program.

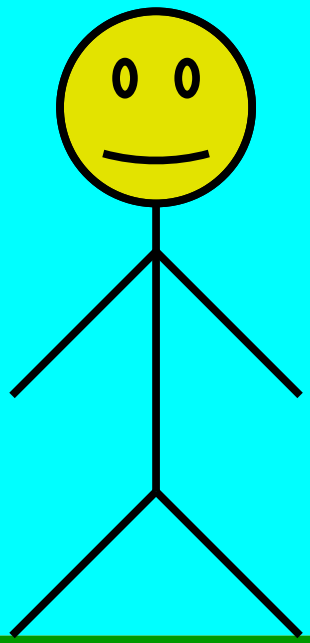
Arrays of `int`s, `double`s, `char`s, or `bool`s initially have garbage values. Other types use good defaults.

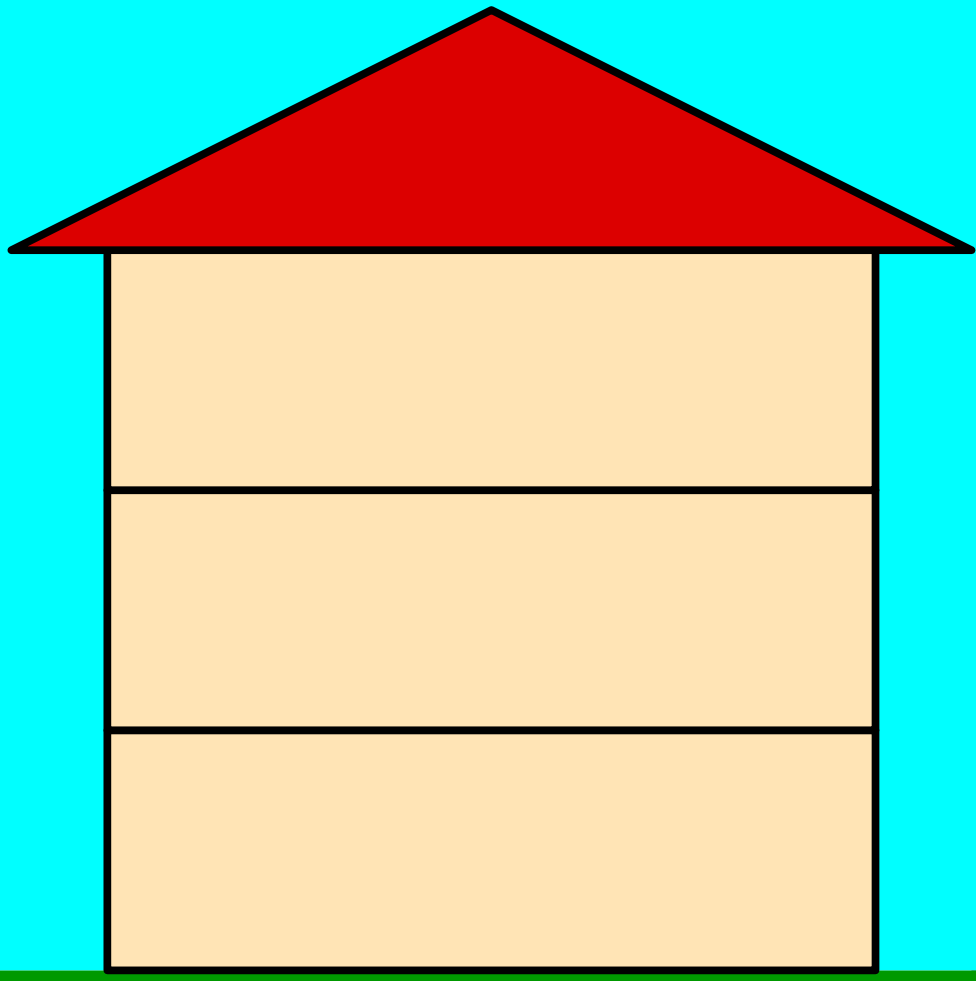




`ptr`

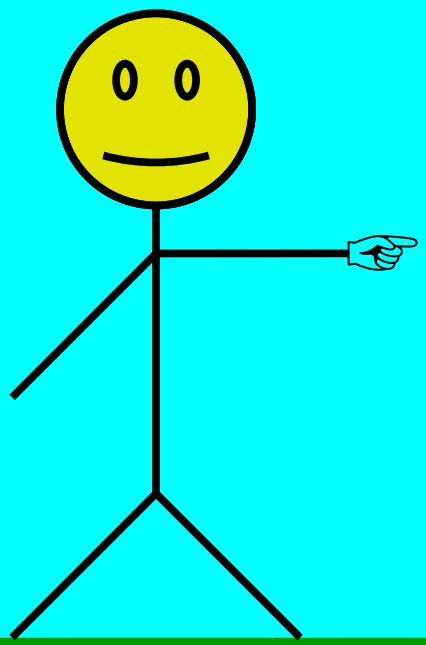
```
string* ptr = new string[3];
```

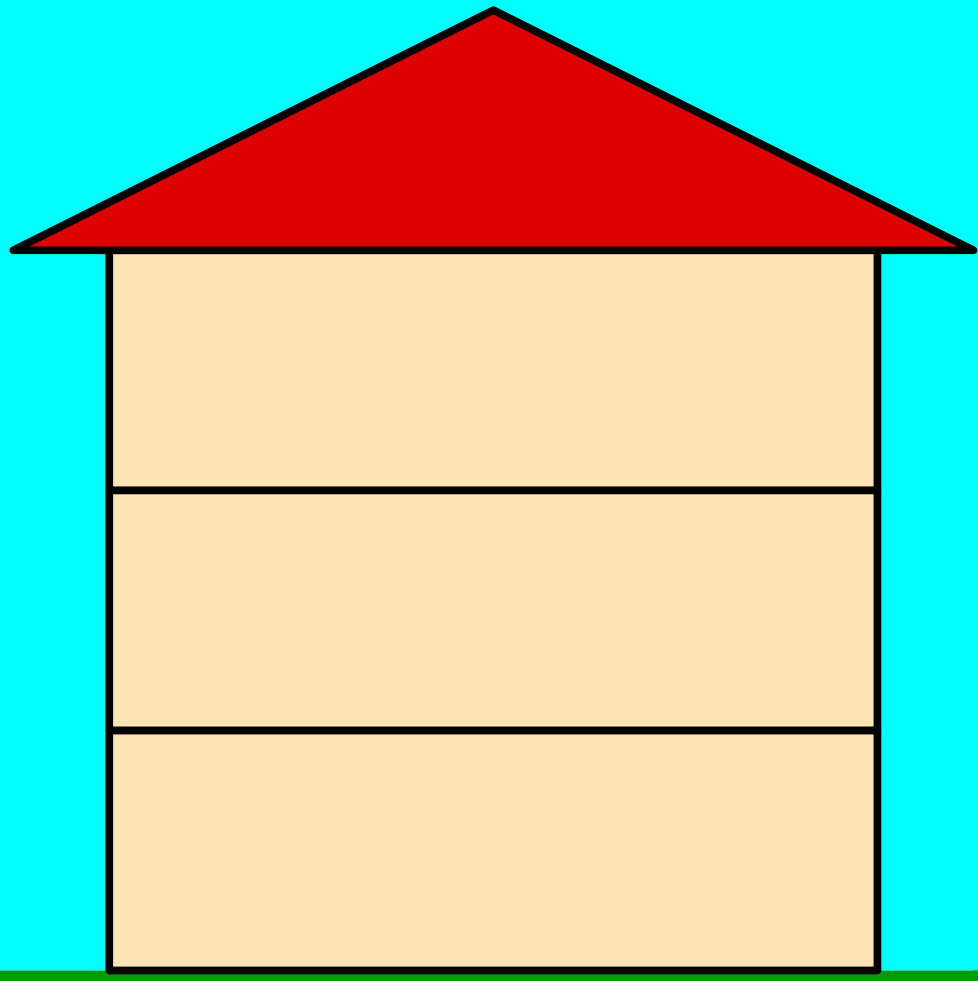




ptr

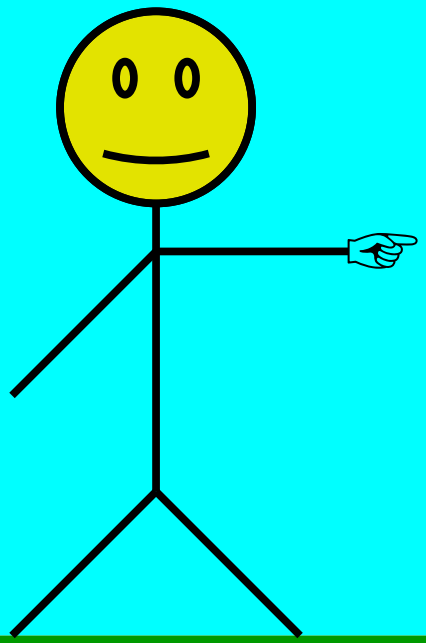
```
string* ptr = new string[3];
```

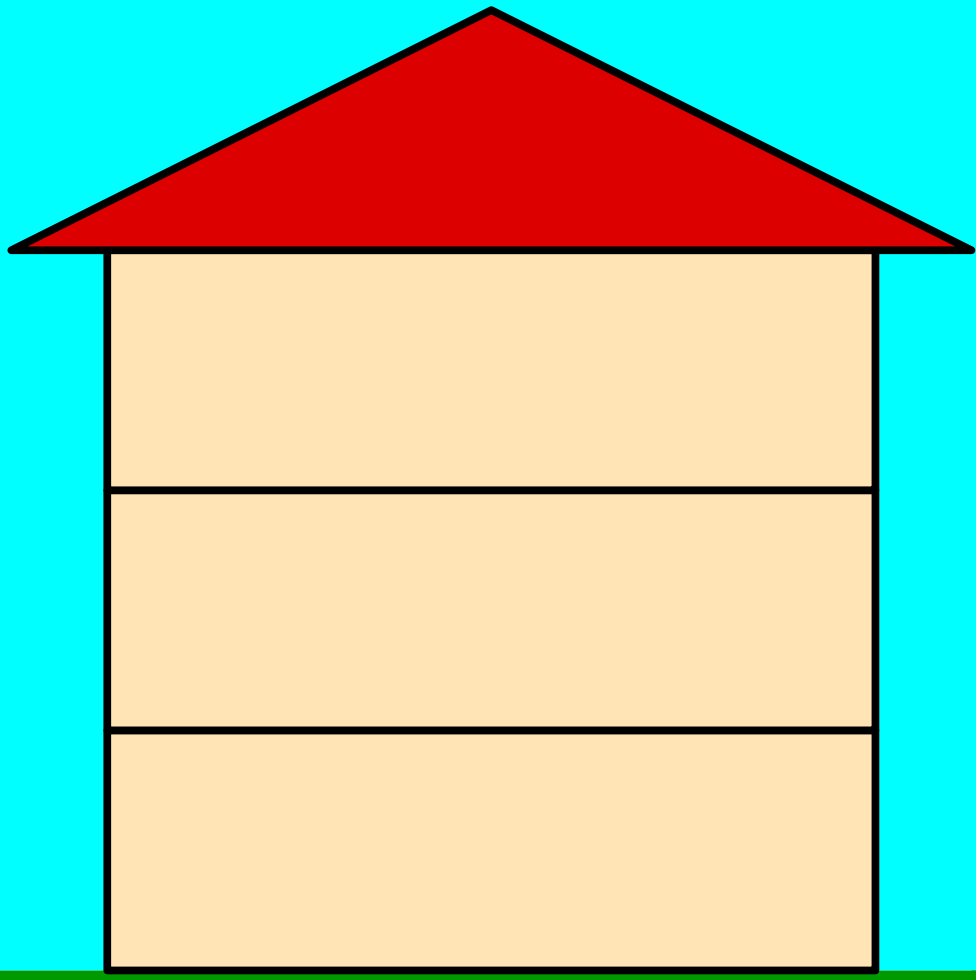




ptr

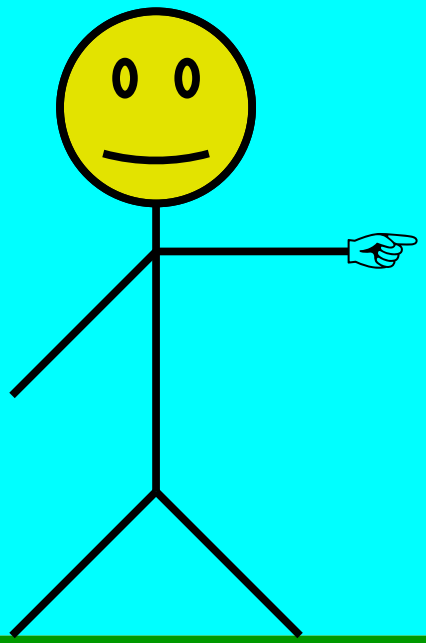
```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";
```

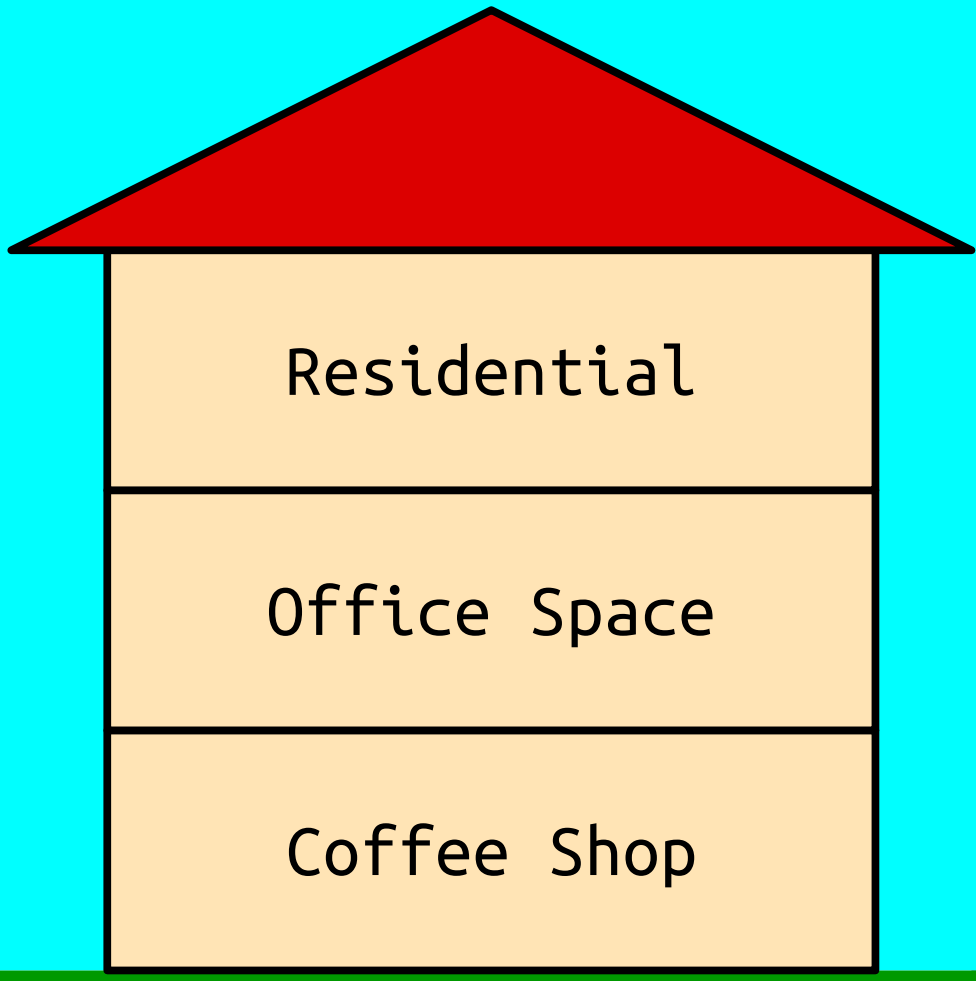




ptr

```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";
```

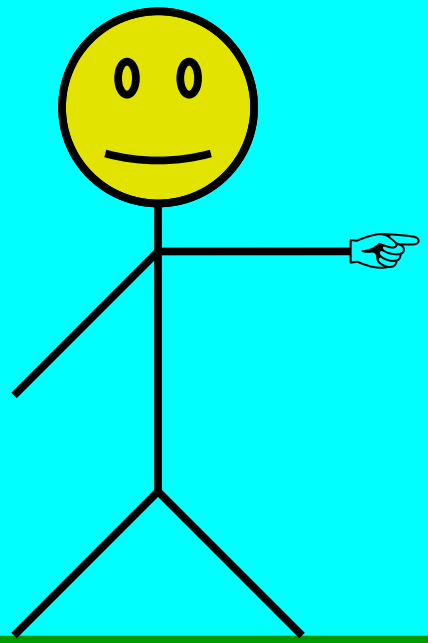


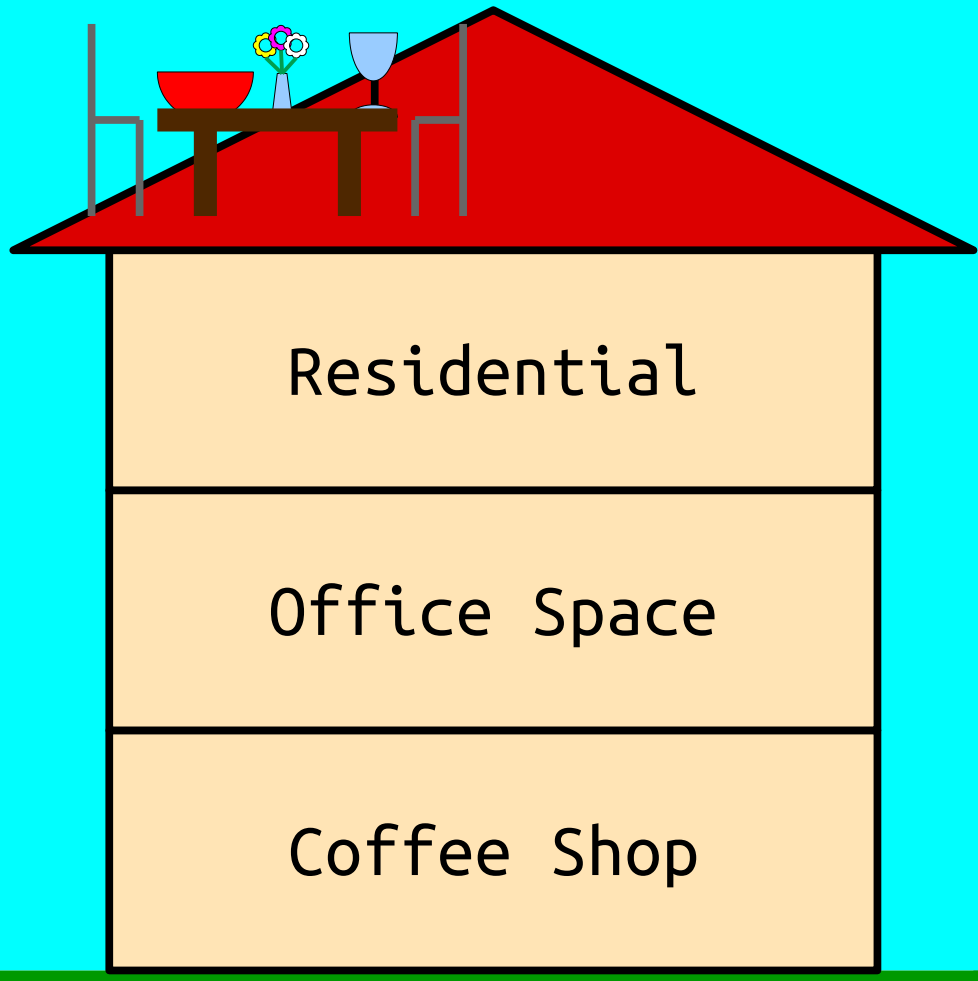


ptr

```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";

ptr[3] = "Restaurant"; // Uh…
```

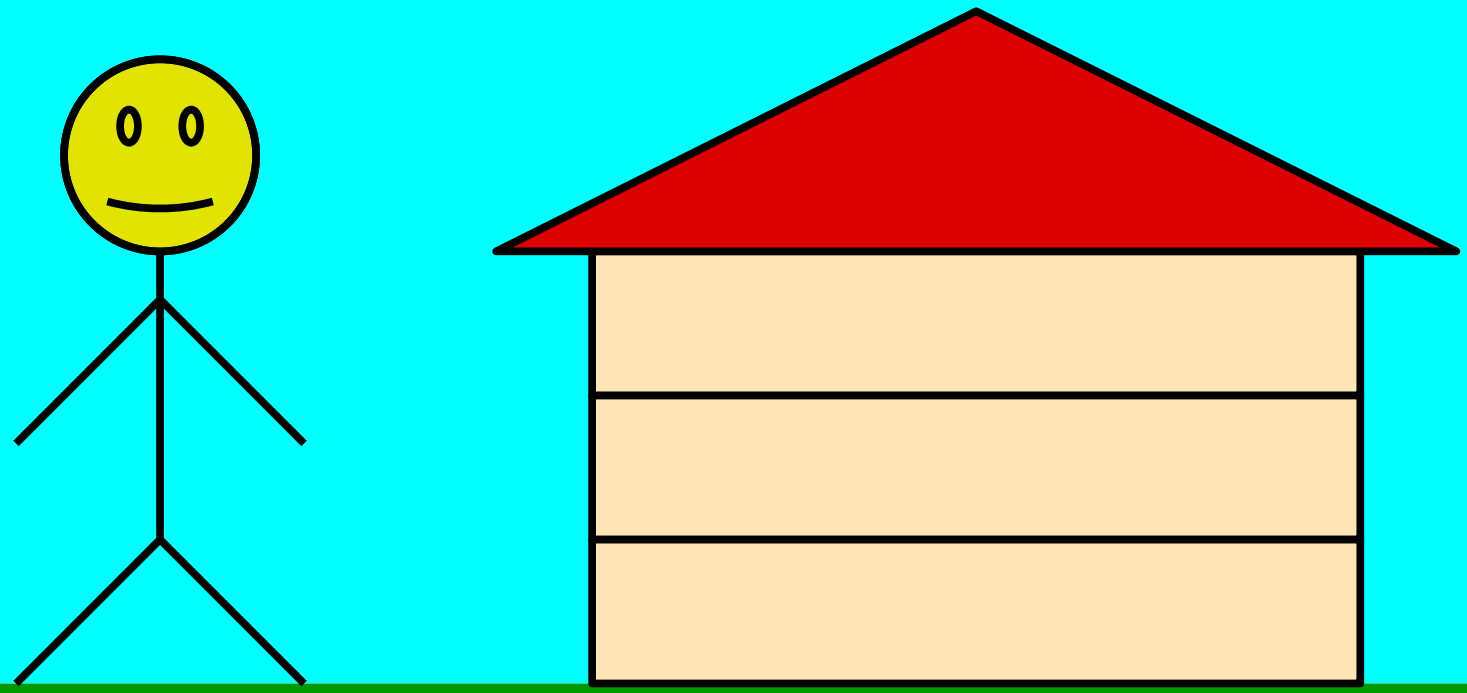
Arrays in C++ do not do any bounds checking. Writing off the end of an array might crash, might corrupt other data, or might do nothing.





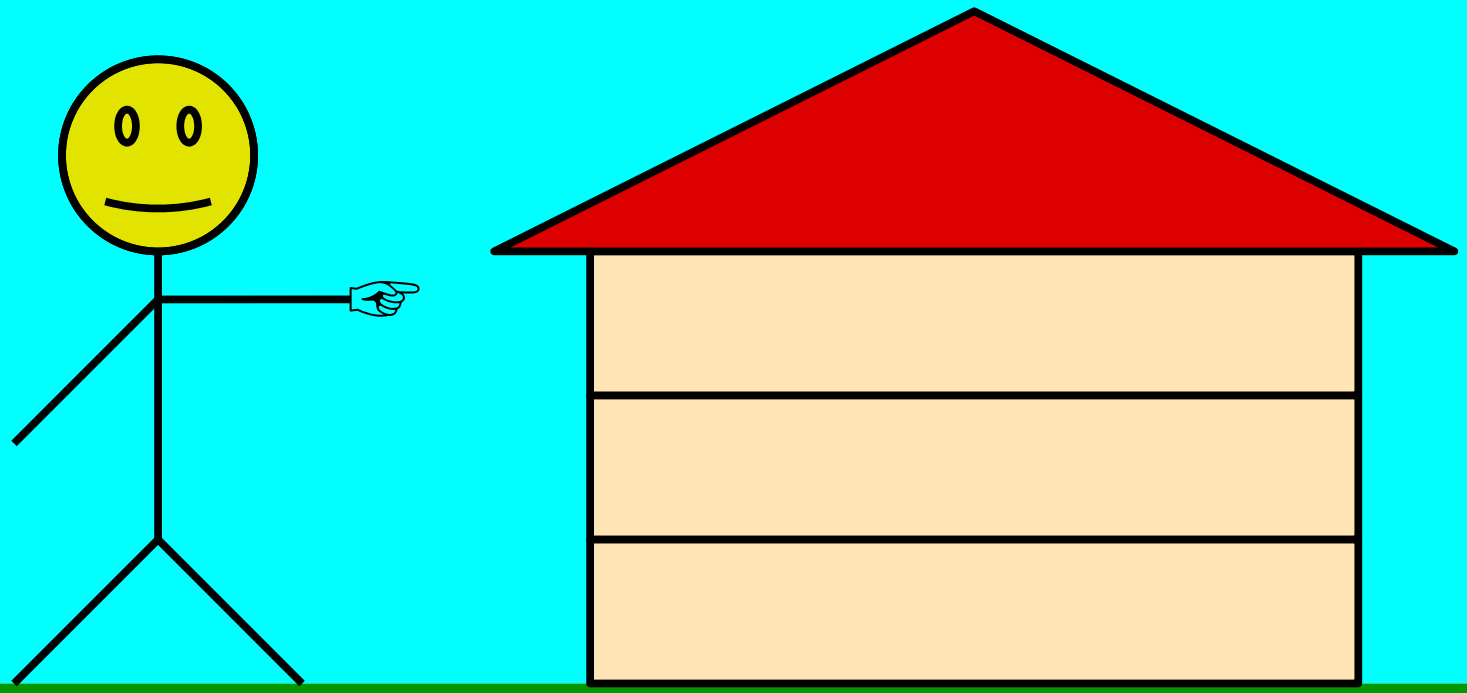
ptr

```
string* ptr = new string[3];
```



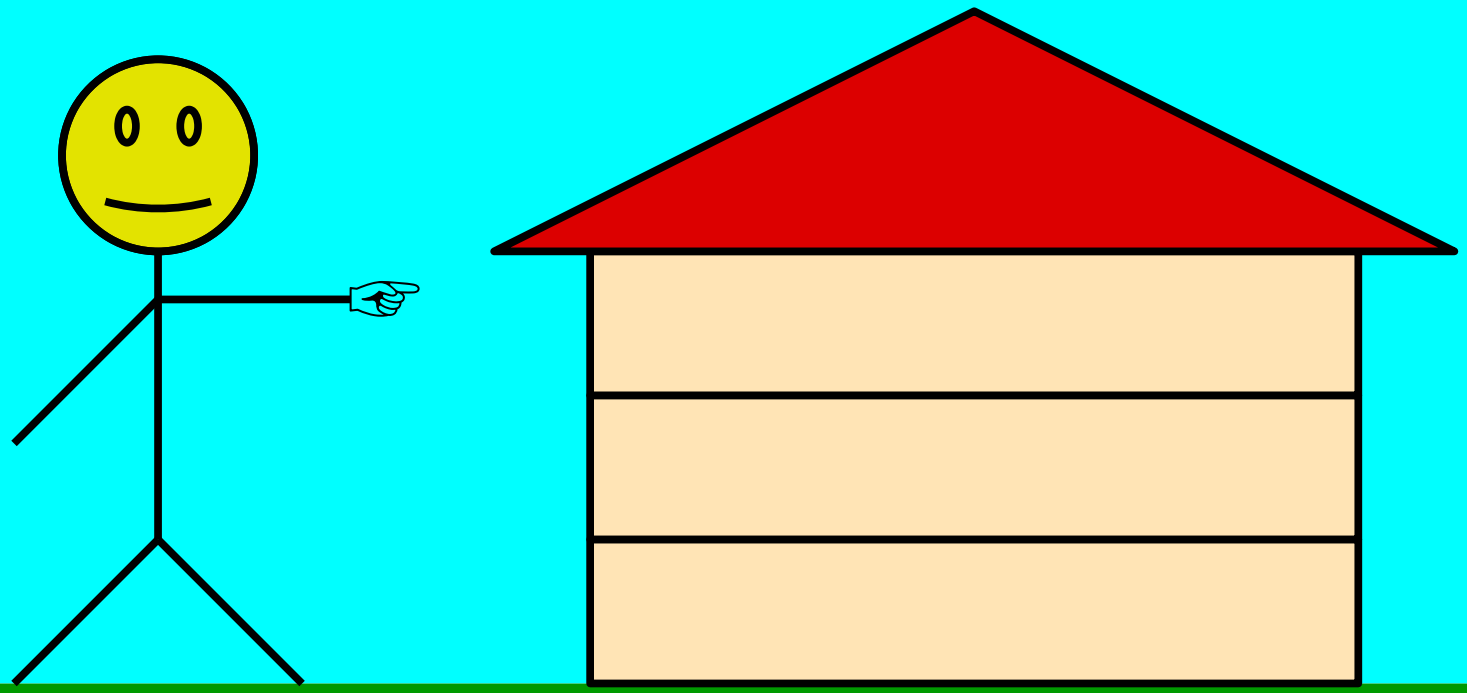
ptr

```
string* ptr = new string[3];
```



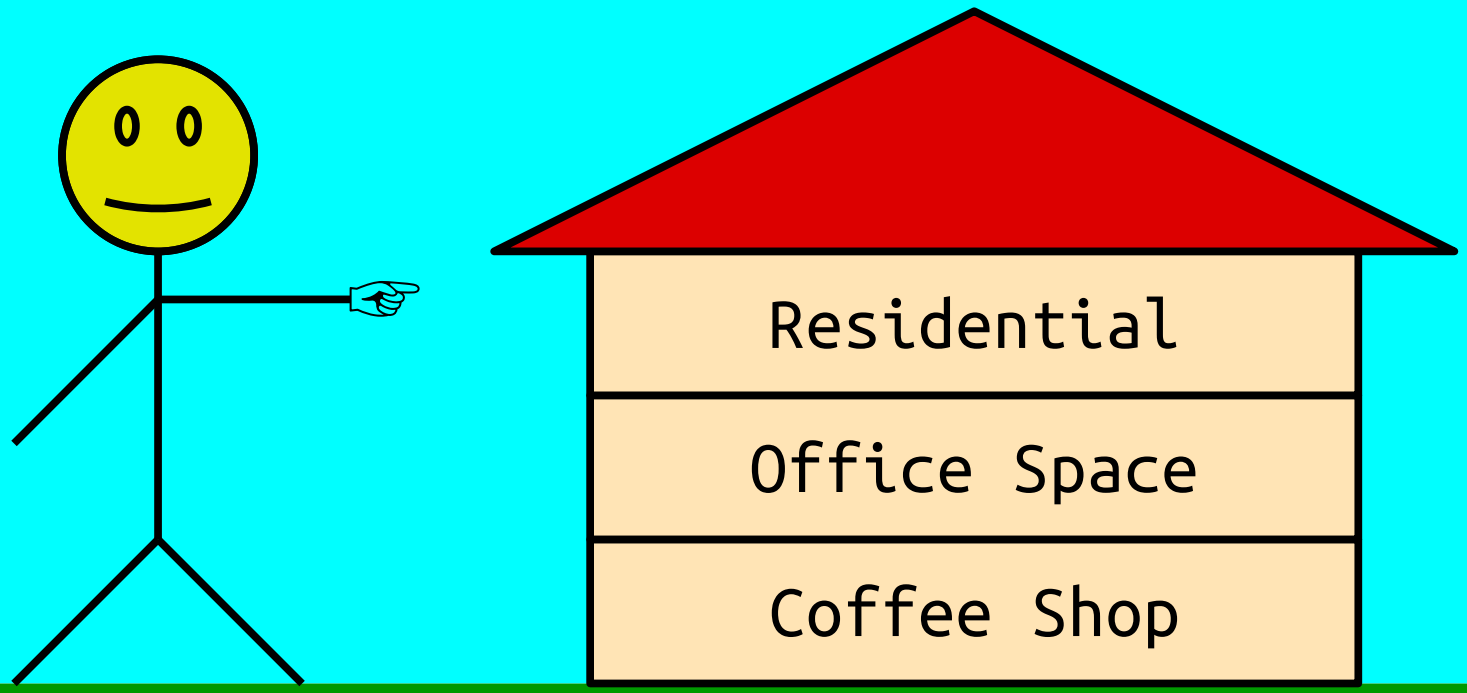
ptr

```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";
```

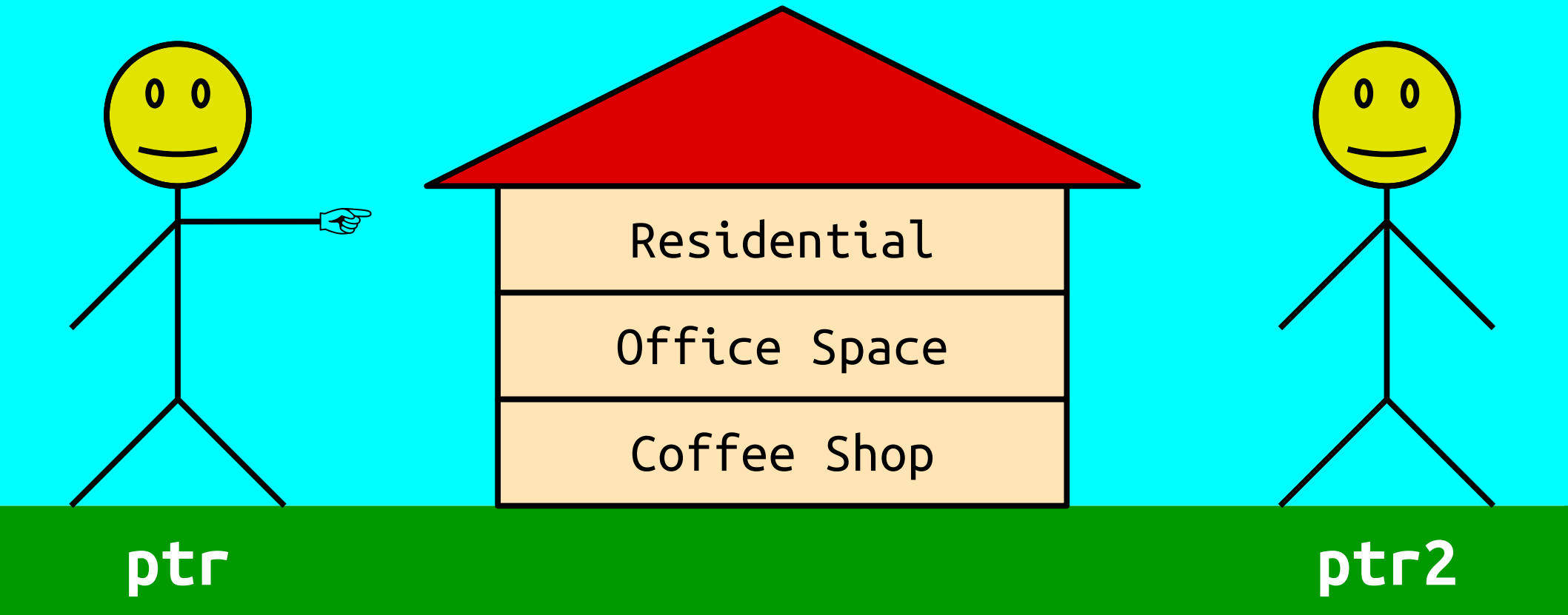


ptr

```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";
```

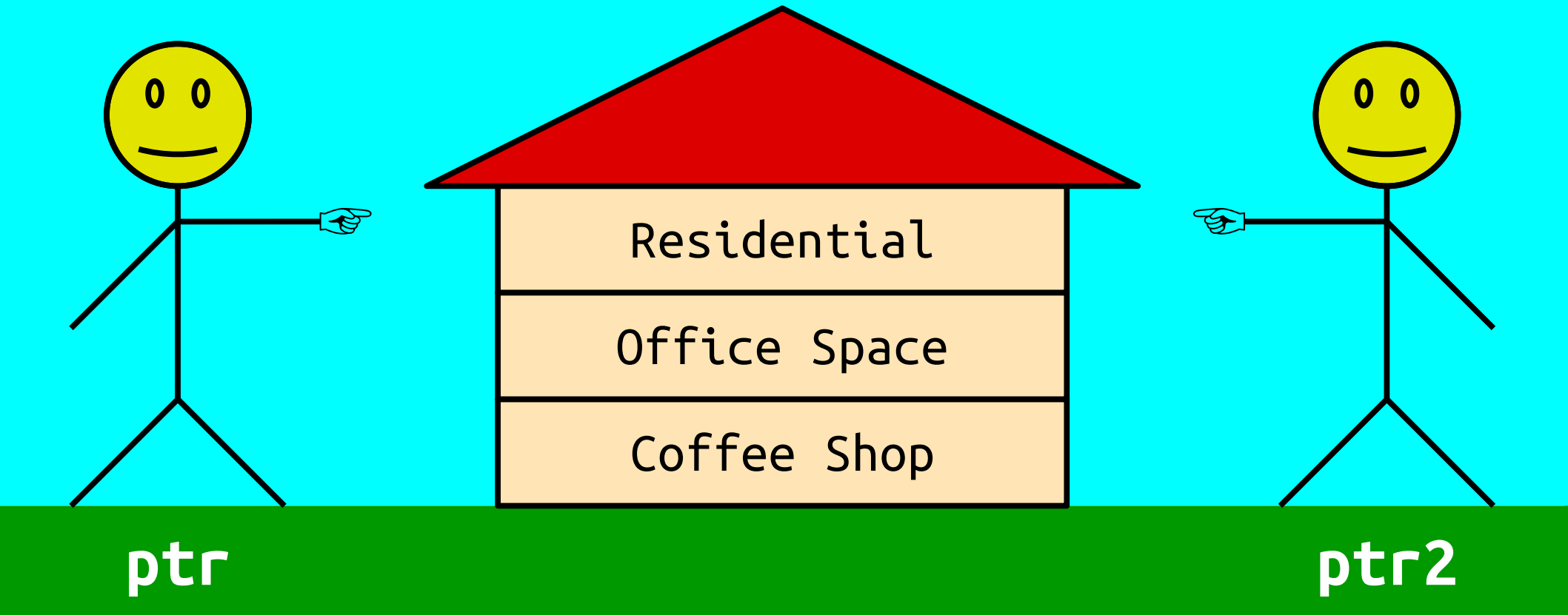


```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";

string* ptr2 = ptr;
```

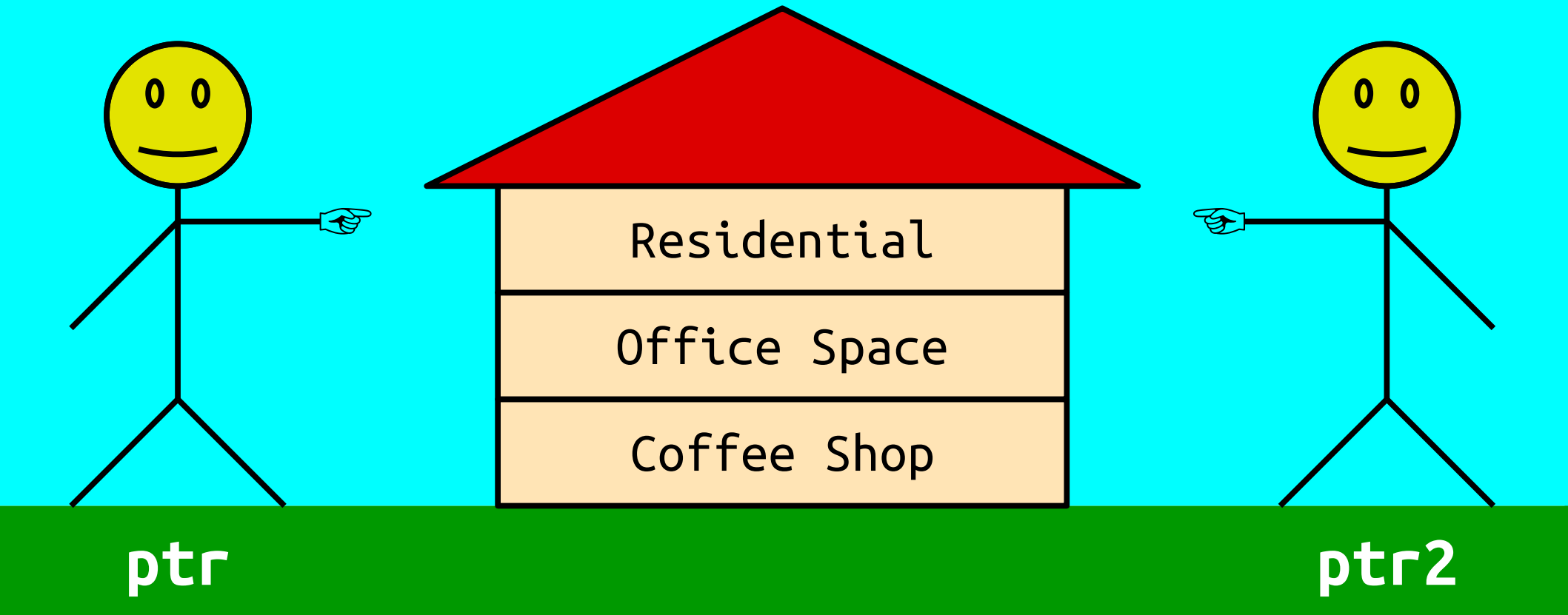


```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";

string* ptr2 = ptr;
```

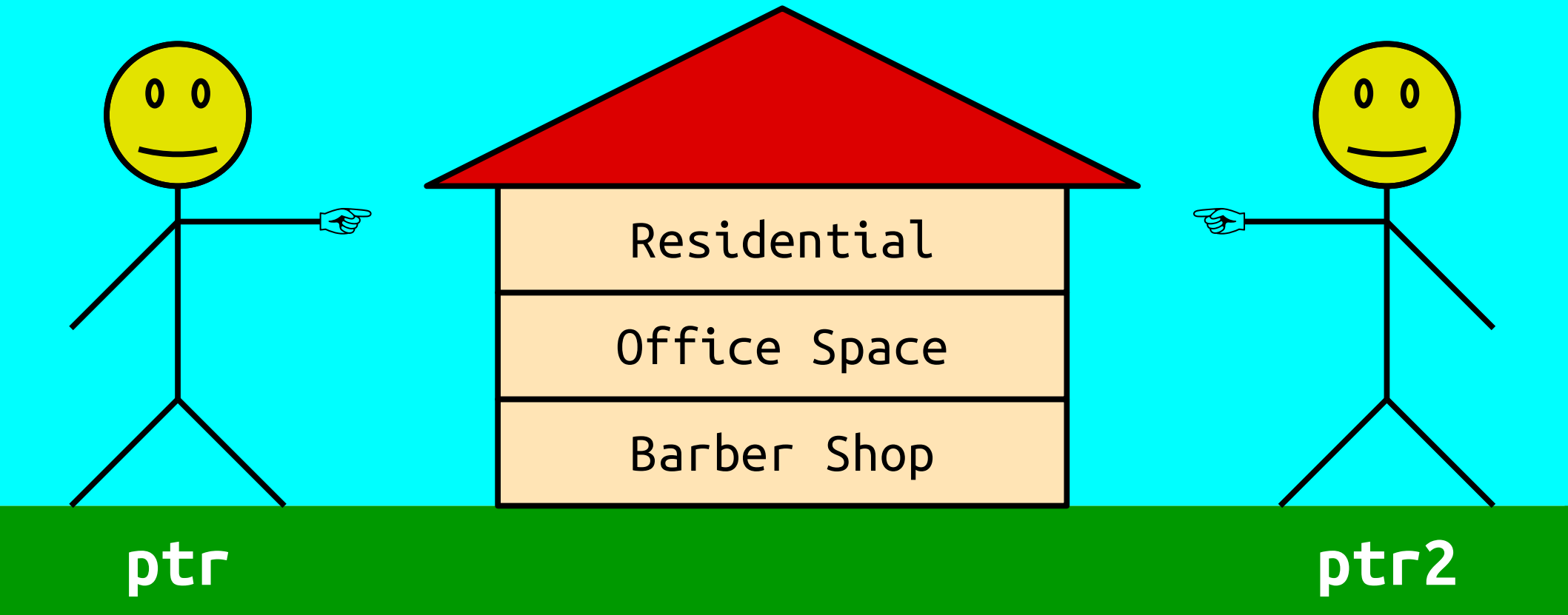
Assigning one pointer to another makes them both point to the same array.



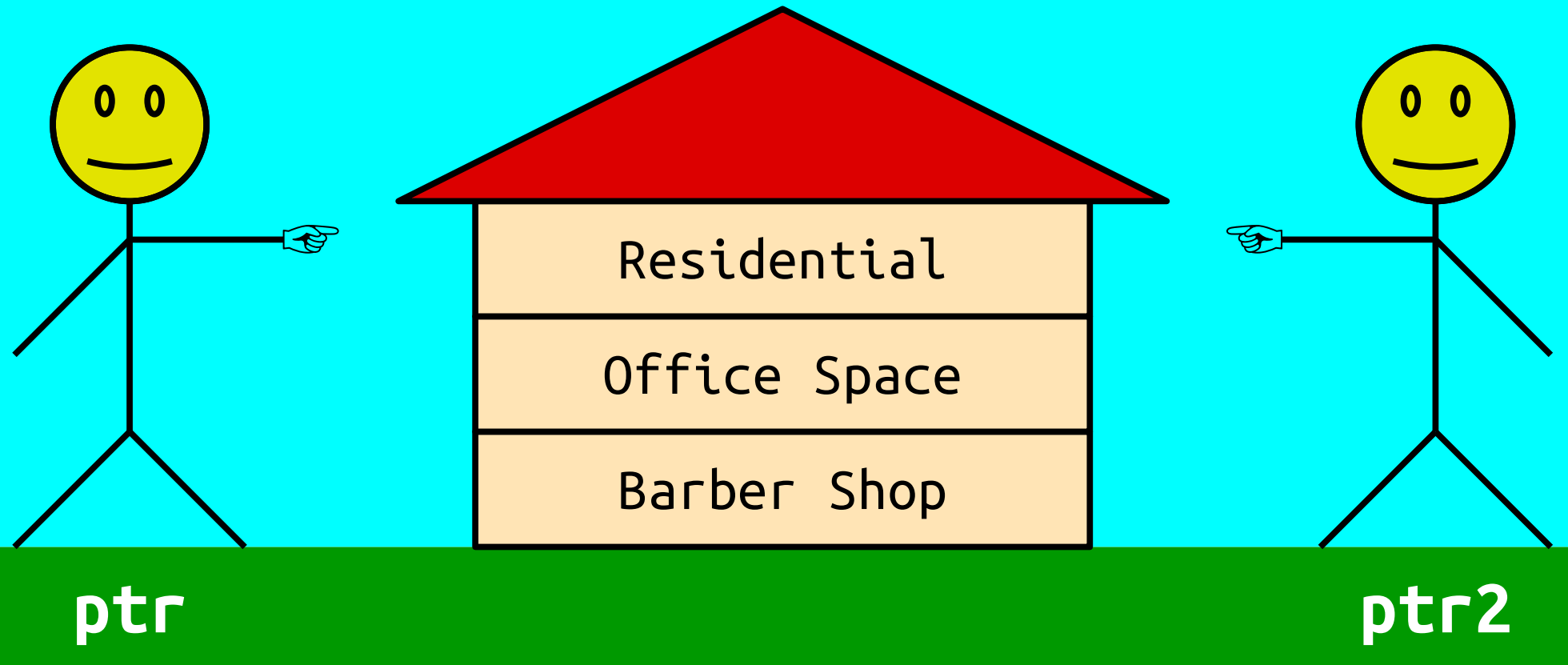
```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";

string* ptr2 = ptr;
ptr2[0] = "Barber Shop";
```



```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";

string* ptr2 = ptr;
ptr2[0] = "Barber Shop";
```



```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";

string* ptr2 = ptr;
ptr2[0] = "Barber Shop";
cout << ptr[0] << endl;
```

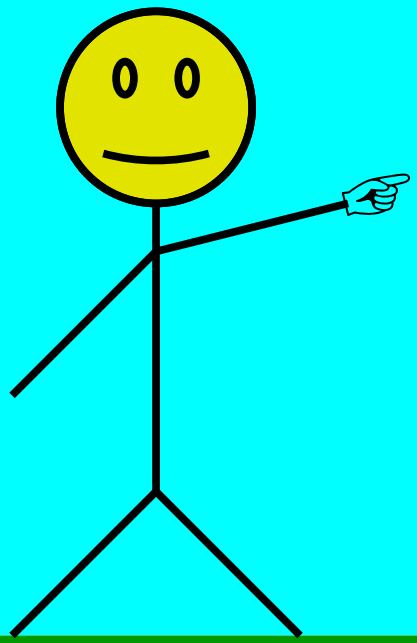


```
string* ptr;
```

Pointers always point somewhere, even if you don't initialize them.

An ***uninitialized pointer*** (sometimes called a ***garbage pointer***) is a pointer that hasn't been assigned to point to anything.

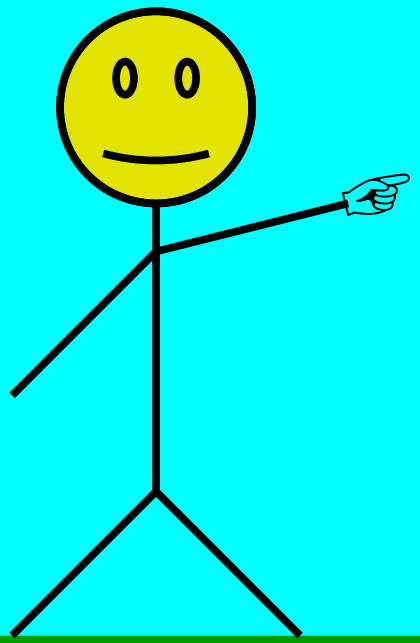
Uninitialized pointers point somewhere, but there's no way to predict exactly where.



ptr

```
string* ptr;
ptr[1] = "Day Care"; // Uh…
```

Day Care



ptr

C++ does not do any safety checks when writing through uninitialized pointers. It might crash your program. It might corrupt data. Or it might seemingly do nothing.

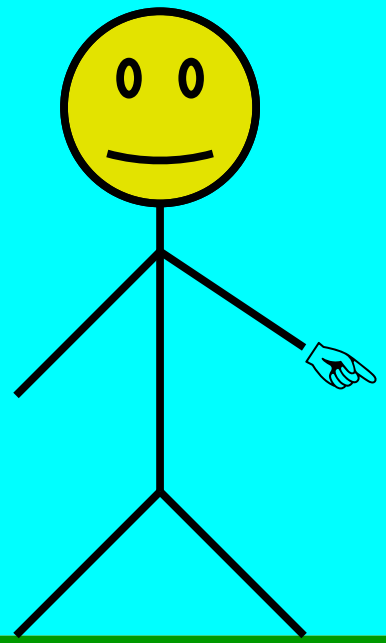
```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```

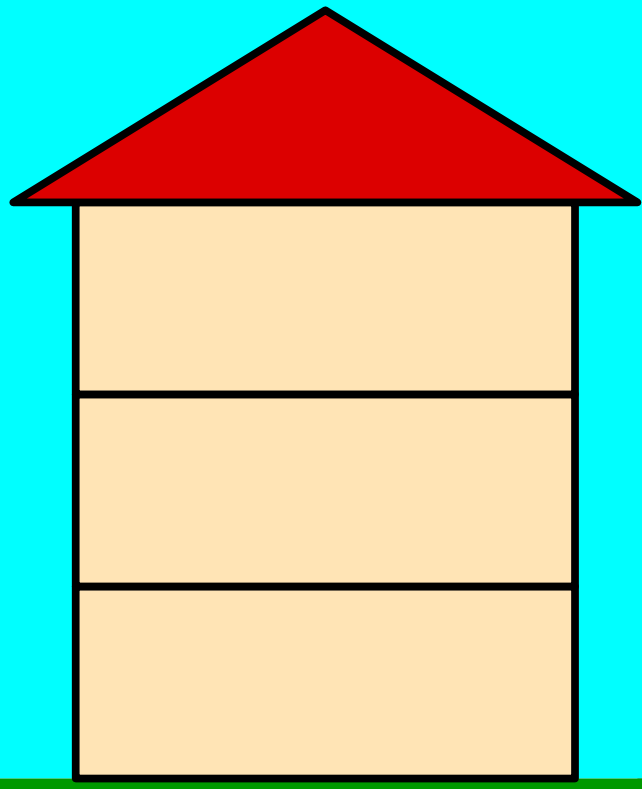
```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```

```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```

```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```

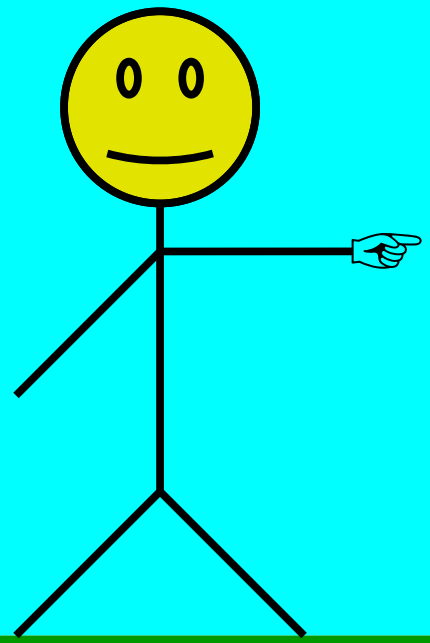
```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```

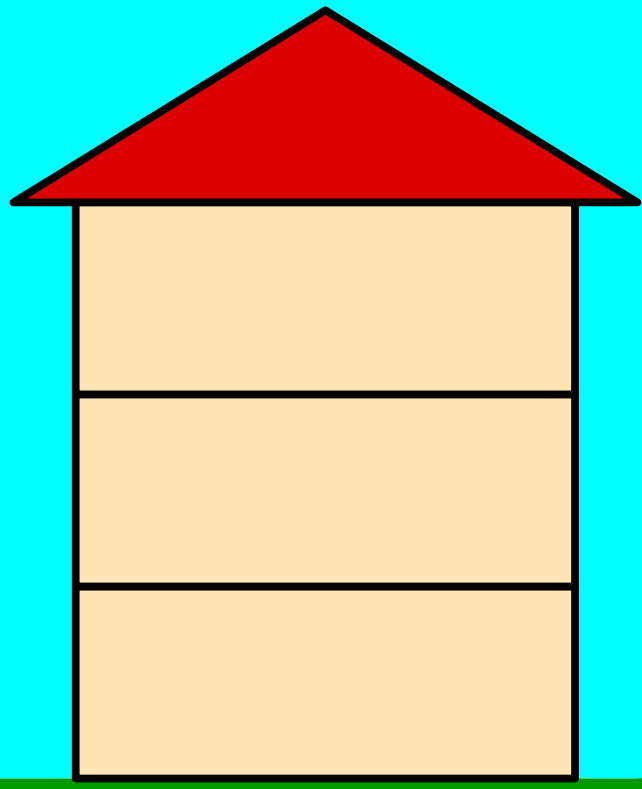




ptr

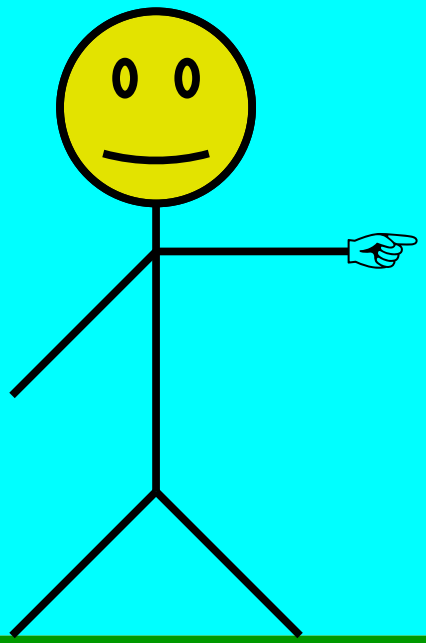
```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```

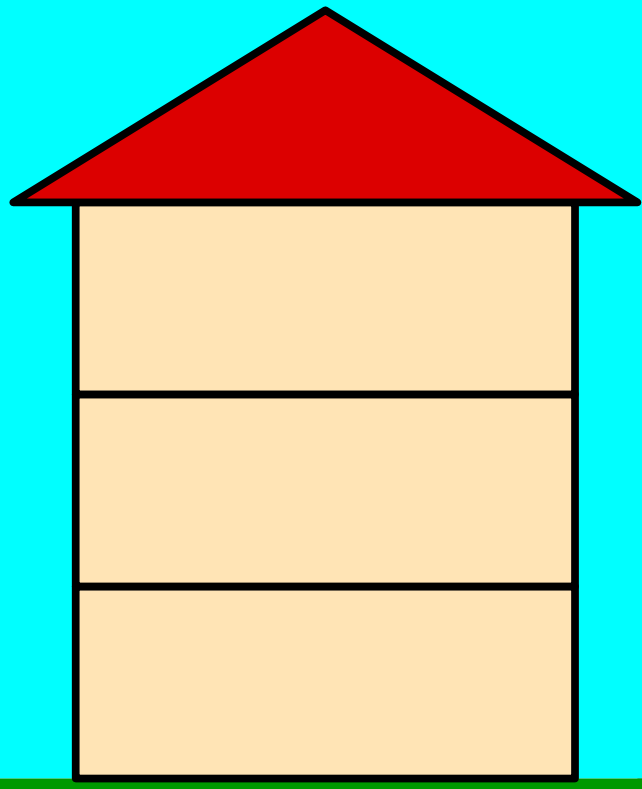




ptr

```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```





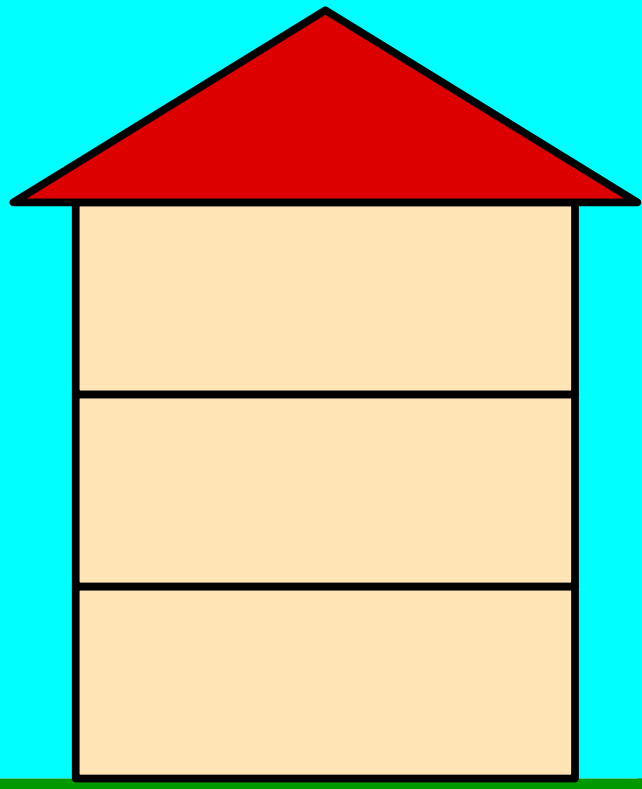
ptr

```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```

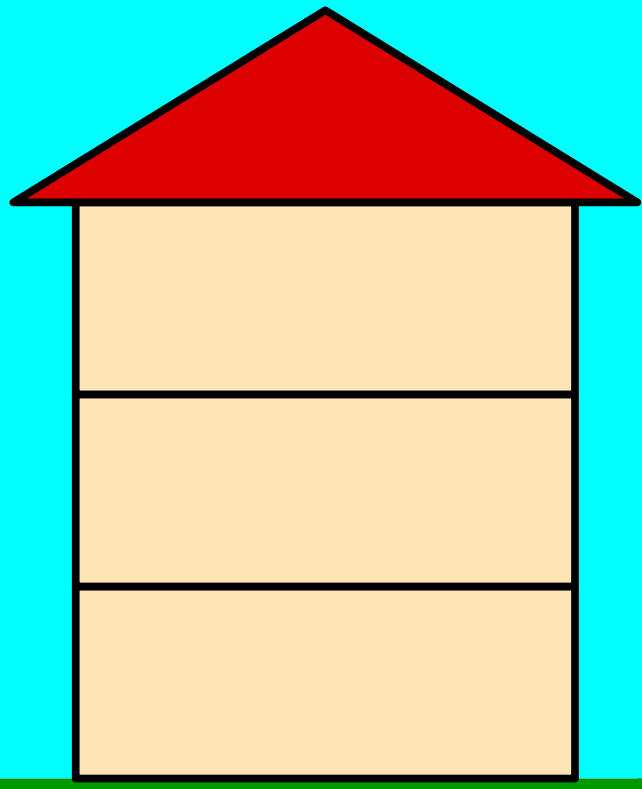
The local variable `ptr` is cleaned up once the function returns – but the array itself remains!

This is different than how the container types work.

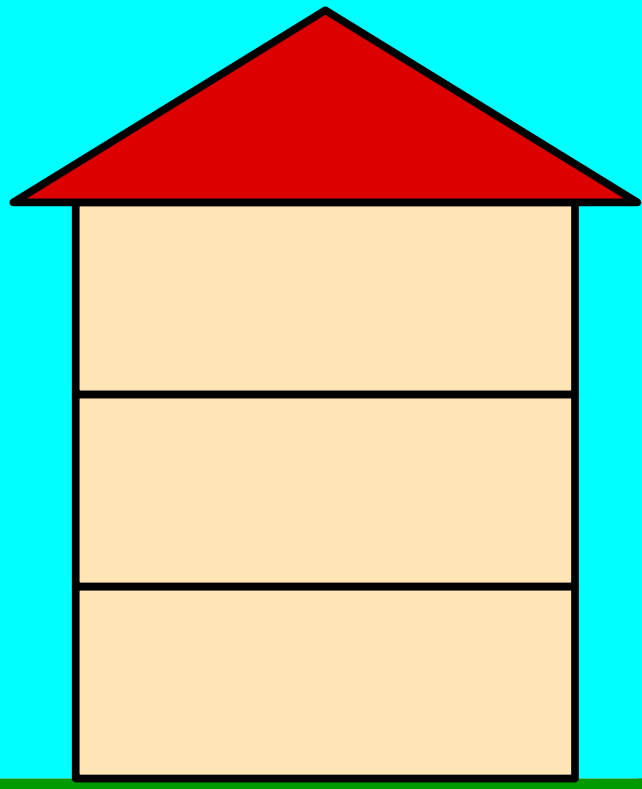
Anything created with `new[]` persists until explicitly cleaned up.



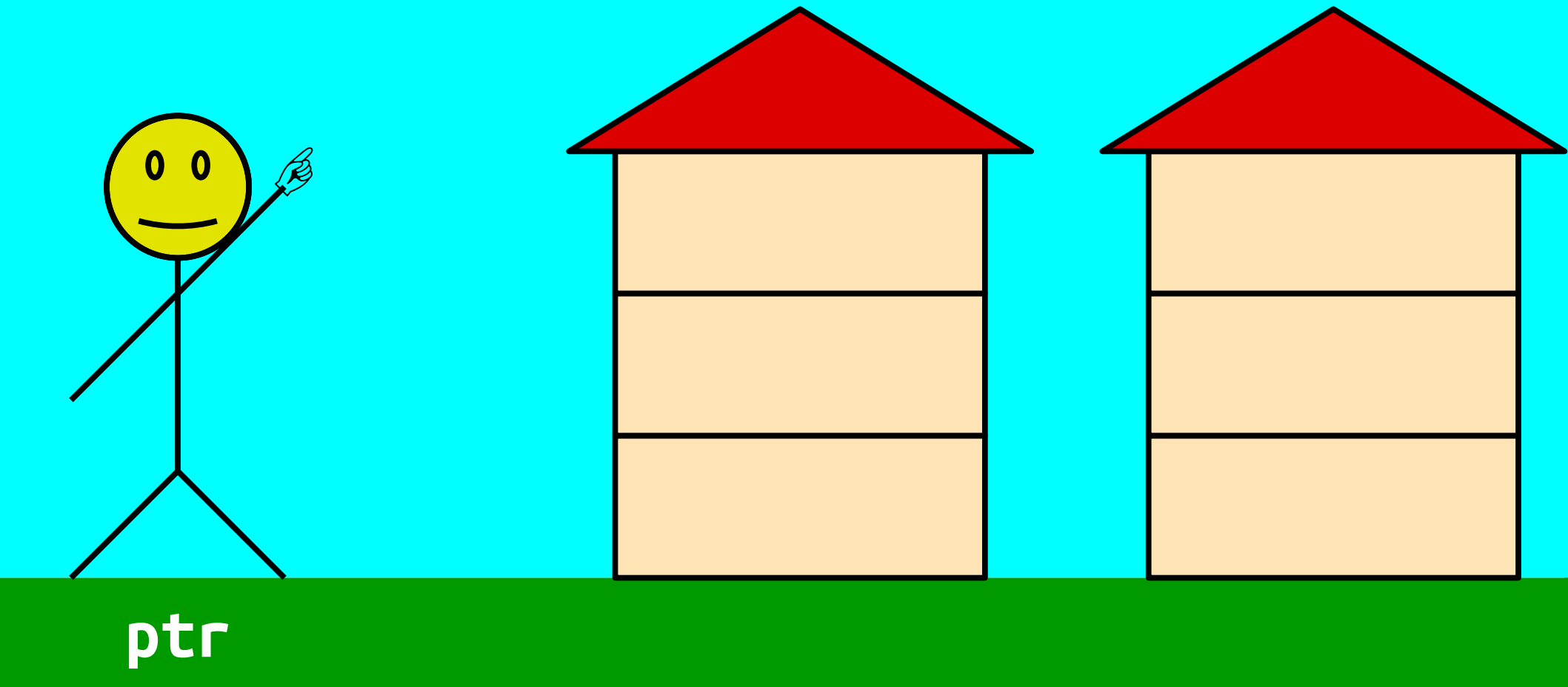
```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```



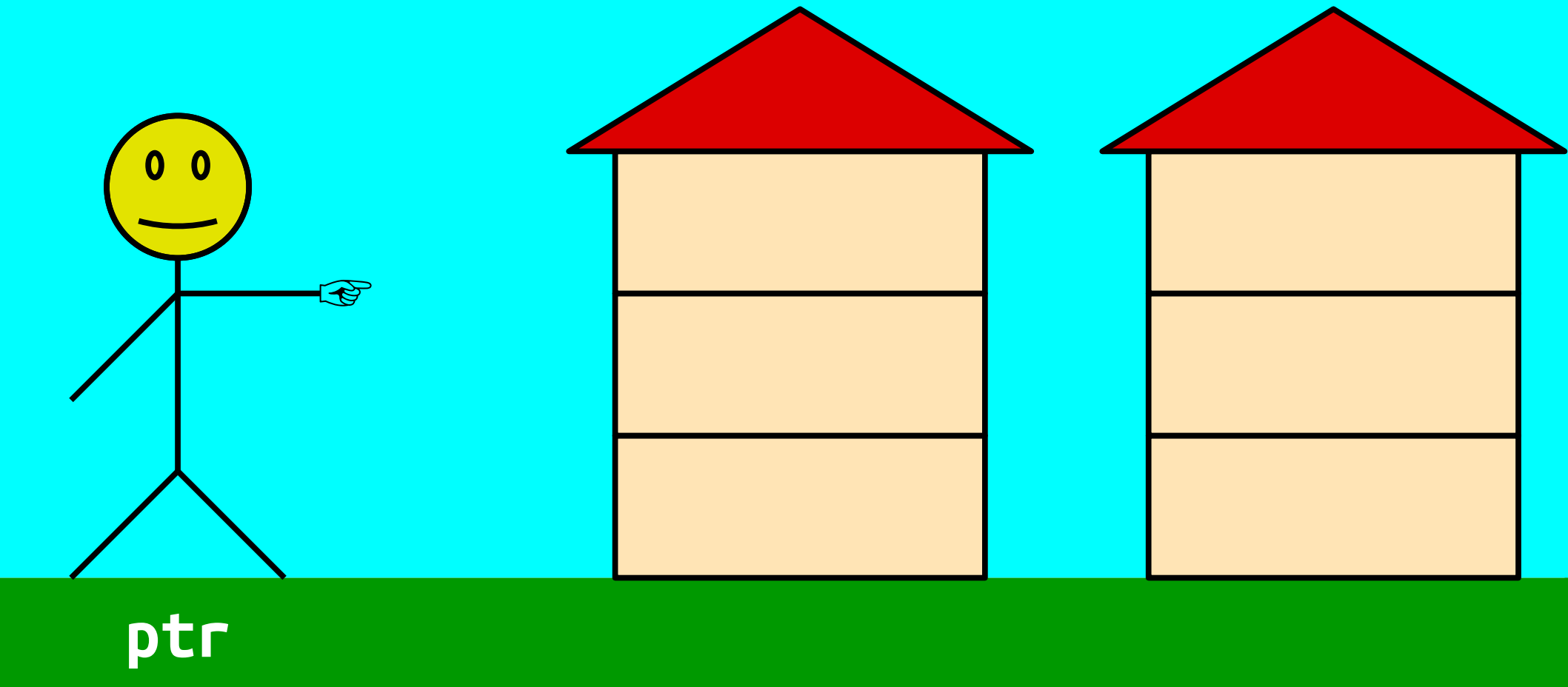
```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```



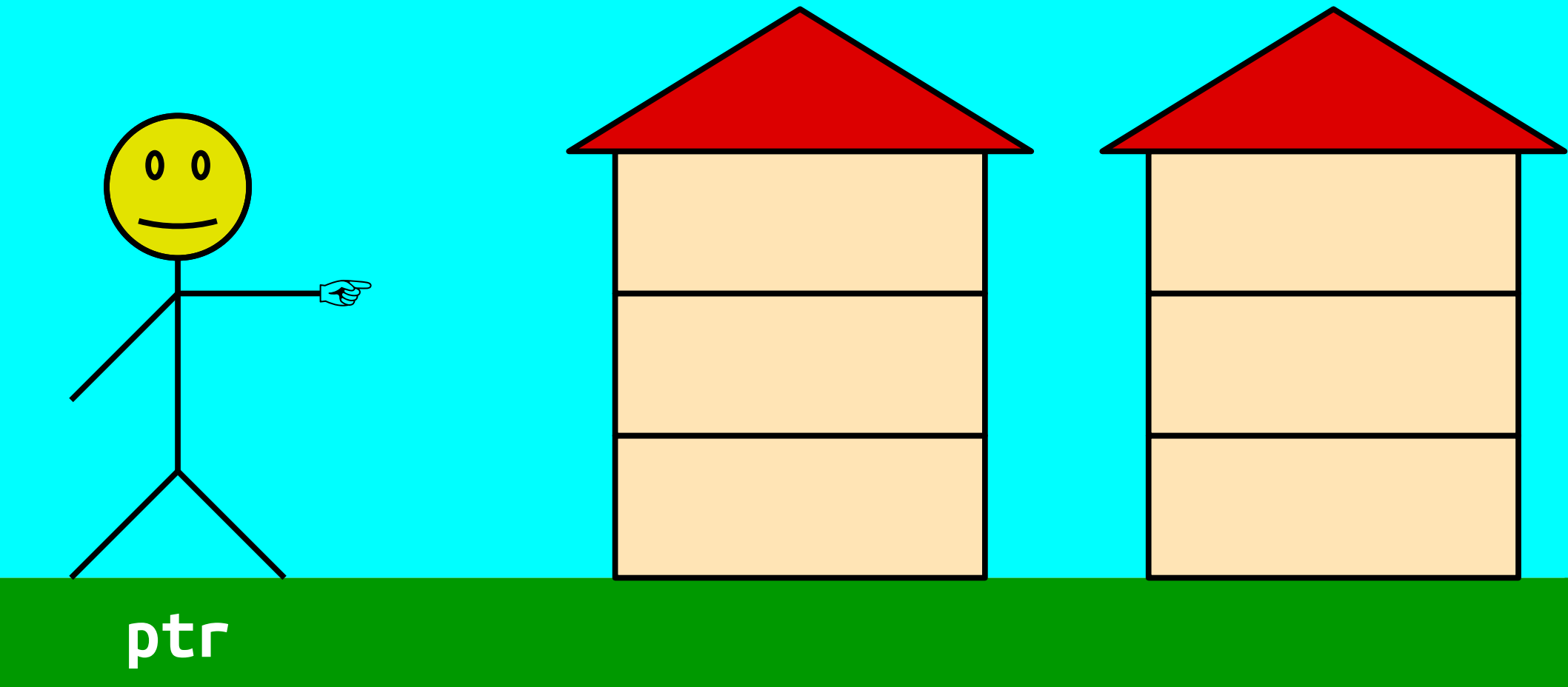
```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```



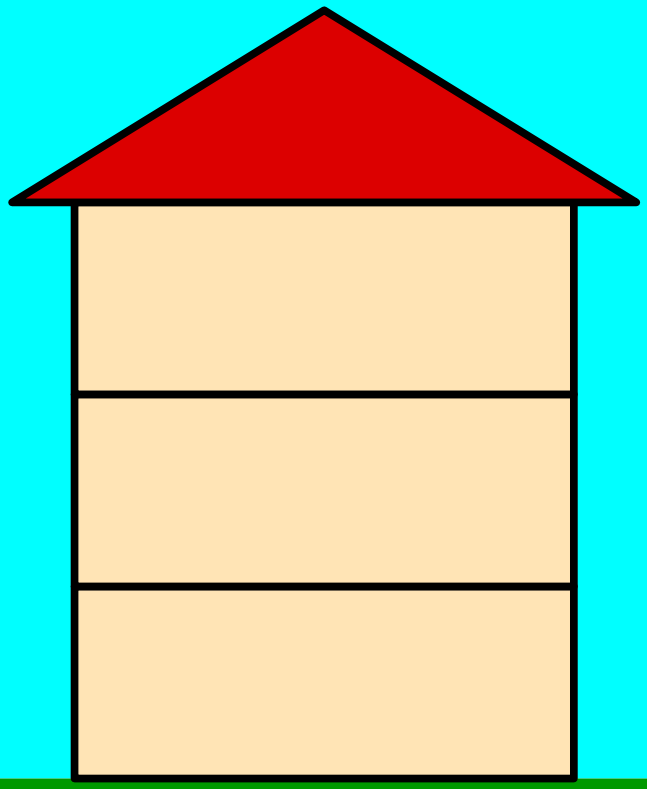
```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```

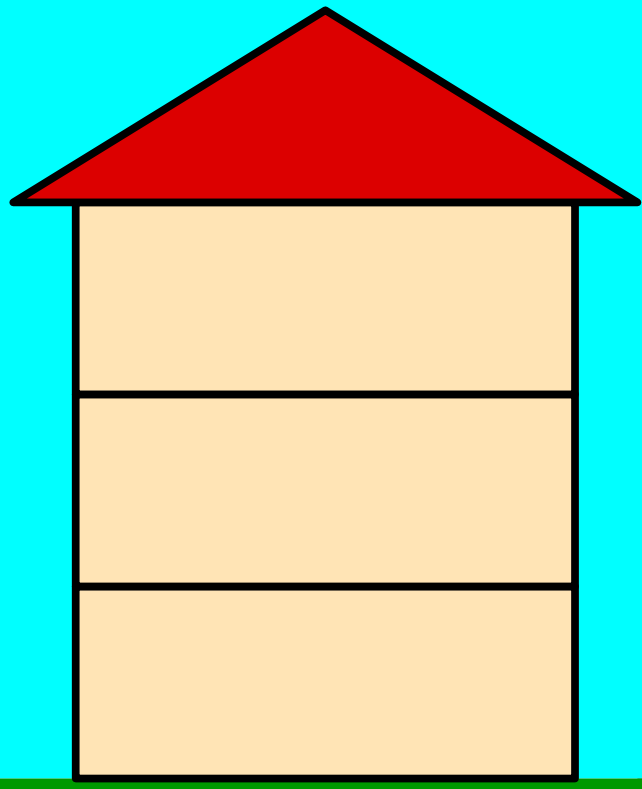


```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```

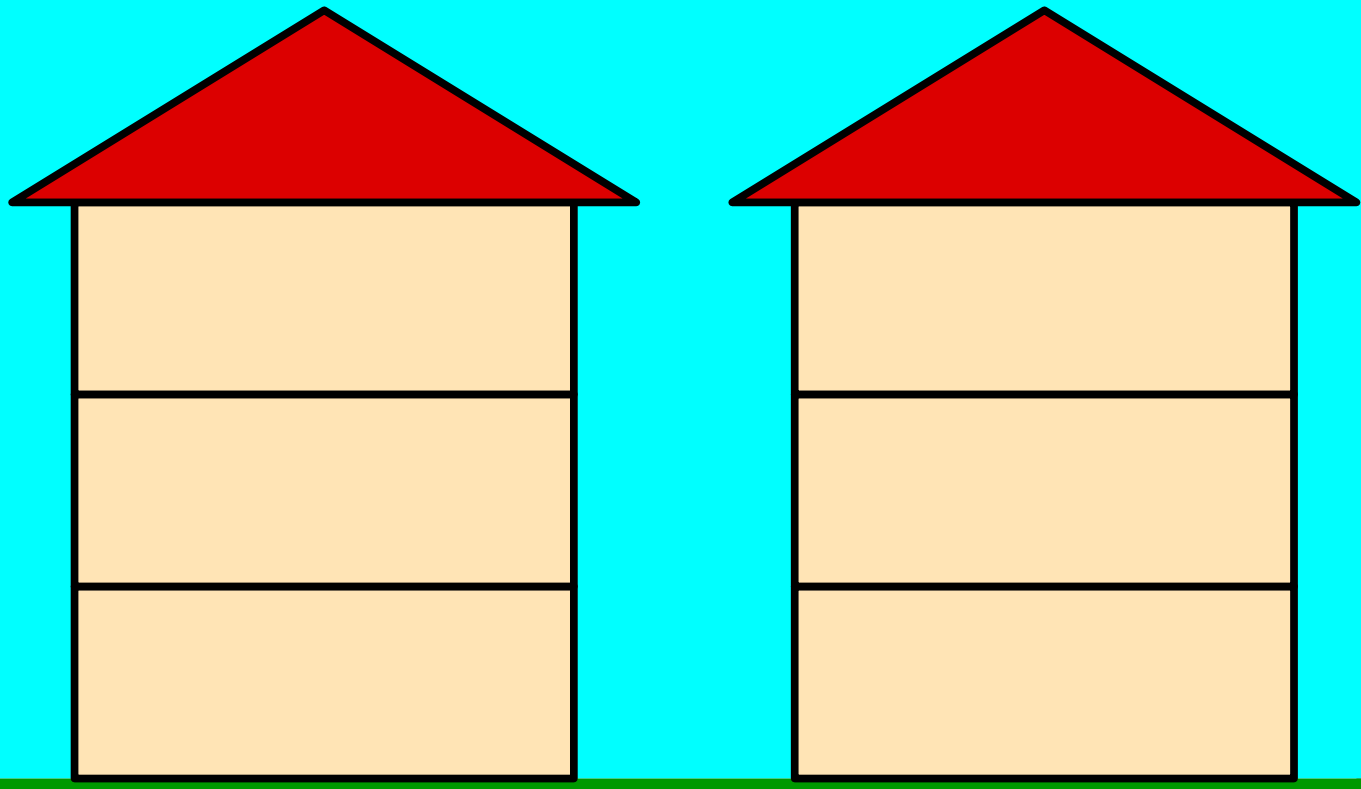


```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```

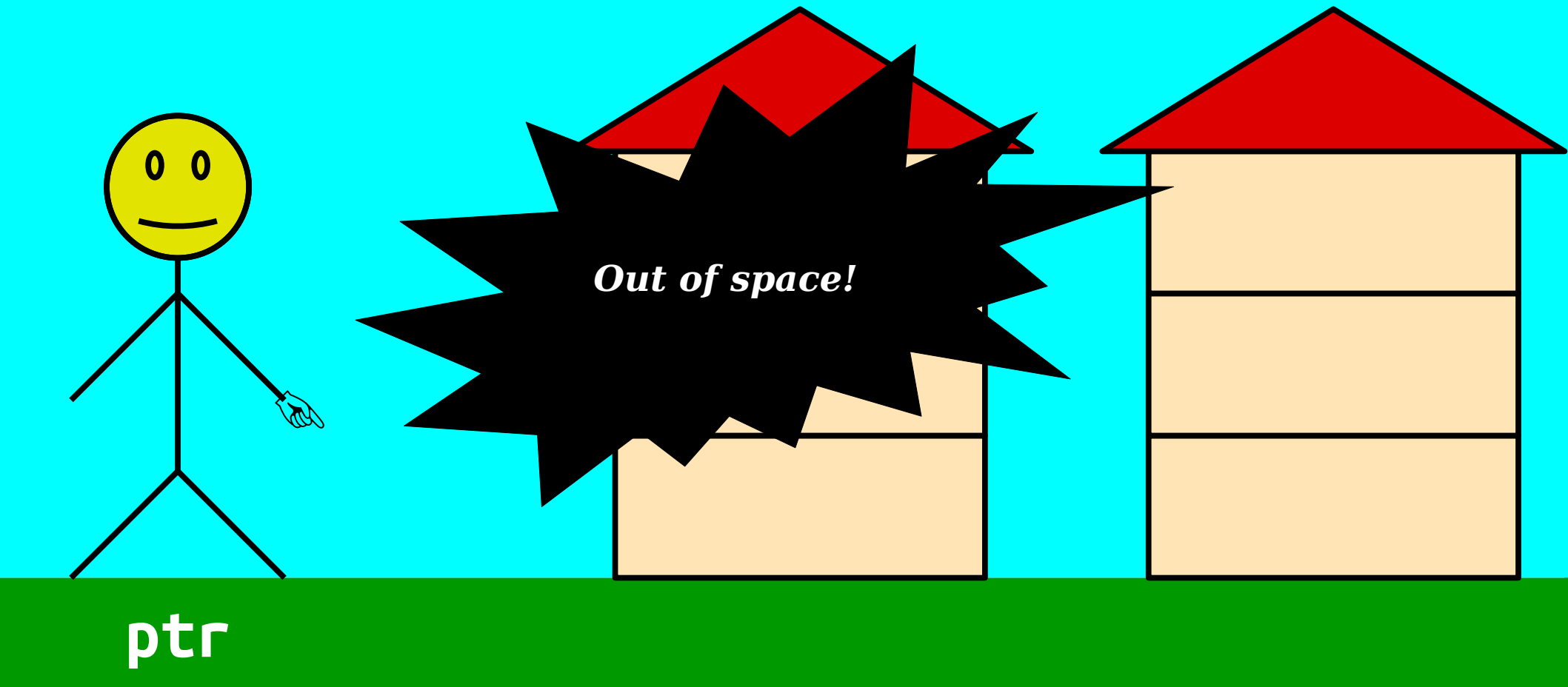




```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```



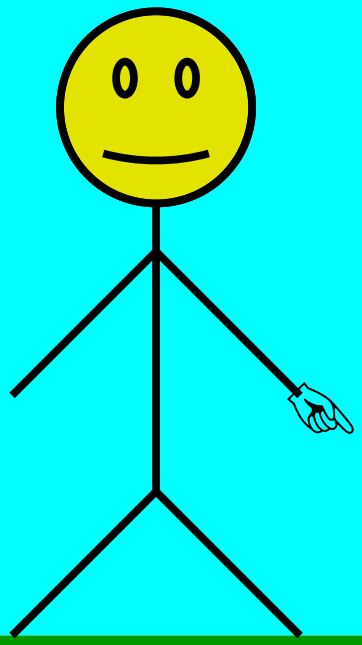
```
void makeAnArray() {
    string* ptr = new string[3];
}

int main() {
    for (int i = 0; i < 5; i++) {
        makeAnArray();
    }
}
```

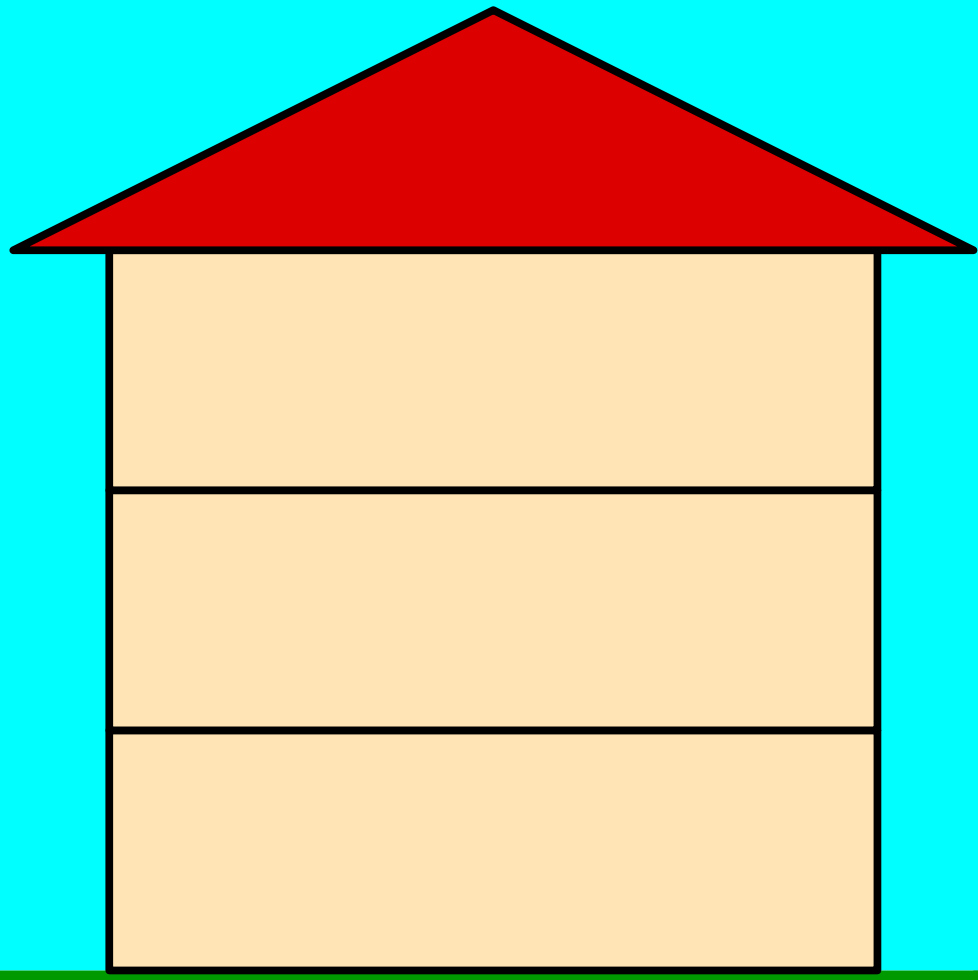


Cleaning Up

- When declaring local variables or parameters, C++ will automatically handle memory allocation and deallocation for you.
- When using `new[]`, you are responsible for deallocating the memory you allocate.
- If you don't, you get a ***memory leak***. Your program will never be able to use that memory again.
  - Too many leaks can cause a program to slow down and eventually crash as memory becomes more and more scarce!
- (Realistically, that previous example wouldn’t allocate enough memory to crash the program. You need to leak a bunch of memory before that will happen.)

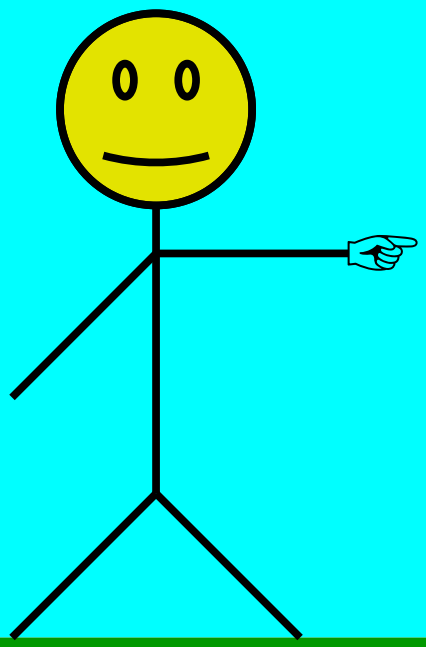
```
string* ptr = new string[3];
```

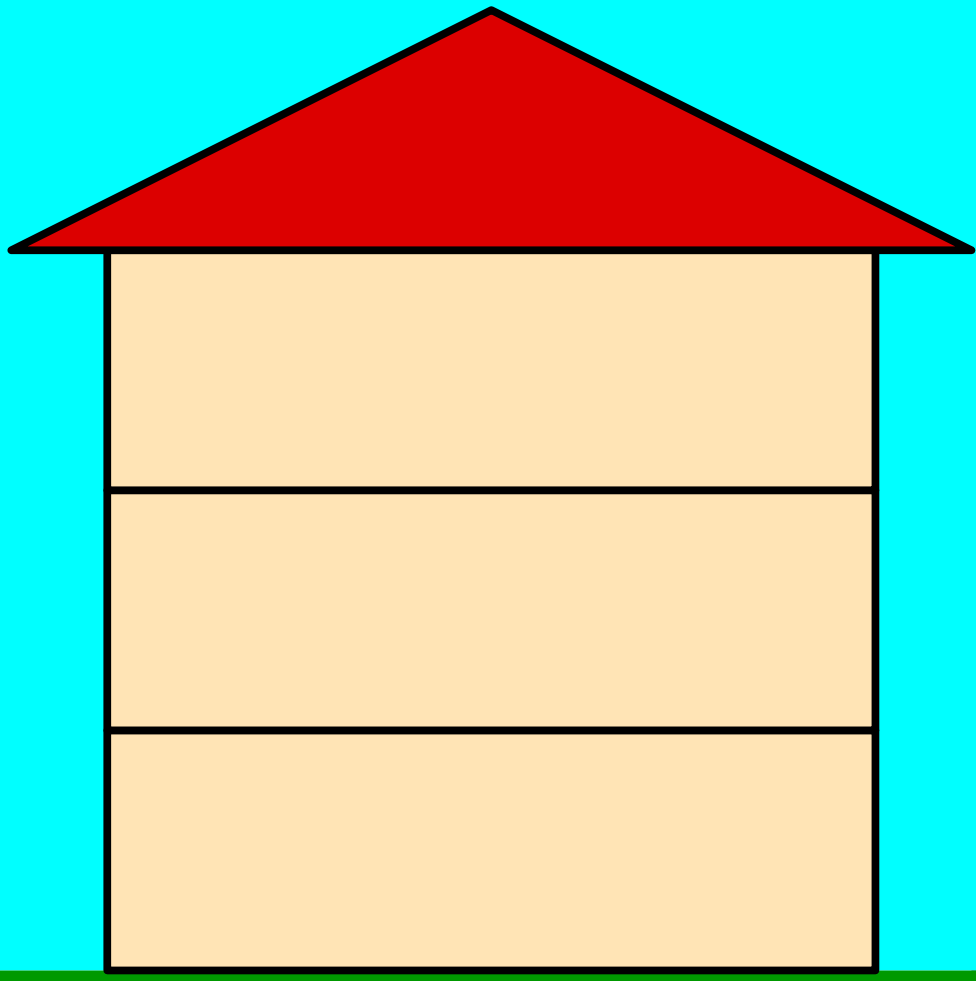




ptr

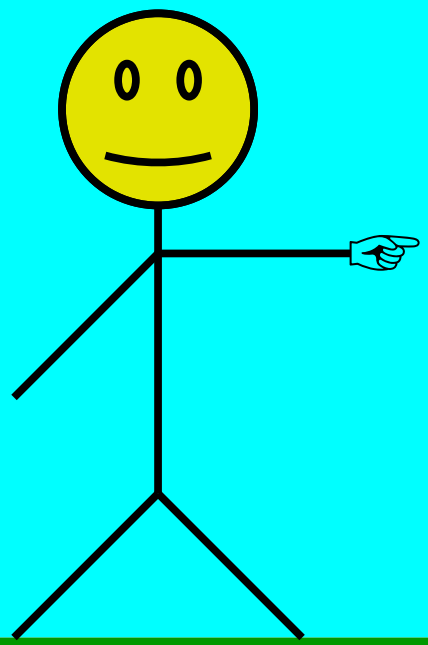
```
string* ptr = new string[3];
```

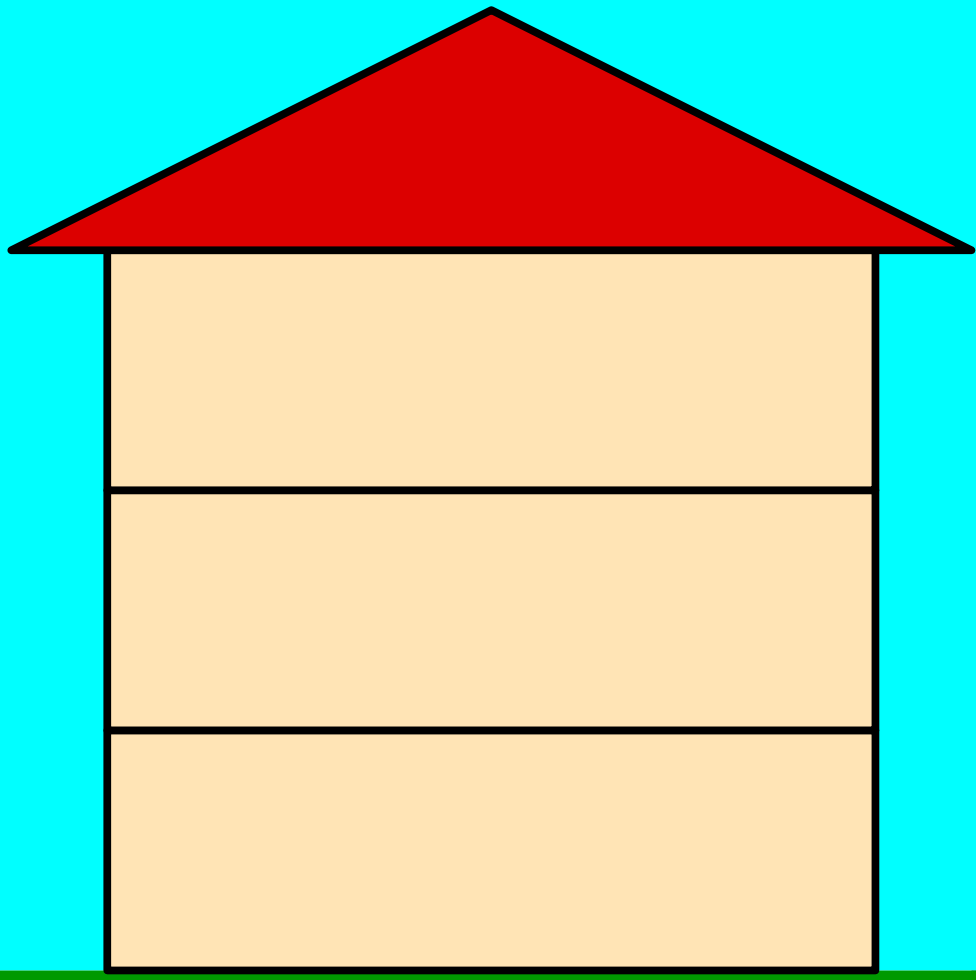




ptr

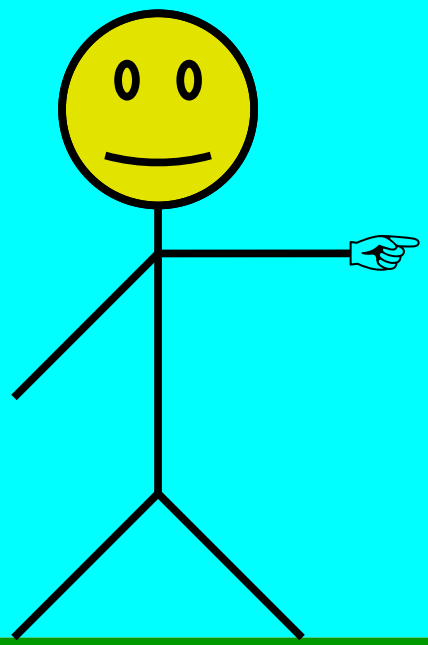
```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";
```

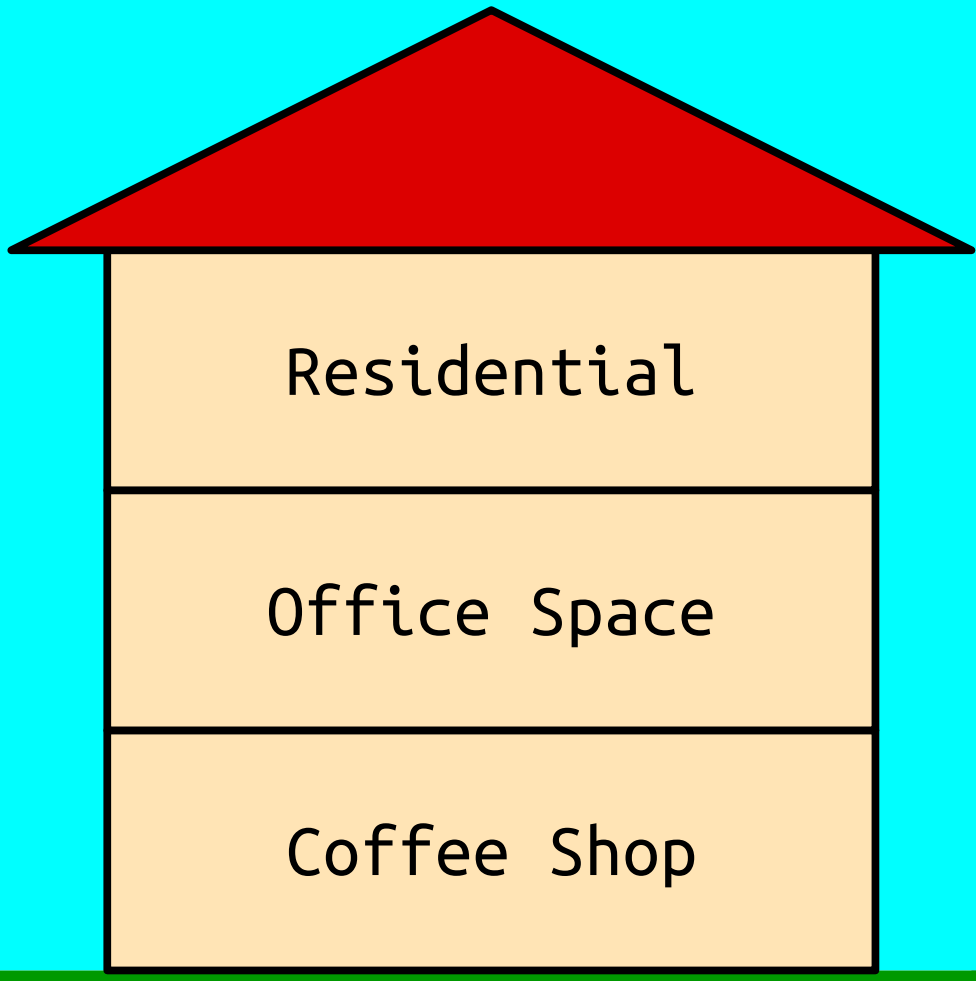




ptr

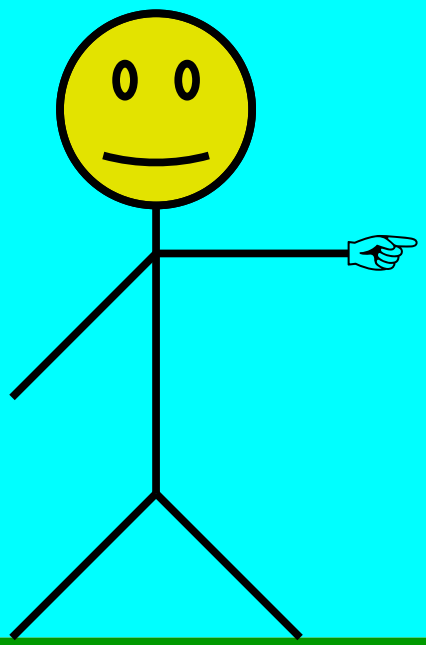
```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";
```



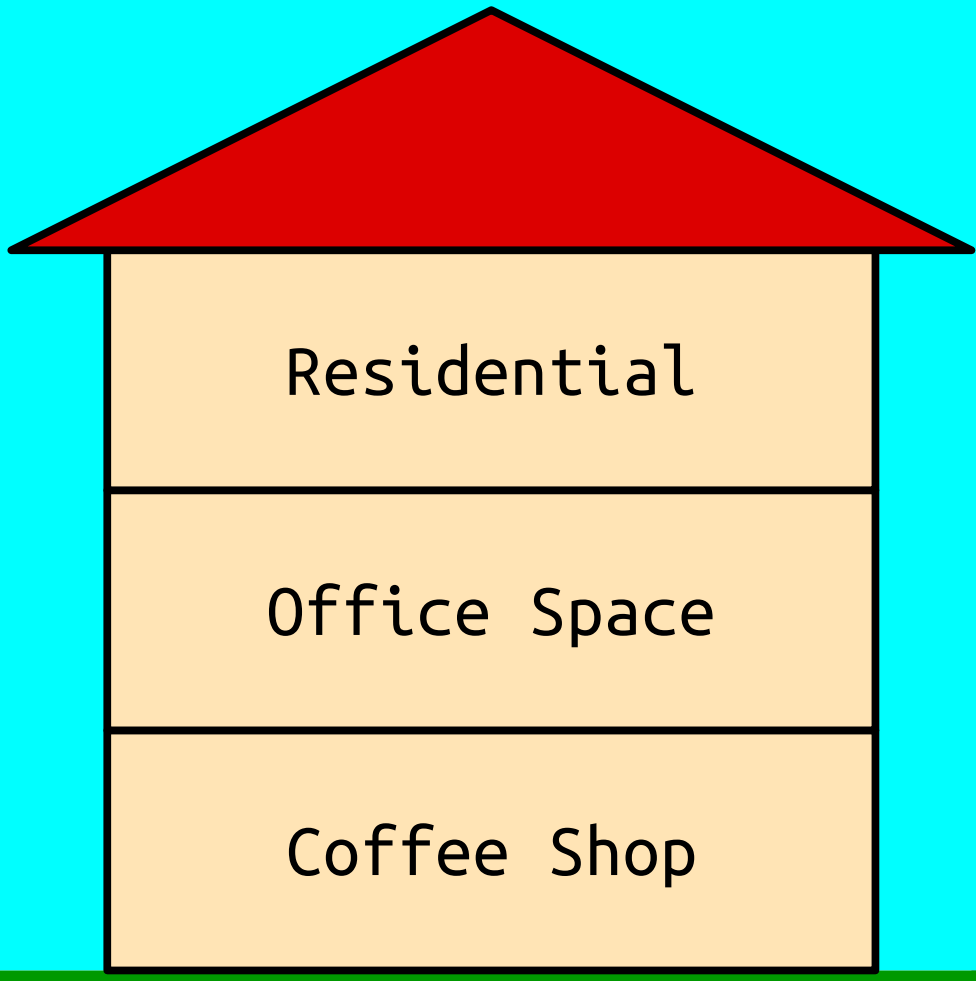


ptr

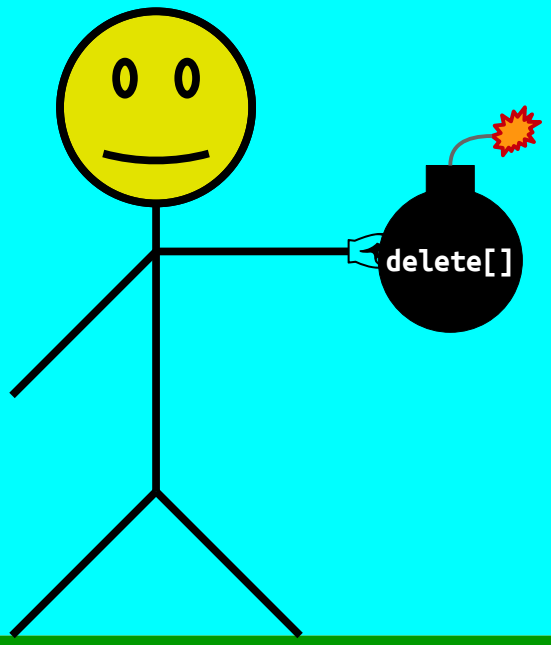
```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";

delete[] ptr;
```



ptr



```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";

delete[] ptr;
```

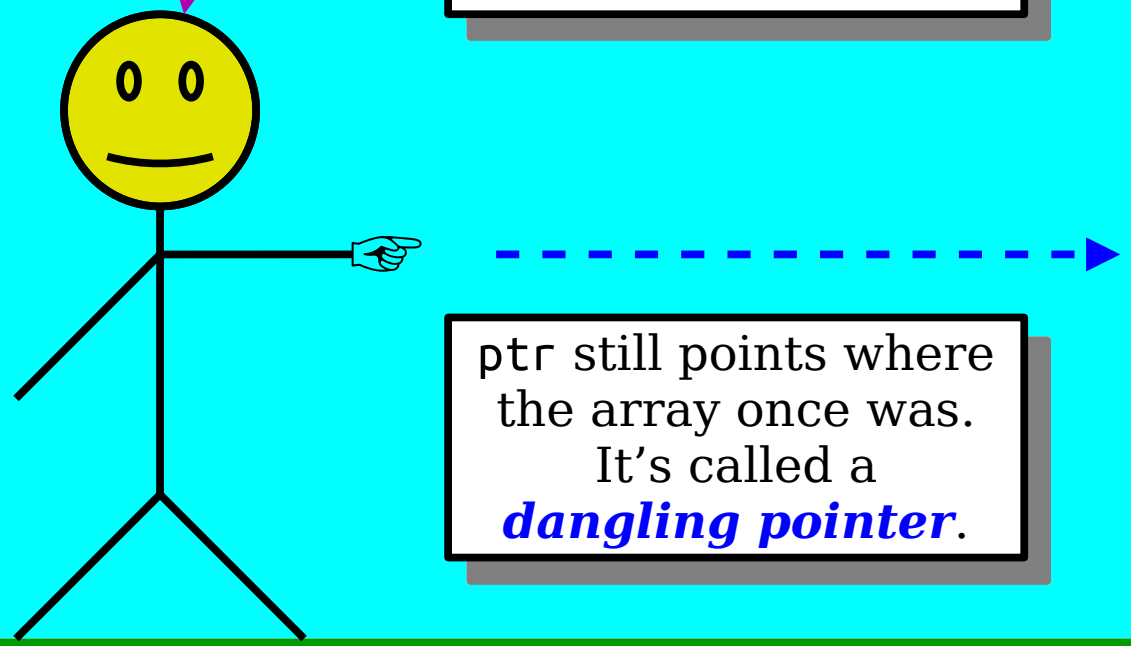


ptr

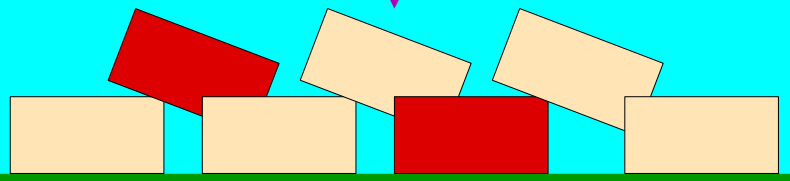
```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";

delete[] ptr;
```

The variable `ptr` is still here. It will stick around until the function returns.

We've freed up space for future buildings (arrays).



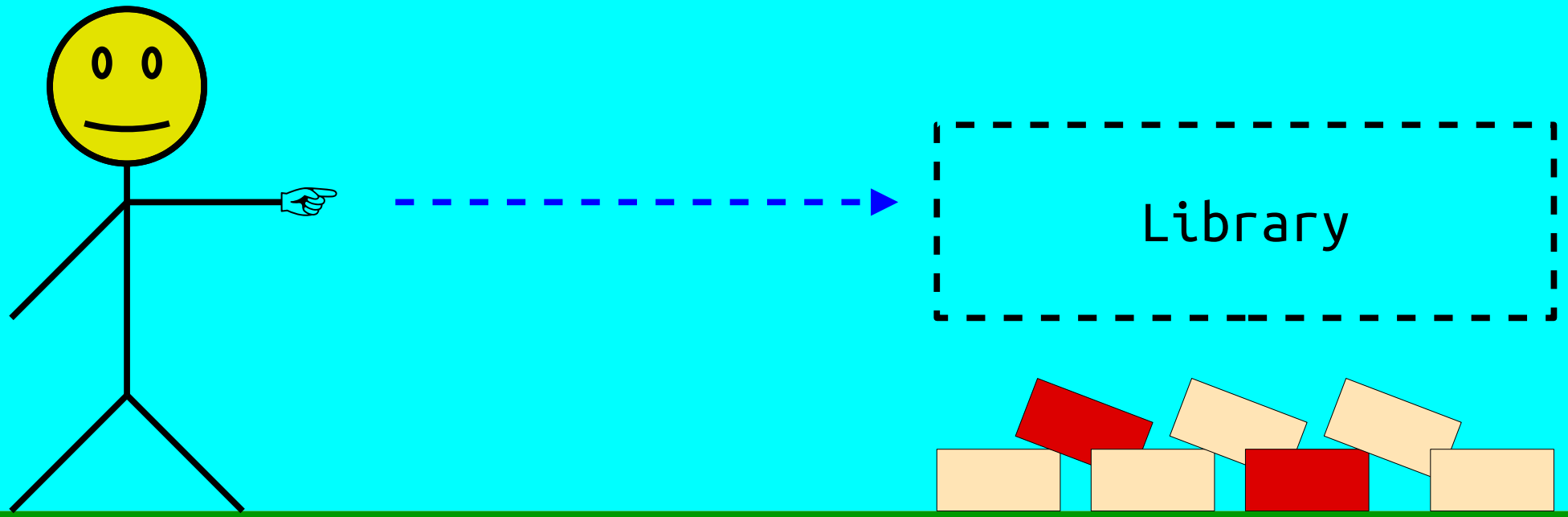
`ptr` still points where the array once was. It's called a ***dangling pointer***.



`ptr`

```
string* ptr = new string[3];
ptr[0] = "Coffee Shop";
ptr[1] = "Office Space";
ptr[2] = "Residential";

delete[] ptr;
ptr[1] = "Library"; // Uh…
```

C++ has no safety checks for reading/writing deallocated memory. It might crash your program, it might do nothing, or might corrupt data.



To Summarize

- You can create arrays of a fixed size at runtime by using `new[]`.
- C++ arrays don't know their lengths and have no bounds-checking. With great power comes great responsibility.
- You are responsible for freeing any memory you explicitly allocate by calling `delete[]`.
- Once you've deleted the memory pointed at by a pointer, you have a dangling pointer and shouldn't read or write from it.

Implementing `Stack`

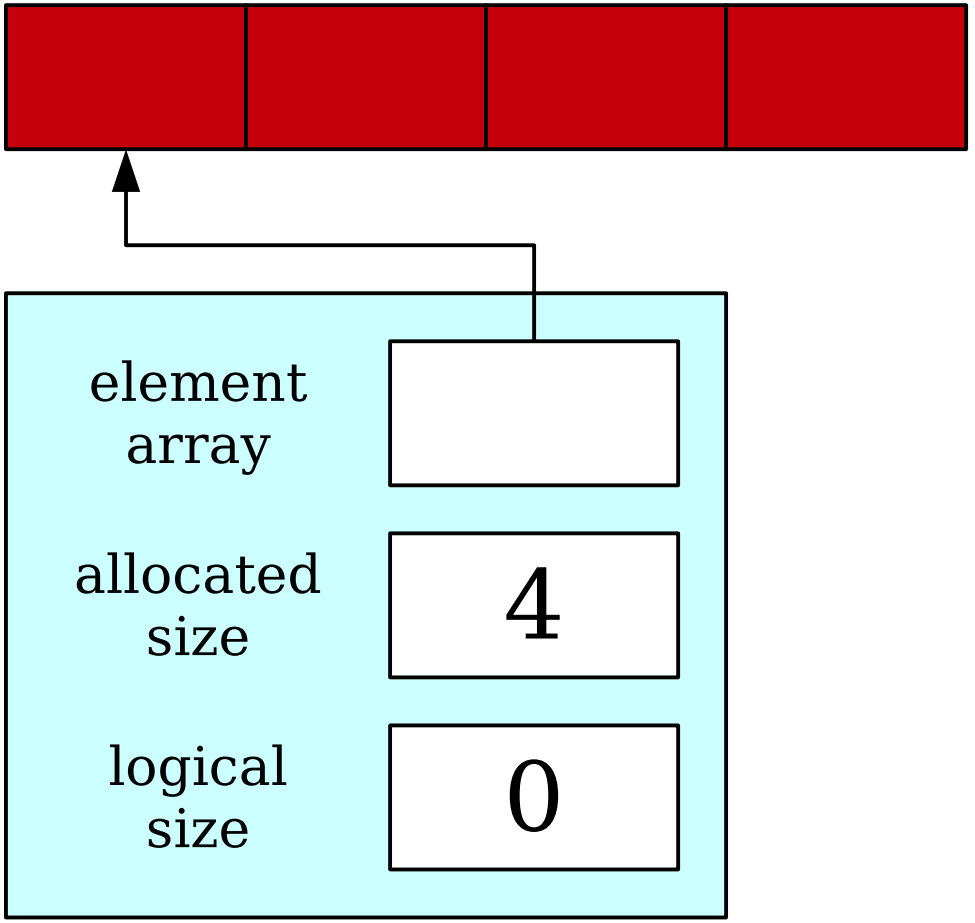
Implementing `Stack`

- Last time, we saw how to implement `RandomBag` in terms of `Vector`.
- We could also implement `Stack` in terms of `Vector`.
- What if we wanted to implement the `Stack` without relying on any other collections?
- Let's build the stack directly!

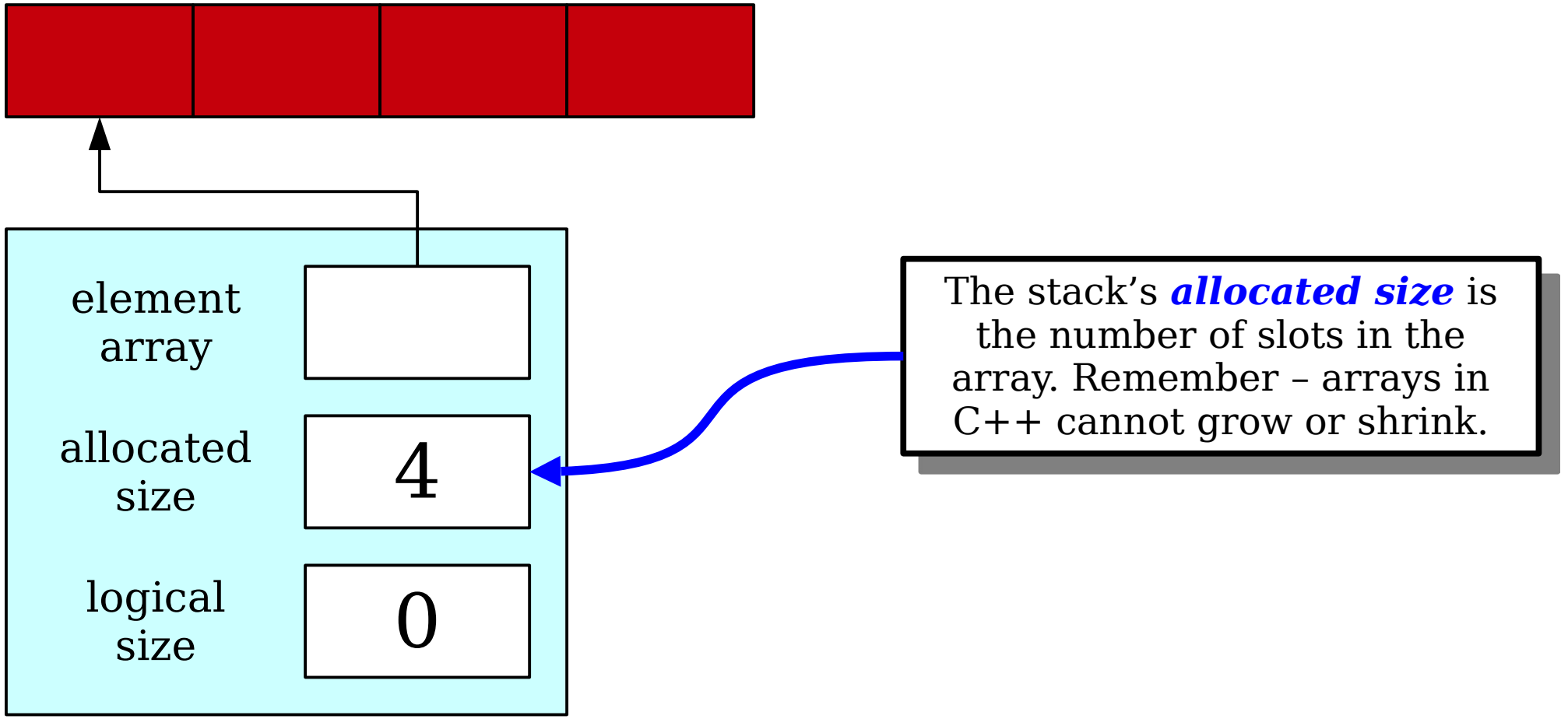
You Gotta Start Somewhere

- Our initial implementation of the stack will be a *bounded* stack with a maximum capacity.
- We'll allocate a fixed amount of storage space for the elements, then write them into the array as they're pushed.
- If we run out of space, we'll report an error.
- Next time, we'll update this code so that we can have a stack without any fixed maximum capacity.

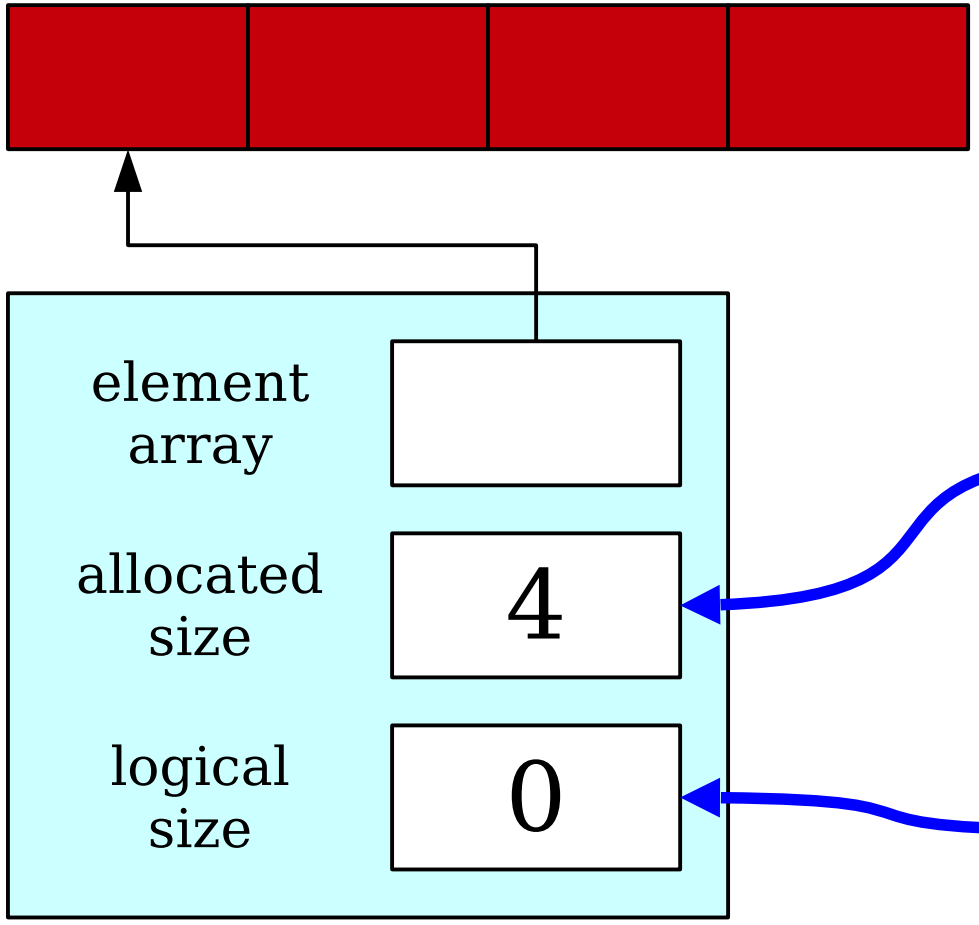
An Initial Idea



An Initial Idea



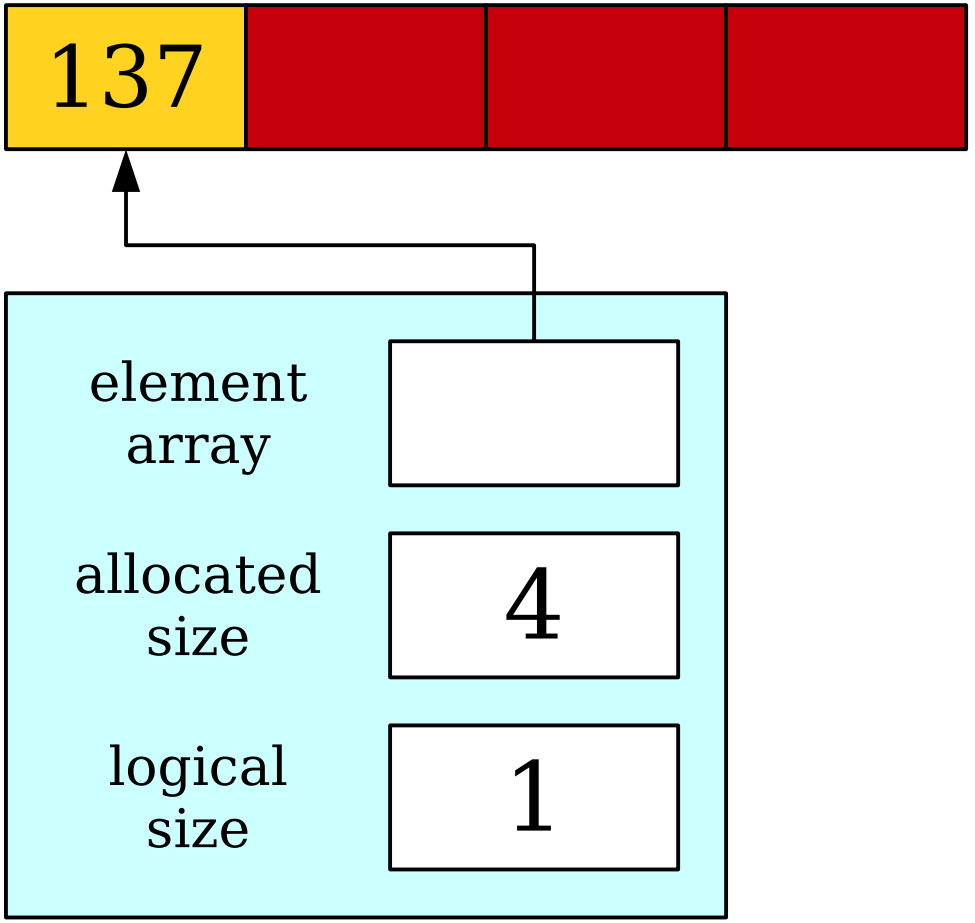
An Initial Idea



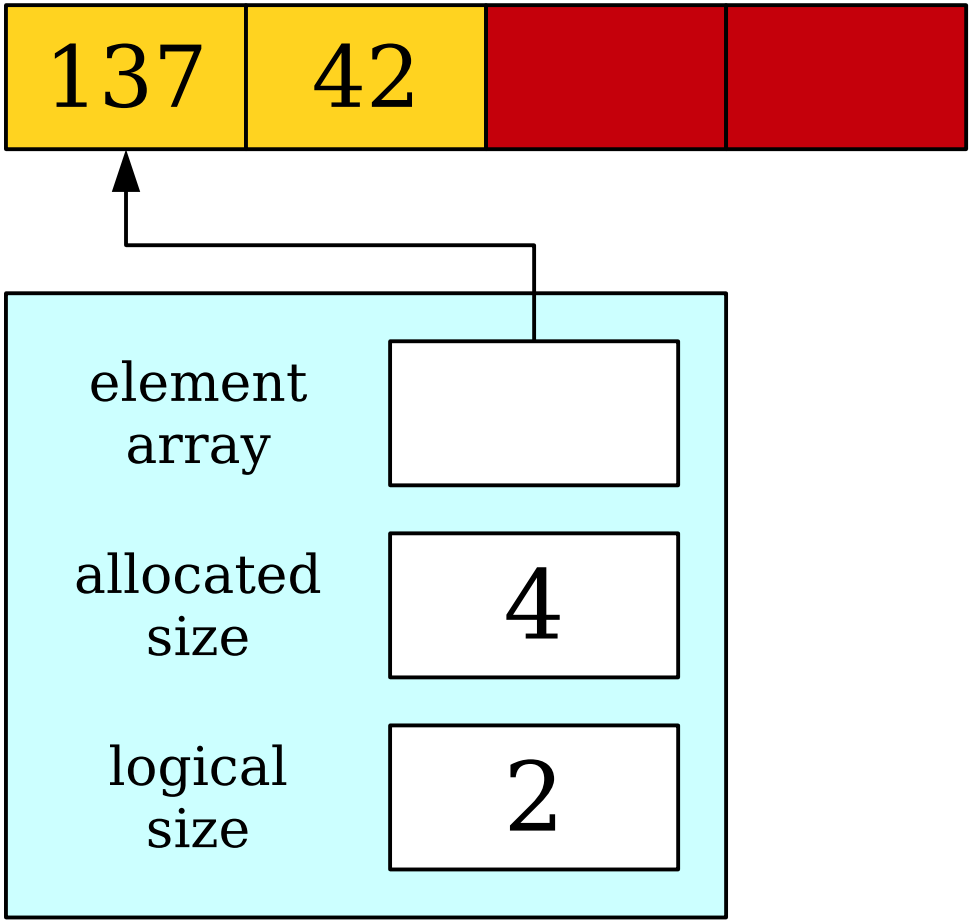
The stack's ***allocated size*** is the number of slots in the array. Remember – arrays in C++ cannot grow or shrink.

The stack's ***logical size*** is the number of elements actually stored in the stack. This lets us track how much space we're actually using.

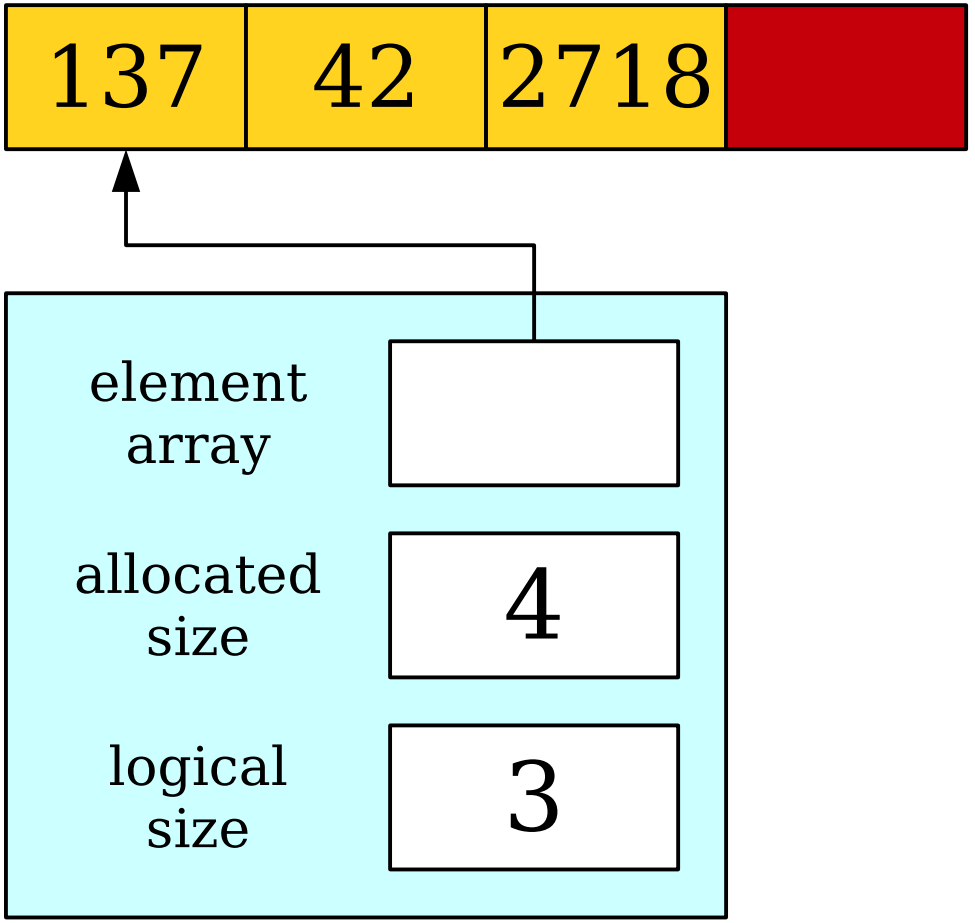
An Initial Idea



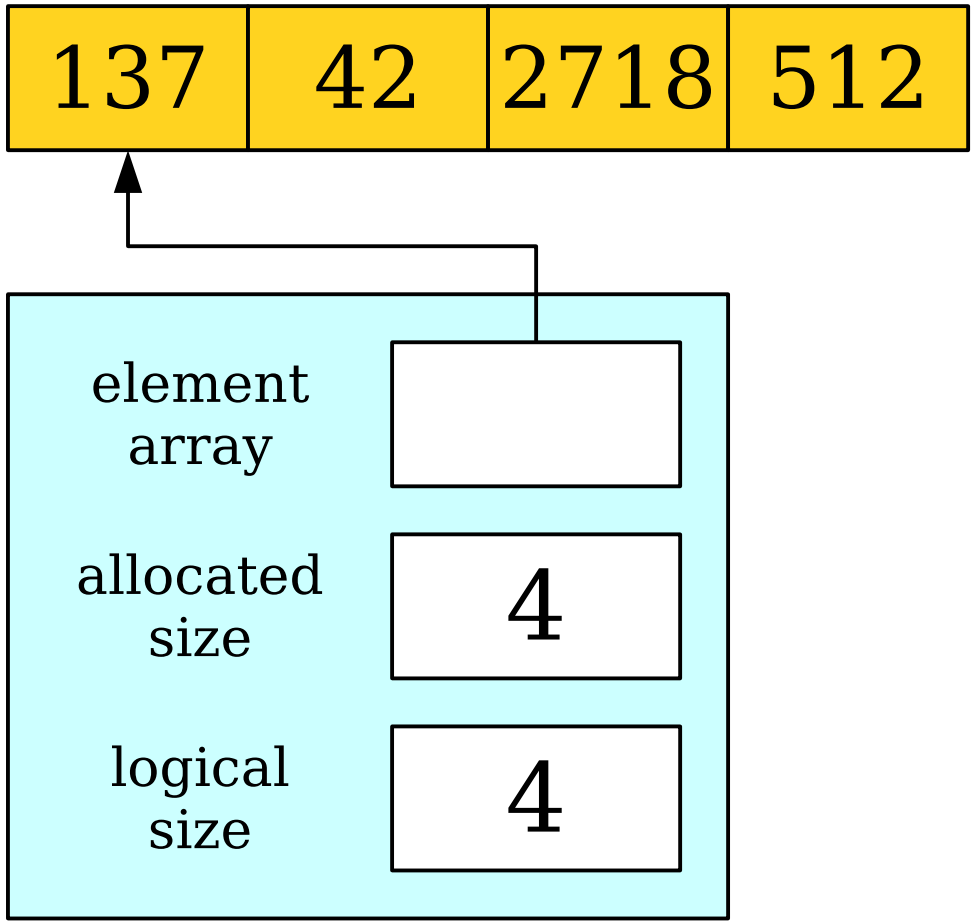
An Initial Idea



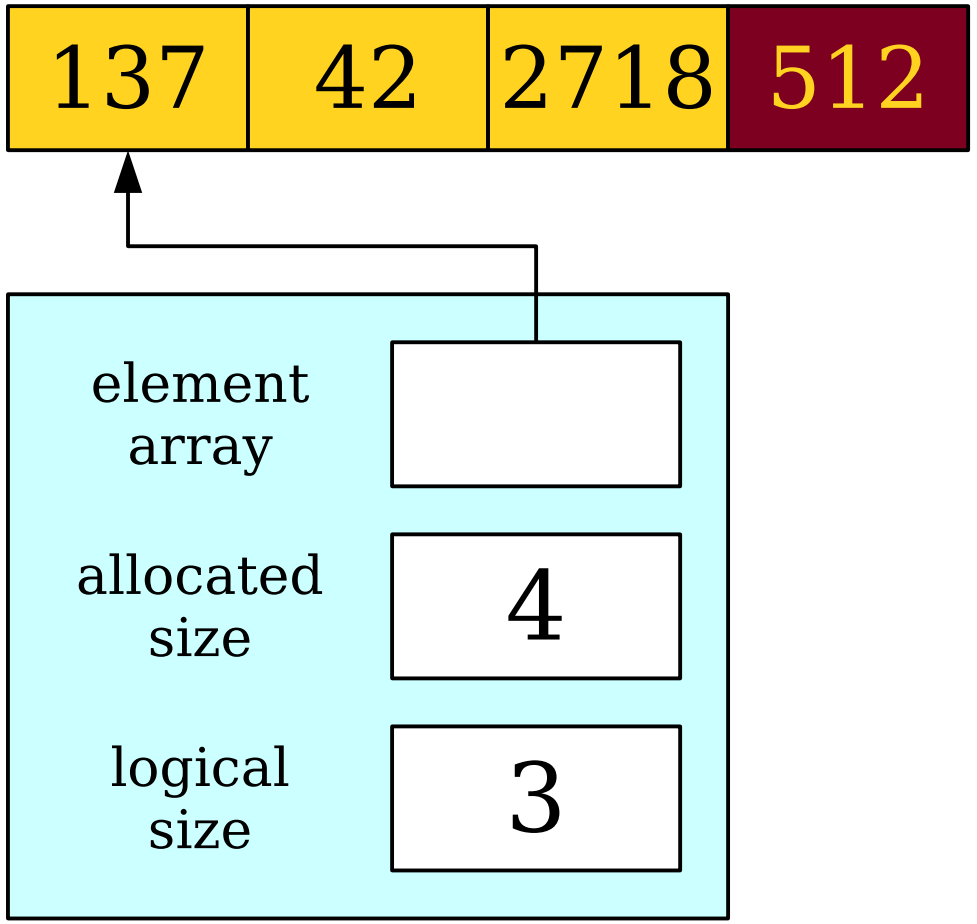
An Initial Idea



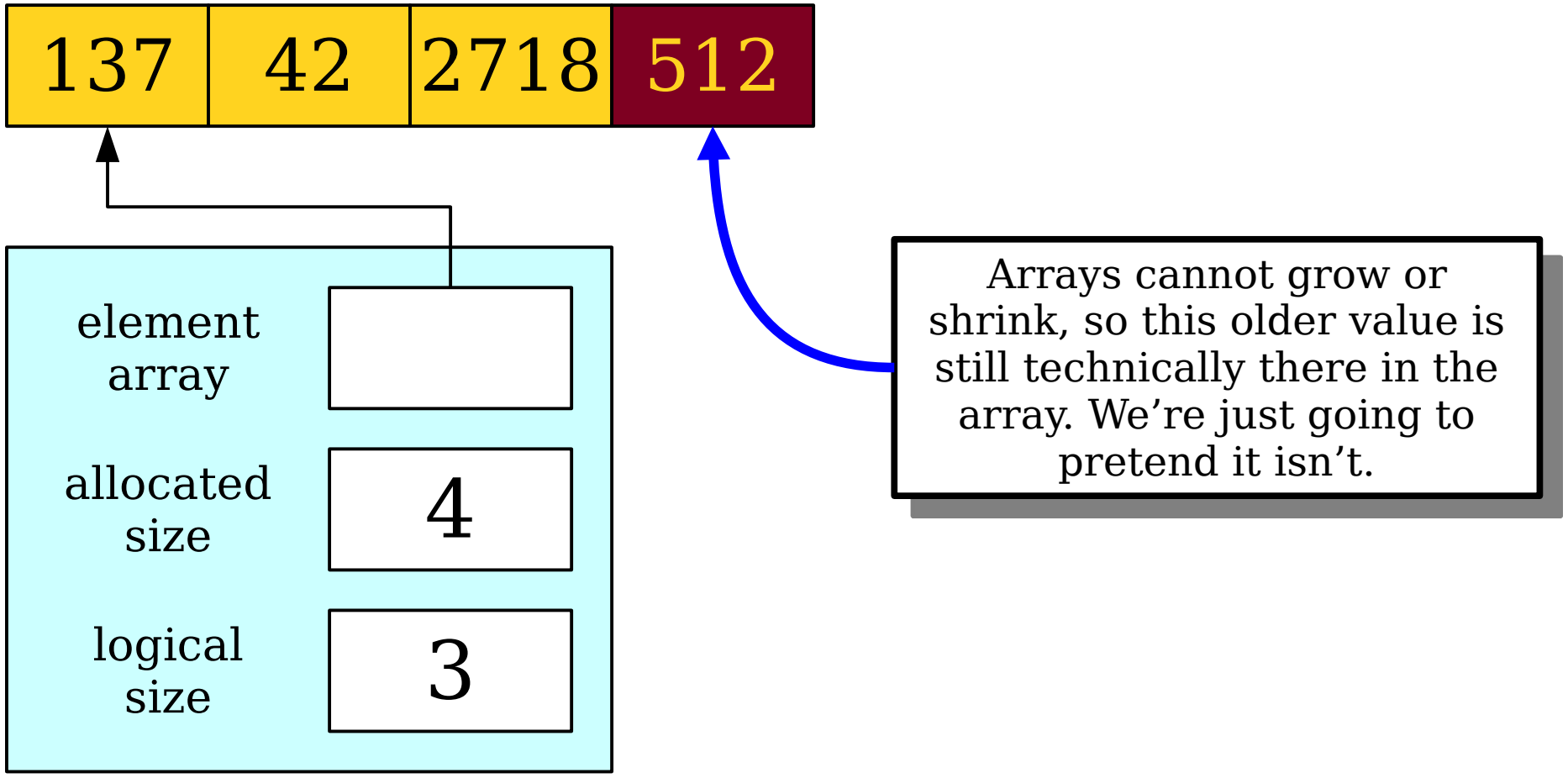
An Initial Idea



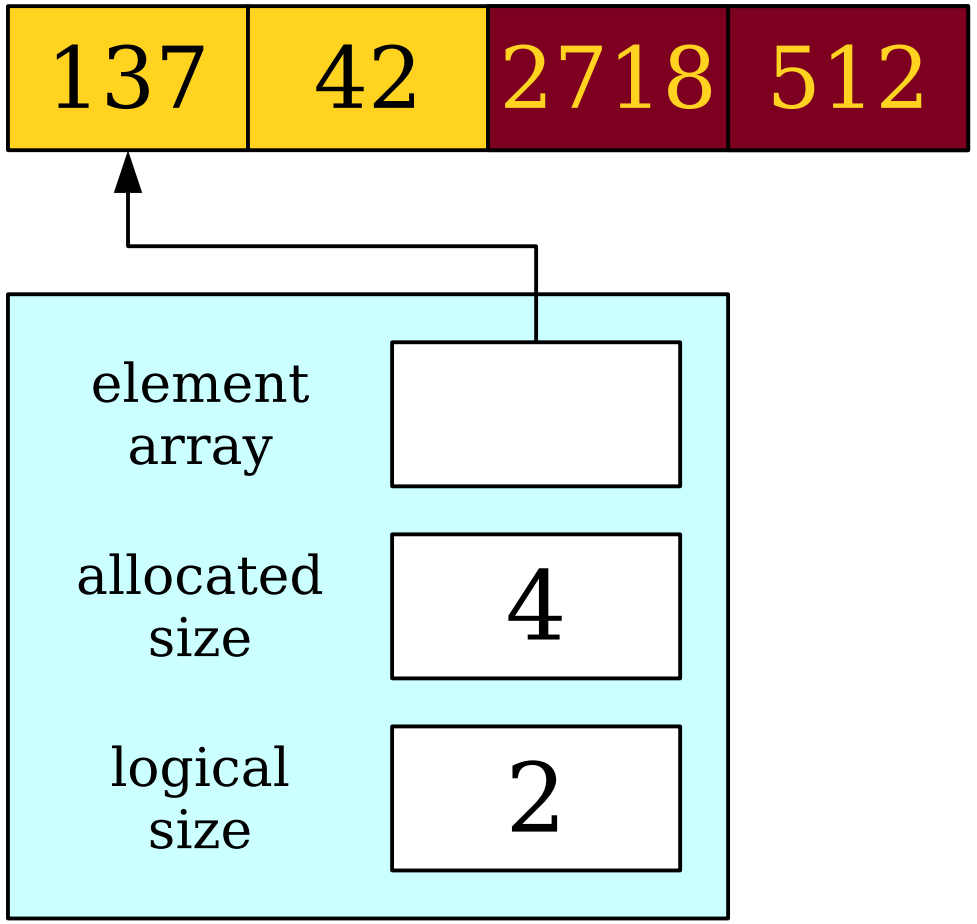
An Initial Idea



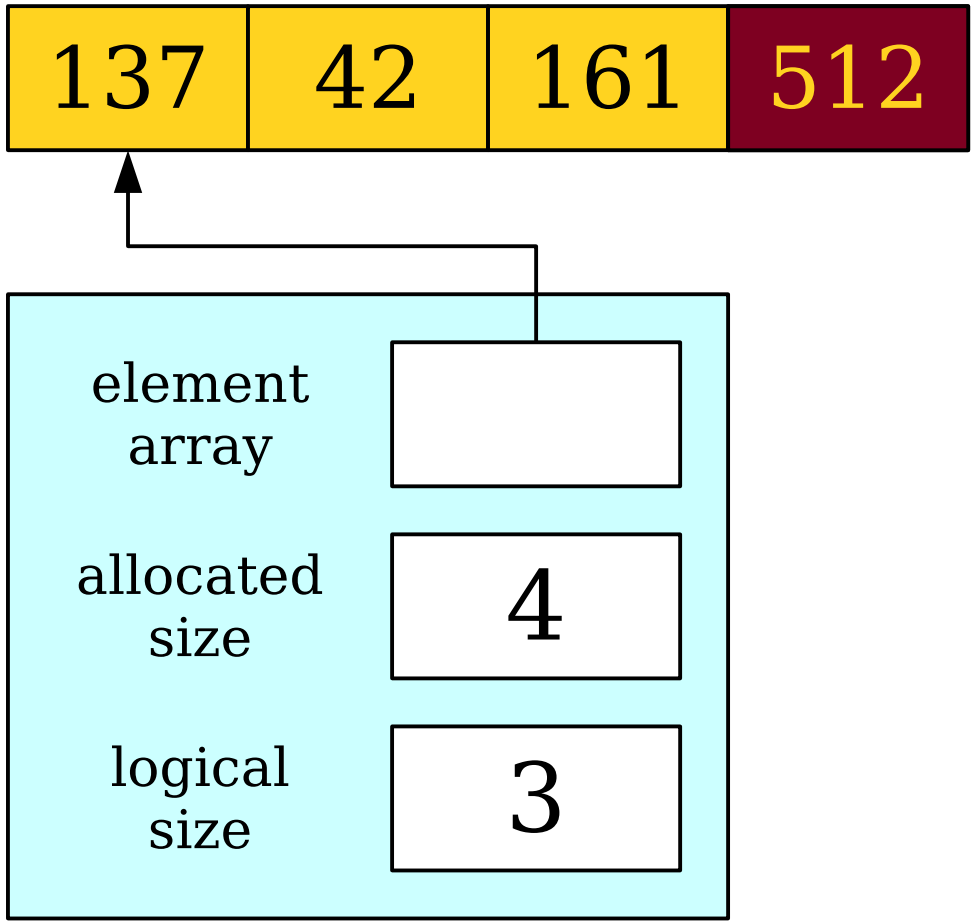
An Initial Idea



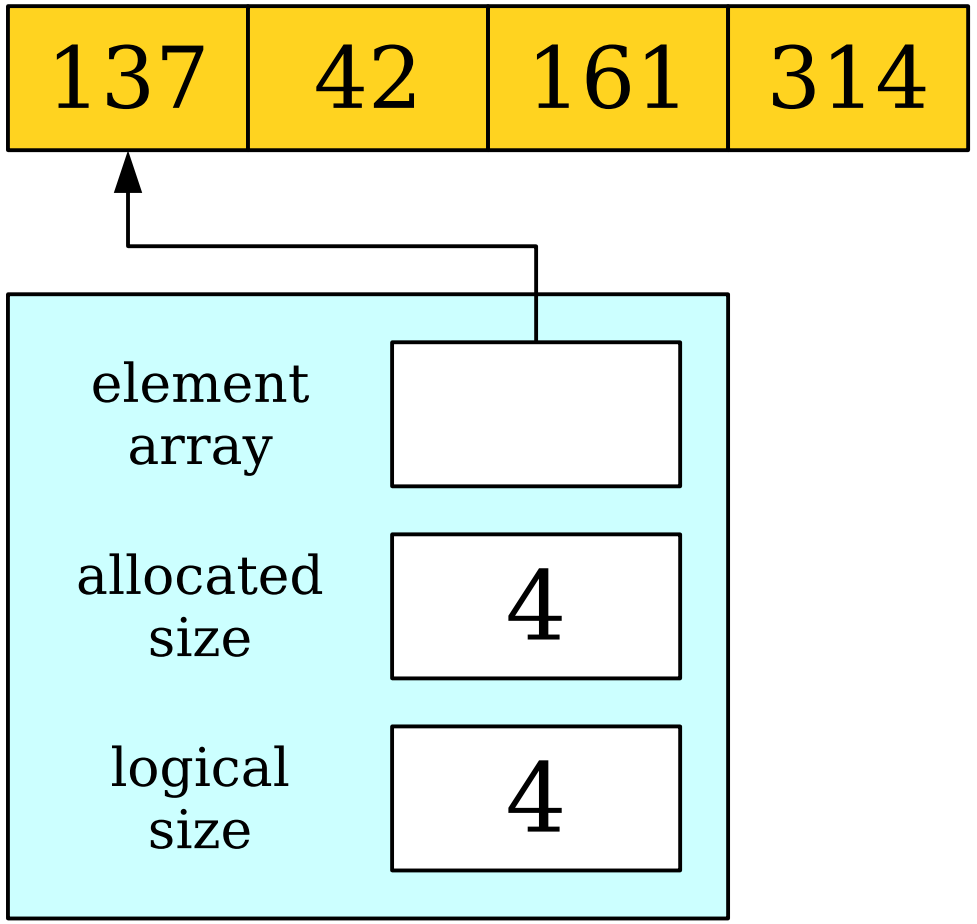
An Initial Idea



An Initial Idea



An Initial Idea



What We Have

Before We Start: A Problem

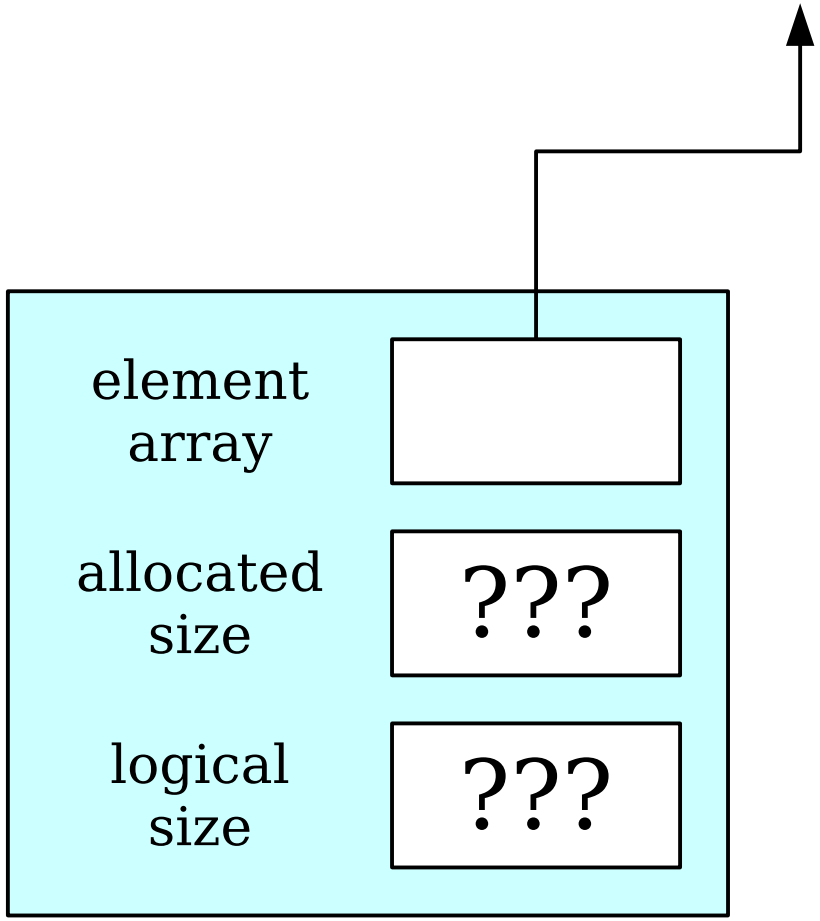
Cradle to Grave

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave

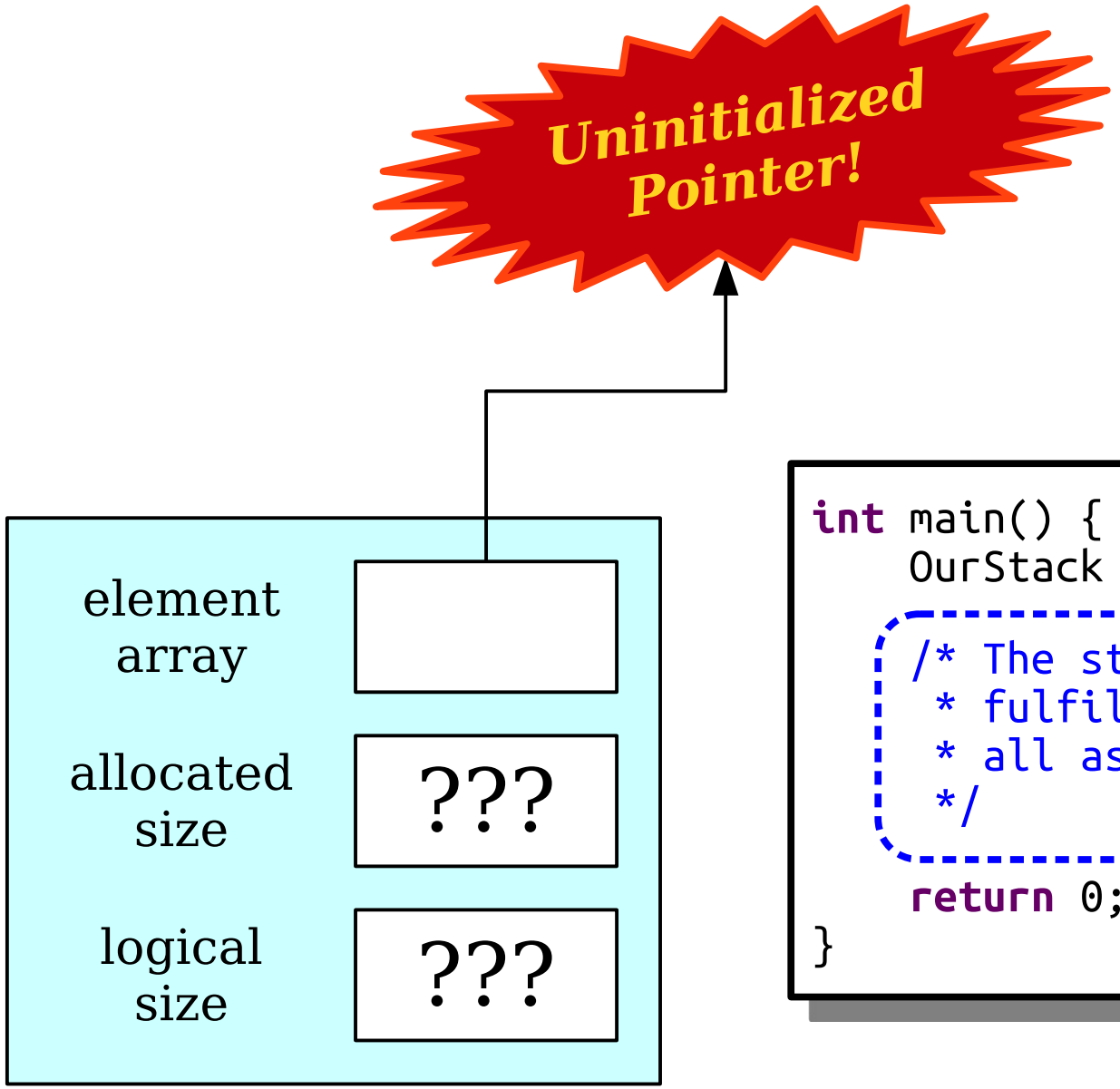
```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave




```
int main() {
   OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave




```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Constructors

- A ***constructor*** is a special member function used to set up the class before it is used.
- The constructor is automatically called when the object is created.
- The constructor for a class named ***ClassName*** has signature

  ***ClassName*(*args*);**

```
class OurStack {
public:


    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

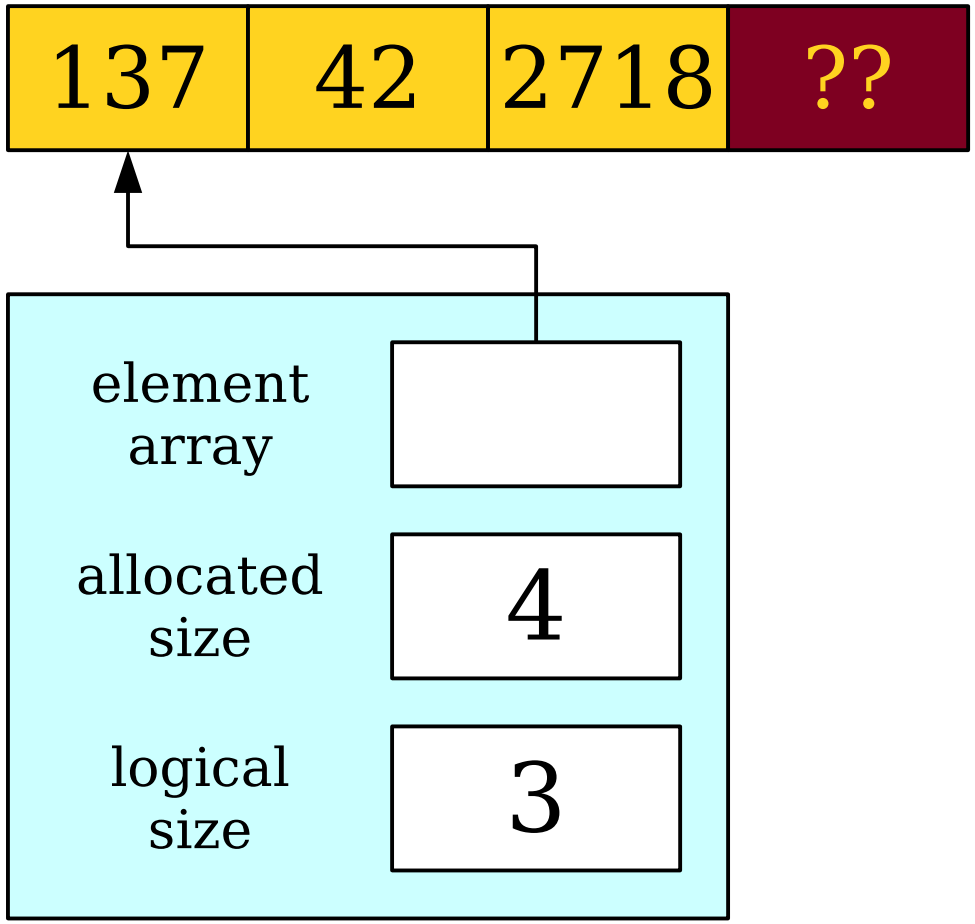
Constructors

- A ***constructor*** is a special member function used to set up the class before it is used.
- The constructor is automatically called when the object is created.
- The constructor for a class named ***ClassName*** has signature

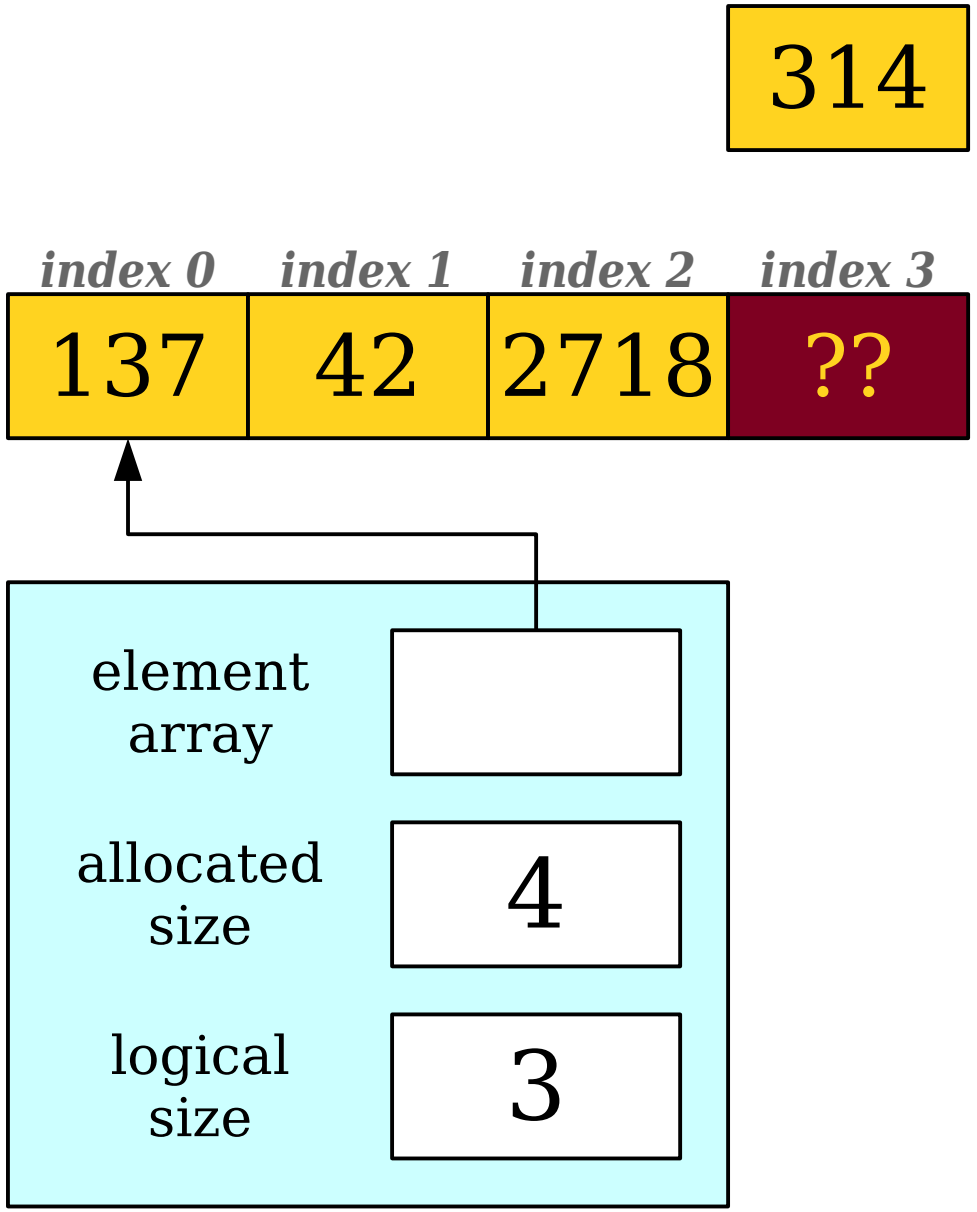
  ***ClassName*(*args*);**

```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

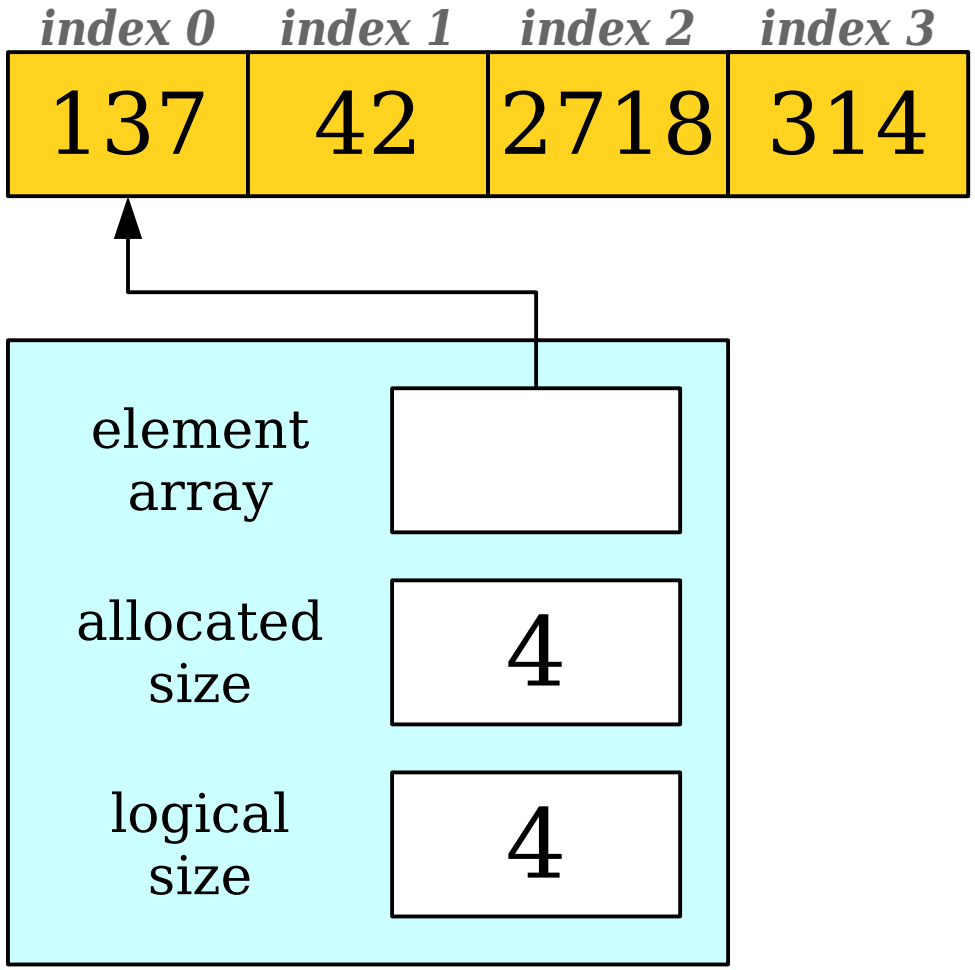
Implementing our Operations



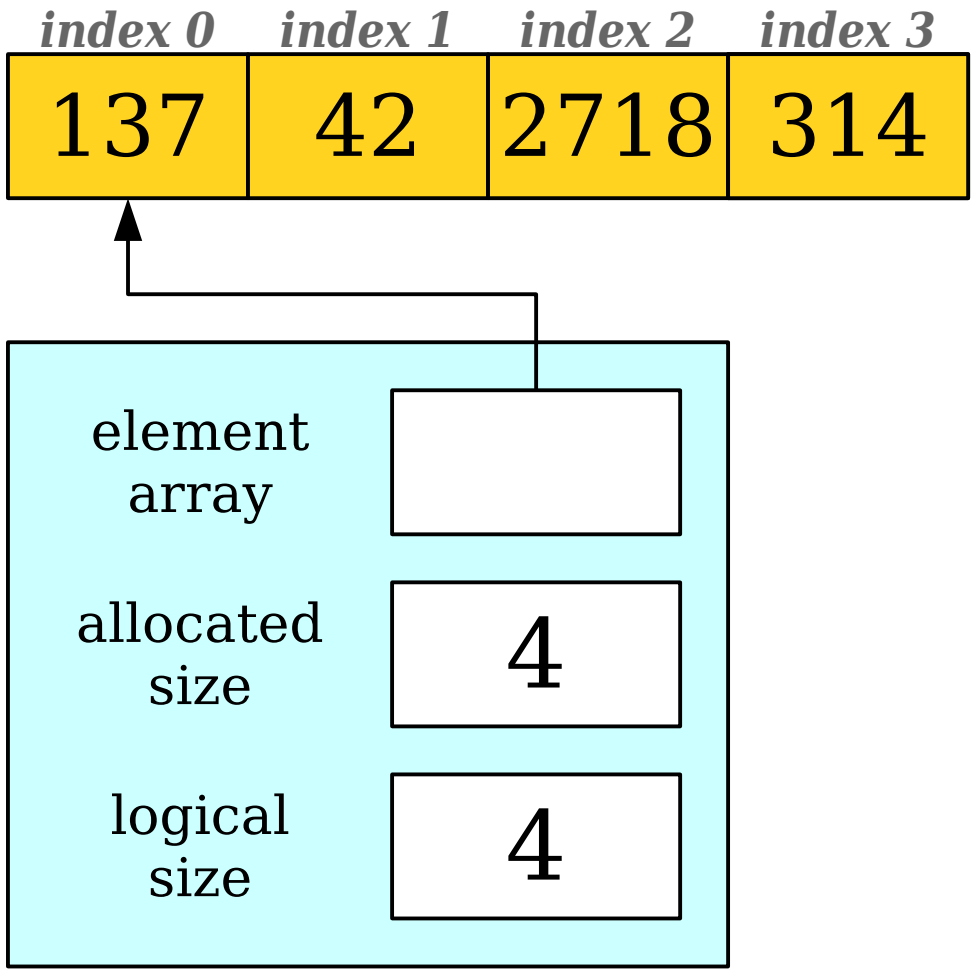

```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```



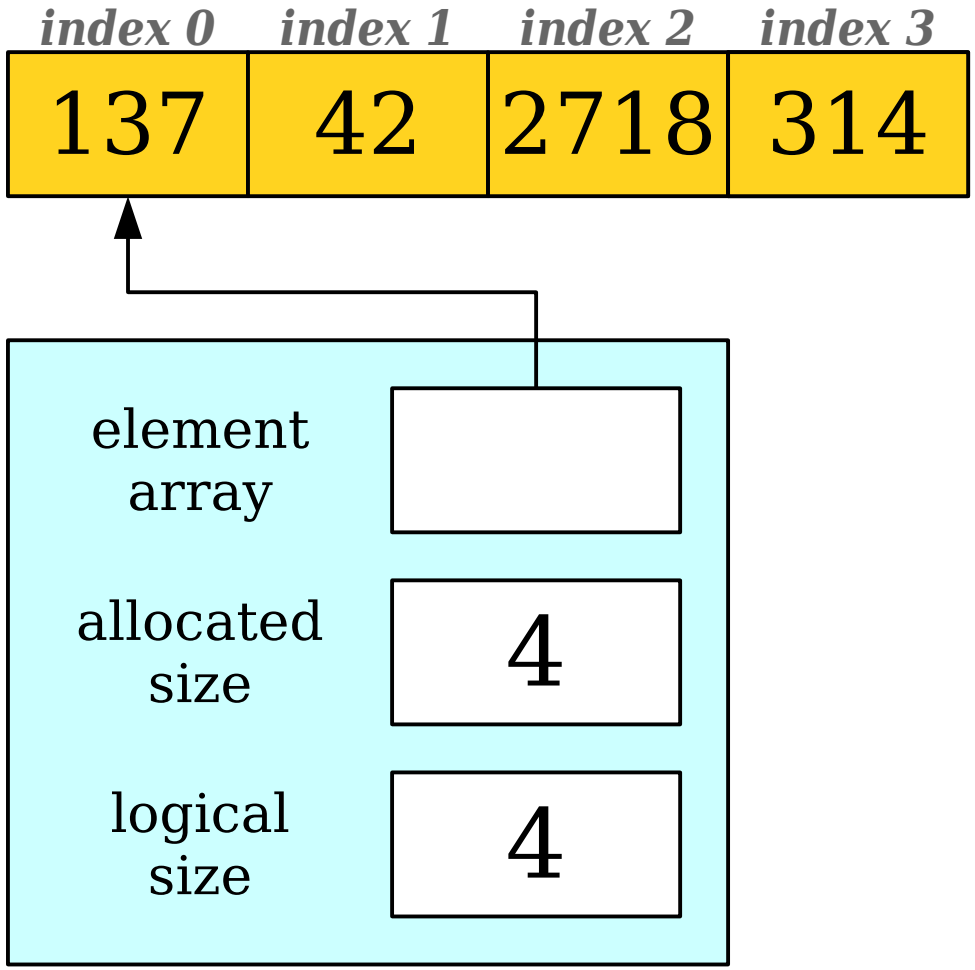

```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```



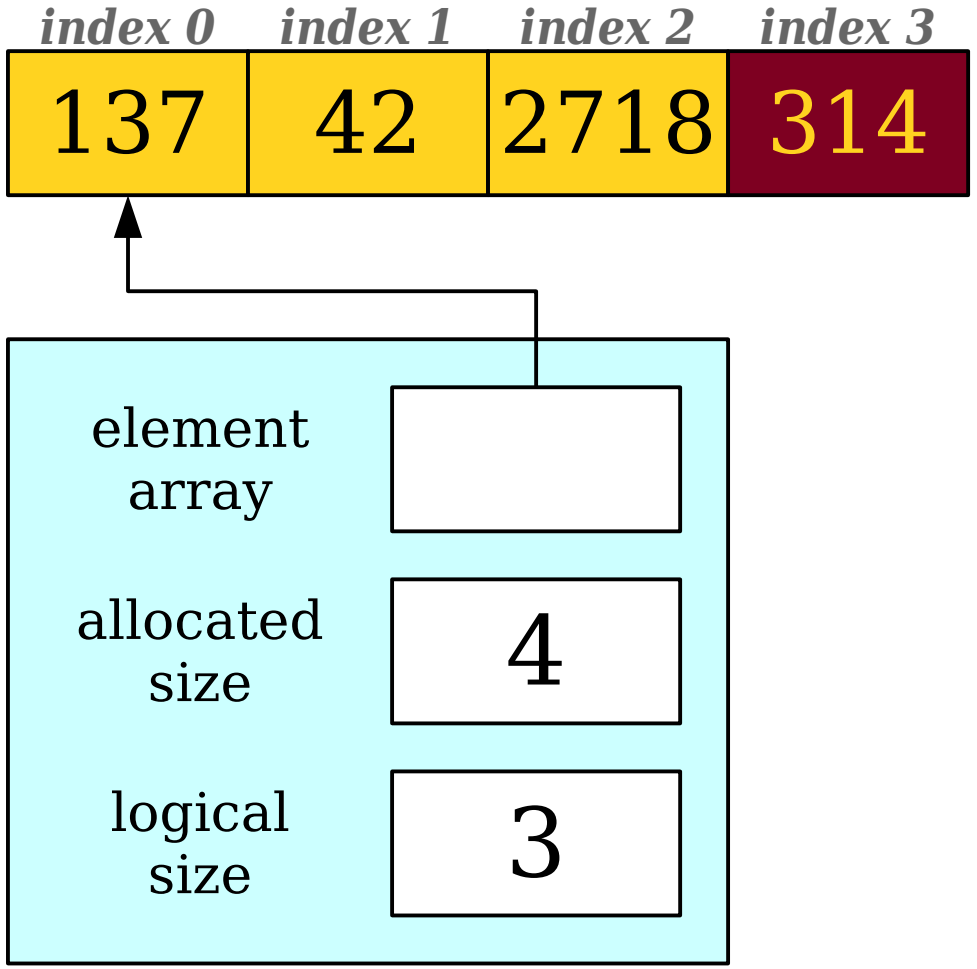

```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```



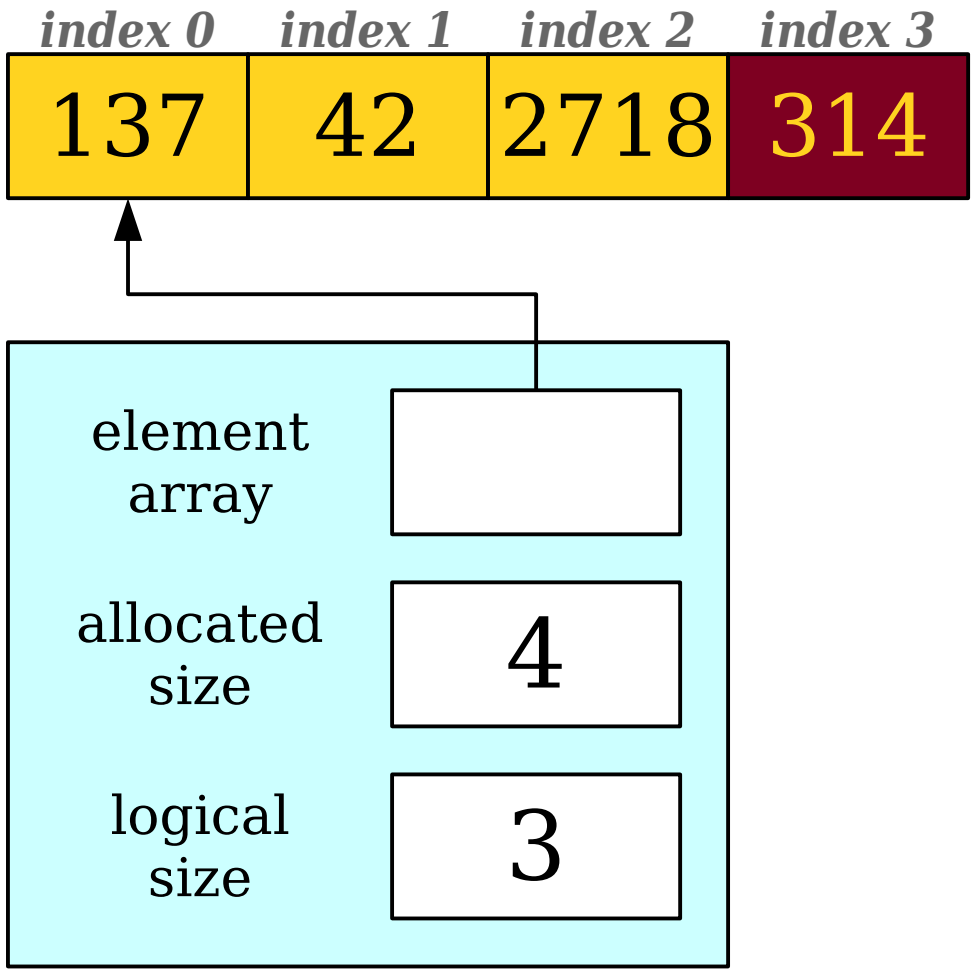

```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```




```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```




```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```




```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

So… we’re done?

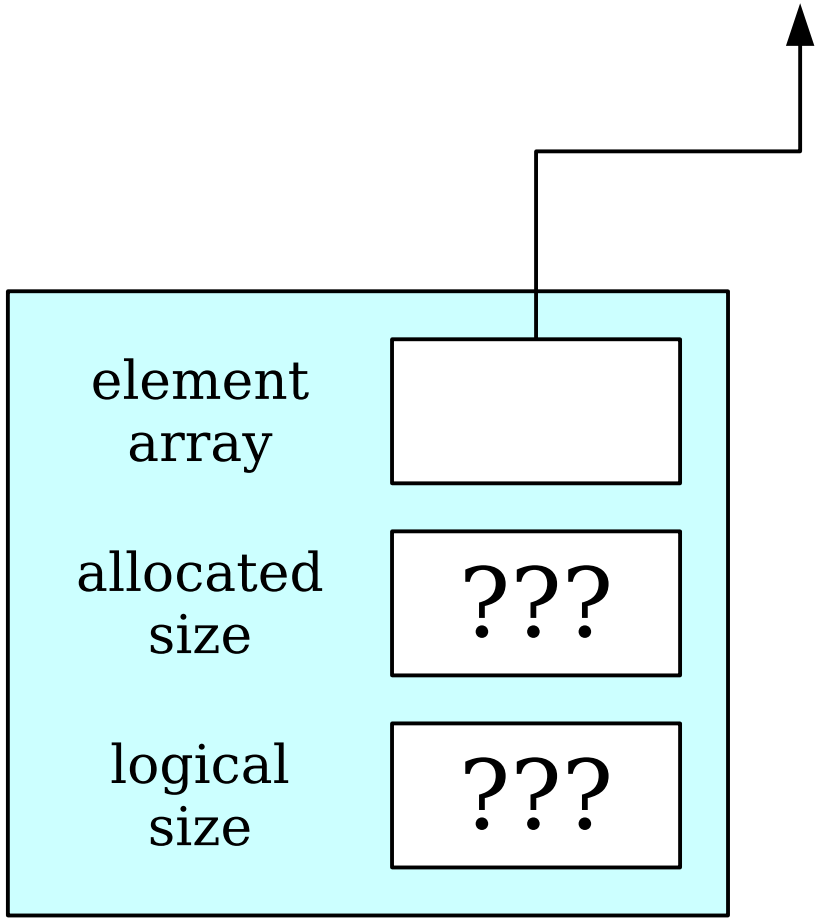
Cradle to Grave, Take II

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave, Take II

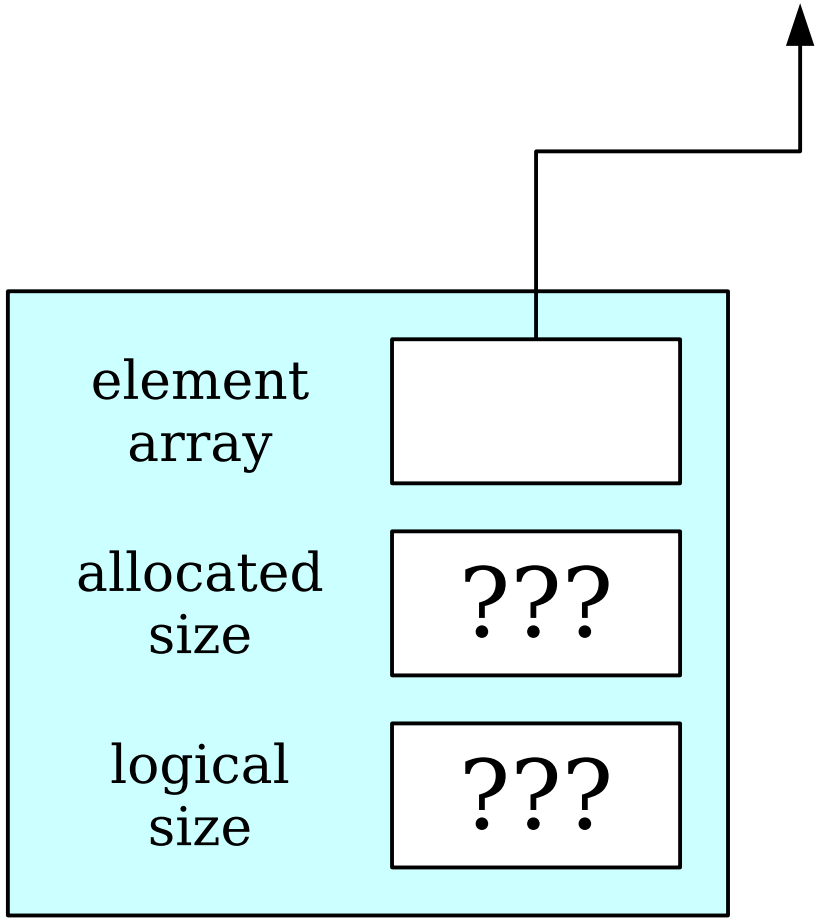
```
int main() {
   OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave, Take II



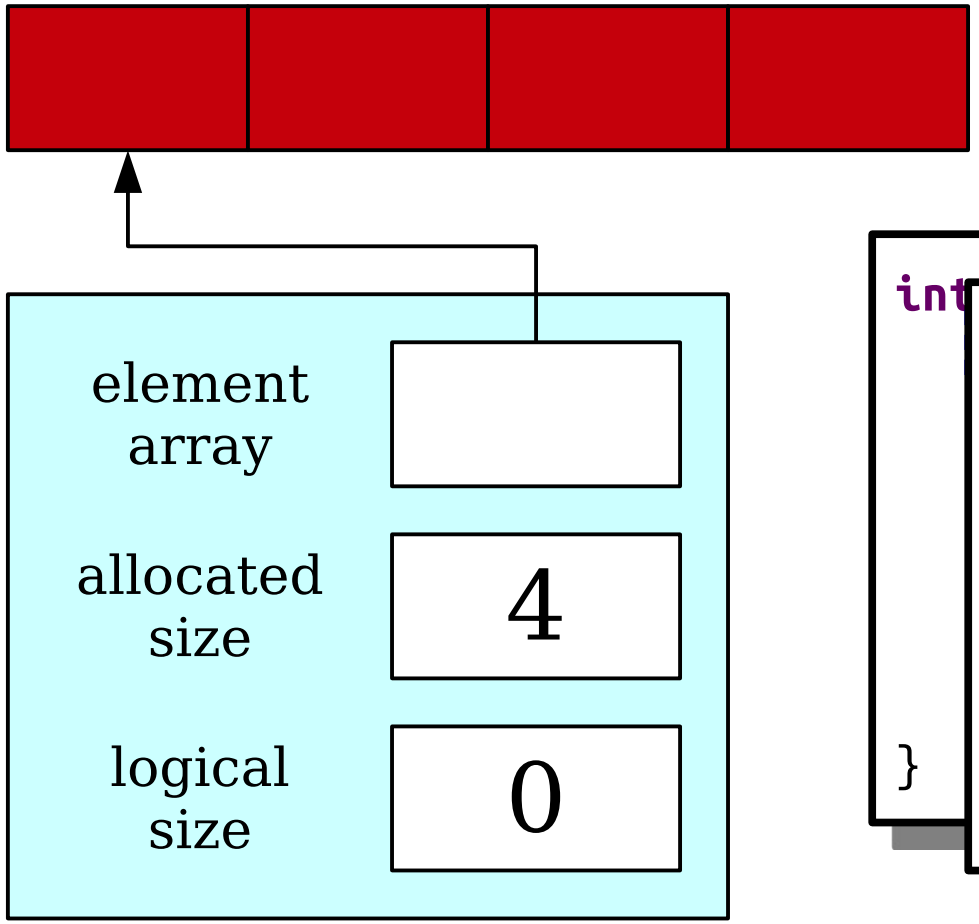

```
int main() {
   OurStack stack;

   /* The stack lives a rich, happy,
    * fulfilling life, the kind we
    * all aspire to.
    */

   return 0;
}
```

Cradle to Grave, Take II




```
int
}

OurStack::OurStack() {
    logicalSize = 0;
    allocatedSize = kInitialSize;
    elems = new int[allocatedSize];
}
```

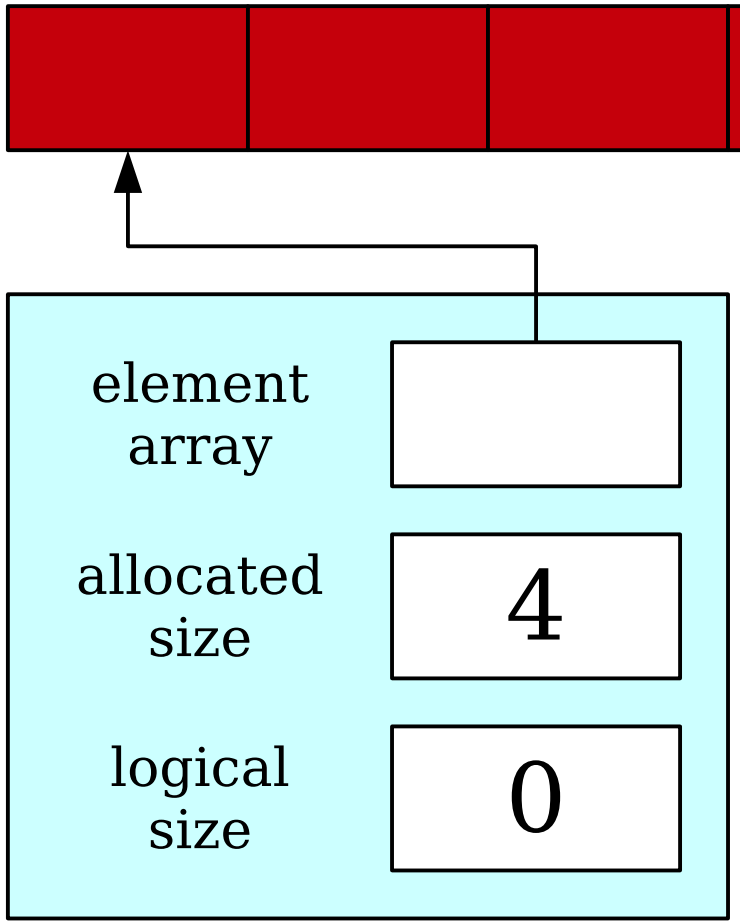
Cradle to Grave, Take II




```
int
}
```

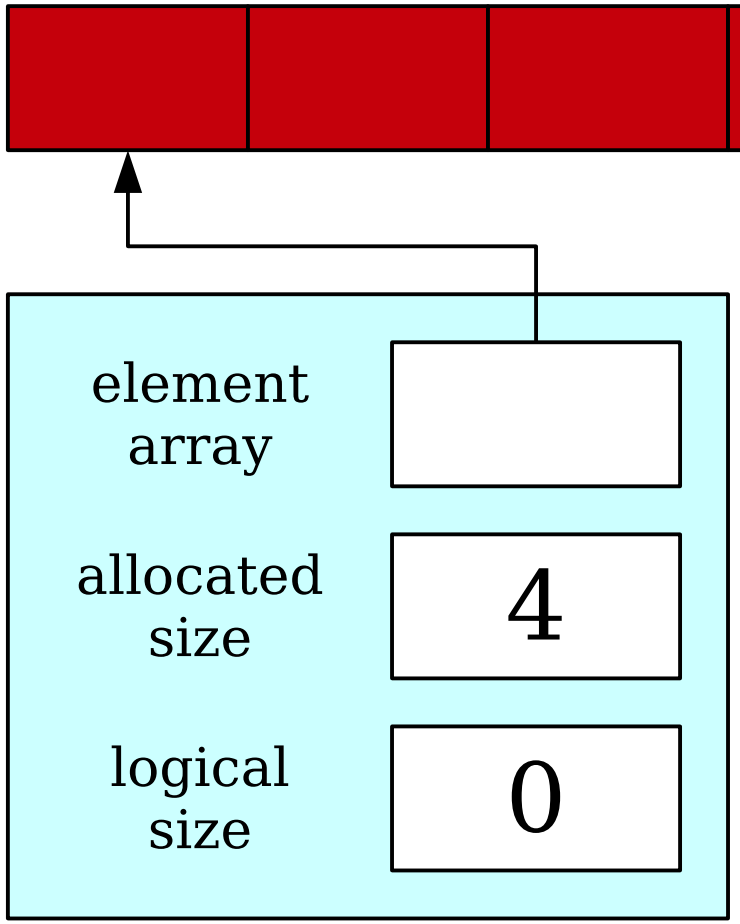
```
OurStack::OurStack() {
    logicalSize = 0;
    allocatedSize = kInitialSize;
    elems = new int[allocatedSize];
}
```

Cradle to Grave, Take II



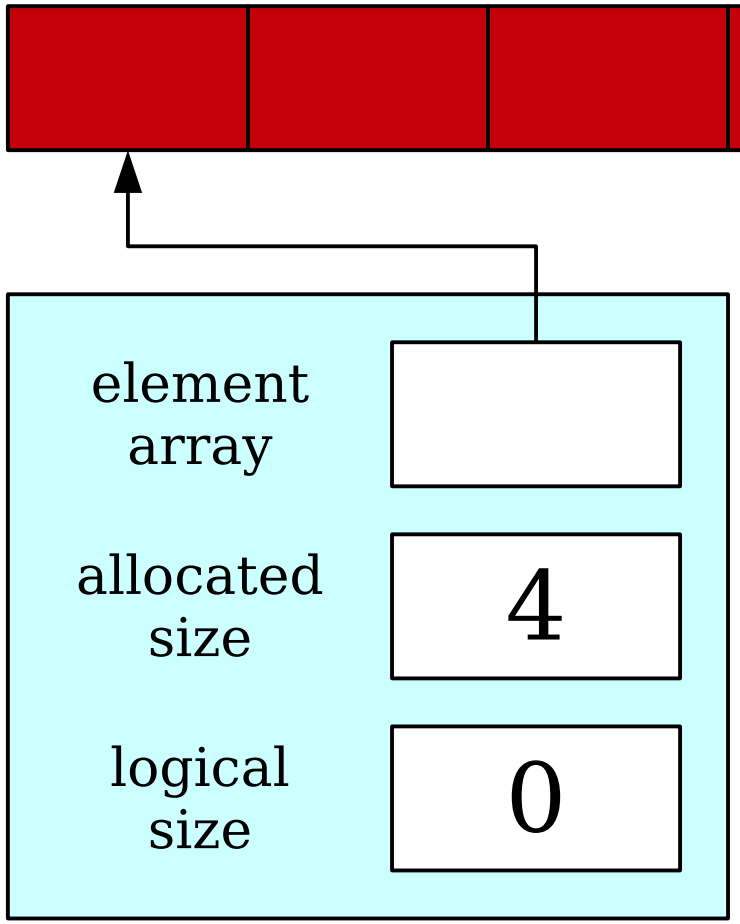

```
int main() {
   OurStack stack;

   /* The stack lives a rich, happy,
    * fulfilling life, the kind we
    * all aspire to.
    */

   return 0;
}
```

Cradle to Grave, Take II



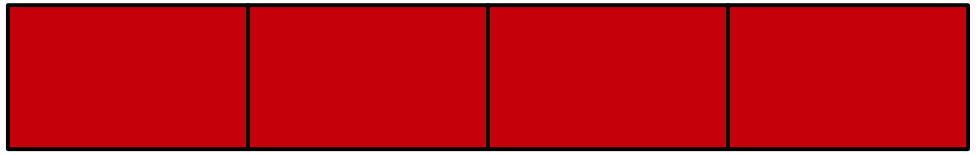

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave, Take II



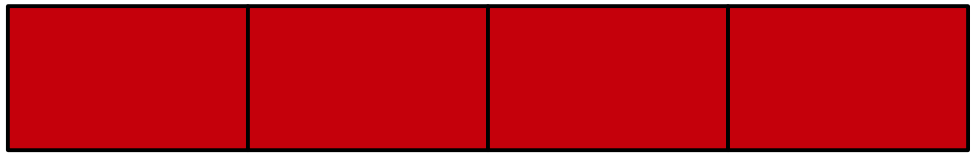

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave, Take II



```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave, Take II




```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Destructors

- A ***destructor*** is a special member function responsible for cleaning up an object's memory.
- It's automatically called whenever an object's lifetime ends (for example, if it's a local variable that goes out of scope.)
- The destructor for a class named ***ClassName*** has signature

  ~***ClassName***();

```
class OurStack {
public:
    OurStack();


    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

Destructors

- A ***destructor*** is a special member function responsible for cleaning up an object's memory.
- It's automatically called whenever an object's lifetime ends (for example, if it's a local variable that goes out of scope.)
- The destructor for a class named ***ClassName*** has signature

  ~***ClassName***();

```
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

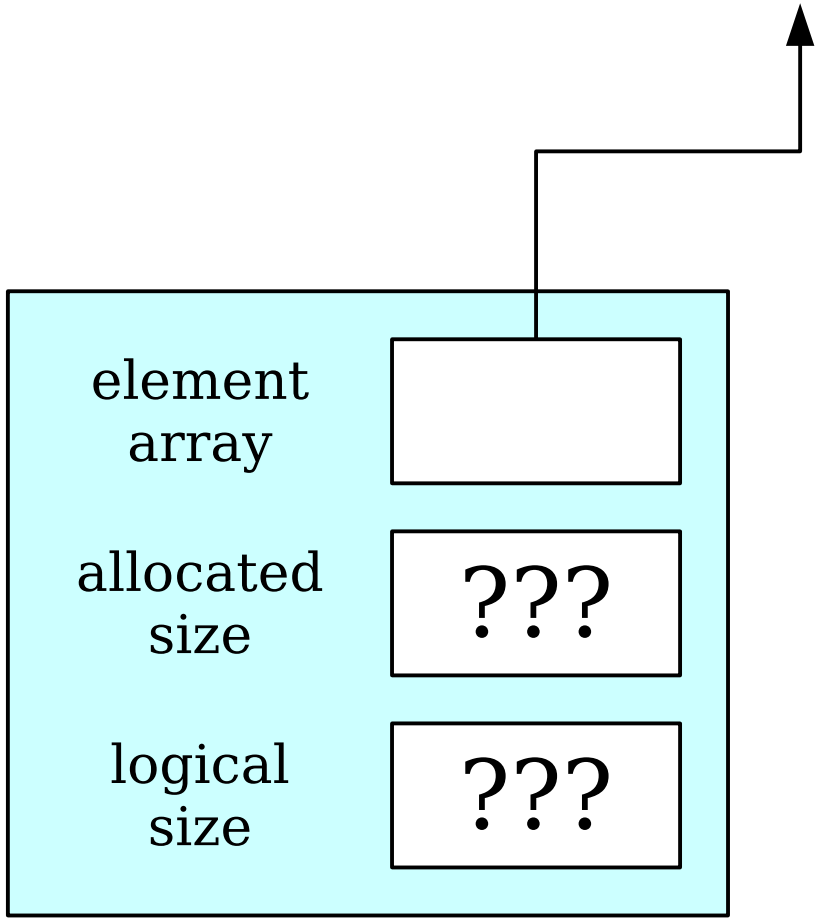
Cradle to Grave, Take III

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave, Take III

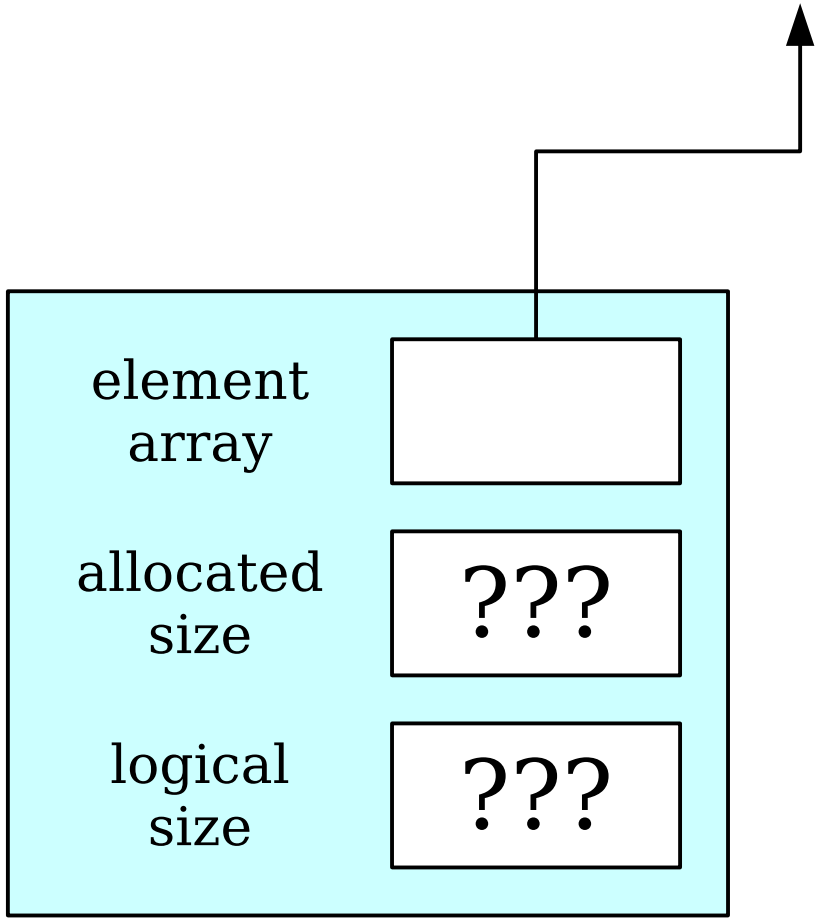
```
int main() {
   OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave, Take III



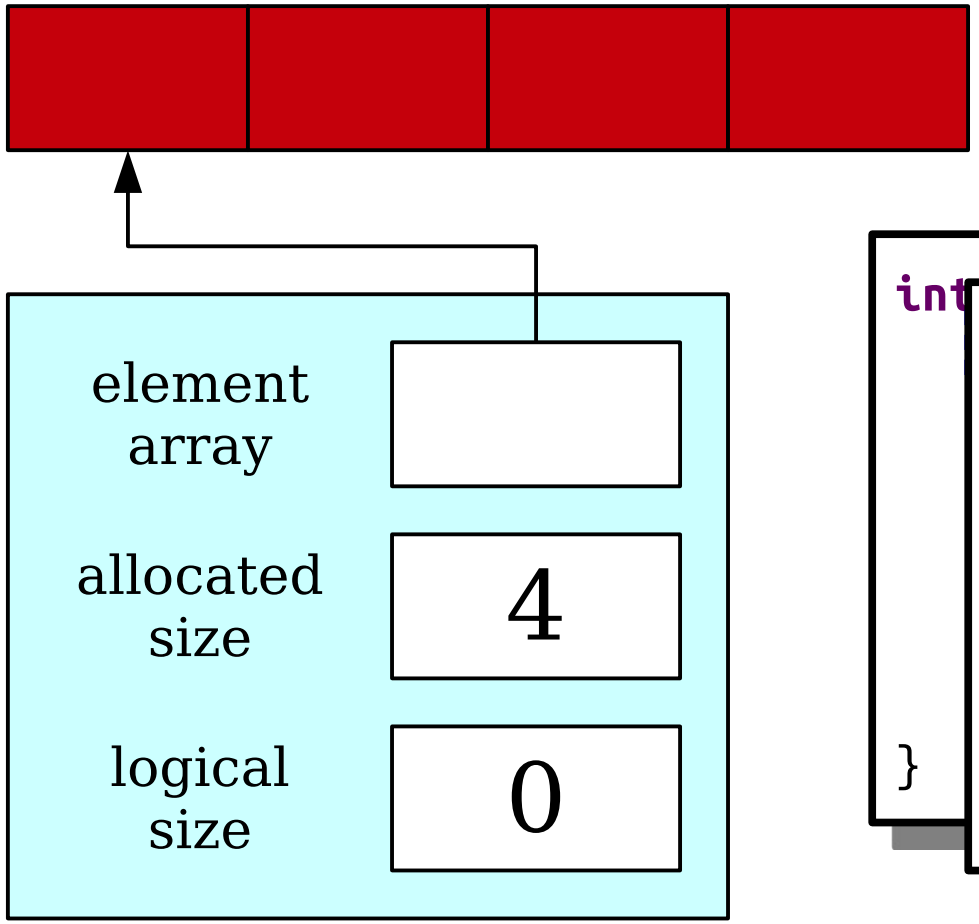

```
int main() {
   OurStack stack;

   /* The stack lives a rich, happy,
    * fulfilling life, the kind we
    * all aspire to.
    */

   return 0;
}
```

Cradle to Grave, Take III




```
int

OurStack::OurStack() {
    logicalSize = 0;
    allocatedSize = kInitialSize;
    elems = new int[allocatedSize];
}

}
```

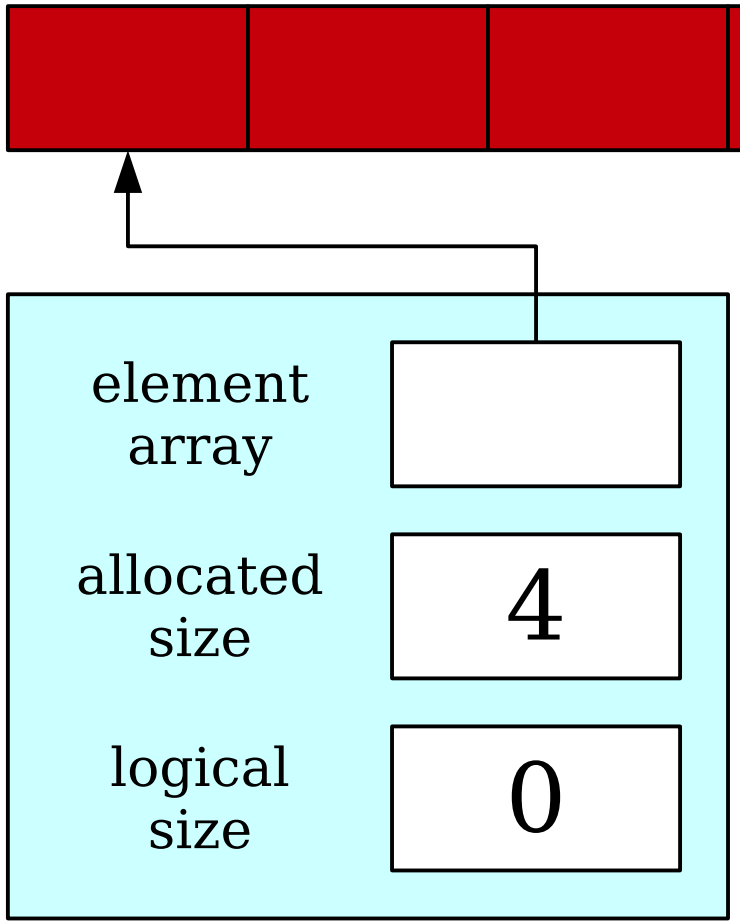
Cradle to Grave, Take III




```
int

}
```

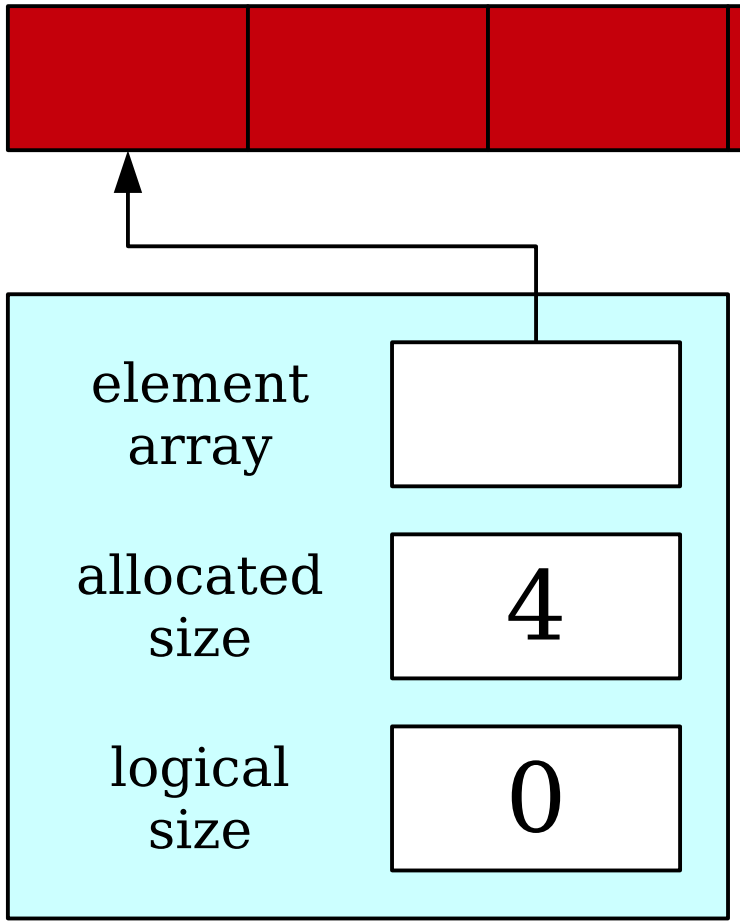
```
OurStack::OurStack() {
    logicalSize = 0;
    allocatedSize = kInitialSize;
    elems = new int[allocatedSize];
}
```

Cradle to Grave, Take III



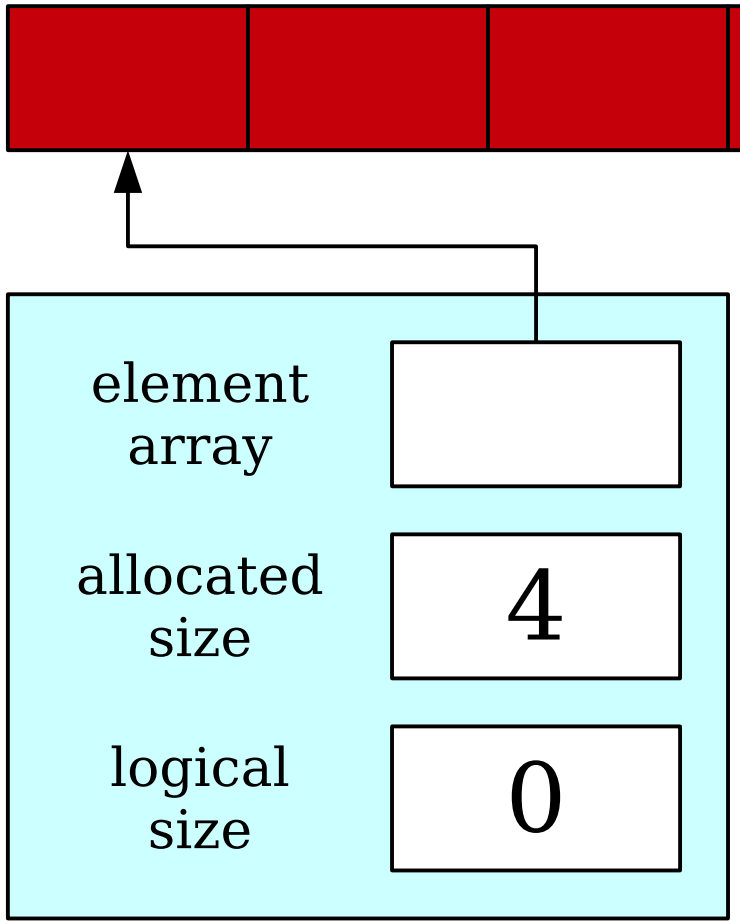

```
int main() {
   OurStack stack;

   /* The stack lives a rich, happy,
    * fulfilling life, the kind we
    * all aspire to.
    */

   return 0;
}
```

Cradle to Grave, Take III



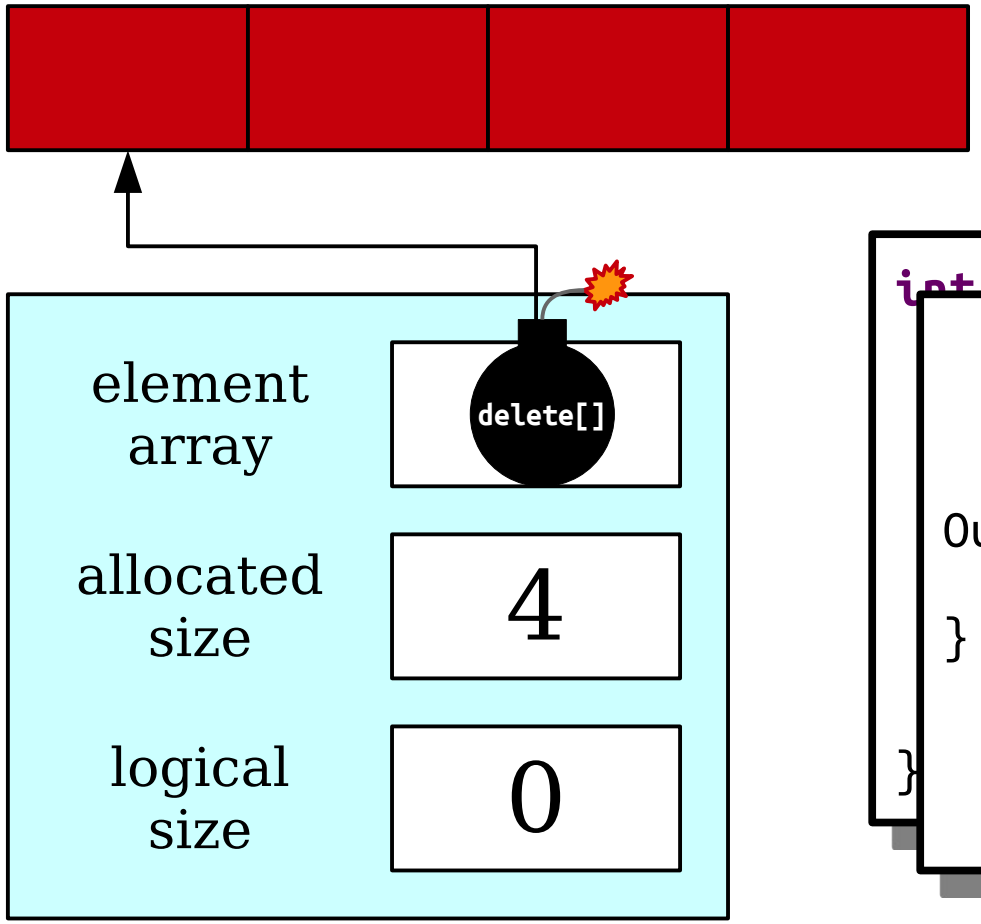

```
int main() {
    OurStack stack;
    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */
    return 0;
}
```

Cradle to Grave, Take III




```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

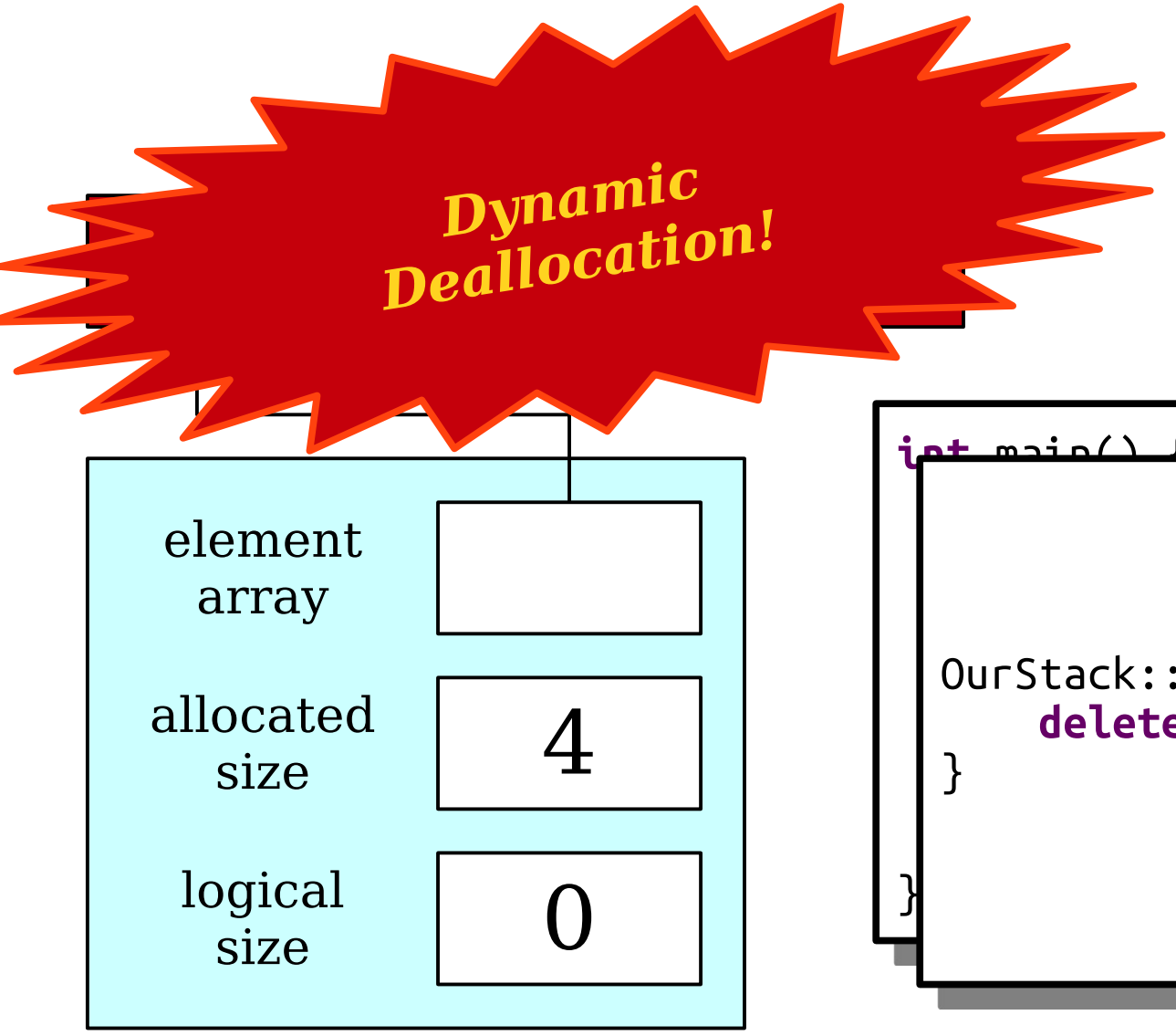
Cradle to Grave, Take III




```
int main() {


}
```

```
OurStack::~OurStack() {
    delete[] elems;
}
```

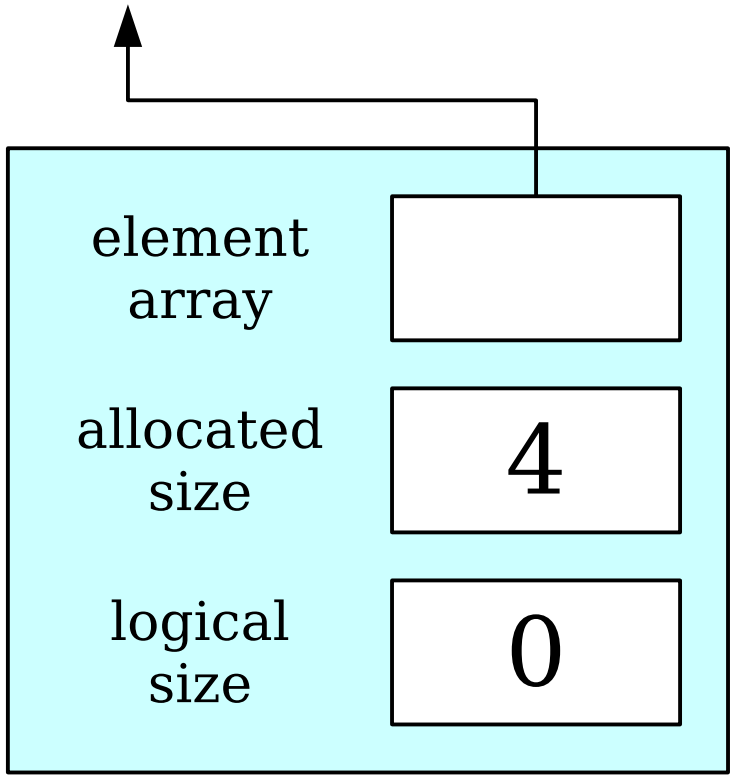
Cradle to Grave, Take III




```
int main() {


}
```

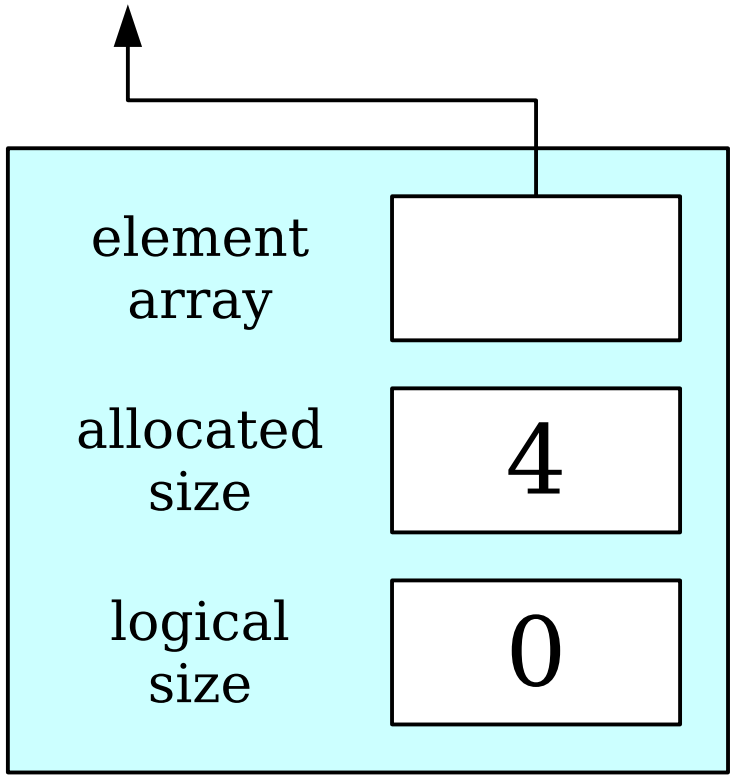
```
OurStack::~OurStack() {
    delete[] elems;
}
```

Cradle to Grave, Take III




```
int main() {


OurStack::~OurStack() {
    delete[] elems;
}


}
```

Cradle to Grave, Take III




```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave, Take III

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

To Summarize

- You can create arrays of a fixed size at runtime by using `new[]`.
- You are responsible for freeing any memory you explicitly allocate by calling `delete[]`.
- Constructors are used to set up a class's internal state so that it's in a good place.
- Destructors are used to free resource that a class allocates.

Your Action Items

- ***Finish Assignment 5.***
  - Need help? Stop by the LaIR, call into office hours, or ask on EdStem!

Next Time

- ***Making Stack Grow!***
  - Different approaches to `Stack` growth.
  - Analysis of these approaches.
  - The reality: *everything is a tradeoff!*