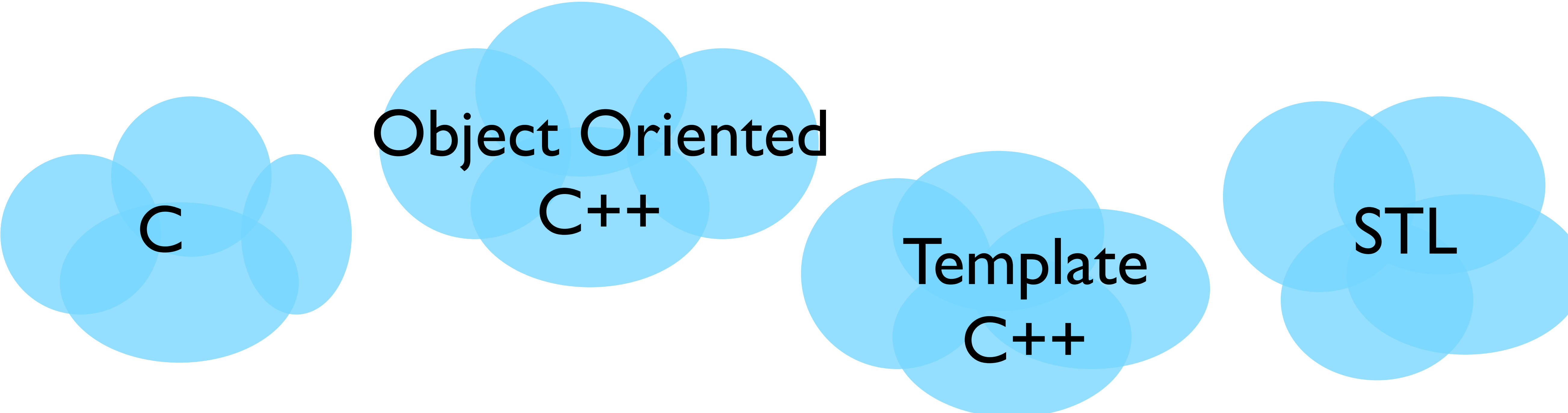
Lecture 3



More C++

- Stack & Heap
- Pointers/reference
- The big five
- Templates
- Functors
- stringstreams
- exceptions

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.

Storing information…

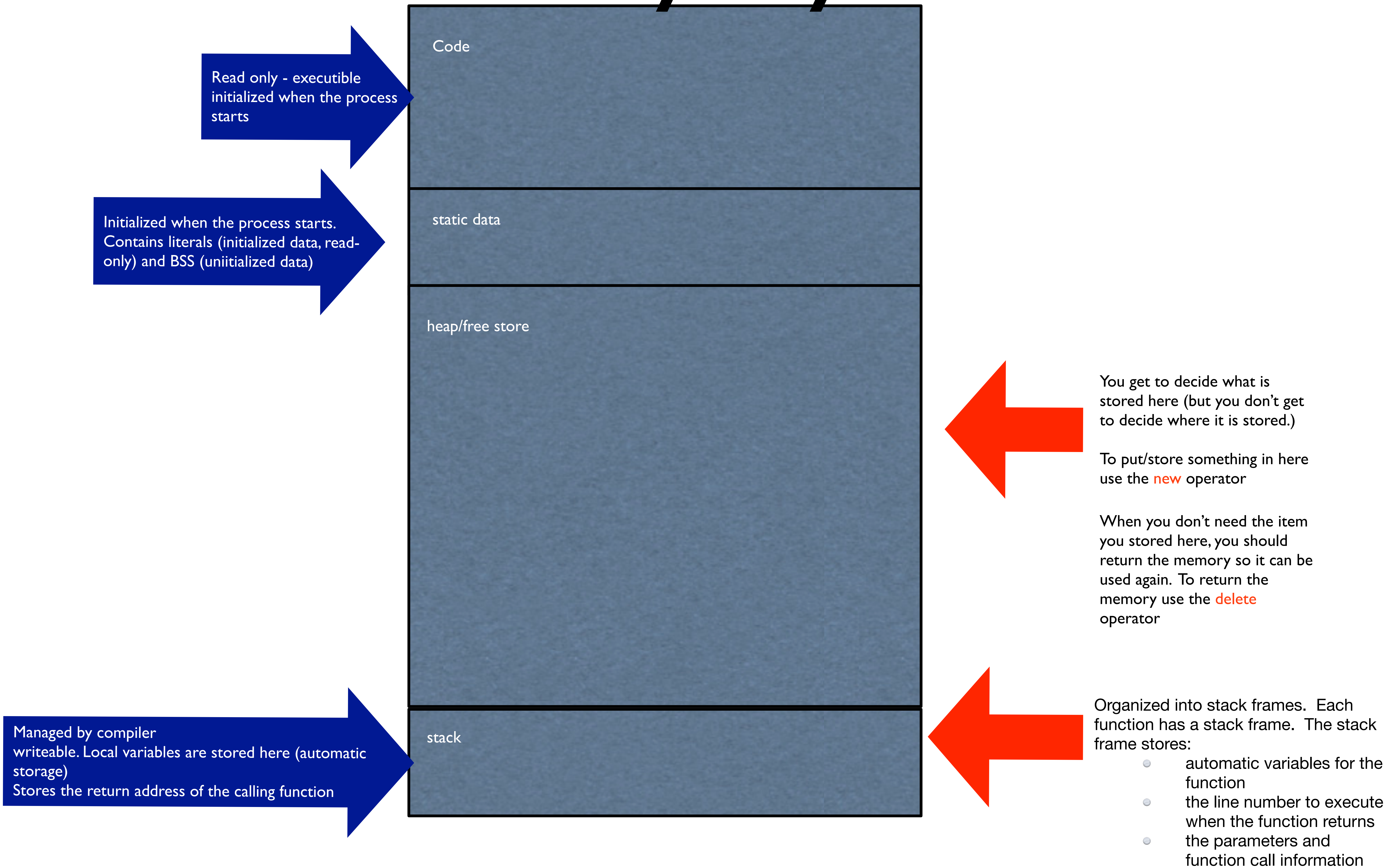
The memory is divided into bytes. Traditionally a memory address is given to every byte.

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

Every letter is given its own number, and the computer is told to interpret it as a letter and not a number (i.e. there is a one to one map between a number and a letter.

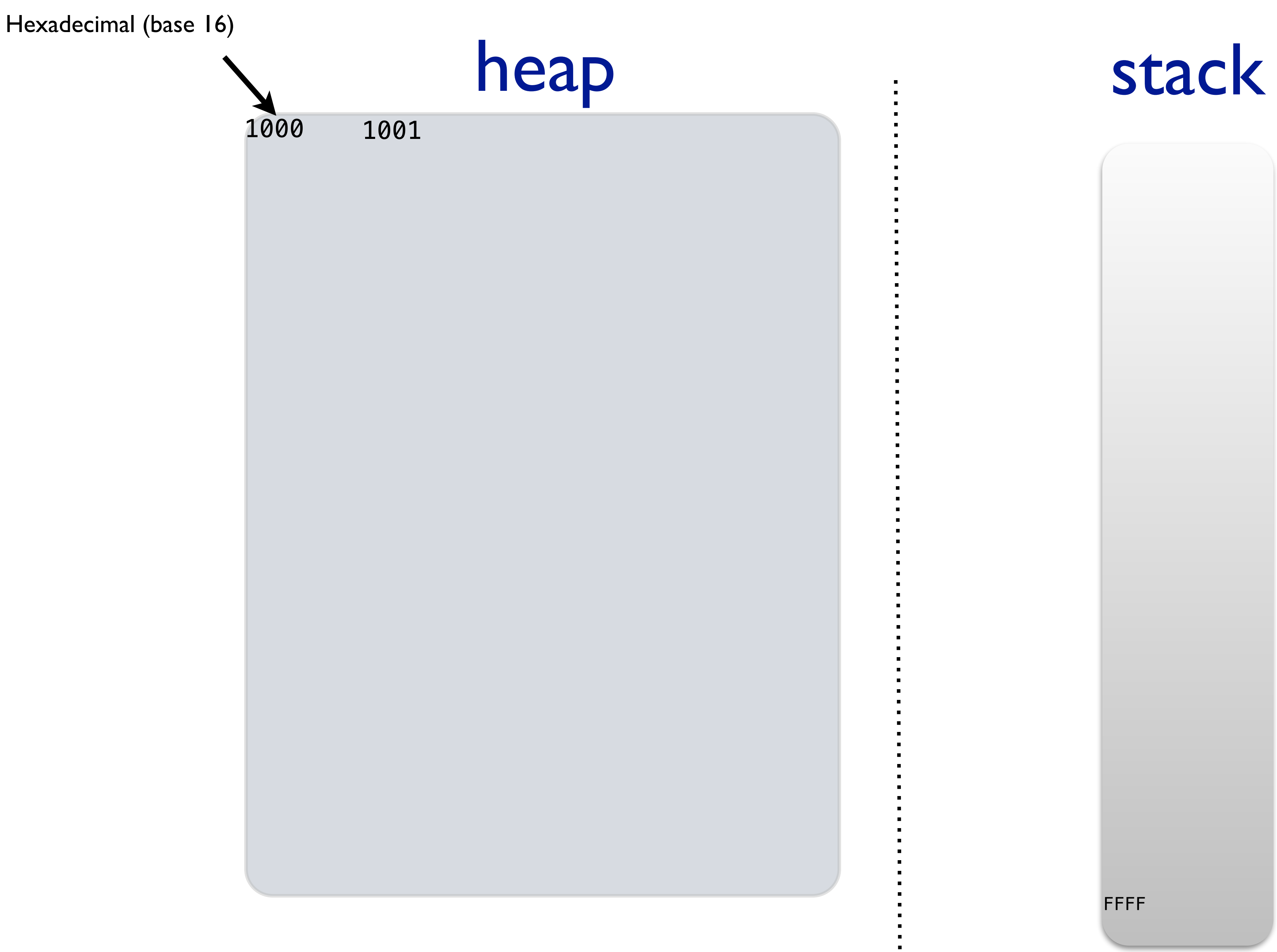
On my computer, C++ uses 1 byte to store a character.

Memory layout



This is an abstraction of
how a compiler
might store items

An abstract view of the heap and the stack



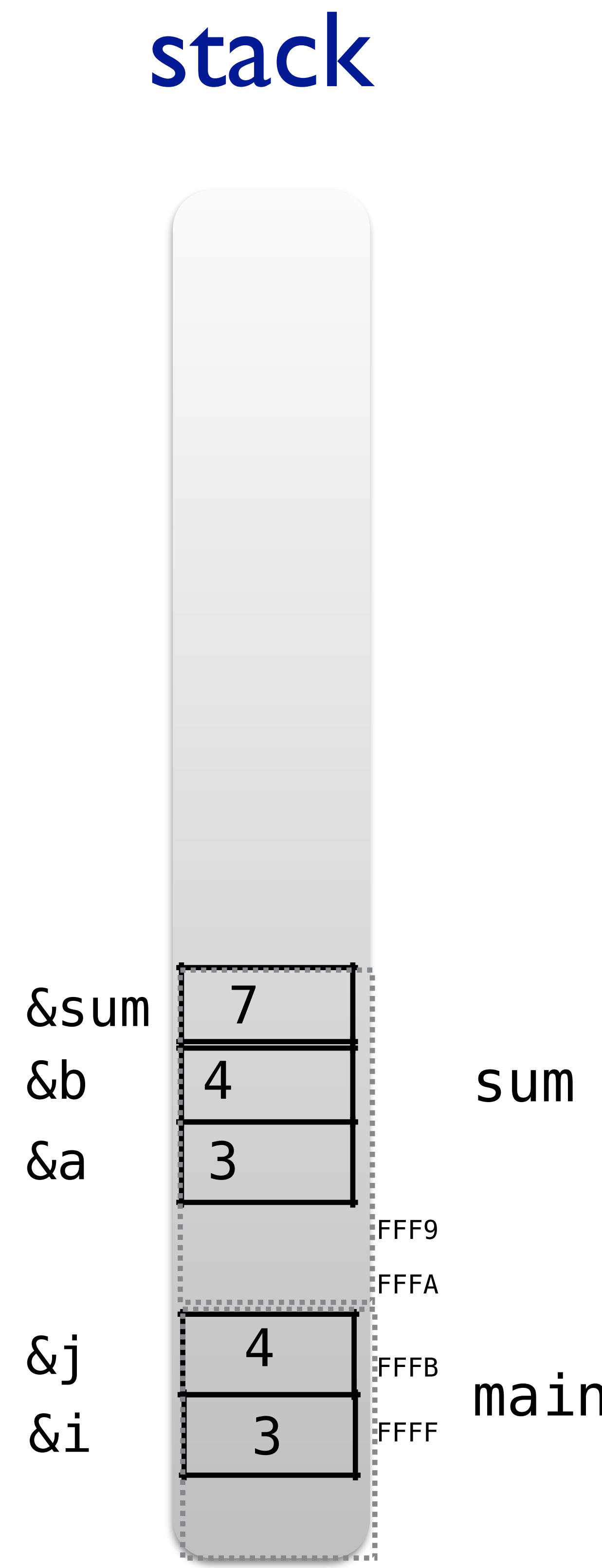
The memory model presented in these slides is for building up your intuition

In your compiler/programming language courses you were learn a more accurate model

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

stack frame for a function call

```
int sum(int a, int b)
{
    int sum = a + b;
    return sum;
}


int main( )
{
    int i = 3;
    int j = 4;

    cout << sum(i, j);
}
```



When we declare:
int i
we allocate memory on the stack.

Dynamic Memory and the Heap

Memory Management:

- allocate memory
- free memory

Allows data structures to expand while the program is running

Accessing Data by its <u>address</u>: Pointers

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

  C *p, *q   //pointers to objects of class C

- operations:

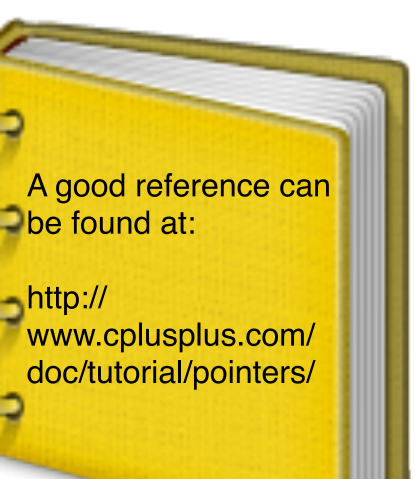
  *p //dereference – gives object at *address* p

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

  p=q // assignment of pointers. Creates alias

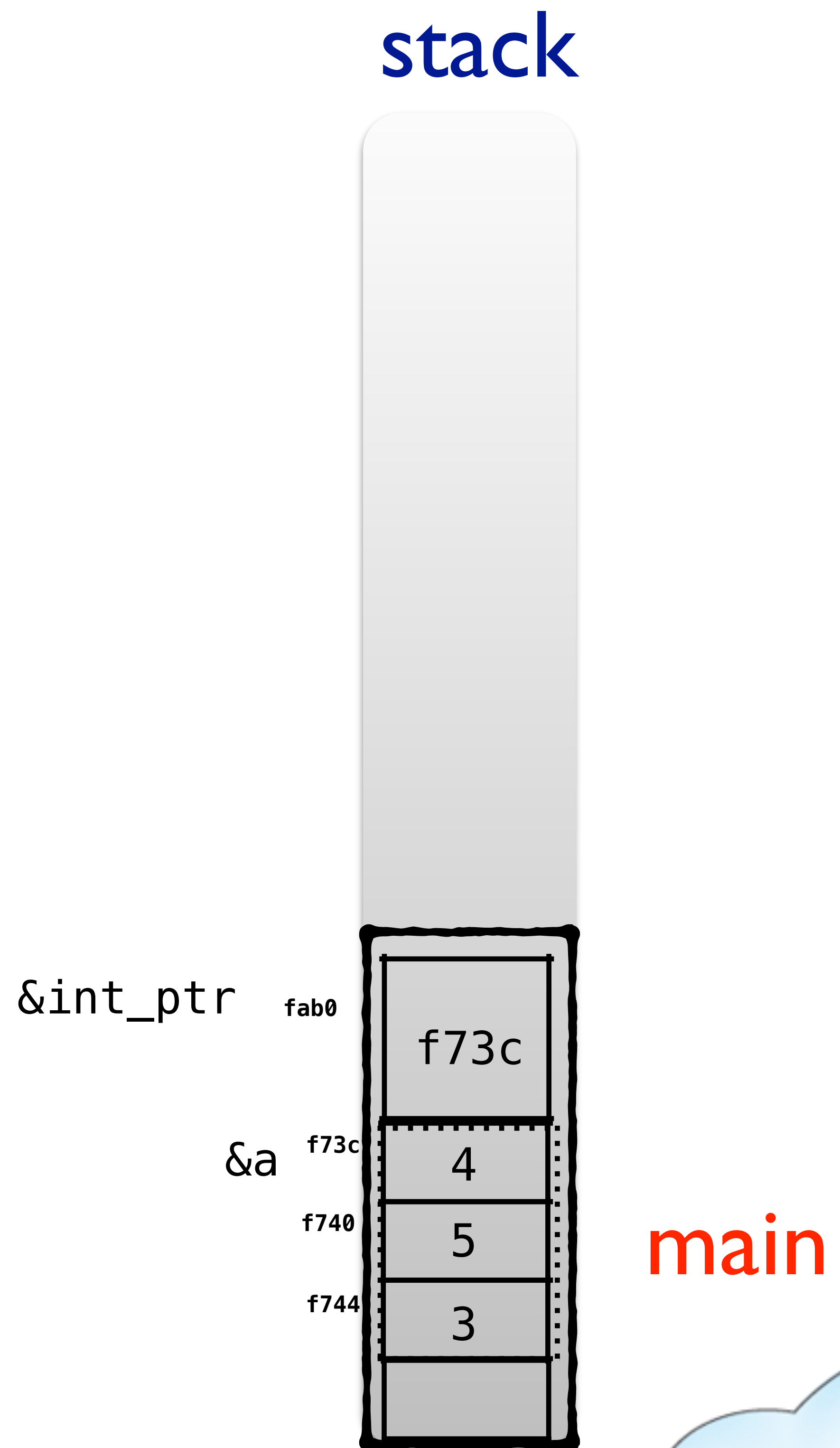
  p = &x   // where x is object of class C

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



“ an array can always be implicitly converted to the pointer of the proper type.”

```
int main () {

   int a[] = {1, 2, 3};
   int * int_ptr = a;
   *int_ptr = 4; // int_ptr[0] = 4;
   *( int_ptr + 1 ) = 5; //int_ptr[1] = 5;

}
```



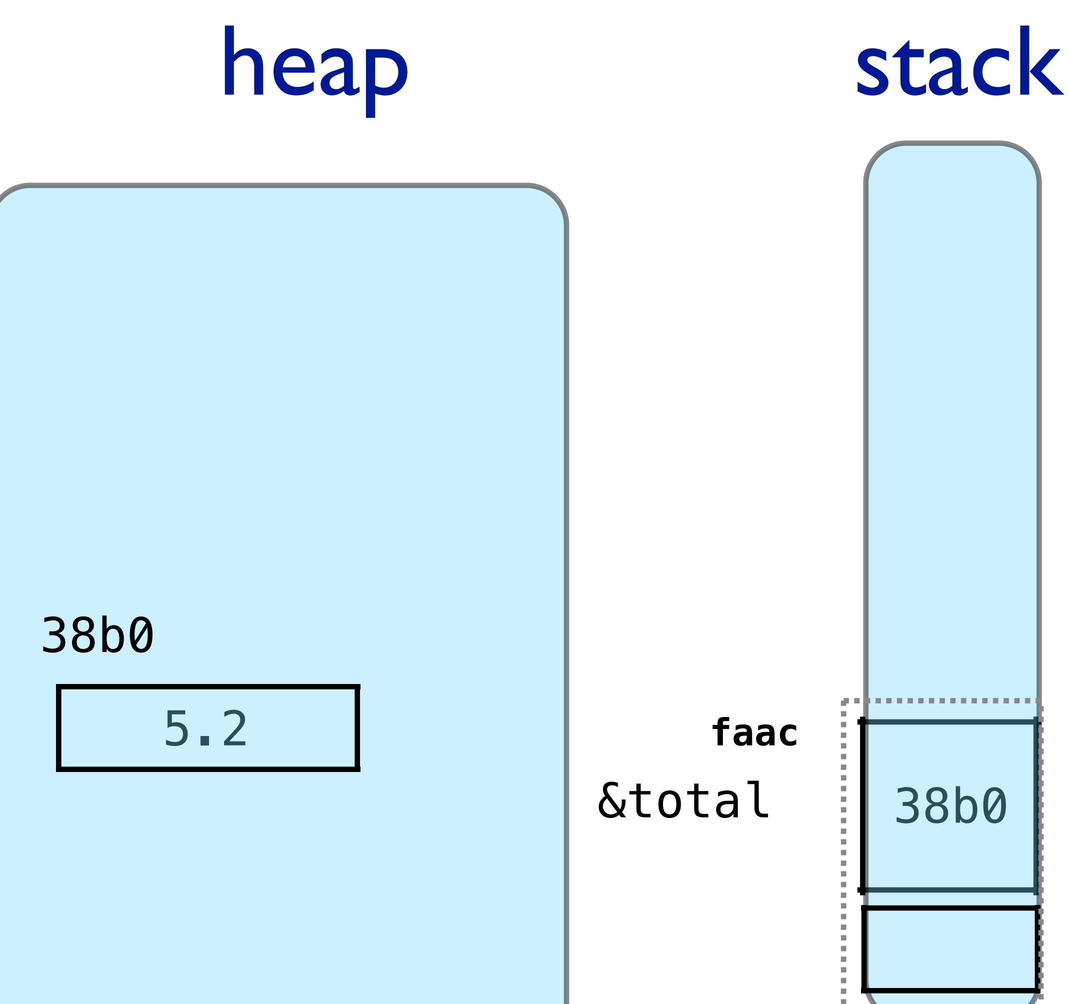
“Like C, C++ doesn't define layouts, just semantic constraints that must be met.“ Bjarne Stroustrup

Memory Management and the Heap

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

heap

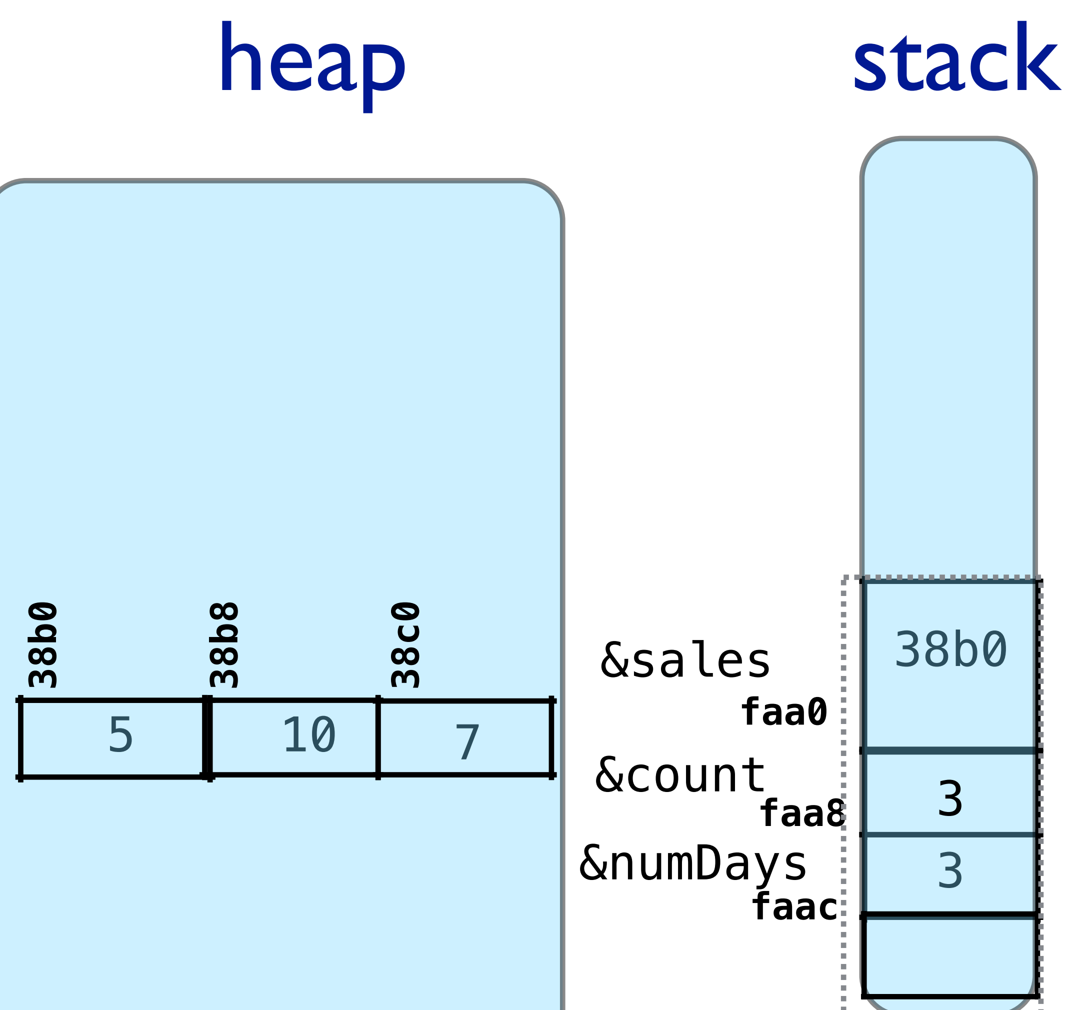
heap




```
int main() {
   int * total = nullptr,
   total = new double;

   cin >> *total;

   delete total;

   total = nullptr;
```

Memory Management and the Heap

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




```
int main() {
   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;
```

Beware of:

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

If you forgot what these are…go to this Friday's recitation 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

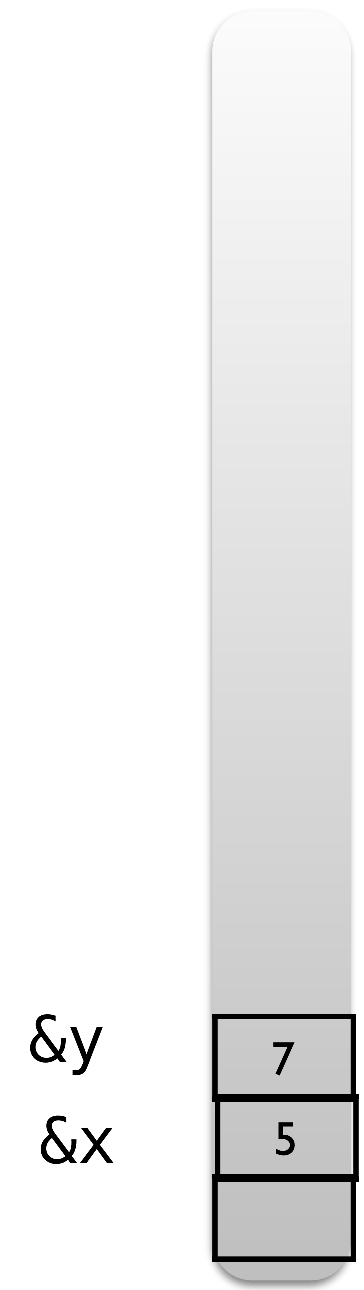
- Call by value (default)
  - allocates (formal) parameter and initializes it by copying argument (actual parameter)
  - changes to parameter do not affect argument
  - appropriate for small objects that should not be changed
- Call by lvalue reference
  - creates alias between argument and parameter
  - 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 constructor*

Swapping values

Call by value

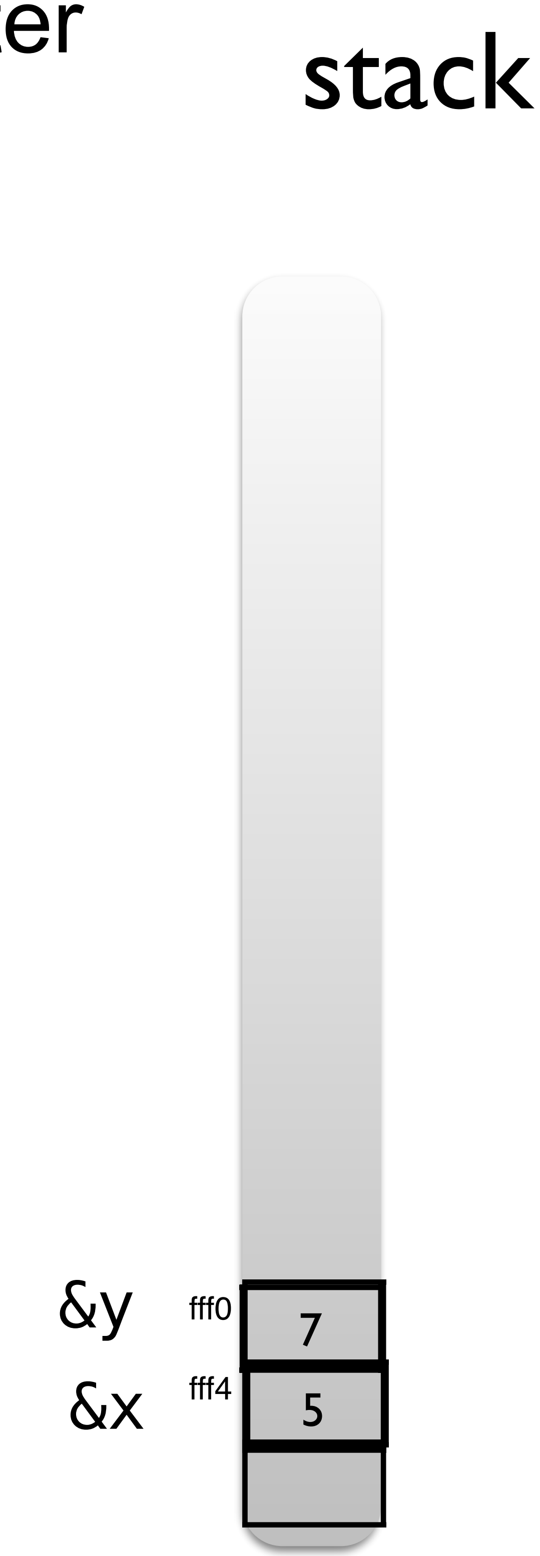
stack

```
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;
```



Call by pointer

```
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;
```



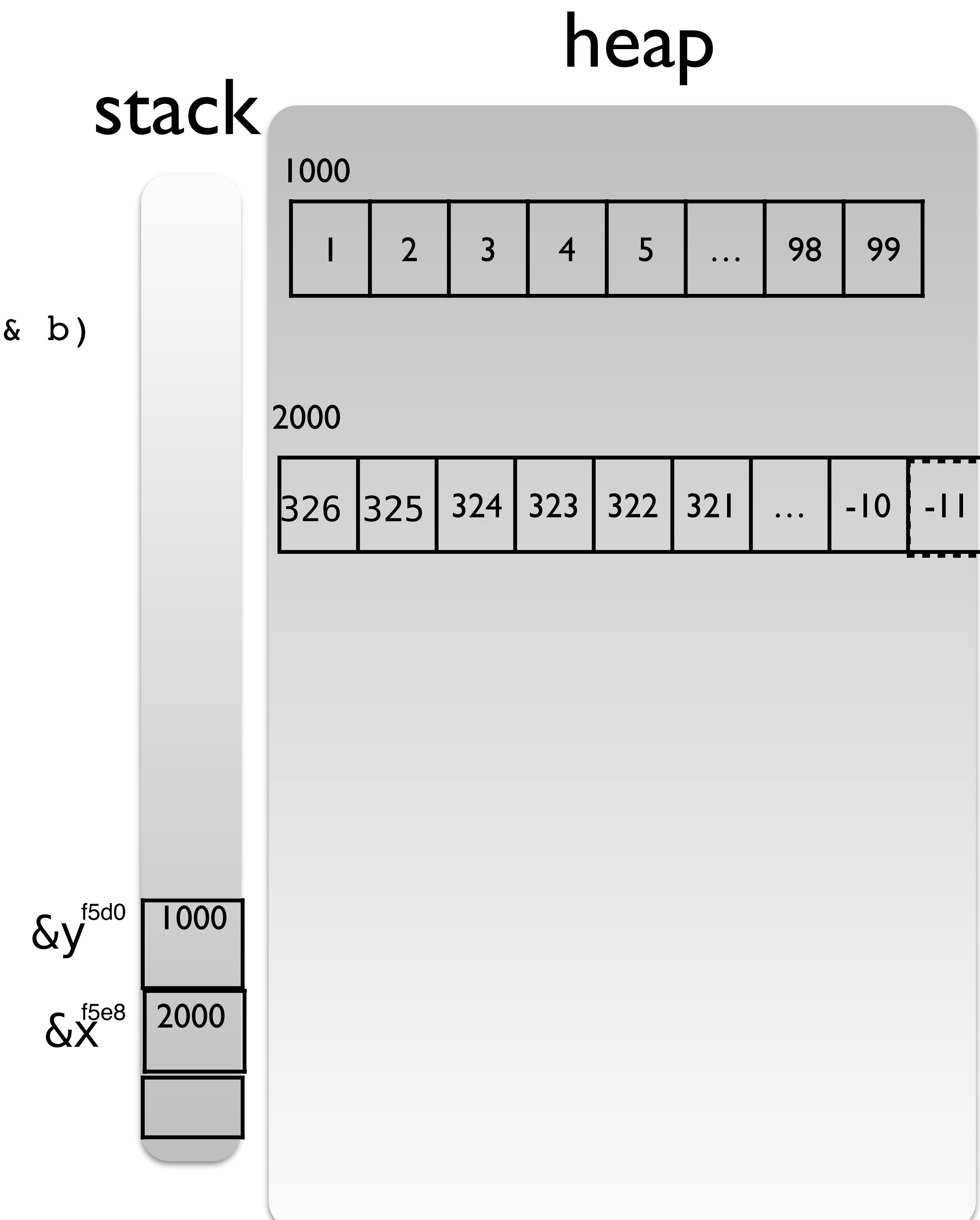
Call by reference

```
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 call by reference 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).

What we want to happen!

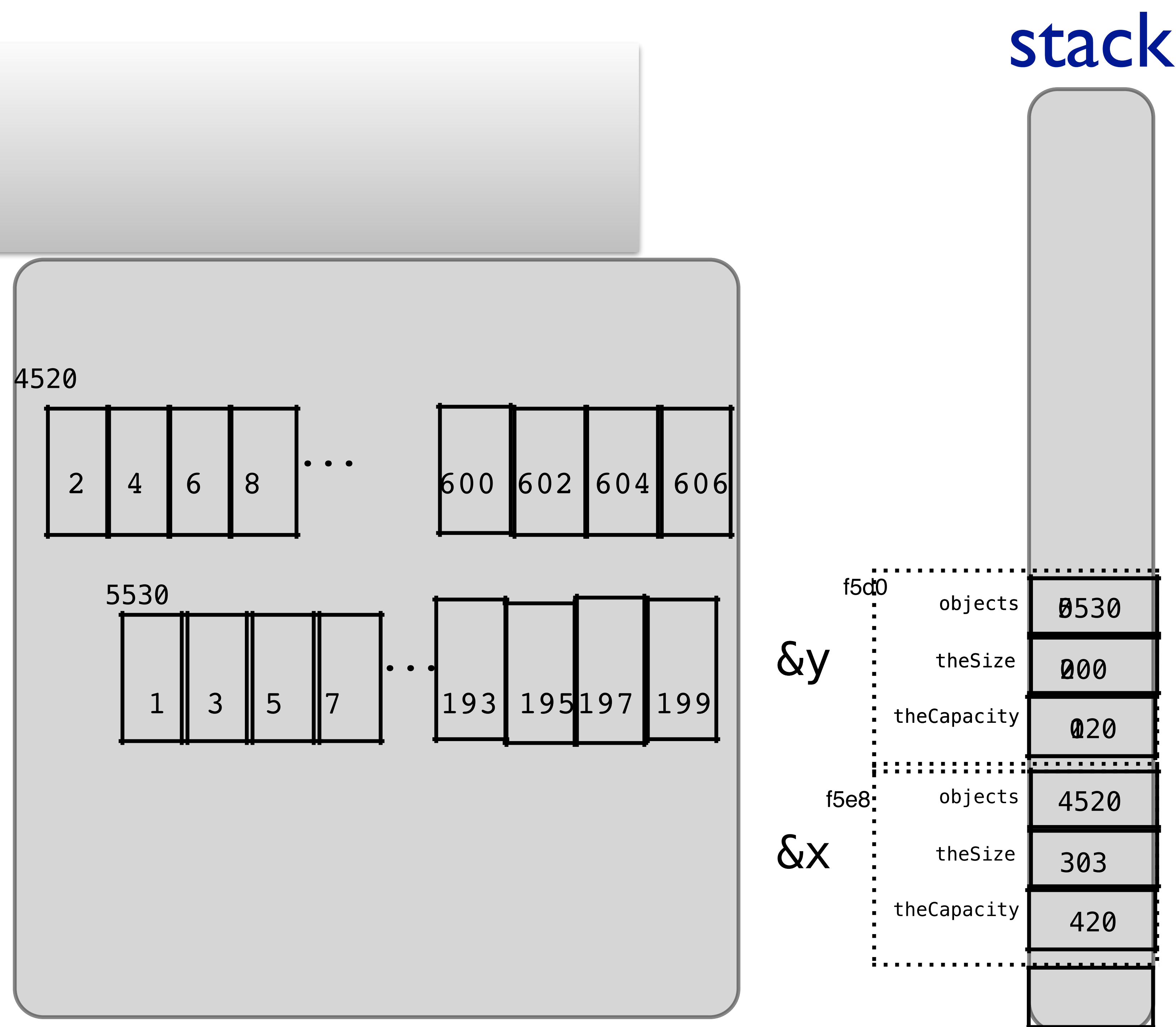
What if we could tell the compiler it could "steal" the resources from another variable.

Last semester, the students preferred thinking about asking the compiler to "recycle" the resources from another variable.

Stealing the resources… (recycling)

```
void swap(vector<int> & a, vector<int> & b)
{
    /* new code written here */
}
```

```
int main( )
{
  vector<int> x(303);
  vector<int> y(200);
  // code …
  swap( x, y );
```



When do you think it would be “safe” to recycle (steal) the resources?

This optimization becomes possible in C++11

Move Semantics

"a way of transmitting information without copying" Bjarne Stroustrup

E.g. Data in the heap

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

se·man·tics
səˈman(t)iks/
noun
the branch of linguistics and logic concerned with meaning. There are a number of branches and subbranches of semantics, including formal semantics, which studies the logical aspects of meaning, such as sense, reference, implication, and logical form, lexical semantics, which studies word meanings and word relations, and conceptual semantics, which studies the cognitive structure of meaning.

To understand move semantics you need to understand which expressions are lvalues and which are rvalues

The program might depend on these values

The program will not be affected if the resources are recycle

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);
const double z;
bool r;
```

lvalue

lvalue (even if you cannot modify it)

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;
bool r = true;
```

rvalue

rvalue

lvalue

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

lvalue

rvalue

rvalue

```
int x;
x = 1;
```

lvalue

```
int chooseRandom(vector<int> & v)
{ return v[ rand() % v.size() ]; }
```

lvalue

lvalue

```
int *ptr = new int;
*ptr = chooseRandom(v);
```

lvalue

(operator[ ] returns a lvalue reference to the type the vector holds)

return value is a rvalue

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

Lvalue and Rvalue Reference Types

&, &&

- lvalue references (what you have been using):
  - –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

- rvalue references (the new type of reference):
  - –rvalues may bind to rvalue reference
  - –lvalues may not bind to rvalue references

```
string && greeting1 = s + “!”;
string && greeting2 = greeting.substr(0,3);
```

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 &&)
}
```

```
#include <algorithm>
```

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

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

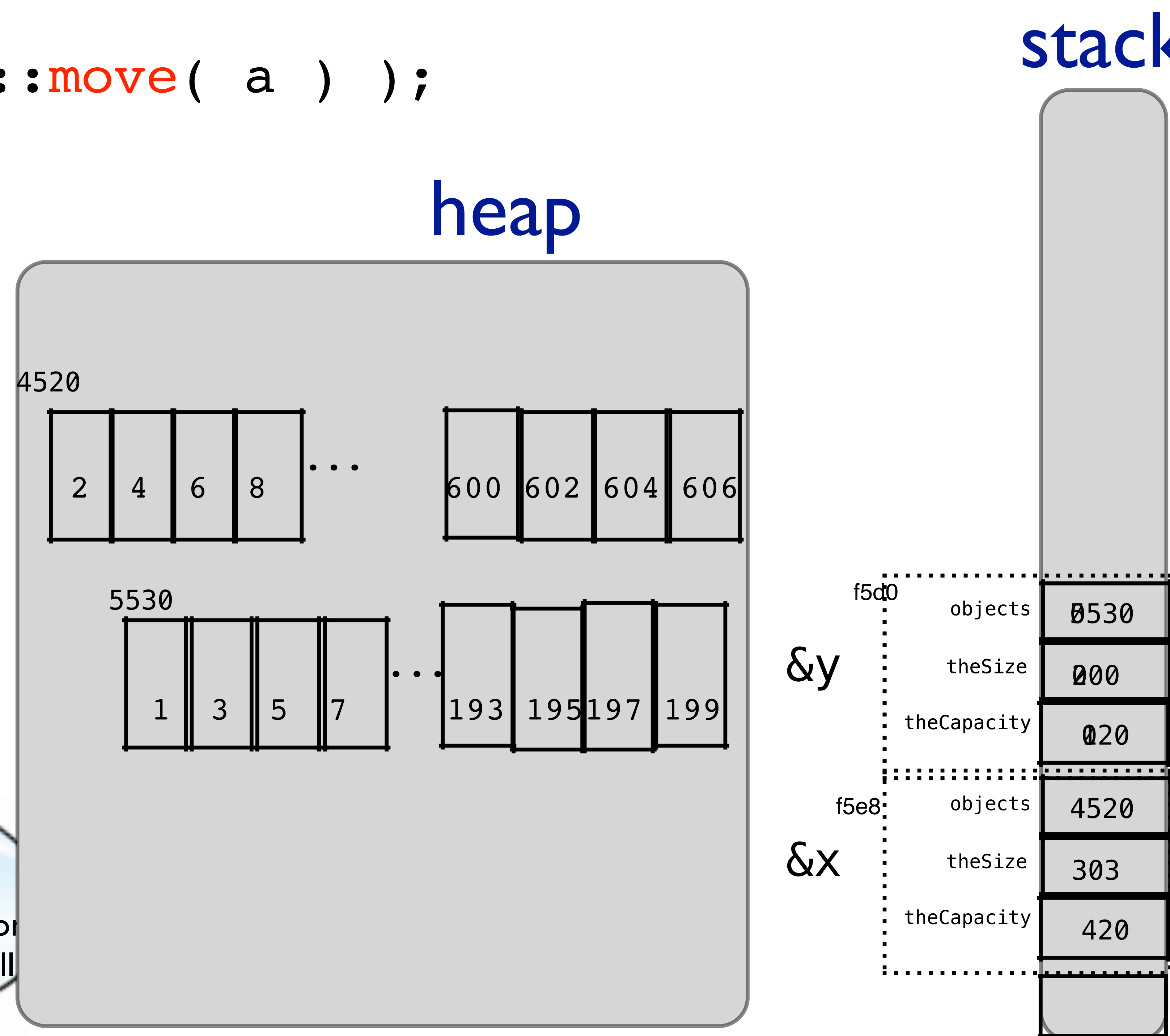
move function

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

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

```
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(303);
  vector<int> y(200);
  // code …
  swap( x, y );
```

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



C++ Classes

A class is a user defined type that allows the

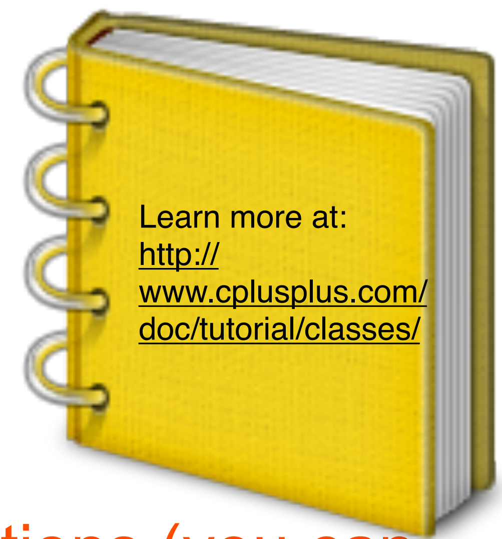
- interface to reflect what requests can be made of the type
- implementation to be hidden, allowing for it to change AND to protect the object from the client

C++11 Shallow vs Deep

C++98 had the big-three:
- copy assignment operator
- copy constructor
- destructor

```
class C
{
    public:
      C(C2 x, C3 * y): x(x),y(y){ }
    private
      C2 x;
      C3 *y;
};

int main()
{
   C *o1 = new C(...);
   C *o2 = new C(...);
   if (*o1==*o2) {…}
   *o1 = *o2;
   delete o1;
   C o3(*o1);
```

C++11 classes have five functions already created:
- Copy Assignment operator=
- <u>Move Assignment operator=</u>
- Copy Constructor
- <u>Move Constructor</u>
- Destructor



Often you can use these five functions (you can choose to not use these by writing your own function or by telling the compiler not to use the default). If your object has one or more member variables which are pointers, the behavior of these five default functions will probably not be what you intended.

> e.g. copy assignment operator will copy pointers not dereferenced pointers.

As a good rule of thumb, if you need to define any of the "big 5" you should define all of them.

Creation of a very simple class

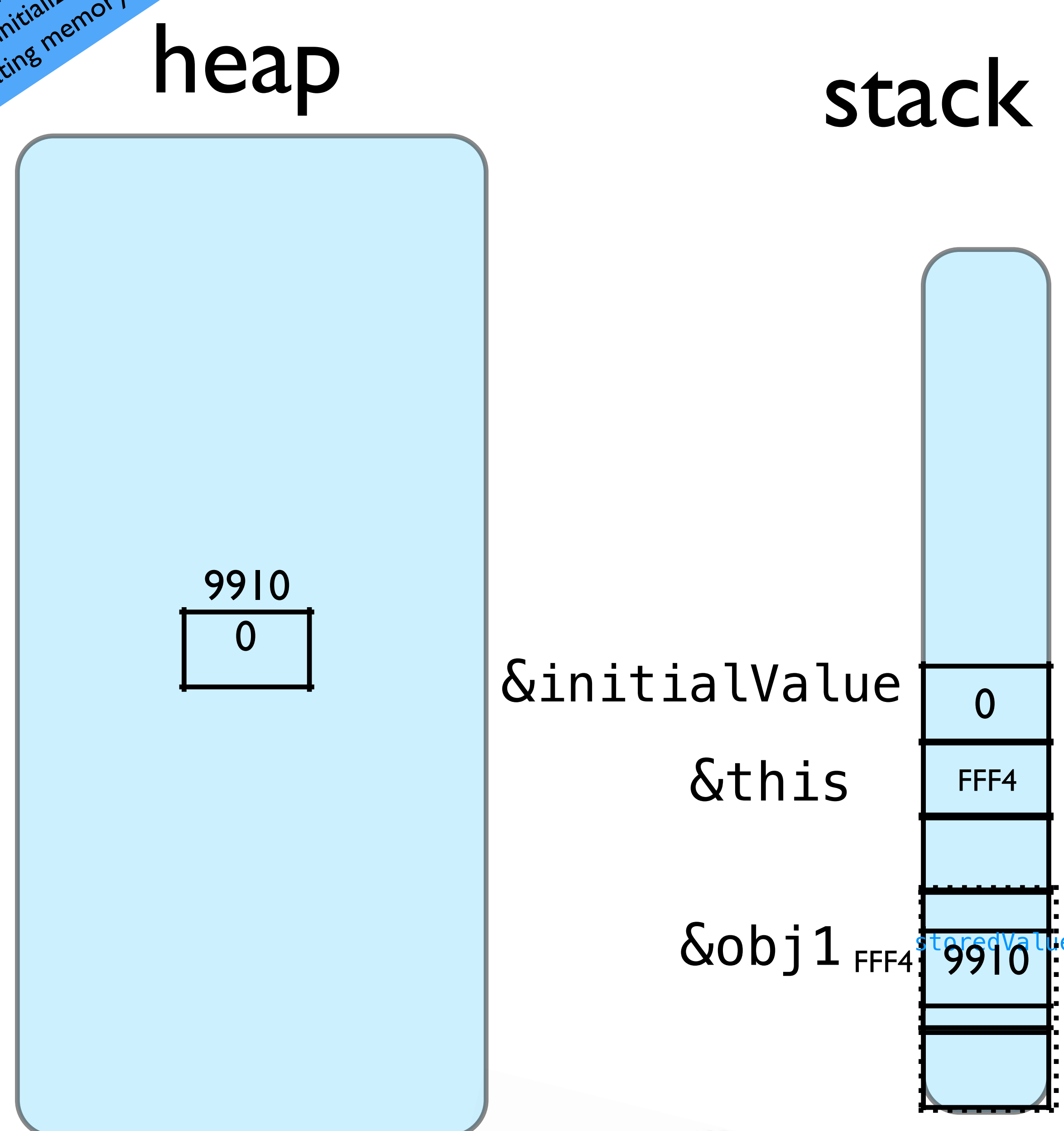
IntCell

A very simple class to show why we need to define the big five when a data member is a pointer

Without the word "explicit" a one-parameter constructor defines an implicit type conversion.

The constructor is called when the object is declared.
In the constructor you decide what the new item should "look like" by initializing member variables and/or allocating memory

```
class IntCell
{
public:
   explicit IntCell(int initialValue = 0)
     {storedValue = new int(initialValue);}

   int read() const {return *storedValue;}
   void write(int x) {*storedValue = x;}
    …
 private:
   int* storedValue;
};


int main{

    IntCell  obj1;
    …


 }
```



The constructor, like other functions, can be overloaded.

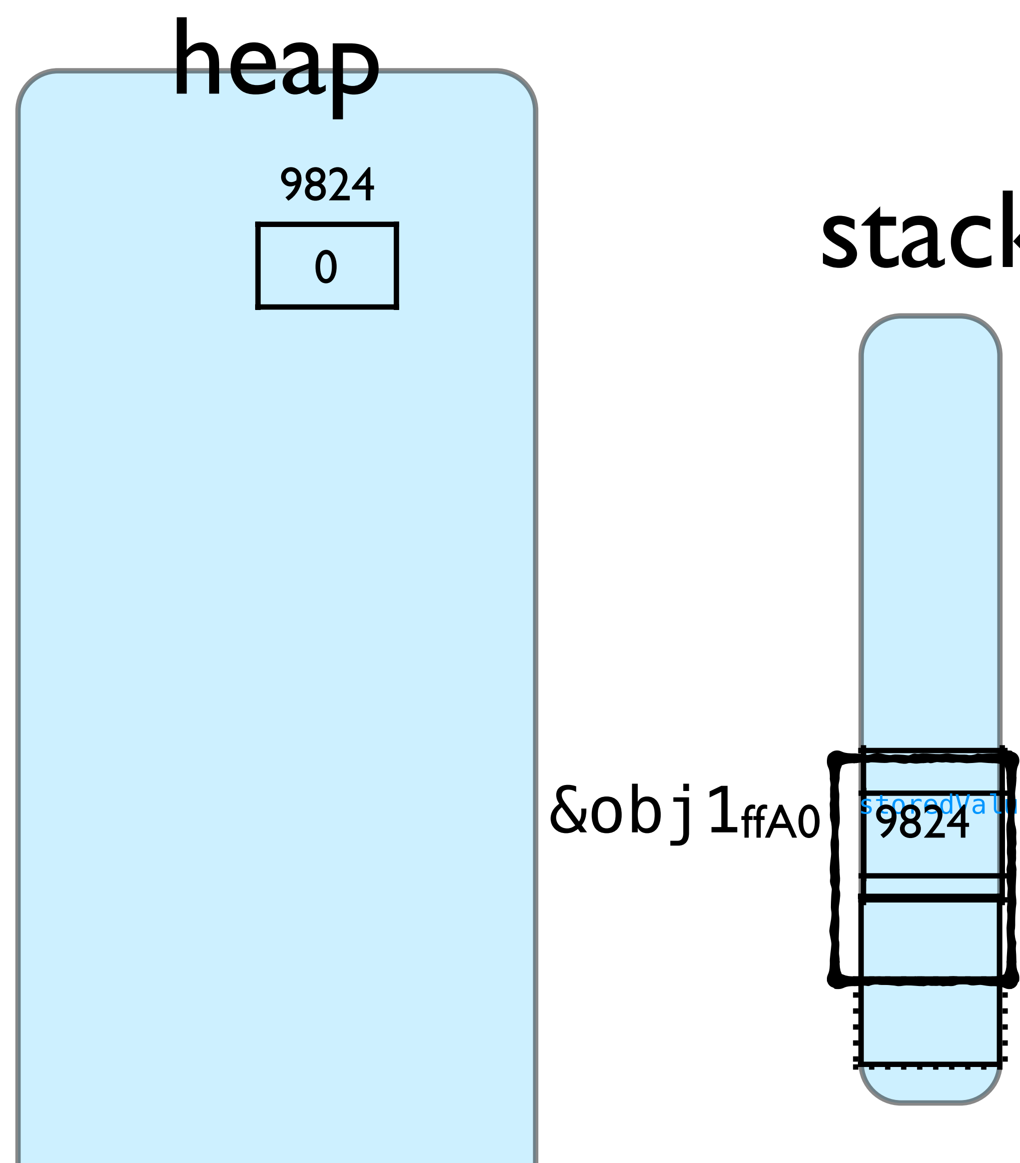
Destructor

Called when an object goes out of scope,
or when it is subjected to a delete

The Destructor

The destructor does the cleanup. One of the most important jobs is freeing memory in the heap created by the object. By writing a destructor we solved the memory leak problem we saw in the last slide

```
class IntCell
{
public:
   explicit IntCell(int initialValue = 0)
   {storedValue = new int(initialValue);}

   IntCell(const IntCell& rhs);
    ~IntCell();

   int read() const;
   void write(int x);

private:
   int* storedValue;
};

IntCell::~IntCell()
{
  delete storedValue;
}

void silly()
{
  IntCell  obj1;
  return;
}

int main ()
{
   silly();
```

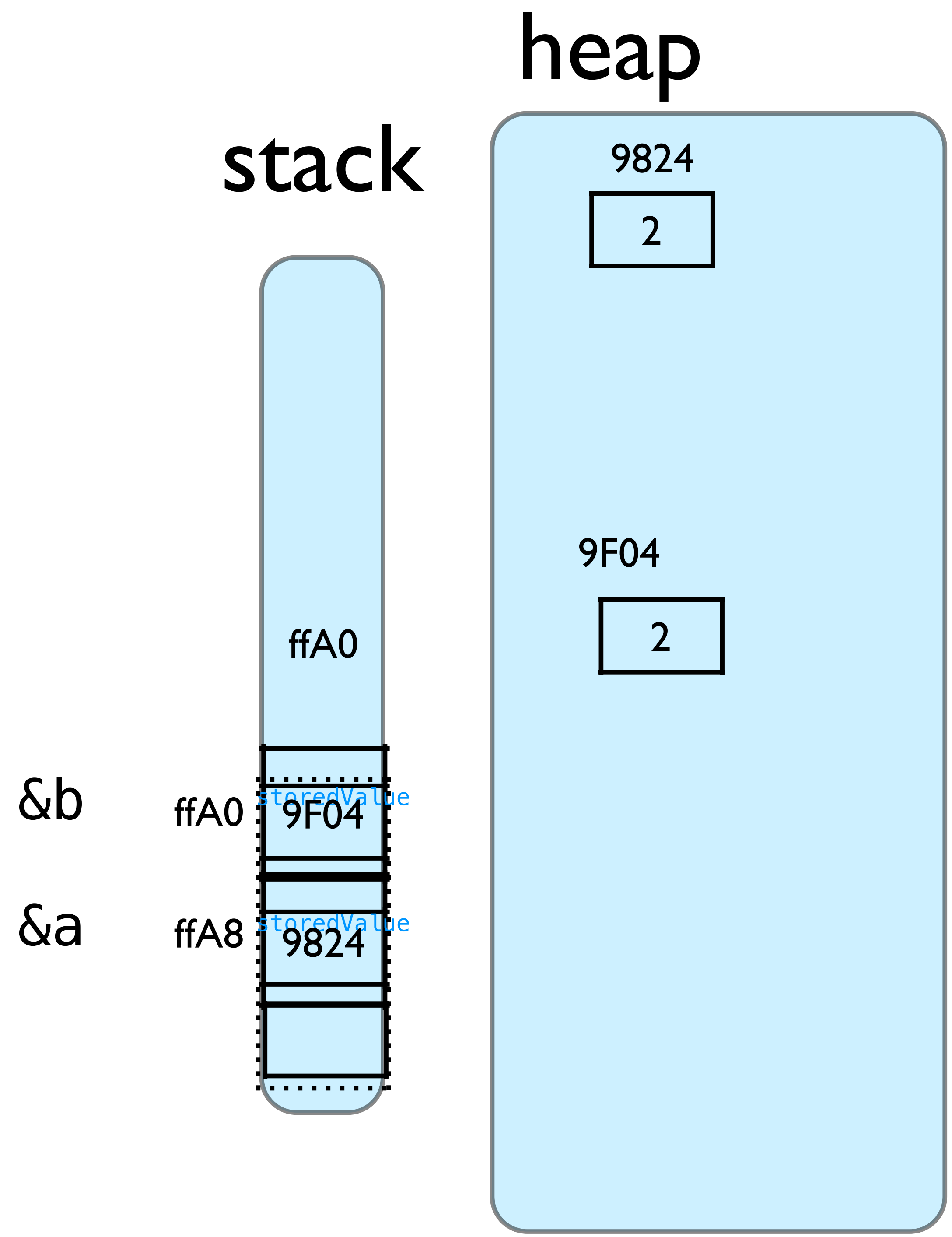


Copy Constructor, Move Constructor

- Called when constructing a new object to be initialized to the same state as another object of the same type
- For each example below, the copy constructor is call if C is an lvalue, otherwise the move constructor is called if C is an rvalue
  - IntCell B = C;
  - IntCell B {C};
- Defaults typically don't work when a data member is a pointer

The Copy Constructor

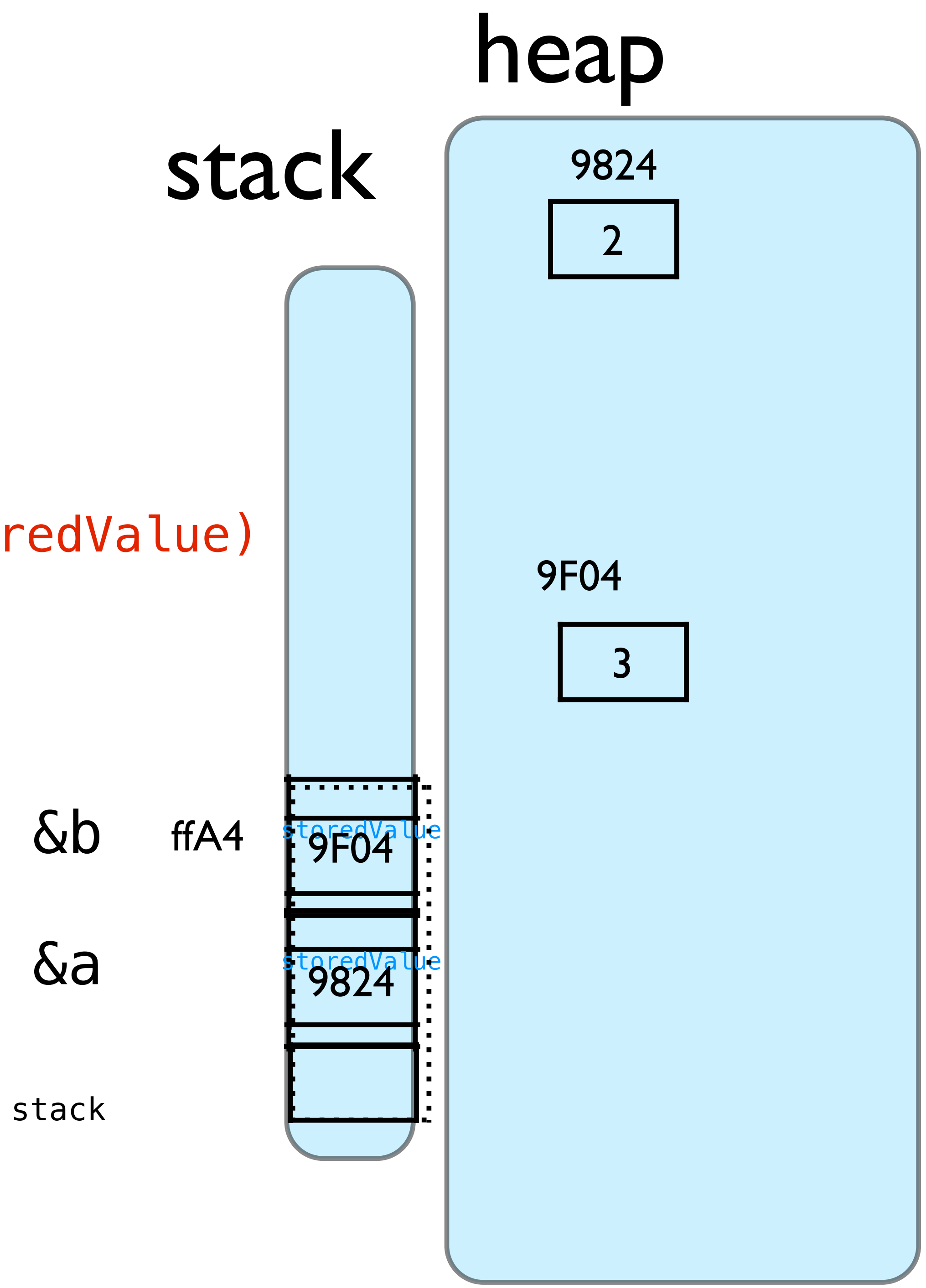
```
class IntCell
{
public:
   explicit IntCell(int initialValue = 0)
   {storedValue = new int(initialValue);}

   IntCell(const IntCell& rhs);

   int read() const;
   void write(int x);
    …
private:
   int* storedValue;
};


IntCell::IntCell(const IntCell & rhs)
{
     storedValue = new int( *rhs.storedValue );
}


int main ()
{
    IntCell a(2);
    IntCell b(a);
```



The Move Constructor

```
class IntCell
{
public:
   explicit IntCell(int initialValue = 0)
   {storedValue = new int(initialValue);}

   IntCell(const IntCell& rhs);
   IntCell(IntCell && rhs);

   int read() const;
   void write(int x);

private:
   int* storedValue;
};


IntCell::IntCell(IntCell && rhs):storedValue(rhs.storedValue)
{
    rhs.storedValue = nullptr;
}


int main ()
{
    IntCell a(2); // I am not showing the steps in the function call stack
    IntCell b = IntCell(3);
}
```

Some compilers optimize…
They do return value optimization - which omits certain copies when returning a value



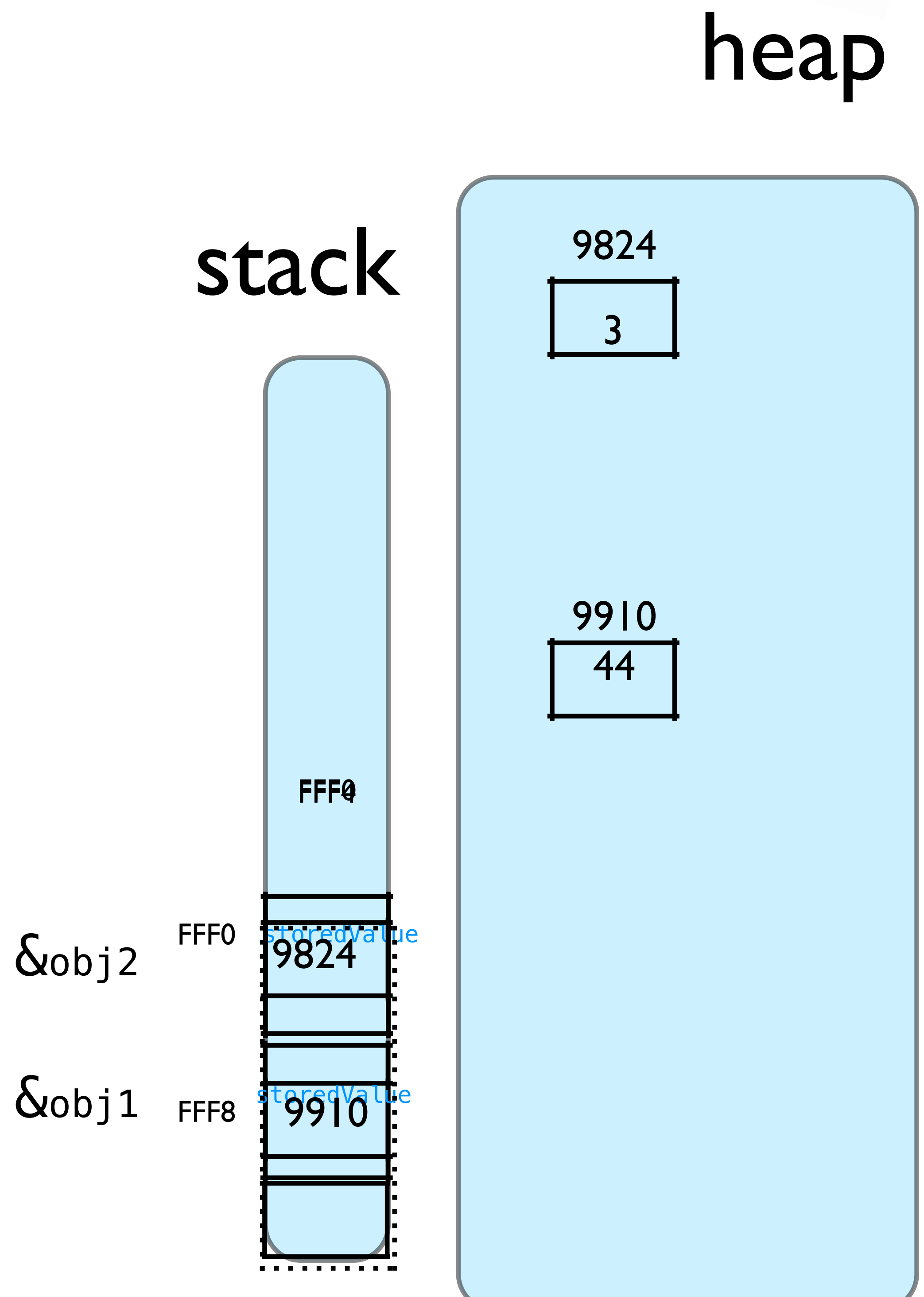
Copy Assignment, Move Assignment

- For the example below, the copy assignment is called if rhs is an lvalue, otherwise the move assignment is called if rhs is an rvalue
  - lhs = rhs; // where lhs and rhs are previous constructed objects
- Defaults typically don't work when a data member in the class is a pointer

Copy Assignment Operator=

```
class IntCell
{
public:
   explicit IntCell(int initialValue = 0)
   {storedValue = new int(initialValue);}

   IntCell & operator=(const IntCell & rhs);

    int read() const {return *storedValue;}
    void write(int x) {*storedValue = x;}
   …
private:
   int* storedValue;
};
IntCell & IntCell::operator=(const IntCell& rhs)
{
 if( this != & rhs )
    *storedValue = *rhs.storedValue;
 return *this;
}
int main ()
{

   IntCell  obj1(44);
   IntCell  obj2;
   cout << obj1.read() << endl;
   obj2 = obj1;
   obj2.write(3);
   cout << obj1.read() << endl;
```

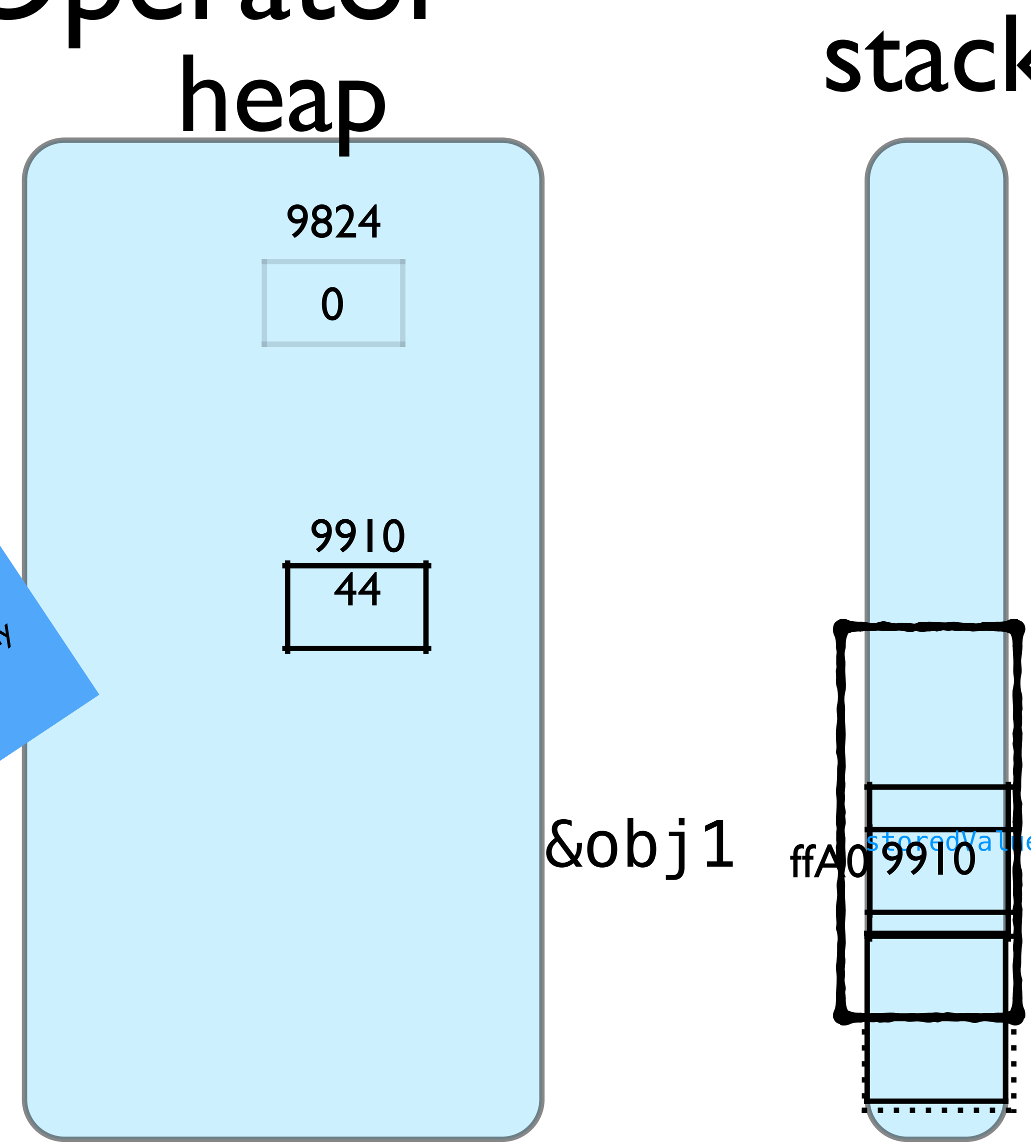
By declaring our own copy assignment operator we ensure that each IntCell points to its own memory location in the heap.



Move Assignment Operator=

```
class IntCell
{
public:
   explicit IntCell(int initialValue = 0)
   {storedValue = new int(initialValue);}

   IntCell & operator=(const IntCell & rhs);
   IntCell & operator=(IntCell && rhs);

   int read() const;
   void write(int x);
    …
private:
   int* storedValue;
};
```

"… moving implies that the moved-from object is left in a valid but unspecified state. Which means that, after such an operation, the value of the moved-from object should only be destroyed or assigned a new value; accessing it otherwise yields an unspecified value."




```
IntCell & IntCell::operator=(IntCell && rhs)
{
    int * tmp(storedValue);
    storedValue = rhs.storedValue;
    rhs.storedValue = tmp;
    return *this;
}
```

$\}$ `std::swap( storedValue, rhs.storedValue );`

```
int main ()
{
  IntCell  obj1;
  obj1 = Intcell(44);
}
```