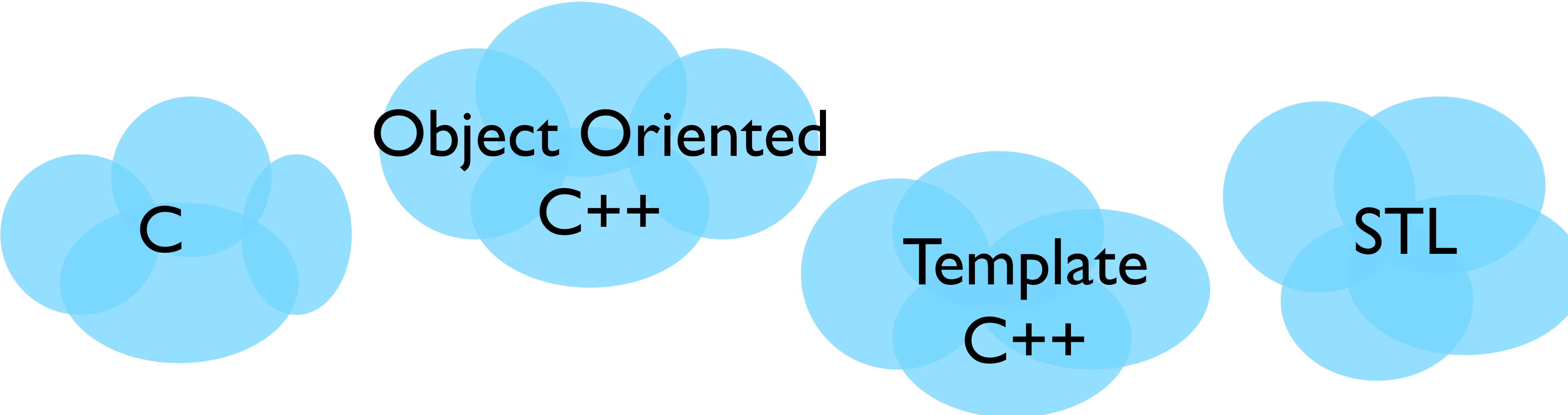
Lecture 3



C++ Review+

Pointers, Arrays/Strings, Classes, Templates & Functors

C++

Chapter 1 in Data Structures and Algorithmic Analysis C++ Fourth Edition, by Mark Allen Weiss

- Review notes from CS1124
- Recitation on Friday from 11:00 - 11:50
- Tutoring Center for C++ questions. Located 3rd floor JAB 373.

Other resources: books, (Some examples presented in class will be from different books, or code I found on the web, or …), ...

The code in class does not have sufficient error checking or comments because we are focusing on the concept being presented. In your hw you MUST include error checking and comments.

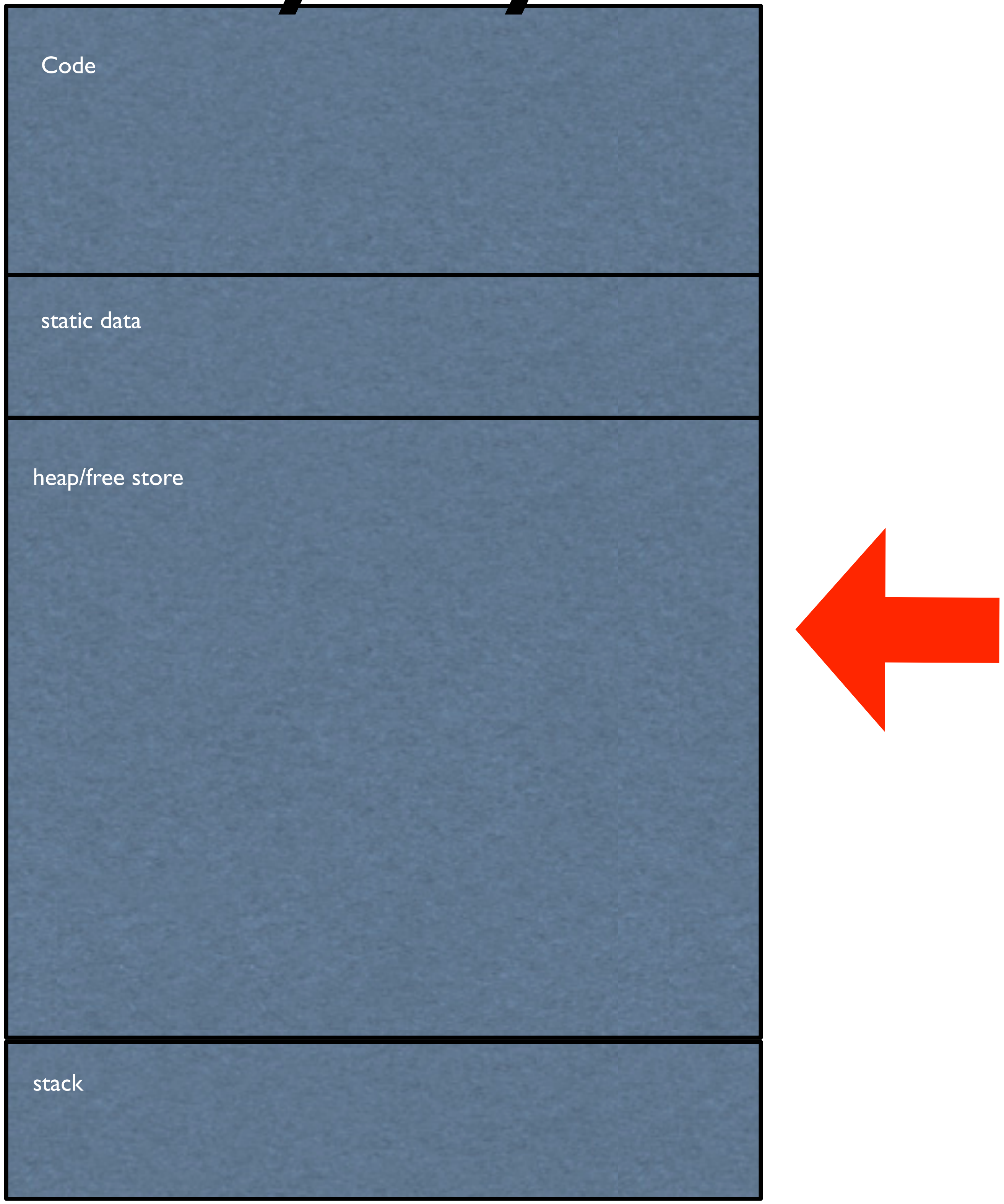
8 bits is a byte

The memory is divided into bytes. Traditionally a memory address is given to every byte.

C++ stores the values of variables in the memory by knowing the address of where the information is stored

On my computer, C++ uses 1 byte to store a character.

Memory layout

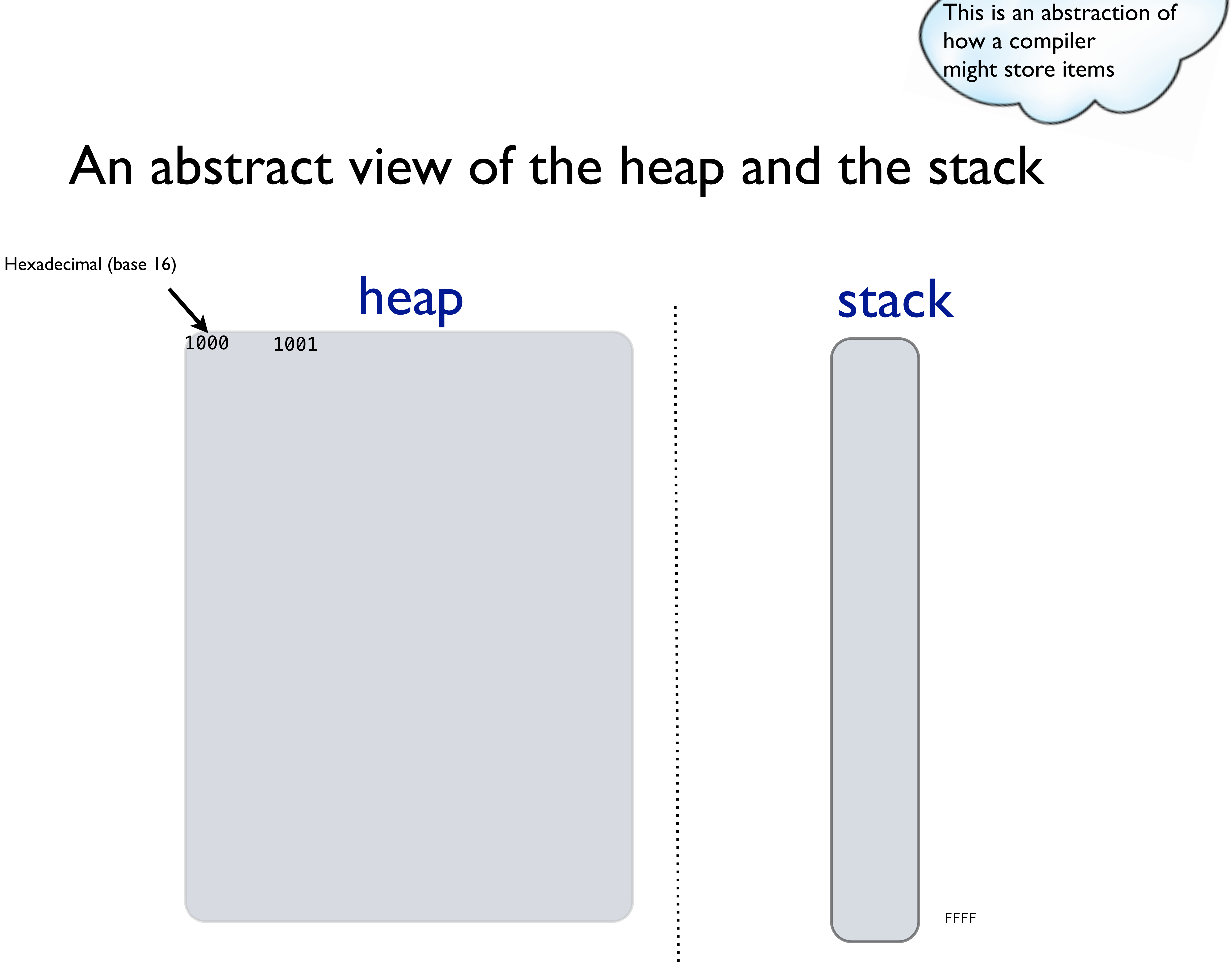


You get to decide what is stored here (but you don't get to decide where it is stored.)

To put/store something in here use the new operator

When you don't need the item you stored here, you should return the memory so it can be used again. To return the memory use the delete operator

An abstract view of the heap and the stack



Pointers

- value of a pointer variable is address or NULL
- pointer declarations based on type of object the pointer references:

  C *p, *q   //pointers to objects of class C
- operations:

  *p //dereference – gives object at address p

  *p=*q   // assignment of objects of class C

  p=q // assignment of pointers. Creates alias

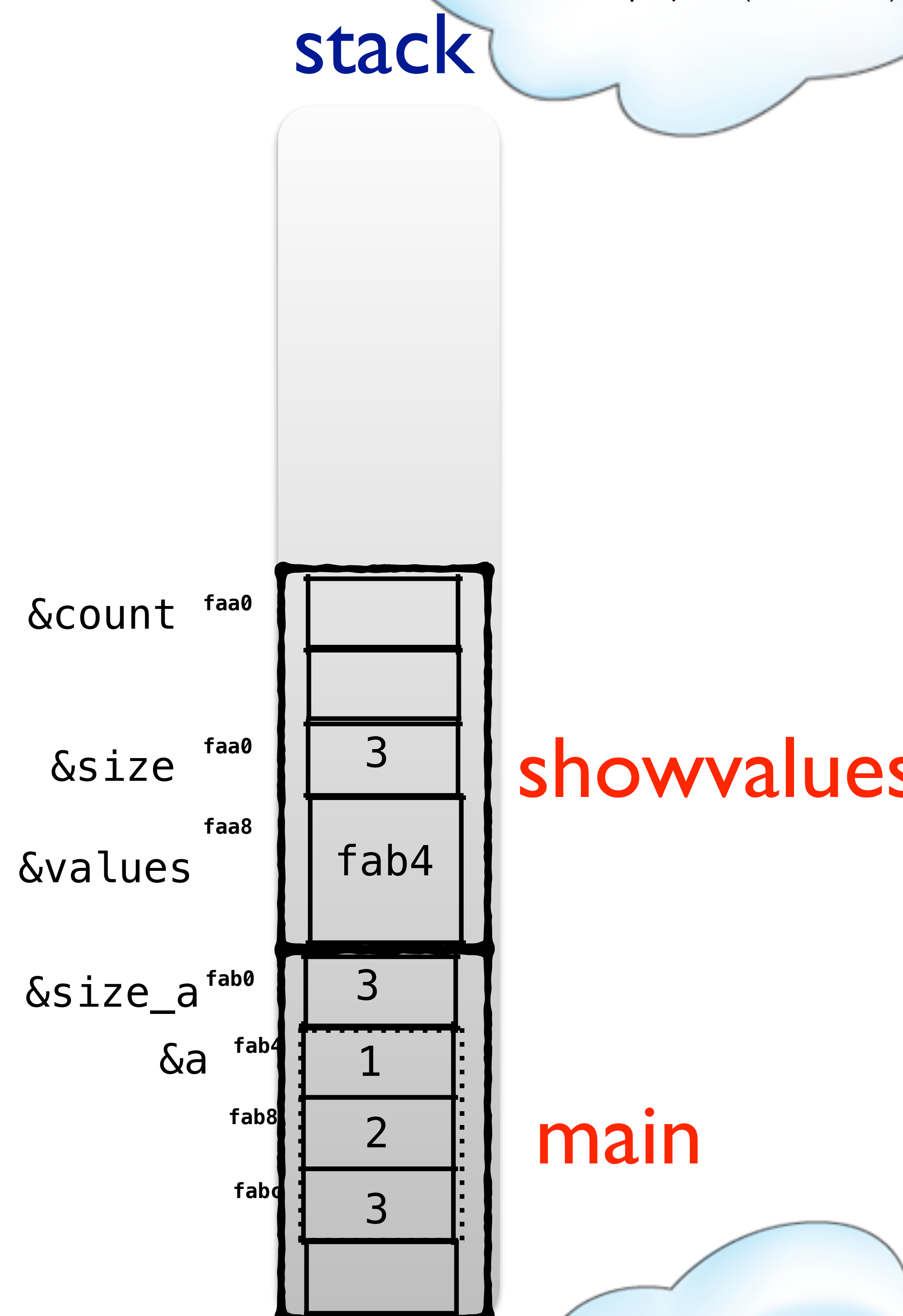
  p = &x   // where x is object of class C

  p->f   // shorthand for (*p).f where f is member of C

> “When used as a function argument, the first dimension of an array is simply treated as a pointer.”
>
> Stroustrup, Bjarne (2013-07-10)

```
void showvalues(int values[], int size)
{
   int count;
   for (count = 0; count < size; count++)
    cout << values[count] << endl;
   }
```

```
 int main () {

    int a[] = {1, 2, 3};
    int size_a = 3;
    showvalues(a, size_a);

 }
```



> “Like C, C++ doesn't define layouts, just semantic constraints that must be met.“
> Bjarne Stroustrup

Memory Management

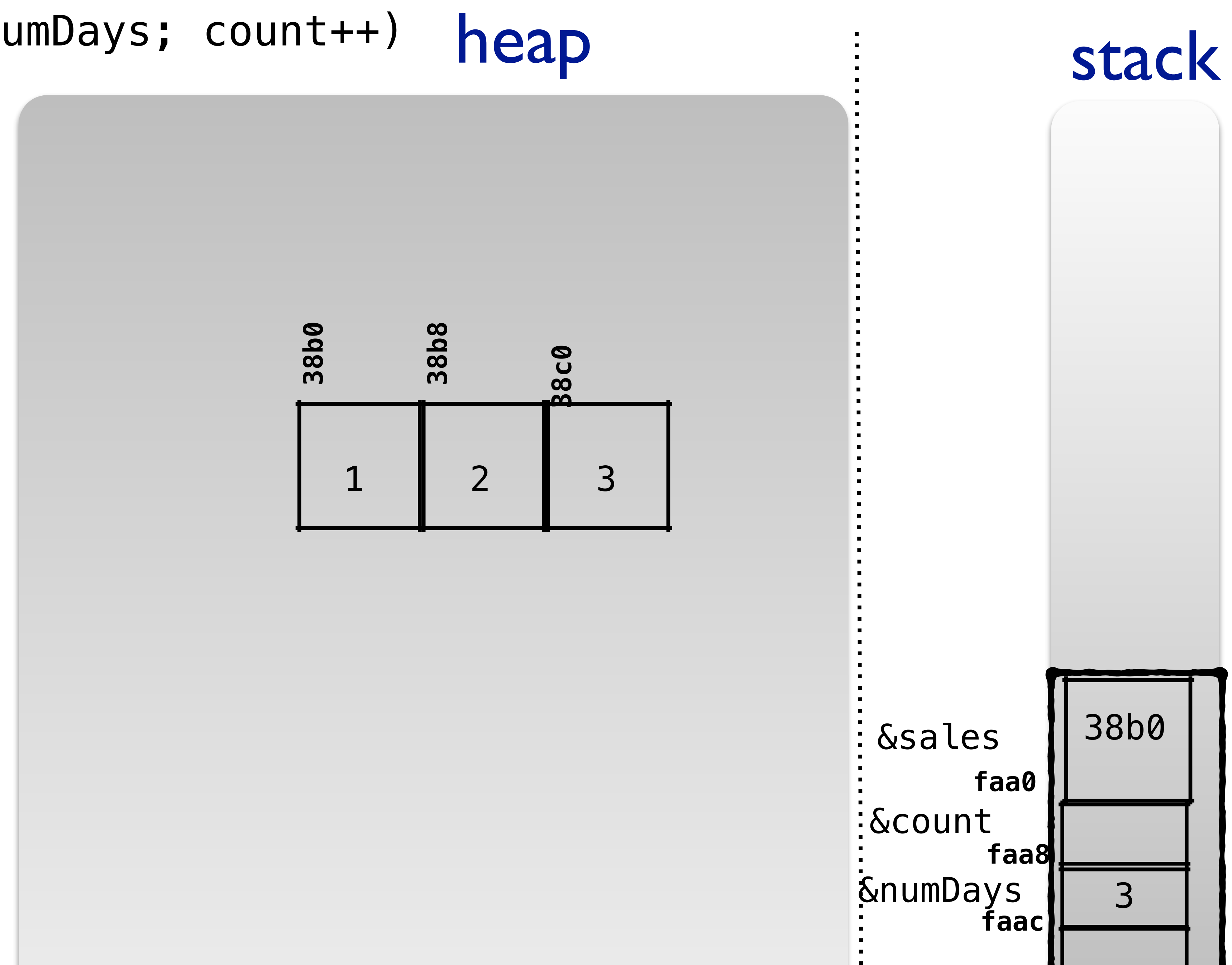
```
C  *p;
p = new C;  // calls constructor of class C
 ...
delete p;  // frees memory occupied by *p;
         // calls destructor if there is one.
```

Beware of:

- dangling references
- double delete
- garbage (memory leaks)

Dynamic Memory Example

```
int numDays,
int count;
double *sales = nullptr;

cin >> numDays;

sales = new double[numDays];

for (count = 0; count < numDays; count++)
{
   cin >> sales[count];
}

delete [] sales;

sales = nullptr;
```



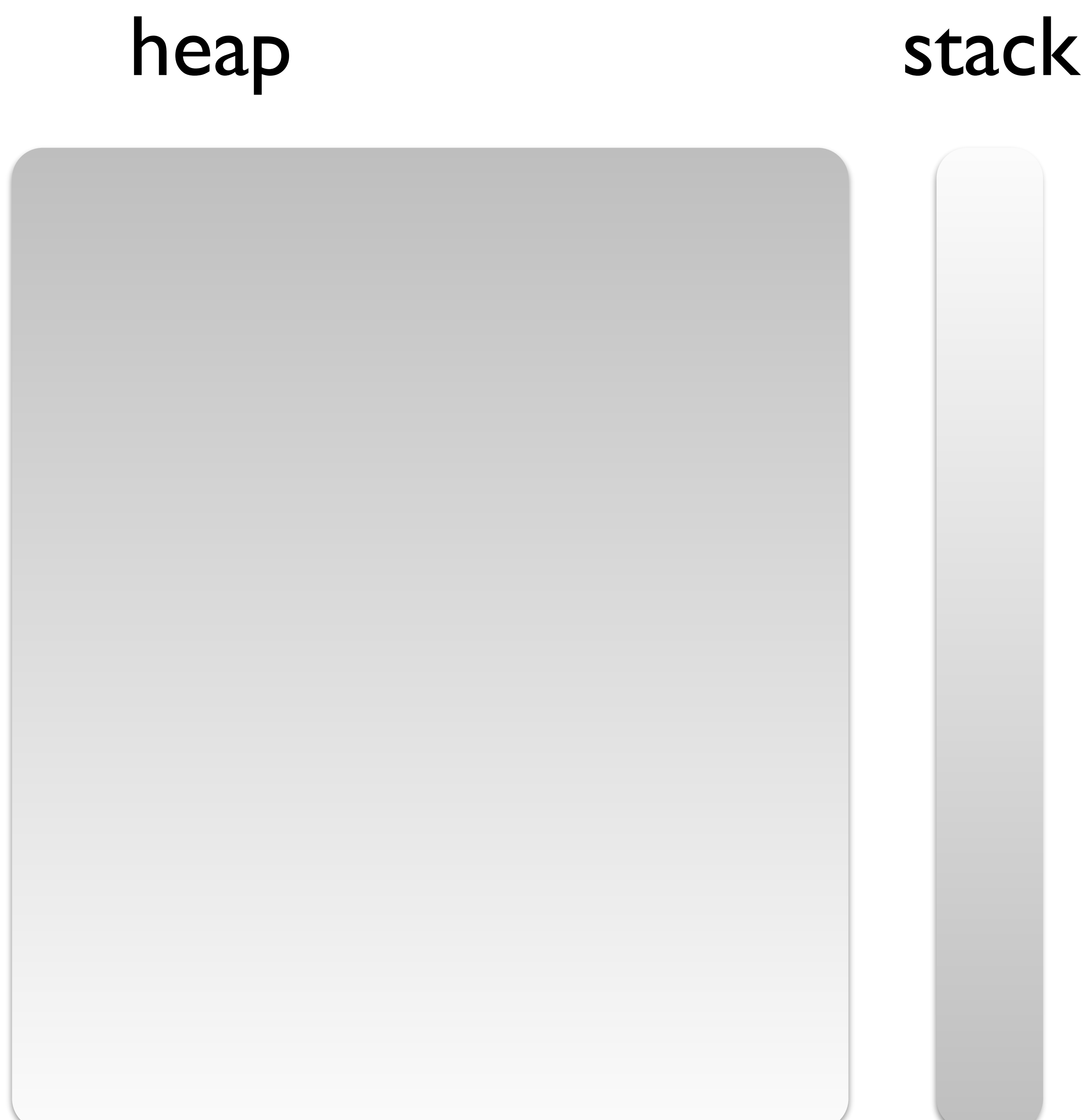
Anything wrong the the following code?

Dangling Reference and Double Delete Example

```
int main ()
{
     int* p1 = new int{7};
     int* p2 = nullptr;

     p2 = p1;

     delete p2;
     cout << *p1;
     delete p1;
```

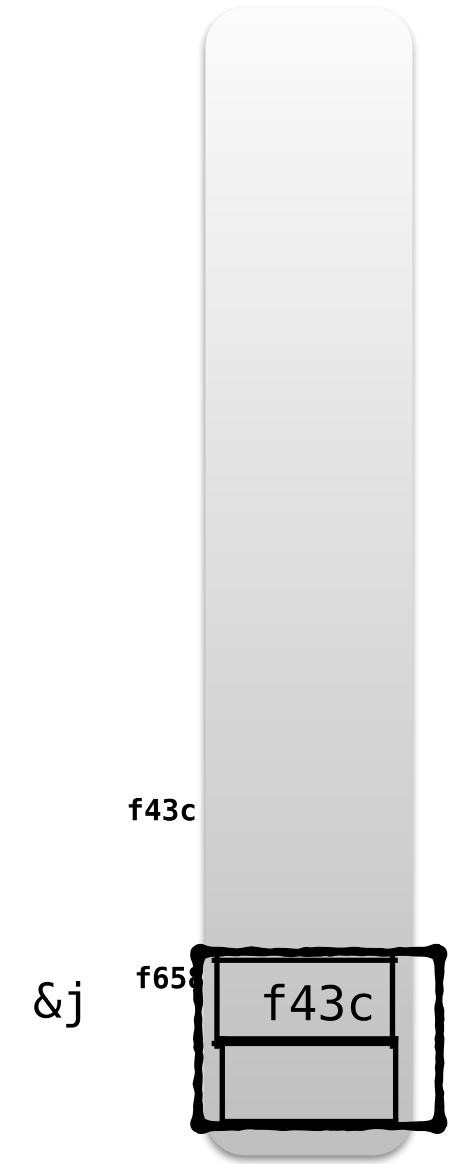
Memory Leak Example

```
int main ()
{
     int* p1 = new int{4};
     int* p2 = nullptr;

     p2 = new int{3};
     p2 = p1;
```



Dangling Reference Example

```
int * oops()
{
    int i = 1;
    return &i;
}
int main ()
{
    int *j = nullptr;
    j = oops();


    cout << *j << endl;
    return 0;
}
```



http://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html

References…

lvalue references & rvalue references

Lvalue Reference

- pointer constant that is always implicitly dereferenced
- creates alias
- useful for call by reference

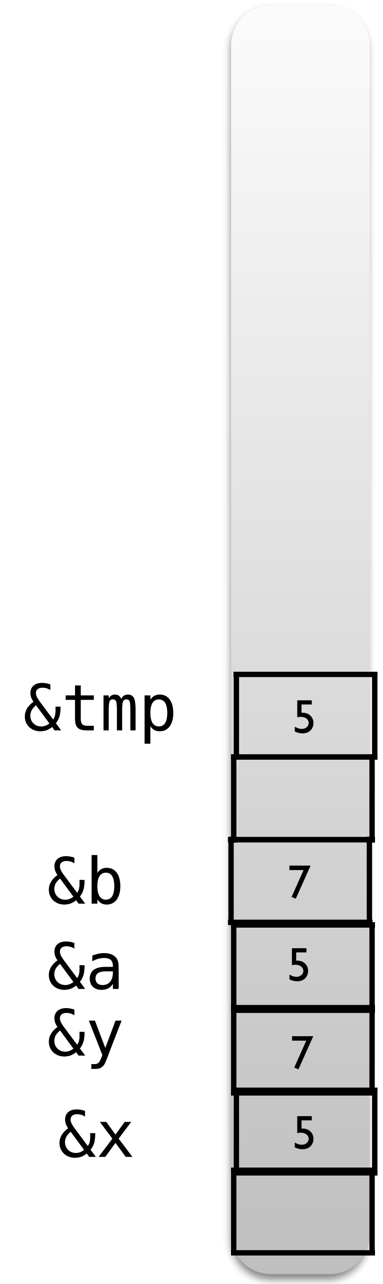
```
int x = 0;
int& y=x;
y++;        // increments x
cout << x;
```

Parameter Passing

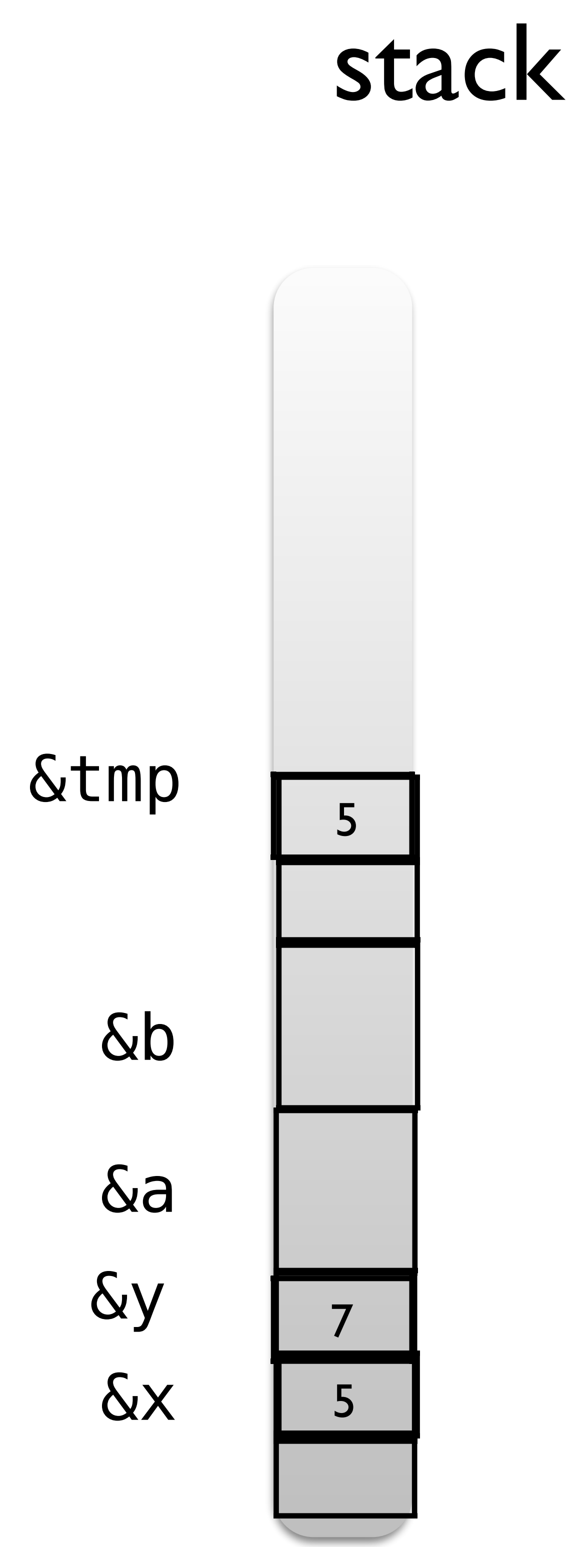
- Call by value (default)
  - allocates (formal) parameter and initializes it by copying argument (actual parameter)
  - changes to parameter do not affect argument
  - appropriate for small objects that should not be changed
- Call by lvalue reference
  - creates alias between argument and parameter
  - changes to parameter DO affect argument
  - appropriate for all objects that may be changed
- Call by const lvalue reference
  - call by reference, but compiler prevents modification of the parameter
  - appropriate for large objects that should not be changed and are expensive to copy
- Call by rvalue reference
  - if the item passed as a parameter is a temporary object that is about to be destroyed
  - most common use is *overloading operator*= *and copy constructor*

Swapping values   Call by value   stack

```
void swapWrong( int a, int b )
{
    int tmp = a;
    a = b;
    b = tmp;
}


int main( )
{
    int x = 5;
    int y = 7;

    swapWrong( x, y );
    cout << "x=" << x << " y=" << y << endl;
```

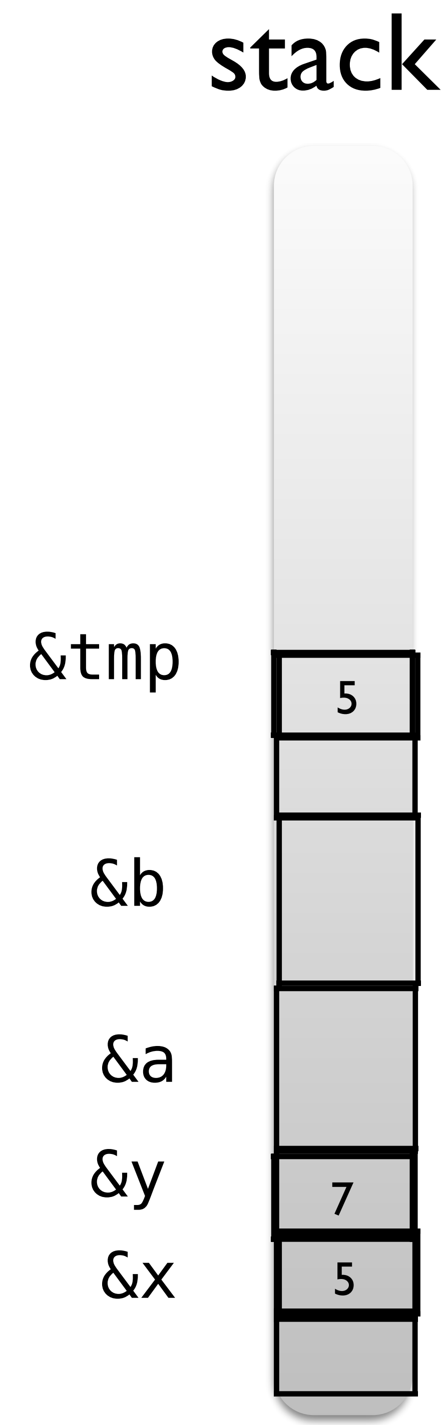


```
void swapPtr( int *a, int *b )
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}


int main( )
{
    int x = 5;
    int y = 7;

    swapPtr( &x, &y );
    cout << "x=" << x << " y=" << y << endl;
```



Call by reference

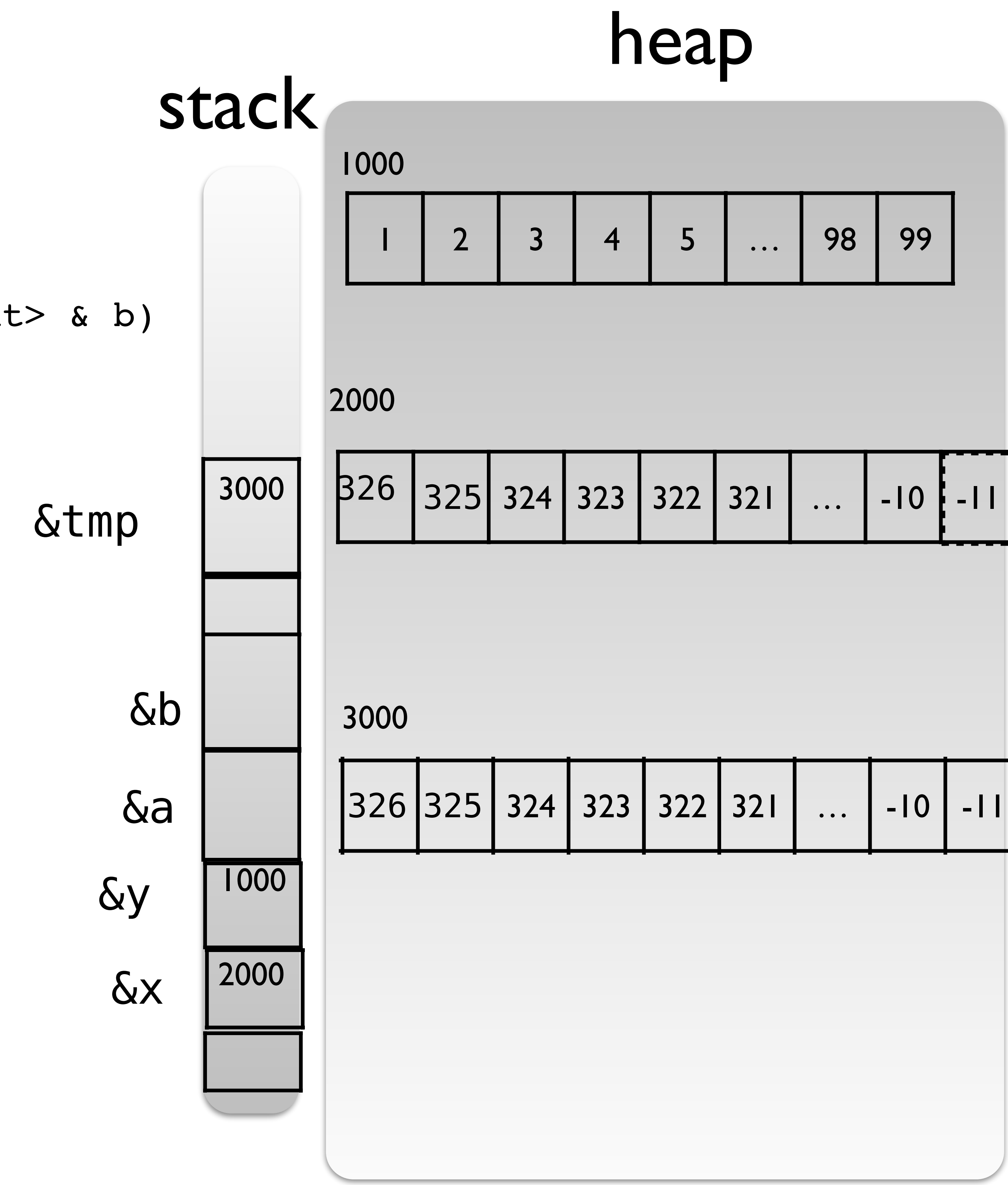
```
void swapRef( int & a, int & b )
{
    int tmp = a;
    a = b;
    b = tmp;
}


int main( )
{
    int x = 5;
    int y = 7;

    swapRef( x, y );
    cout << "x=" << x << " y=" << y << endl;
```



Our swap function…

```
void swapRef(vector<int> & a, vector<int> & b)
{
    vector<int> tmp(a);
    a = b;
    b = tmp;
}
```

```
int main( )
{
    vector<int> x;
    vector<int> y;
    //code to enter values into x and y

    swapRef( x, y );

    return 0;
}
```



That was a very inefficient way to swap!

Constructing a large object takes time. Typically it involves memory allocation and a loop.

This is fine if we need two copies - but often we don't need the old copy as seen in the swap function (or return by value from a function, or a temporary object used in an expression).

Move Semantics

"a way of transmitting information without copying" Bjarne Stroustrup

works by **not** moving the *primary* data, instead changes ownership of the data

When does it make sense to change the ownership of an expression’s resources?

Lvalues and Rvalues

lvalue
void f(string s);
// code …
f(``hi”);
lvalue
temporary string created for copy constructor is an rvalue

In general

- lvalues are objects you can take the address of. e.g. named objects, objects accessible from a pointer, or reference objects

function is a lvalue
return value is a lvalue
string & f(const string & s);
parameter is an lvalue

vector<string> a(10); lvalue
const double z; lvalue (even if you cannot modify it)
bool r; lvalue

not permitted* to moved (*potentially accessible from more than one location in source code*)

- rvalues are objects you cannot take the address of. e.g. temporary objects

return value is a rvalue
string f(const string & s);

const double z = 3.14; rvalue
bool r = true; rvalue

may be moved from (*accessible from only one place in source code*)

lvalue
int x;
x = 1;
rvalue
rvalue

lvalue
int chooseRandom(vector<int> & v)
{ return v[rand() % v.size()]; }
lvalue
(operator[ ] returns a lvalue reference to the type the vector holds)

lvalue
lvalue
int *ptr = new int;
*ptr = chooseRandom(v);
return value is a rvalue

* It is possible to cast an lvalue to an rvalue.

Lvalue and RvalueReference Types

&, &&

- lvalue references:
  - –lvalues may bind to lvalue references
  - –rvalues may bind to const lvalue references

```
string s = ”hello”;
string & greeting = s;
bool same = (&s == &greeting);
```

evaluates to true since they <u>are</u> the same object

string & greeting ~~=~~ string(”hello”);

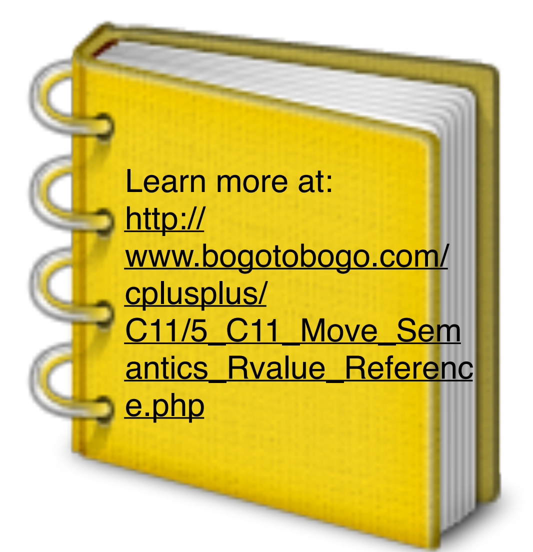
string & greeting2 ~~=~~ s + ”!”;

string & greeting3 ~~=~~ s.substr(0,3);

- rvalue references:
  - –rvalues may bind to rvalue reference
  - –lvalues may not bind to rvalue references

```
string && greeting = string(“hello”);
bool same = (&s == &greeting);
string && greeting2 = greeting + ”!”;
string && greeting3 = greeting.substr(0,3);
```

evaluates to false since they <u>are not</u> the same object



The most common use of a rvalue reference in an overloaded move assignment operator and overloaded move constructor

Reference Types
lvalue &, rvalue &&

Officially && is always an rvalue reference, but it doesn't always act that way. If the type needs to be deduced it uses reference collapsing rules.

Scott Myer came up with the idea of a universal reference.

We will not cover this topic in the course

If you are interested in learning more: https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Scott-Meyers-Universal-References-in-Cpp11

- every expression is a lvalue or rvalue

```
string g( )
{ return "Hi!"; }

void f(string & v)         lvalue reference overloaded
{ cout << "lvalue reference"; }

void f(string && v)        rvalue reference overloaded
{ cout << "rvalue reference"; }

void main{
    string s = "Hello!";
    f(s);                  argument is an lvalue, calls f(T &)
    f(string("Hello"));    argument is an rvalue, calls f(T &&)
    f( g( ) );             argument is an rvalue, calls f(T &&)
}
```

Changing from an lvalue to an rvalue

```
vector<int> b = {1, 2, 3, 4};

vector<int> a;
```

~~a = static_cast<vector<int> &&>(b~~);

```
a = std::move(b);
```

Move function

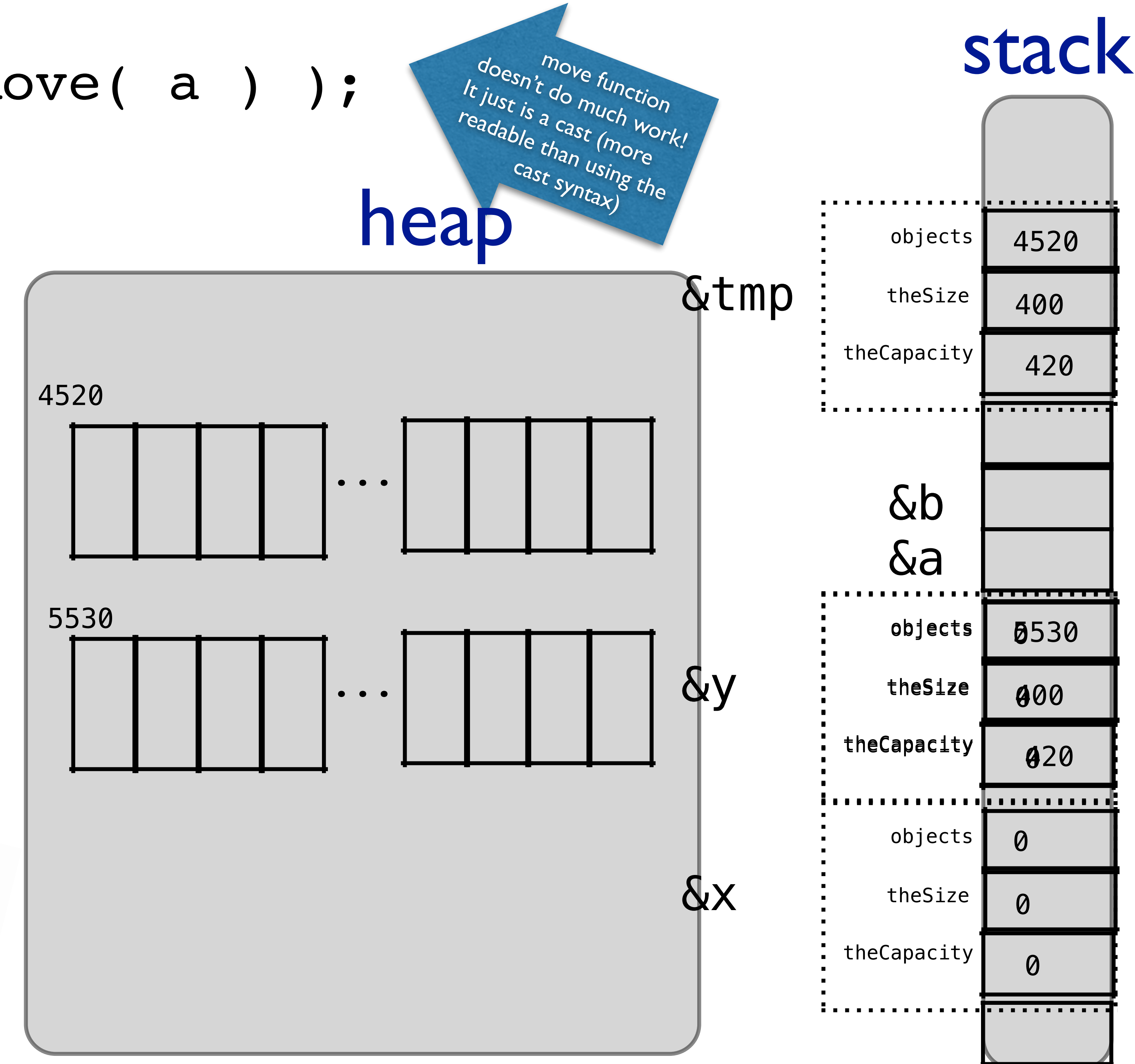
The overloaded move operator= and the move constructor does the moving of the resources

After applying the move function to a lvalue object it can be moved

The move function doesn't move anything!
The move function does an rvalue cast (that is all)!

```
void swap(vector<int> & a, vector<int> & b)
{
    vector<int> tmp(std::move( a ) );
    a = std::move(b);
    b = std::move(tmp);
}

int main( )
{
  vector<int> x(400);
  vector<int> y(400);
  // code …
  swap( x, y );
```

move function doesn't do much work! It just is a cast (more readable than using the cast syntax)



If the type of the object you want to move the resources from doesn't support moving the resources, you will copy the object