More C++

- The big five
- Templates
- Functors

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=
- Move Assignment operator=
- Copy Constructor
- Move Constructor
- Destructor

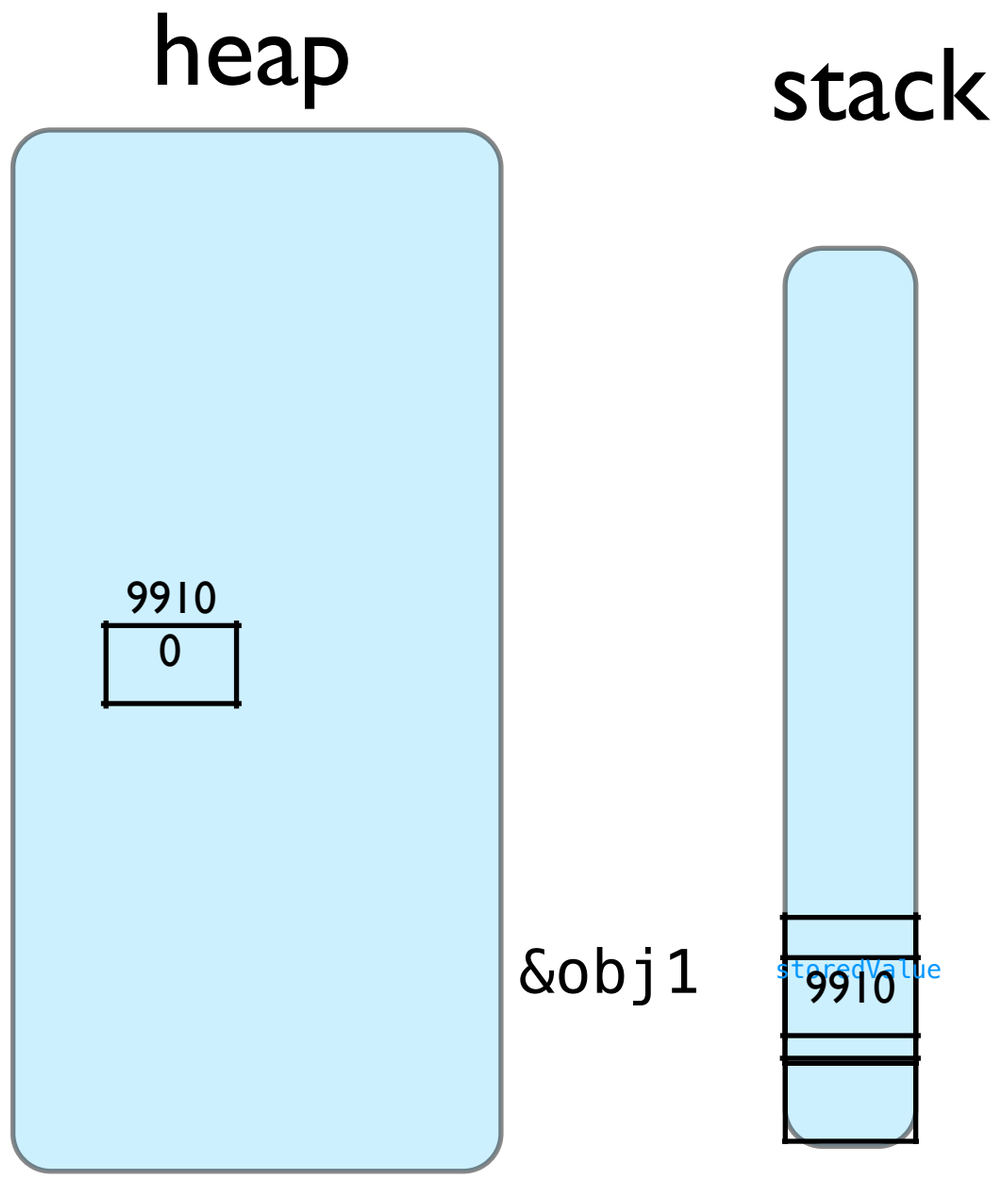
Often you can use these five functions (you can choose to not use these by writing your own function or by telling the compiler not to use the default). If your object has one or more member variables which are pointers, the behavior of these five default functions will probably not be what you intended.

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.

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

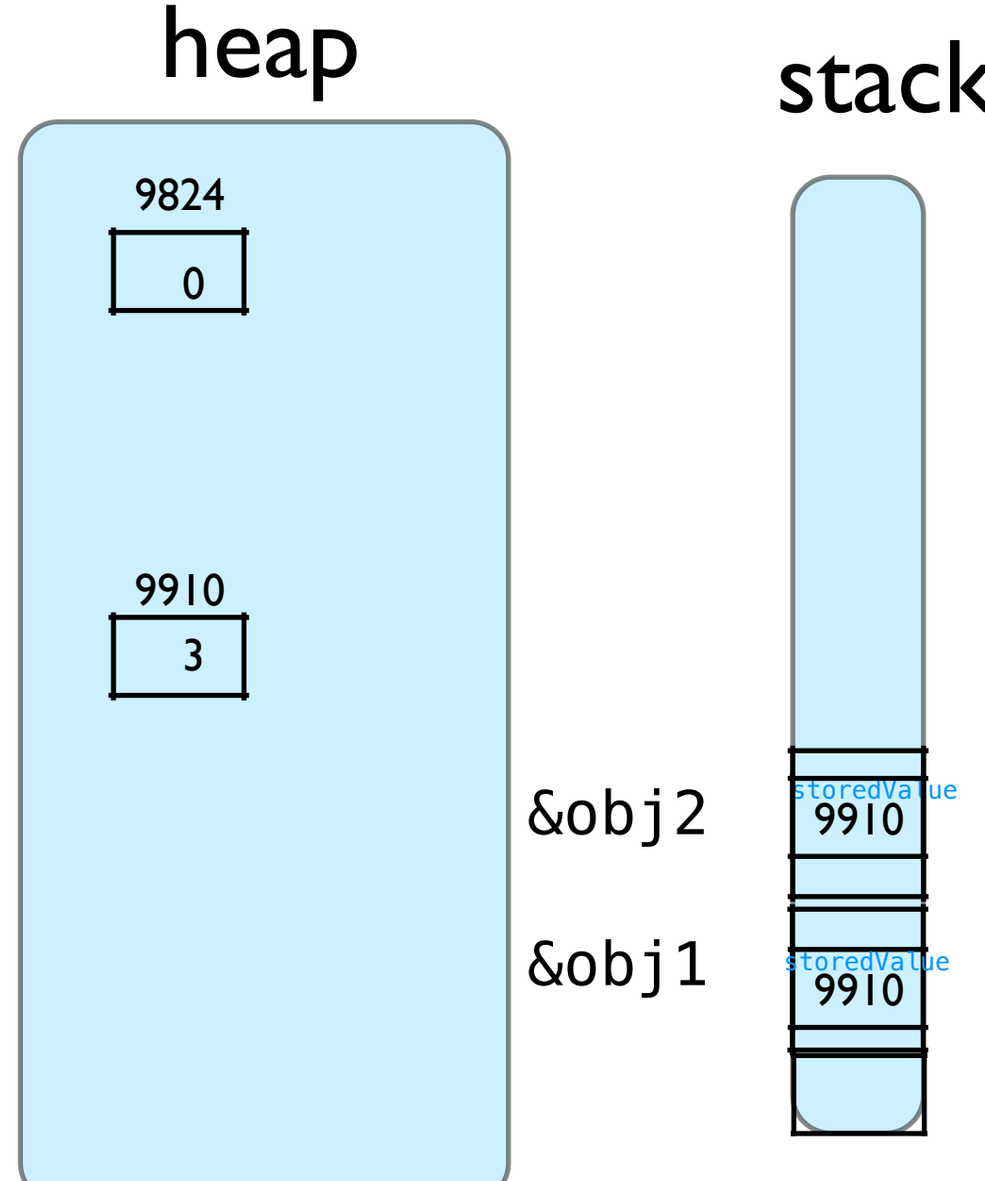
```
class IntCell
{
public:
   explicit IntCell(int initialValue = 0)
     {storedValue = new int(initialValue);}

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


int main{

    IntCell  obj1;
    …


 }
```



Is this what we expected?

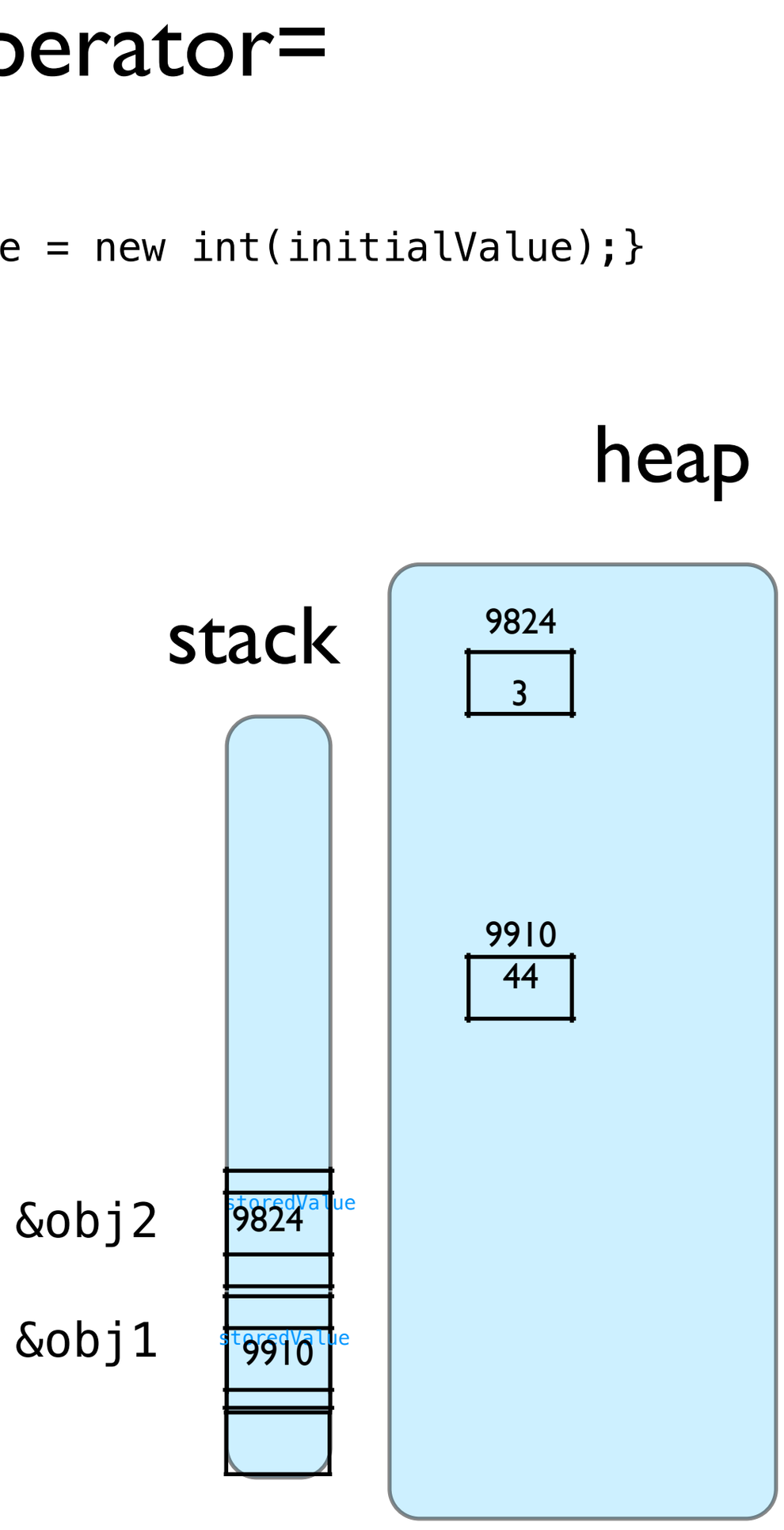
```
class IntCell
{
 public:
    explicit IntCell(int initialValue = 0)
      {storedValue = new int(initialValue);}

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



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

?



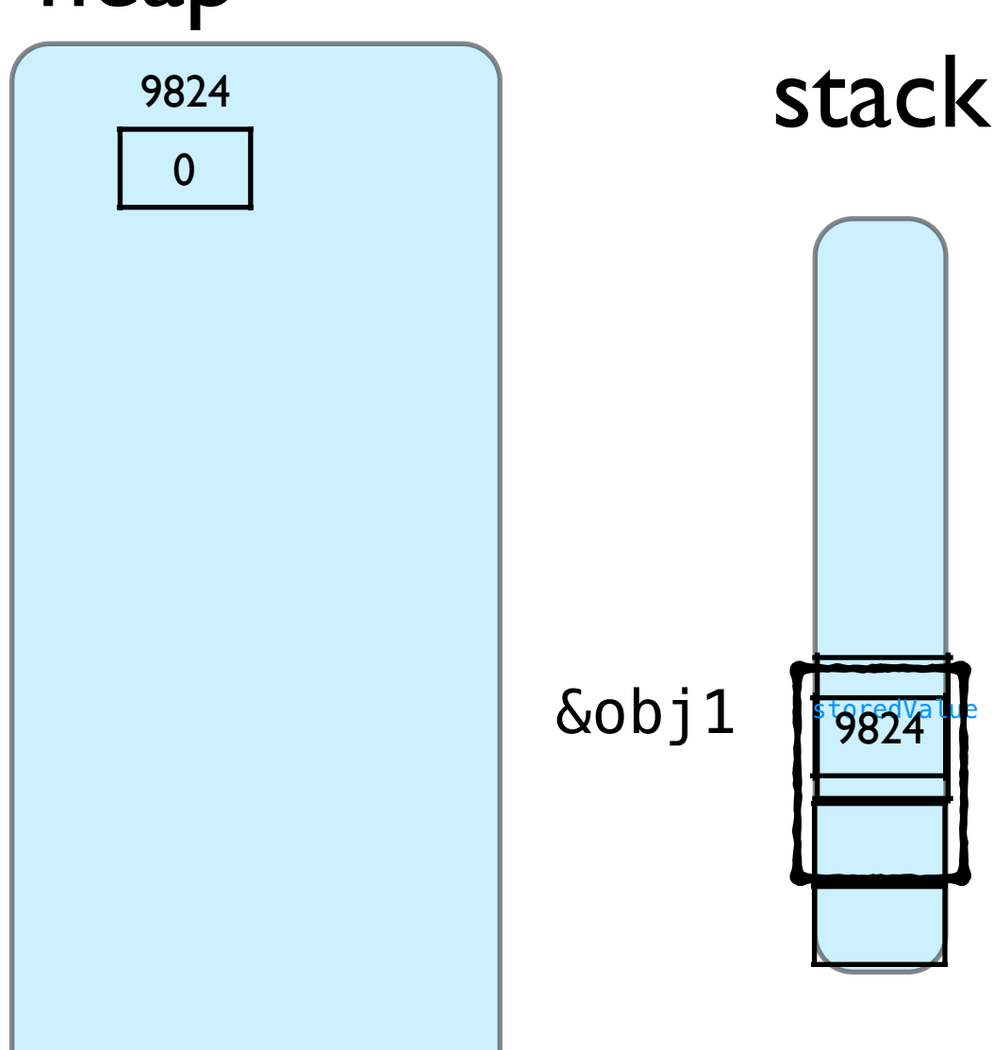
Copy Assignment Operator=

```
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;
```



Is this what we expected?

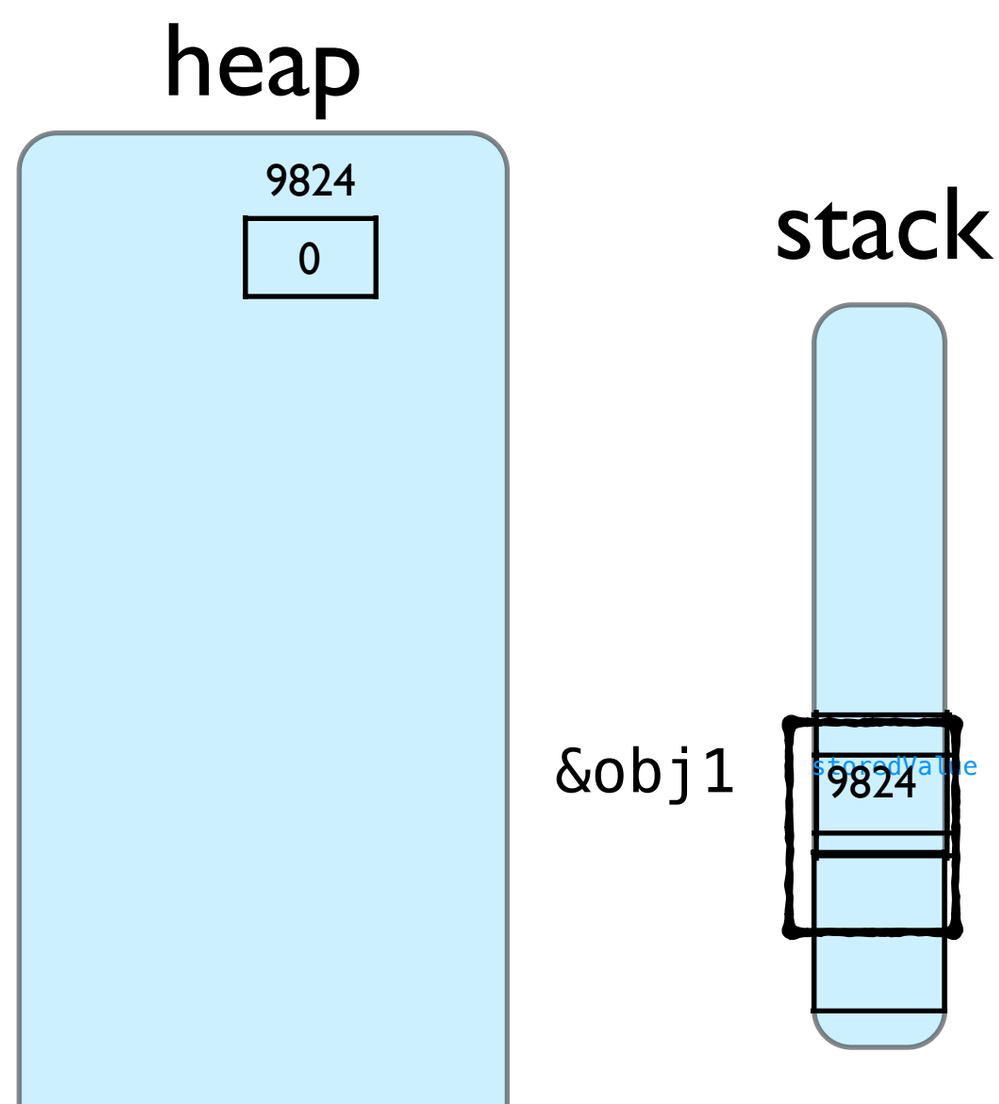
```
class IntCell
{
public:
   explicit IntCell(int initialValue = 0);

   IntCell(const IntCell& rhs);

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

private:
   int* storedValue;
 };


void silly()
{
   IntCell  obj1;
   return;
}


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



The Destructor

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



Move Assignment Operator=

If you are interested in learning more:
http://thbecker.net/articles/rvalue_references/section_01.html
or
https://channel9.msdn.com/Series/C9-Lectures-Stephan-T-Lavavej-Standard-Template-Library-STL-/C9-Lectures-Stephan-T-Lavavej-Standard-Template-Library-STL-9-of-n

```
class IntCell
{
public:
   explicit IntCell(int initialValue = 0);
   IntCell & operator=(const IntCell & rhs);
   IntCell & operator=(IntCell && rhs);

   int read() const;
   void write(int x);
   …
private:
   int* storedValue;
};
IntCell & IntCell::operator=(IntCell && rhs)
{
    int * tmp(storedValue);
    storedValue = rhs.storedValue;
    rhs.storedValue = tmp;
    return *this;
}
int main ()
{
   IntCell  obj1;
   obj1 = Intcell(44);
}
```

} std::swap(storedValue, rhs.storedValue);

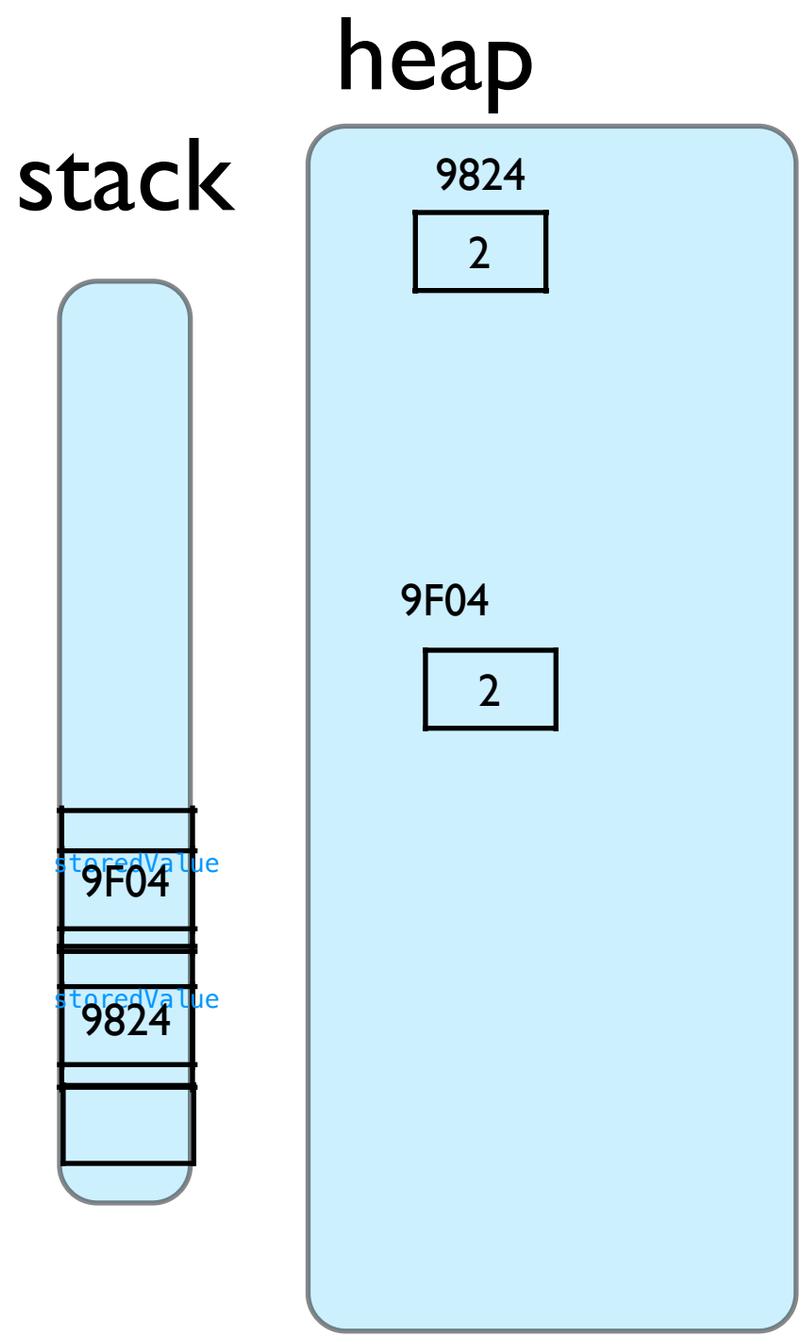
The move assignment operator needs to leaves the object in a state that can be destructed.

The Copy Constructor

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

   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

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

```
class IntCell
{
public:
   explicit IntCell(int initialValue = 0);
   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);
    IntCell b = IntCell(2);
}
```

Generic Programming

CS2134

Template Motivation

```
void Swap(string& x; string& y)
{ string tmp(move(x));
  x = move(y);
  y = move(tmp);
}
void Swap(vector<int>& x; vector<int>& y)
{ vector<int> tmp (move(x));
  x = move(y);
  y = move(tmp);
}
```

Almost identical.
Can sometimes avoid writing "same" code twice by defining Swap(Object&, Object&) and using
typedef string Object or
typedef vector<int> Object

We cannot do this if we want to swap strings and swap vectors in same program.

template

"a preset format for a document or file, used so that the format does not have to be recreated each time it is used: *a memo template*."

from the dictionary on my computer

Templates

- Logic doesn't depend on type.
- Not an actual function/class!
- Compiler instantiates the function template (one instantiation for each type used.) Compiler deduces the type.
- Used in *classes* and *functions*.

Syntax:

template<class Type>
function declaration

template<class Type>
class declaration

The template parameter cannot* be deduced from the return value

*There is one exception to this rule

Brace initializer does not have a type… e.g f({1,2,3}) doesn't work since template type deduction fails.

Have the compiler write the function!

```
void Swap( string& lhs, string& rhs)
{
   string tmp(move(lhs));
   lhs = move(rhs);
   rhs = move(tmp);
}
```

```
void Swap( vector<int>& lhs, vector<in>& rhs)
{
   vector<int> tmp(move(lhs));
   lhs = move(rhs);
   rhs = move(tmp);
}
```

A function template is sometimes called a parameterized function or an algorithm.

```
template <class Object>
void Swap( Object& lhs, Object& rhs)
{
   Object tmp(move(lhs));
   lhs = move(rhs);
   rhs = move(tmp);
}
int main()
{
   string f = 'Hello';
   string l = 'Hi';
   Swap(f,l);
   vector<int> a = {4,5};
   vector<int> b = {1,2};
   Swap(a,b);
   Swap(f,b);
   return 0;
}
```

<u>Function Template</u>
Design for a function. Not an actual function.

The compiler can deduce the parameter type from the arguments!

<u>Instantiation</u>
Compiler generates a new function with the correct type. The compiler then checks to make sure the function works with this type.

How would you return the larger of two items?

```
const string& Max(const string& first,
                  const string& second)
{
   return (first > second)? first: second;
}
```

```
const int& Max(const int& first,
               const int& second)
{
   return (first > second)? first: second;
}
```

```
template<class Object>
const Object& Max(const Object& first,
                  const Object& second)
{
    return (first > second)?first: second;
}

 int main()
 {
    cout << Max( 7, 9);

    cout << Max(string("Hello"), string("World"));

    cout << Max(7, string("World"));


 }
```

Finding an item in a vector

Template example: finding an item in a vector

When writing the template - provide a comment to describe the requirements!

```
int find(vector<int> & a, const int & search_item)
{
   for(int i=0; i < a.size(); i++)
    if (a[i] == search_item)
        return i;
   return -1;
}
```

```
int find(vector<char> & a, const char search_item)
{
   for(int i=0; i < a.size(); i++)
    if (a[i] == search_item)
        return i;
   return -1;
}
```

```
int find(vector<double> & a, const double & search_item) }
{
   for(int i=0; i < a.size(); i++)
    if (a[i] == search_item)
        return i;
   return -1;
}
```

```
int find(vector<string> & a, const string & search_item)
{
   for(int i=0; i < a.size(); i++)
    if (a[i] == search_item)
        return i;
   return -1;
}
```

```
/**
* Return the index of the search_item
* in array a if it exists,ow return -1
*
* Object must have
*    operator==
*/
 template < class Object>

 int find(vector<Object> & a,
          const Object & search_item)
 {
    for(int i=0; i < a.size(); i++)
     if (a[i] == search_item)
         return i;
    return -1;
```

```
int main()
{
    vector<string> a(3);
    a[0] = “cat”;
    a[1] = “dog”;
    a[2] = “zebra”;
    string my = “dog”;
    if( find(a, my) == -1)
       cout << “...”;
   vector<IntCell> b(3);
    b[0].write(1);
    b[1].write(2);
    b[2].write(3);

    if( find(b,IntCell(2))==-1 )
       cout << “...”;
```

When instantiating a function from a template the compiler checks to make sure the function works with this type.

Error!!!

Template example: functions to find the largest item in a vector

```
int findMax(const vector<int> & a)
{
   int maxIndex = 0;

   for(int i=1; i < a.size(); i++)
    if (a[maxIndex] < a[i])
        maxIndex = i;
   return maxIndex;
}
```

```
int findMax(const vector<char> & a)
{
  int maxIndex = 0;

   for(int i=1; i < a.size(); i++)
    if (a[maxIndex] < a[i])
        maxIndex = i;
   return maxIndex;
}
```

```
int findMax(const vector<double> & a)
{
  int maxIndex = 0;

   for(int i=1; i < a.size(); i++)
    if (a[maxIndex] < a[i])
        maxIndex = i;
   return maxIndex;
}
```

When instantiating a function from a template the compiler checks to make sure the function works with this type.

Error!!!

```
/**
* Return the index of the maximum item
* in the array a
* Assume a.size() > 0
* Comparable objects must have
* operator<
*/
 template < class Comparable>
 int findMax(const vector<Comparable> & a
 {
   int maxIndex = 0;

    for(int i=1; i < a.size(); i++)
     if (a[maxIndex] < a[i])
         maxIndex = i;
    return maxIndex;
 }
int main()
{
   vector<char> a(5);
   ... // code to enter 5 items into a
   int i = findMax(a);
      cout << "...";
   vector<int> b(5);
   ... // code to enter 5 items into b
   int j = findMax(b);
      cout << "...";

  vector<IntCell> c(5);
   ... // code to enter 5 items into c

   int k = findMax(c);
     cout << "...";
```

A class template is a type generator.

A class template is sometimes called a parameterized type or parameterized class.

Similarly, we defi mplate classes:

No type deduction!

```
template <class Object>
class MyClass
{   Object   data[5];
    ...
}

int main()
{
    MyClass<int> x;
    MyClass<double> y;
    ...
}
```

Notice that the compiler cannot deduce the type!

Generating a class from a templated class is Specialization or template instantiation. Remember that in a templated class you need to provide the template arguments.

A simpler IntCell Class:

```
class IntCell2
{
public:
   explicit IntCell2(int initialValue = 0):storedValue(initialValue){}

   int read() const {return storedValue;}
   void write(int x){storedValue = x;}

private:
   int storedValue;
};
```

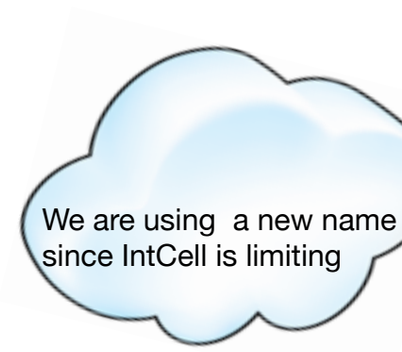
Could we store a double instead of an int?

How about storing a char instead of an int?

```
class MemoryCell
{
public:
   explicit MemoryCell(double initialValue = 0)
     :storedValue(initialValue){}

   double read() const {return storedValue;}
   void write(double x){storedValue = x;}

private:
   double storedValue;
};
```

```
class MemoryCell
{
public:
   explicit MemoryCell(char initialValue = ‘’)
     :storedValue(initialValue){}

   char read() const {return storedValue;}
   void write(char x){storedValue = x;}

private:
   char storedValue;
};
```

```
template<class Object>
class MemoryCell
{
public:
   explicit MemoryCell(const Object& initialValue = Object())
     :storedValue(initialValue){}
   //public member functions

   Object& read() const {return storedValue;}
   void write(const Object& x){storedValue = x;}

private:
   Object storedValue;
};
int main()
{
   MemoryCell<double> d;
   d.write( 5.0 );
   cout << “Cell contents are “ << d.read() << endl;
   MemoryCell<char> c(‘a’);
   cout << “Cell contents are “ << c.read() << endl;
   return 0;
```



Typical layout for member implementation

```
template<class Object>
class MemoryCell
{
public:
   explicit MemoryCell(const Object& initialValue = Object()):storedValue(initialValue){}
   //public member functions
    const MemoryCell& operator=(const MemoryCell& rhs);
   const Object& read() const {return storedValue;}
   void write(const Object& x){storedValue = x;}

private:
   Object storedValue;
};
```

```
template<class Object>
const MemoryCell<Object>& MemoryCell<Object>::operator=(const MemoryCell<Object>& rhs)
{
   if (this != &rhs)
    storedValue = rhs.storedValue;
   return *this;
}

int main()
{
   MemoryCell<int> m;

   m.write( 5 );
   cout << “Cell contents are “ << m.read() << endl;

   return 0;
}
```

Notice that MemoryCell is not a class!
It is a class template.
MemoryCell<int> and MemoryCell<char> are the actual classes.

Functors

A functor is used for:
* Customizing sorting algorithms
* In comparing two items, determining what it means for two items to be equal
* Customizing numerical algorithms
* Customizing searching algorithms

Flexibility & Generality
A functor is the main method of parameterization in the STL.

Functor Motivation

```
class student
{
  private:
    string name;
    double gpa;
   ...
  public:
     string get_name();
     double get_gpa();
     bool operator<(const student& rhs){ ...};
  ...
};
```

```
{ return (name<rhs.name)? 1: 0;}
```

```
{ return (gpa<rhs.gpa)? 1: 0;}
```

```
template<class Comparable>
Const Comparable & findMax(const vector<Comparable> & a)
{
        Int maxIndex = 0;
        for( int i = 1; i < a.size( ); ++i )
           if (a[maxIndex] < a[i])
                maxIndex = i;
        return a[maxIndex];
}
```

Meaning of student::operator< fixed at compile time.
What if we want to compare students by name and by gpa at different points within the same program?

```
class LessThanByGPA
{ public:
    bool IsLessThan(const student& lhs, const student& rhs) const
    {return lhs.get_gpa() < rhs.get_gpa();}
};


class LessThanByName
{ public:
    bool IsLessThan(const student& lhs, const student& rhs) const
    {return lhs.get_name() < rhs.get_name();}
};
```

The second template parameter is a class which has a member function called isLessThan

```
// Generic findMax, with a function object.

// Precondition: a.size( ) > 0.

// class Comparator functor with method IsLessThan

template <class Object, class Comparator>

const Object & findMax( const vector<Object> & a,  Comparator comp )

{   int maxIndex = 0;

    for( int i = 1; i < a.size( ); i++ )

        if( comp.isLessThan( a[ maxIndex ], a[ i ] ) )

            maxIndex = i;

    return a[ maxIndex ];

}
int main()
{ vector<student> a;
  ...
  student  i = findMax(a, LessThanByName());
  student  j = findMax(a, LessThanByGPA());
  ...
}
```

Functors (Function Objects)

- Class that can be useful with no data members and a single method
- Can pass a functor as a template parameter, just like other classes
- This effectively passes the member function as a parameter
- Example: pass LessThanByGPA or LessThanByName to findmax

Cleaner Syntax

- Overloaded operator() as a method in a function object
- This is the convention used in STL
- IsLessThan(....) is call to IsLessThan.()

Both classes have the overloaded operator()

```
class LessThanByGPA
{ public:
    bool operator( )(const student& lhs, const student& rhs) const
    {return lhs.get_gpa() < rhs.get_gpa();}
};

class LessThanByName
{ public:
    bool operator( )(const student& lhs, const student& rhs) const
    {return lhs.get_name() < rhs.get_name();}
};

int main()
{ vector<student> a;
  ...
  student i = findMax(a, LessThanByName());
  student j = findMax(a, LessThanByGPA());
  ...
}
```

```
template <class Object, class Comparator>
const Object & findMax( const vector<Object> & a, Comparator comp )
{
    int maxIndex = 0;

    for( int i = 1; i < a.size( ); i++ )
        if( comp( a[ maxIndex ], a[ i ] ) )
            maxIndex = i;

    return a[ maxIndex ];
}

class LessThanByGPA
{ public:
     bool operator()(const student& lhs, const student& rhs) const
     {return lhs.get_gpa() < rhs.get_gpa();}
};
```

```
class LessThanByName
{
public:
   bool operator()(const student & lhs,  student & rhs) const
   {return lhs.get_name() < rhs.get_name();}
};

int main()
{
    vector<student> classList;
   //some code to fill the classList, etc
    student st1 = findMax( classList, LessThanByName() );
    student st2 = findMax( classList, LessThanByGPA() );
}
```

function object examples less

```
template <class Object>
class less
{ public:
      bool operator()(const Object& lhs, const Object& rhs) const
          {return lhs < rhs;}
};
```

greater_equal

```
template <class T>
class greater_equal
{
  public:
      bool operator() (const T& lhs, const T& rhs) const
        {return lhs >= rhs;}
};
```

We can pass any object to a function

```
template<class Object, class Comparator>
Object choose(Object item1, Object item2, Comparator comp)
{
    return comp(item1, item2)? item1: item2;
}

template <class Object>
class less
{ public:
     bool operator()(const Object& lhs, const Object& rhs) const
        {return lhs < rhs;}
};

template <class T>
class greater_equal
{
  public:
     bool operator() (const T& lhs, const T& rhs) const
       {return lhs >= rhs;}
};
```

Functor Example

Modified from http://www.stroustrup.com/bs_faq2.html#this

```
class Sum {
   int val;
public:
   Sum(int i=0) :val(i) { }
   operator int() const { return val; }    // extract value
   int operator()(int i) { return val+=i; }   // application
};
```

functor
Capable of maintaining a state. The state can be examined from the outside (static variables cannot be examined from the outside.)

Conversion operator is a member function. It cannot modify the member variables. Note that the syntax is odd. It has no return type:
operator type()const;