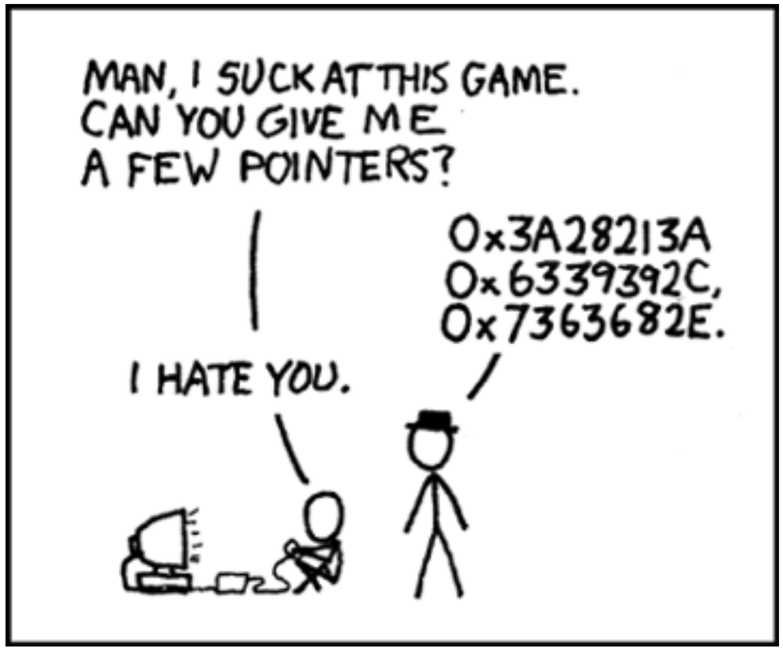
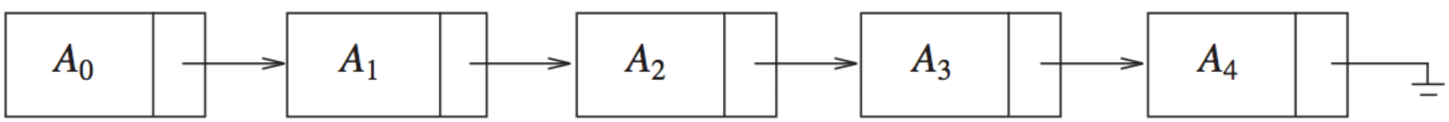
Intro to C++

- Just like Java... except for...
  - Full read/write access to memory pointers and references
    - Java only allows re-assignment
  - Manual memory allocation/deallocation
    - i.e. no garbage collection
  - Compilation directly to machine code
  - Different built-in libraries (of course)
  - Interfaces don't exist – multiple inheritance of classes
    - "virtual" classes can serve as interfaces
  - Where Java passes primitives by value and objects by reference, in C++ you get to choose

Pointers


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

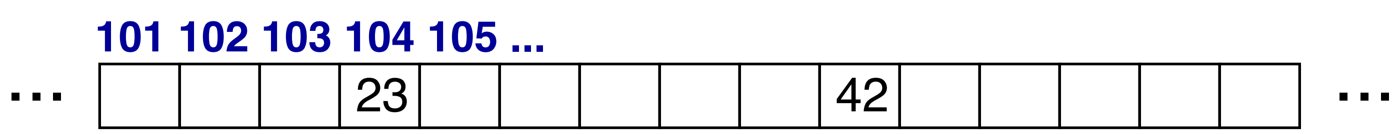

A_0
A_1
A_2
A_3
A_4

Pointers

- A **pointer variable** is a variable that stores the memory address where another object resides. It *points* to a memory location
- Used as fundamental tool in many data structures. Why are pointers useful?
- Many reasons! Most notably:
  - Dynamic-sized structures
  - Lower memory overhead from function arguments
  - Non-contiguous data representations (e.g. linked-list)

Memory Address vs. Value Stored

- Consider memory to be a single huge array:
  - Each cell of the array has an address associated with it.
  - Each cell also stores some value.
- Don’t confuse the address referring to a memory location with the value stored in that location.



C++ pointer syntax

Declaration:

<type> * <variable name>;

Examples:

```
int* counter;
Dog* dogs;    (where dogs is an array of Dogs)
```

Why does the <type> need to be there?

We can access the data directly from the pointer, and to do that we need to know the size of the data it points to.

C++ pointer syntax

- As seen in pointer declaration, the ‘*’ on the left side of the assignment means it is a pointer type
- However, when the ‘*’ is not in the declaration, it is called the **dereference operator**, which returns the data at the memory address that the pointer stores.
  - Example:

```
Dog* dog1;
Dog dog2 = *dog1;
```

C++ pointer syntax

- **&**, the **address-of operator**: returns the virtual memory address of any variable – the opposite of the dereference operator
  - Example:
    ```
    int x = 5;
    int * y = &x;
    ```
- We can use the dereference operator on the left side to change values, like this:
  ```
  *y += 2;
  ```
  What are the values of x and *y ?

C++ pointer syntax

Initialization:

<type> * <variable name> = & (<type>) variable;

Example:
```
int blah = 5;
int* counter;
counter = &blah;
```

- OR -

<type> * <variable name> = new <type>();

Example:

```
Dog * dog1 = new Dog(); // calls the
                        // constructor for Dog
```

What's up with new?

- As you (hopefully) remember from Java, the 'new' keyword is used to allocate memory
- In C++, 'new' returns the **address-of** the newly allocated object — not the object itself
  - Note: even though it is valid in C++, do **NOT** use 'malloc', 'realloc', or 'calloc' – these are for C code and 'new' is used in C++
- The following are **valid** in C++:
  - `Dog dog1 = Dog();`
  - `Dog* dogptr = new Dog();`
- The following are **invalid** in C++:
  - `Dog dog1 = new Dog();  // valid in Java`
  - `Dog* dogptr = Dog();`

Memory Allocation

- Two ways:
  - On the stack
  - On the heap
- Reminder:
  - The heap **is not** like the heap data structure: a collection of memory blocks that may be fragmented (because of **manually** deallocating memory)
  - The stack **is** like the stack data structure: a LIFO structure that stores the local variables, and no manual deallocation is necessary

Memory Allocation (Stack)

```
int main(void) {
  int x(5);
  if (x > 3) {
    int y(6);
    cout << (x + y) << endl;
  }
}
```

Memory Allocation (Heap)

```
int main(void) {
  int *x = new int(5);
  if (*x > 3) {
    int *y = new int(6);
    cout << (*x + *y) << endl;
  }
}
```

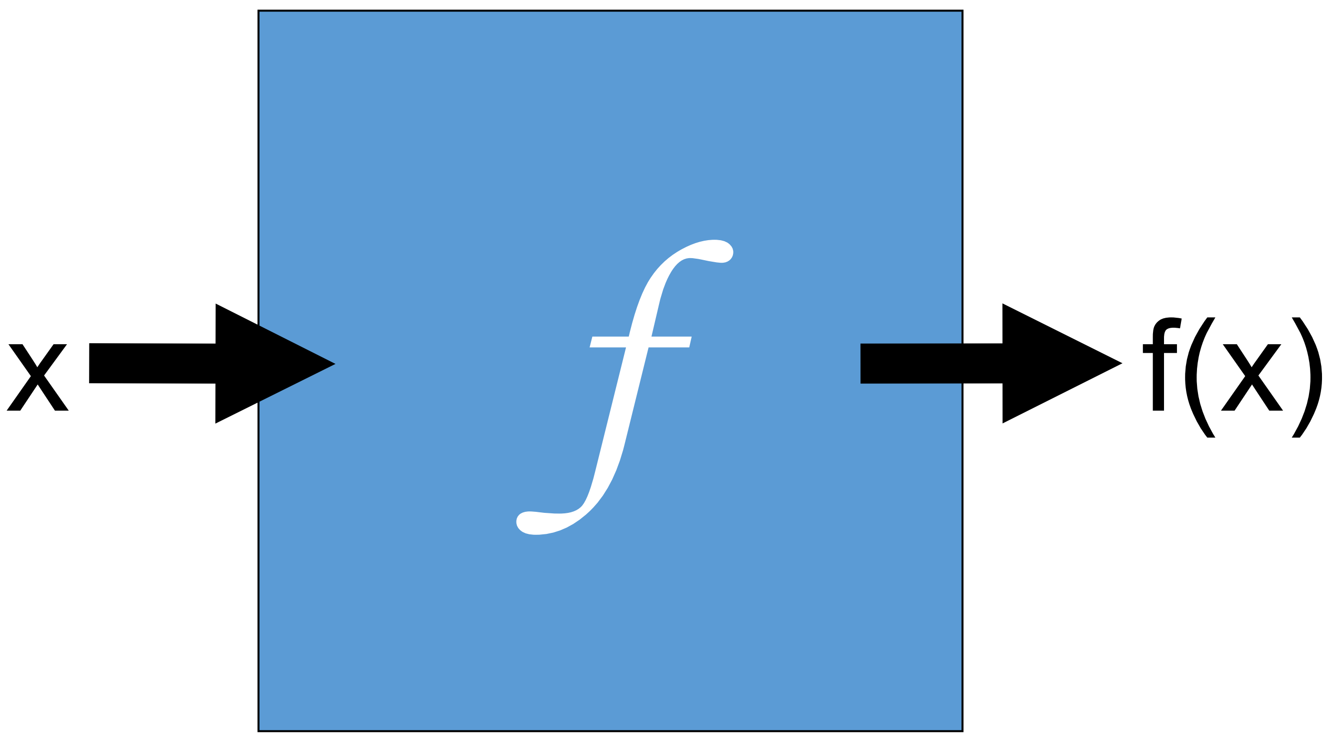
Memory De-allocation (Heap)

```
int main(void) {
  int *x = new int(5);
  if (*x > 3) {
    int *y = new int(6);
    cout << (*x + *y) << endl;
     delete y;
  }
 delete x;
}
```

Array Allocation (Heap)

```
int main(void) {
  int *x = new int[5];
  *x = 5;
  if (*x > 3) {
    int *y = new int(6);
    cout << (*x + *y) << endl;
    delete y;
  }
  delete [] x;
}
```

Functions



Call-by-value

```
int sum (int x, int y) {
  return x + y;
}

int main(void) {
  cout << sum(5,6) << endl;
}
```

Still call-by-value

```
int sum (int* x, int* y) {
  int sum = *x + *y;
  *x += 5; // actually changes blah1 to 10
  x += 4;  // these changes are not reflected
  y -= 7;  //   on the addresses passed
      return sum;
}

int main(){
 int blah1 = 5;
 int blah2 = 7;
 std::cout << sum(&blah1,&blah2) << std::endl;
}
```

Call-by-reference

```
int sum(const int& x, const int& y) {
  return x + y;
}

int main(void) {
  int x (5);
  int y (6);
  cout << sum(x,y) << endl;
}
```

Question 1

What is the output from the following code:

```
double x = 5.5;
double *px = &x;
cout << *px << endl;
*px = 10.0;
cout << x << endl;
```

Question 1

What is the output from the following code:

```
double x = 5.5;
double *px = &x;
cout << *px << endl;
*px = 10.0;
cout << x << endl;

// 5.5
// 10
```

Question 2

What is the output from the following code:

```
double x = 5.5;
double y = 10.0;
double* px, py;
px = &x;
py = &y;
cout << *px << endl << *py << endl;

// will not compile! py is actually a double, not
// a pointer so line 5 throws an error
```

Question 2

What is the output from the following code:

```
double x = 5.5;
double y = 10.0;
double* px, py;
px = &x;
py = &y;
cout << *px << endl << *py << endl;
```

Question 3

What is the output from the following code:

```
double x = 5.5;
double *px = &x;
*px = 3.14;
double& r = *px;
r = 99.44;
cout << x << endl;
```

Question 3

What is the output from the following code:

```
double x = 5.5;
double *px = &x;
*px = 3.14;
double& r = *px;
r = 99.44;
cout << x << endl;

// 99.4
```

Lvalues and Rvalues

- An lvalue is an expression that identifies a non-temporary object
- An rvalue is an expression that identifies a temporary object, or a value not associated with any object
- As examples, consider the following:

```
vector<string> arr( 3 );
const int x = 2;
int y;
 ...
int z = x + y;
string str = "foo";
vector<string> *ptr = &arr;
```

lvalues:

arr, str, arr[x], &x, y, z, ptr, *ptr, (*ptr)[x]

rvalues:

2, "foo", x+y, str.substr(0,1)

References

- A reference type allows us to define a new name for an existing value; it's an **alias**
  - They even have the same memory address!
- Declared as: <type> & <name> = <var>;
  - Normally the <var> needs to be an **lvalue**, but in C++11 we can also have **rvalue references**.
  - lvalue reference example:

    ```
    int x = 5;int & y = x
    ```
  - rvalue reference example:

    ```
    int && y = 5;
                 // note the extra '&' !
    ```

Examples

```
string str = "hell";
string & rstr = str;                //legal
string & sub = str.substr( 0, 3 );  //illegal
rstr += ’o’;    // change string to “hello”
bool cond = (&str == &rstr);        // legal (true)
string & bad1 = "hello";            // illegal
string & bad2 = str + "";           // illegal

string str = "hell"; // change back to “hell”
string && bad1 = "hello";           // legal
string && bad2 = str + "";          // legal
string && sub = str.substr( 0, 4 ); // legal
```

When to use references and when to use values in functions

- If the formal parameter should be able to change the value of the actual argument, then you *must use call-by-reference*
- Otherwise, the value of the actual argument cannot be changed by the formal parameter
  - If the type is a primitive type, use call-by-value
  - Otherwise, the type is a class type and is generally passed using call-by-constant-reference
    * unless it's an unusually small type (e.g., a type that stores <= two primitives)

Rvalue usage example

```
// returns random item in lvalue arr
string randomItem( const vector<string> & arr );
// returns random item in rvalue arr
string randomItem( vector<string> && arr );

vector<string> v { "hello", "world" };
cout << randomItem( v ) << endl; // call lvalue method
cout << randomItem( { "hi world" } ) << endl; // call rvalue method
```

About const pointers

- const is used to declare something as constant, but becomes tricky in pointers
- Is the pointer (memory location) constant, the value it points to constant, or both?
- The syntax is:
  - <const for value> <type>* <const for pointer> <name>;
  - Examples:
    - const int* x –or– int const * x // these are the same!!
    - int* const x;
    - const int* const x;
    - int const * x

Const pointer Examples

Syntax: `<const for value> <type>* <const for pointer> <name>;`

Which lines are invalid in the following code?

```
int w,y,z;     // 3 integers w, y, z
const int* x = &w;
*x += 2;
x += 2;
const int* const u = &y;
*u += 2;
u += 2;
int* const v = &z;
*v += 2;
v += 2;
```

Const pointer Examples

Syntax: `<const for value> <type>* <const for pointer> <name>;`

Which lines are invalid in the following code?

```
int w,y,z;        // 3 integers w, y, z
const int* x = &w;
*x += 2;          // invalid, can't change w
x += 2;
const int* const u = &y;
*u += 2;          // invalid, can't change y
u += 2;           // invalid, can't change u
int* const v = &z;
*v += 2;
v += 2;           // invalid, can't change v
```

Structs

- **Structs** are combined data
- extremely common in C, also found in C++

```
struct Point {
   float x;
   float y;
};
```

Classes

- Structs with methods! (sorta)
  - Technically, structs can have methods—but structs are used as public data wrappers *by convention*
- Syntax is:

```
class <class name> {
public:
 ... Member function declarations
private:
 ... Class variable definitions
};  // the ';' is needed because a class declaration
  // is a statement – all statements end with ;
```

- And initializing methods:

```
<return type> <Class Name>::<function name>(...){
 ...
}
```

More pointer syntax

```
class Dog {
public:
    int legs;
}
Dog* lassie_ptr = new Dog();
```

- More convenient syntax for accessing object pointer members:

```
int legs = lassie_ptr->legs;
```

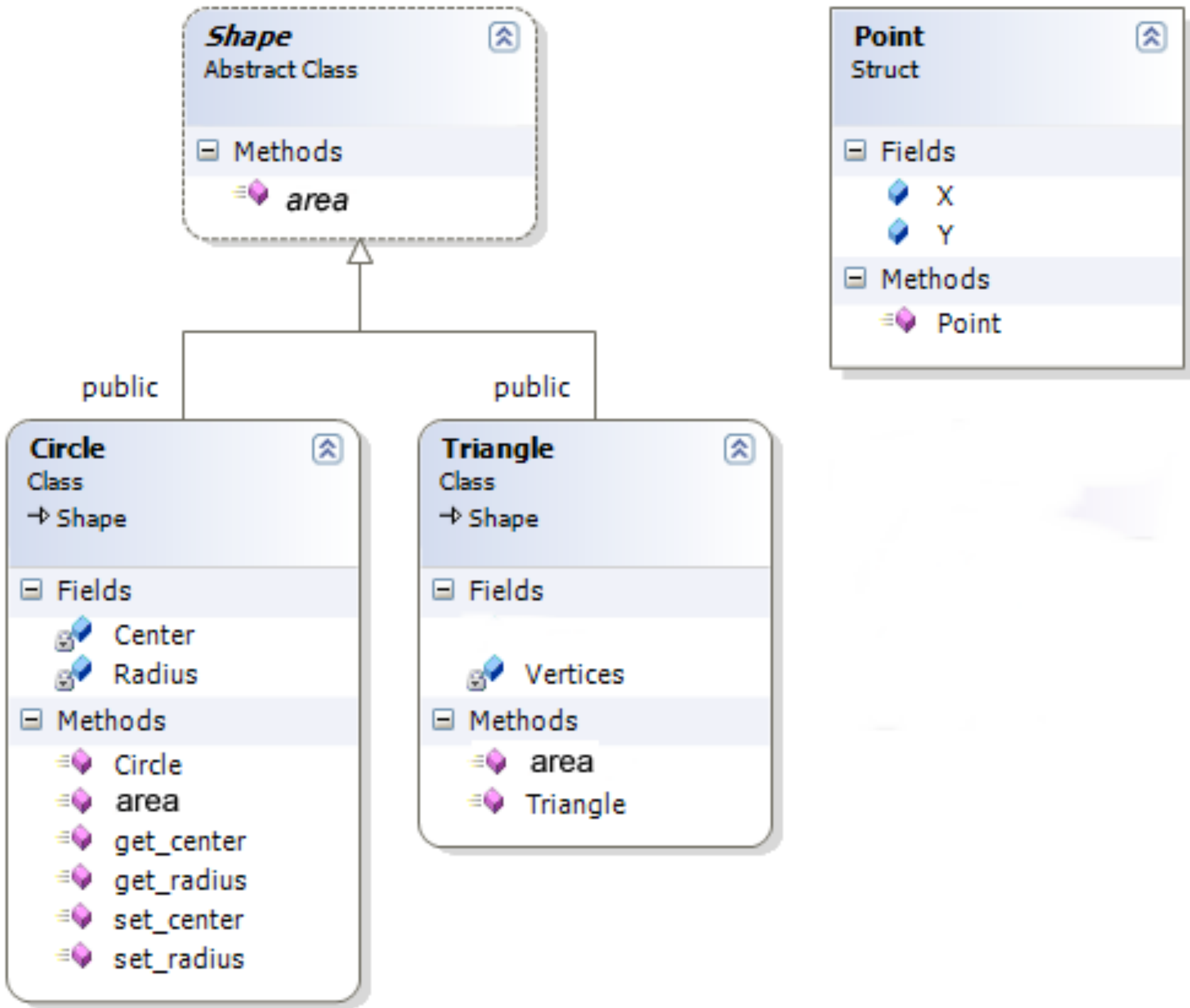
- Is equivalent to:

```
int legs = (*lassie_ptr).legs;
```

Class Example with an ADT

- Normally in C++, collections of data that do not need functions will be structs, and otherwise will be classes
- An Abstract Data Type (ADT) is a combination of data and operations
  - Provides an **interface** for usage and encapsulates **implementation** details

Class Diagram



ADT for Collections of Data

```
class Data {
  public:
    /**
     * Returns 0 if equal to other, -1 if < other, 1 if > other
     */
    virtual int compareTo(Data * other) const = 0;
    // ...
};

class IntegerData : public Data {
  public:
    int value;
    // ...
    int compareTo(Data * other) const ...
};
```

The “public Data” means the inheritance is public; see:
http://stackoverflow.com/questions/860339/difference-between-private-public-and-protected-inheritance

Collection

- add(x)
- remove(x)
- member(x)
- size()

We’ll implement with a fixed-length array:

ArrayCollection

```
class ArrayCollection {
public:
   void add(Data*);
   void remove(Data*);
   bool member(Data*);
   int size();

private:
   Data* data;
   int nextPos;
   int arraySize;
};
```

ArrayCollection

```
void ArrayCollection::add(Data* d){
    if (nextPos < arraySize) {
        data[nextPos++] = *d;
    } else {
        // throw error
    }
}
void ArrayCollection::remove(Data* d){
    bool found = false;
    for (int i = 0; i < nextPos; ++i){
        if ( (data+i) == d || found) {
            found = true;           // copy elements to  location
            data[i] = data[i+1];    // one cell to left
        }
    }
    if (found) {
        delete data[--nextPos];   // delete memory
    }
}
```

ArrayCollection

```
bool member(Data* d){
    for (int i = 0; i < nextPos; ++i){
        if ( !data[i].compareTo(d) ){ // compareTo is 0
            return true;
        }
    }
    return false;
}

int size(){
    return nextPos;
    // this extra
    // space is
    // to trick
    // you  >:)
}
```

Lists

If these methods define a Collection:

- add(x)
- remove(x)
- member(x)
- size()

what is a List?

Lists

- add(x)
- insert(i, x)
- get(i)
- remove(x)
- remove(i)
- member(x)
- size()

where x is a value and i is an index

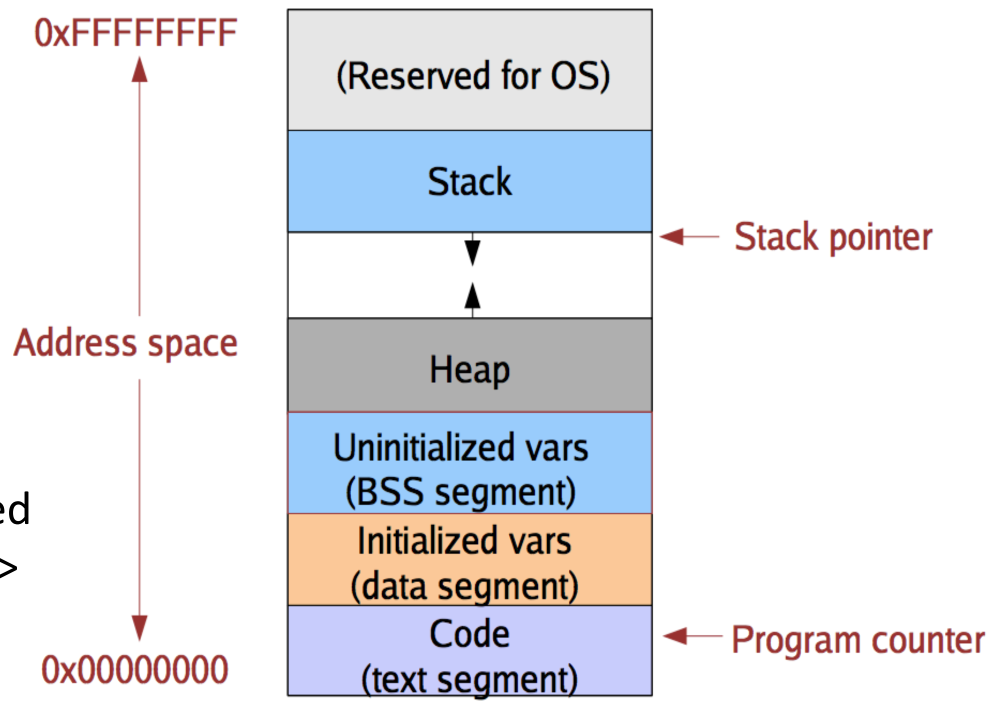
Our LinkedList data structure will use these methods!

Virtual Memory

- Basic abstraction provided by OS for memory management
- Enables programs to run without requiring entire address space to be in physical memory
- Most programs do not use all of their code or data
  - E.g. branches never taken, variables never accessed, objects never created
  - Therefore no need to allocate memory until it's needed
- Also isolates processes from each other
  - Each process gets its own virtual memory space, usually about 4 GB
  - One process cannot access memory addresses in others

Virtual Addresses

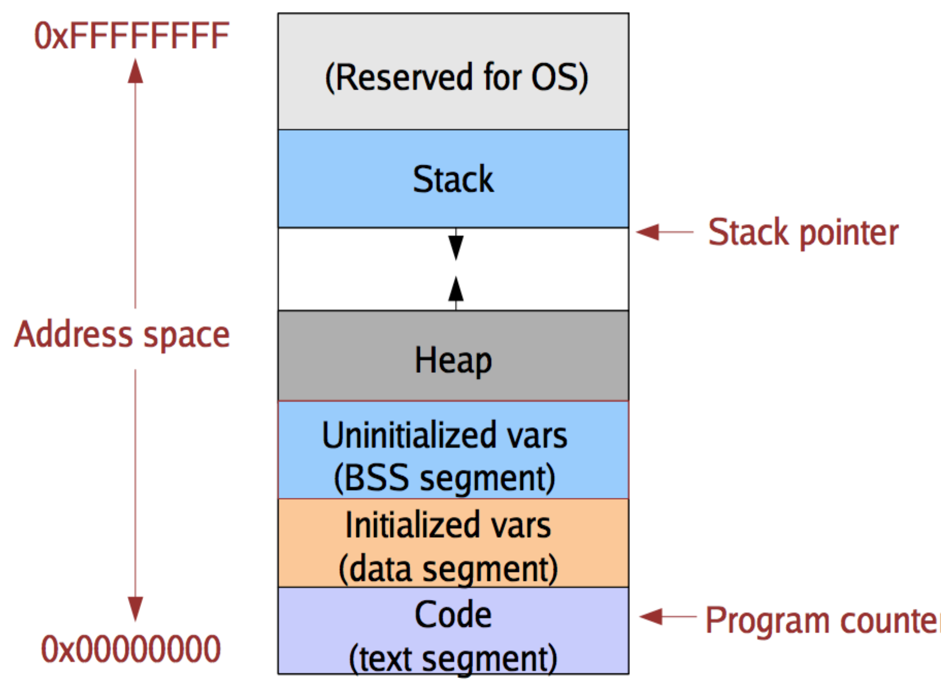
- A virtual address is a memory address that a process uses to access its own memory
  - Which is **not** the same as the address on physical memory
  - The OS determines the mapping from virtual address to physical address



How many values can be represented in this space? --->
16^8 = 4,294,967,296

Virtual Addresses

- We've mentioned Stacks (LIFO) and the Heap already (contiguous block of allocated objects that may be fragmented)
- **BSS**: contains the statically-allocated and uninitialized variables
  - Data bits all set to 0
- **Data**: initialized static variables (including globals)
- Virtual addresses allow **relocation**
  - A process does not (and should not) know the physical address that it uses to run



Question 4

What is the output from the following code:

```
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

int main(void) {
  int a = 0;
  int b = 5;
  swap(a,b);
  cout << a << endl;
}
```

Question 4

What is the output from the following code:

```
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

int main(void) {
  int a = 0;
  int b = 5;          // 0
  swap(a,b);
  cout << a << endl;
}
```

Question 5

Change the code to work correctly using references:

```
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

int main(void) {
  int a = 0;
  int b = 5;
  swap(a,b);
  cout << a << endl;
}
```

Question 5

Change the code to work correctly using references:

```
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}
            // change to swap(int& x, int& y)
int main(void) {
  int a = 0;
  int b = 5;
  swap(a,b);
  cout << a << endl;
}
```

Question 6

What is the value of temp after each assignment?

```
char blocks[3] = {'A','B','C'};
char *ptr = &blocks[0];
char temp;

/*1*/ temp = blocks[0];
/*2*/ temp = *(blocks + 2);
/*3*/ temp = *(ptr + 1);
      ptr = blocks + 1;
/*4*/ temp = *ptr;
/*5*/ temp = *(ptr + 1);
```

Question 6

What is the value of temp after each assignment?

```
char blocks[3] = {'A','B','C'};
char *ptr = &blocks[0];
char temp;

/*1*/ temp = blocks[0];
/*2*/ temp = *(blocks + 2);
/*3*/ temp = *(ptr + 1);
      ptr = blocks + 1;
/*4*/ temp = *ptr;
/*5*/ temp = *(ptr + 1);
```

```
// ‘A’, ‘C’, ‘B’, ‘B’, ‘C’
```

Question 7

What is the value of temp after each assignment?

```
char blocks[3] = {'A','B','C'};
char *ptr = blocks;
char temp;

/*1*/ temp = *++ptr;
/*2*/ temp = ++*ptr;
/*3*/ temp = *ptr++;
/*4*/ temp = *ptr;
```

Question 7

What is the value of temp after each assignment?

```
char blocks[3] = {'A','B','C'};
char *ptr = blocks;
char temp;

/*1*/ temp = *++ptr;
/*2*/ temp = ++*ptr;
/*3*/ temp = *ptr++;
/*4*/ temp = *ptr;
```

1. ‘B’: ptr gets incremented first, then dereference. Ptr now at B.
2. ‘C’: Dereference to get ‘B’ then increment, so char value of ’B’ + 1 = ‘C’. Ptr still at 2nd position, but **array changed** so is now {‘A’,’C’,’C’}.
3. ‘C’: ++ has higher precedence but evaluates at end of expression (postfix), so dereference current position (blocks[1]) to return ‘C’ and ptr ends up pointing to blocks[2], which is also ‘C’.
4. ‘C’: return value at ptr, which is ‘C’.

Before next class

- Review today's slides
- Read the following resources:
  - http://pages.cs.wisc.edu/~hasti/cs368/CppTutorial/NOTES/INTRODUCTION.html
  - http://www.cplusplus.com/doc/tutorial/pointers/
  - http://www.cplusplus.com/doc/tutorial/dynamic/