COMP 3522

Object Oriented Programming in C++

Week 2

Agenda

1. File IO
2. Arrays in C++
3. Random number generation
4. Pointers, nullptr
5. References

COMP 3522

IO Part 2: Files

- Defined in the <fstream> header
  1. **ifstream** for reading from a file
  2. **ofstream** for writing to a file
  3. **fstream** for reading and writing to/from a file

- We can use <<, >>, and manipulators with file streams

Opening a file

```
#include <fstream>
fstream f{“data.txt”}; // Opens data for writing
if (!f.is_open()) { // Or if (!f) …
    cerr << “Unable to open file” << endl;
    exit(1);
}
f << “hello” << 123 << endl; // file closed
                             // automatically
```

Opening a file

```
// Open a file for reading
ifstream fin;
fin.open(“helloWorld.txt”);

// open a file (or create it if it doesn’t exist)
// for writing
ofstream fout;
fout.open(“helloWorld.txt”);

// open a file for reading and writing.
fstream fs;
fs.open(“helloWorld.txt”);
```

How do we close a file

- Too easy for its own slide, but here we are anyway:

```
fin.close();
fout.close();
fs.close();
```

That's it!

Buffers

- Stream objects use an internal buffer
  - Filestreams use a filebuf
    http://www.cplusplus.com/reference/fstream/filebuf/
  - IO streams like cin, cout, cerr use a streambuf
    http://www.cplusplus.com/reference/streambuf/streambuf/
  - Stringstreams use a stringbuf
    http://www.cplusplus.com/reference/sstream/stringbuf/

- We will rarely need to manage the internal buffer directly, but it is a good idea to understand the concepts

Opening streams

- When we open a stream we can specify the “mode”
- This is similar to C
- The mode type is `std::ios_base::openmode`
  1. `ios_base::in` (**input**) Allow input operations on the stream.
  2. `ios_base::out` (**output**) Allow output operations on the stream.
  3. `ios_base::app` (**append**) Set the stream's position indicator to the end of the stream before each output operation.

More open mode flags

- More modetypes
  4. `ios_base::binary` (**binary**) Open in binary mode when file contains binary data.
  5. `ios_base::trunc` (**trunc**ate) Discard the contents of the stream when opening
  6. `ios_base::ate` (**at end**) Set the stream's position indicator to the end of the stream on opening.

Combine modes with bitwise OR (|)

```
ifstream f1{”data”, ios_base::in | ios_base::binary};

ofstream f2{“dest”, ios_base::out | ios_base::app};
```

So how do we read/write char by char?

Similar to C:

1. Use **std::basic_istream::get** to acquire the char
2. Use **std::basic_istream::put** to place the char

```
char c;
while ((c = in.get()) != EOF)
{
    // Do something
}
```

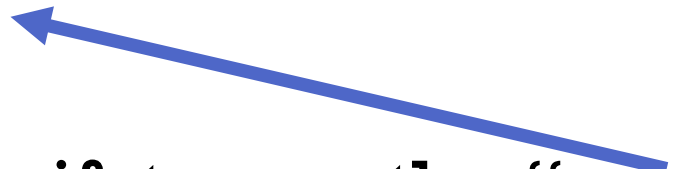
fileIO.cpp

Seeking within a file

- We now know how to read/write using file streams
- But now we need a way to navigate the file
  - We don't always want to read from the beginning of file
  - Maybe we want to jump to the middle, or end

- Imagine a text file that contains the following:

```
Hi class, here is some text
```



- With ifstream the "cursor" is placed at the beginning, position 0

Seeking within a file

These **GET** the position of the current character in the stream:

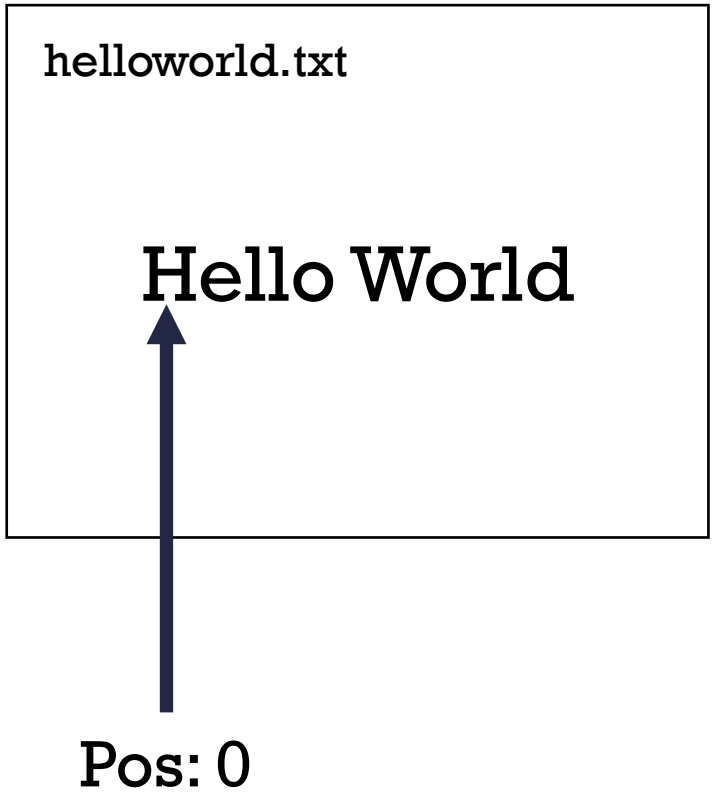
// "Tell put" - Returns the position of the current character in the **output** stream

```
streampos std::ostream::tellp()
```

// "Tell get" Returns the position of the current character in the **input** stream

```
streampos std::istream::tellg()
```

Seeking within a file

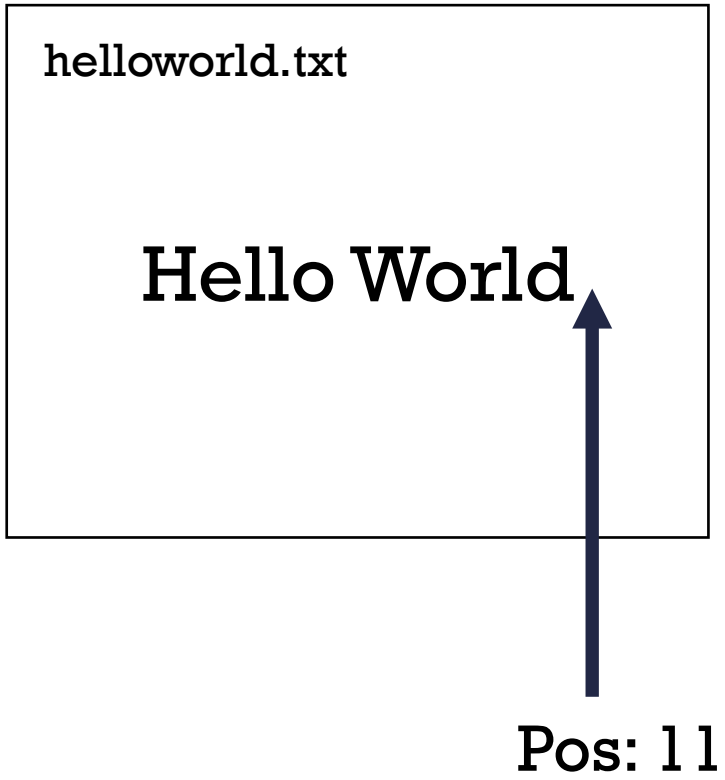



```
ifstream myFile(“helloWorld.txt");

cout << myFile.tellg() << endl;
```

Output is 0
tellg() shows current position of “cursor”

Note we use tellg for **ifstream**

Seeking within a file

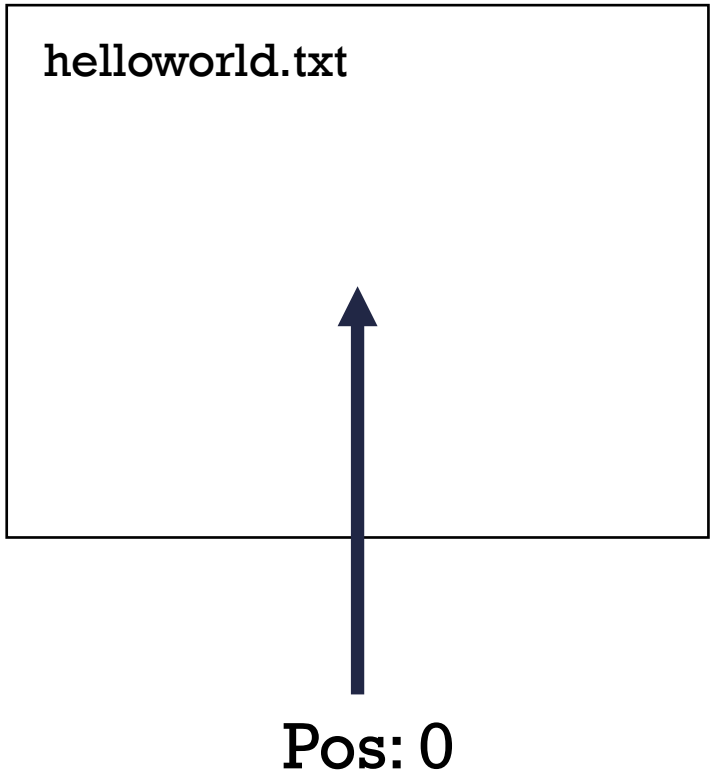



```
ofstream myFile(“helloWorld.txt“, ios::app);

cout << myFile.tellp() << endl;
```

Output is 11
tellp() shows current position of “cursor”

Note we use tellp for **ofstream**

Seeking within a file




```
ofstream myFile("helloWorld.txt");

cout << myFile.tellp() << endl;
```

Output is 0
tellp() shows current position of "cursor"

Note we use tellp for **ofstream**

Seeking within a file

These **SET** the position of the cursor in the stream:

//"Seek put" - use seekp for **output** streams

```
ofstream& seekp(streampos)
ofstream& seekp(streamoff, ios_base::seekdir)
```

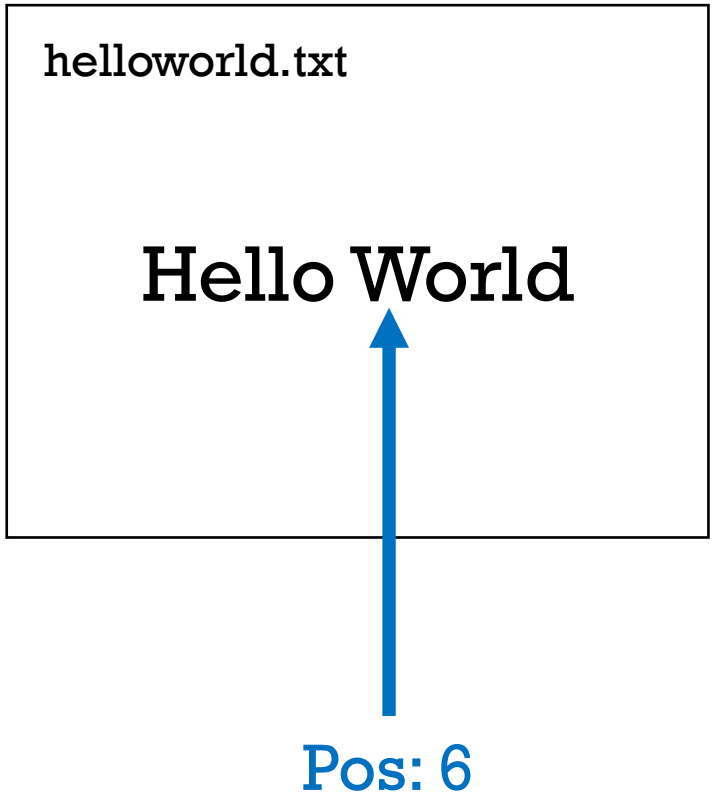
//"Seek get" - use seekg for **input** streams

```
ifstream& seekg(streampos)
ifstream& seekg(streamoff, ios_base::seekdir)
```

Seeking within a file

- Recall C: fseek, ftell, SEEK_SET, SEEK_CUR, SEEK_END, etc.
- Similar in C++:
  - std::ios::**streampos** for storing positions
  - Following two combined to find offset relative to some position
    - std::ios::**streamoff** for storing offsets
    - std::ios_base::**seekdir** represents the seeking direction of a stream-seeking operation
      - ios::**beg** (public member of ios_base class)
      - ios::**cur** (public member of ios_base class)
      - ios::**end** (public member of ios_base class)

Seeking within a file

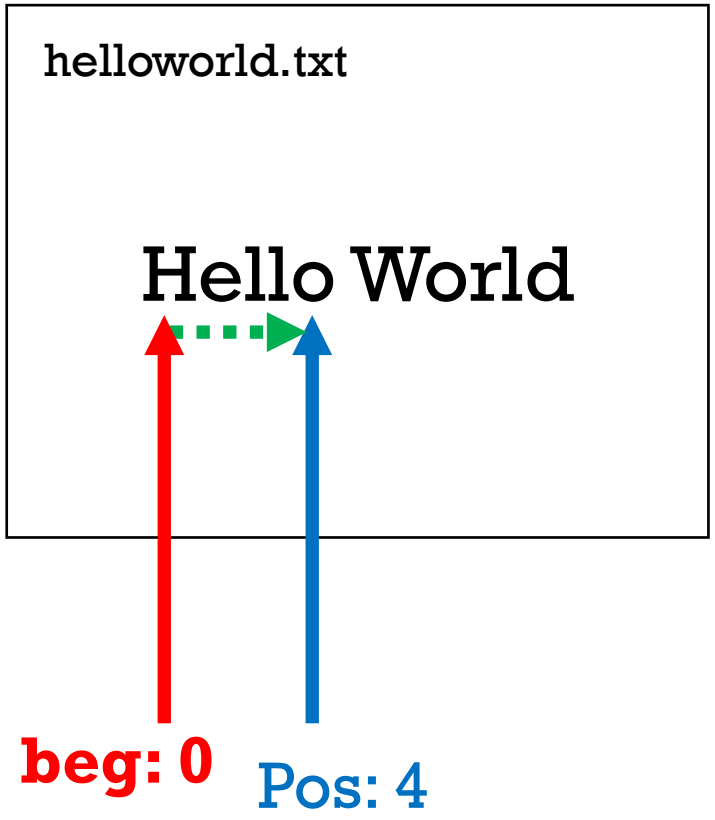



```
ofstream myFile("helloWorld.txt", ios::app);

myFile.seekp(6);
cout << myFile.tellp() << endl;
```

Sets "cursor" to absolute position 6

Output is 6

Seeking within a file

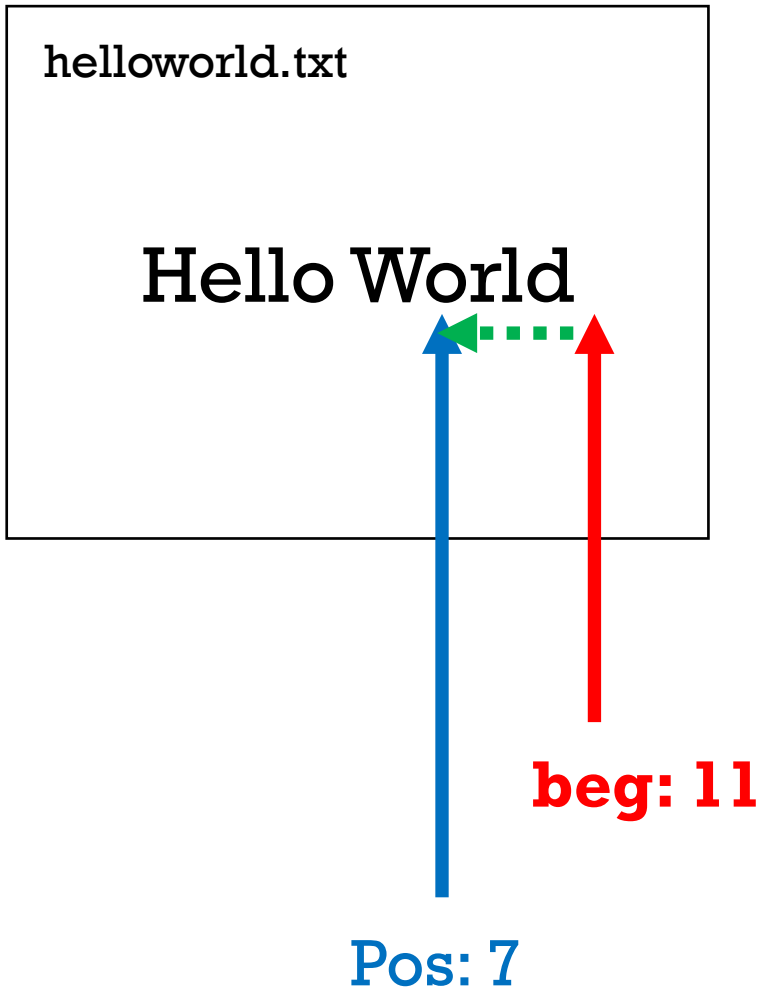



```
ofstream myFile("helloWorld.txt", ios::app);

myFile.seekp(4, ios::beg);
cout << myFile.tellp() << endl;
```

Moves "cursor" **+4** positions relative to the **beg**inning

Output is 4

Seeking within a file

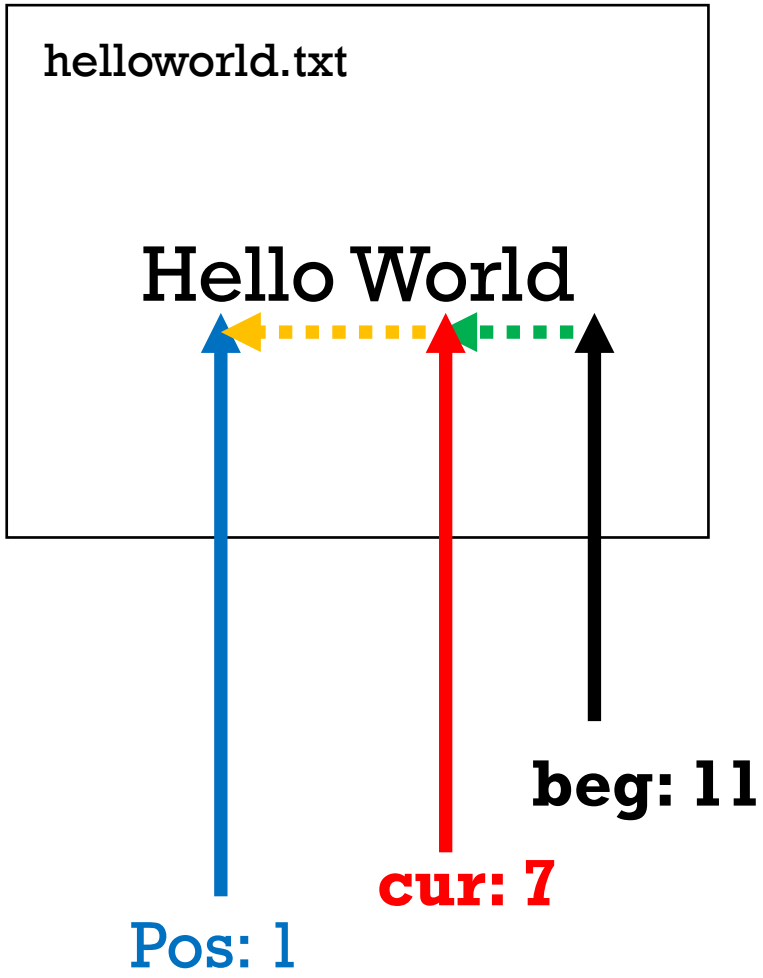



```
ofstream myFile("helloWorld.txt", ios::app);

myFile.seekp(-4, ios::end);
cout << myFile.tellp() << endl;
```

Moves "cursor" **-4** positions relative to the **end**

Output is 7

Seeking within a file




```
ofstream myFile("helloWorld.txt", ios::app);

myFile.seekp(-4, ios::end);
myFile.seekp(-6, ios::cur)
cout << myFile.tellp() << endl;
```

Moves "cursor" -4 positions relative to the end.
Then moves cursor -6 relative to last known "cursor" position to 1

Output is 1

NOTE
use seekp for ofstream
use seekg for ifstream
use seekp or seekg for fstream

Example code: acquiring a file's size!

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
   ifstream myfile{”Macbeth.txt”};
   streampos begin = myfile.tellg();
   myfile.seekg (0, ios::end);
   streampos end = myfile.tellg();
   myfile.close();
   cout << "size is: " << (end-begin) << " bytes.\n";
   return 0;
}
```

fileSize.cpp fileSeek.cpp

C-STYLE ARRAYS

What about arrays? One slide!

```
float values[3] // array of 3 floats
char * names[32] // array of 32 pointers to char

int scores[] = {1, 2, 3, 4};
int some_scores[8] = {1, 2, 3, 4};
     // equivalent to {1, 2, 3, 4, 0, 0, 0, 0}
```

RANDOM NUMBERS

Random numbers

- There are **some fun ways to generate random numbers** in C++
- It would be helpful to review what we've seen and look at some C++ ways to make random numbers:

1. Ye olde tyme C approach

- We can generate random numbers using **rand** and **srand** from the C library:
  - srand initializes the random number generator
  - srand accepts a parameter which is a SEED (use the current time!)
  - rand returns a pseudo-random integer between 0 and RAND_MAX

```
#include <cstdlib>
#include <ctime>

srand (time(NULL)); //seed random num generator only ONCE
const int UPPER_BOUND = 10
int my_int = rand() % UPPER_BOUND;
double zero_to_one = rand() / (double) RAND_MAX;
```

1. Ye olde tyme C approach

- We can generate random numbers using **rand** and **srand** from the C library:
  - srand initializes the random number generator
  - srand accepts a parameter which is a SEED (use the current time!)
  - rand returns a pseudo-random integer between 0 and RAND_MAX

```
#include <cstdlib>
#include <ctime>

srand (time(NULL)); //seed random num generator only ONCE
int random_num_1 = rand() % 100; //random range 0 to 99
int random_num_2 = rand() % 100 + 1; //random range 1 to 100
int random_num_3 = rand() % 25 + 2000; //random range 2000 to 2024
```

2. Uniform distribution of double in [a, b]

```
#include <random>
#include <ctime>

double a = 10;
double b = 100
default_random_engine generator(time(0));
uniform_real_distribution <double> distribution(a, b);
double my_random = distribution(generator);
```

3. Uniform distribution of int in [a, b]

```
#include <random>

random_device rd; // a random number generator
mt19937 generator(rd()); // calls operator()
uniform_int_distribution<> distribution(a, b);
int my_int = distribution(generator);
```

Check it out! We're using a random number generator to generate a random seed for a random number generator!

random_c.cpp, random_int.cpp, random_double.cpp

POINTERS, REFERENCES, AND nullptr

Call by value: will this work?

```
void swap(int arg1, int arg2)
{
      int temp{arg1};
      arg1 = arg2;
      arg2 = temp;
}

int main()
{
      int first{3512};
      int second{2526};

      swap(first, second);
      //does first = 2526 and second = 3512?
}
```

Passing pointers: what about this?

```
void swap(int* arg1, int* arg2)
{
	int temp{*arg1};
	*arg1 = *arg2;
	*arg2 = temp;
}

int main()
{
	int first{3512};
	int second{2526};

	swap(&first, &second);
	//does first = 2526 and second = 3512?
}
```

Introducing the C++ reference (&)

- An **alias** (anything done to the reference is done to the referent)
- Must be initialized when created
- Makes **pass by reference** effortless
- Used for *efficiency* (don't want to make a copy)

References

```
int n{123};
int& ref = n;
int m{345};
ref = m; // same as n = m
cout << n << endl; // 345
cout << ref << endl; // 345
```

References as function parameters

```
void swap(int& first, int& second)
{
      int tmp{first};
      first = second;
      second = tmp;
}
…
int a{3512};
int b{2526};
swap(a, b);
```

Pointers vs references

- **Does our processor know about references?**

- **NO!**
  - Pointers and references produce the same assembly instructions
  - References are for programmers
  - References are converted to pointers when our code is compiled

References to constants

We **cannot** create a reference to a temporary value

```
int& reference{1};          // Will not compile
const int& r{1};            // OK

int n{12};
long& ref = n;              // Won't compile either! (Why not)
const long& ref = n;        // OK
```

pointers.cpp, references.cpp

Pointers and references

- Assignment to a pointer makes the pointer point to a new address

```
int num = 99;
int* numPtr = nullptr; //numPtr pointing to nullptr
numPtr = &num;  //numPtr pointing to address of num
```

- To get a pointer we need to assign to **nullptr** an **existing pointer**, **use & (address of)** or **new (memory allocation)**

```
int* numPtr2 = numPtr;
int* numPtr3 = &num;
int* numPtr4 = new int(123);
```

Pointers and references

- Beware of null pointers (assign empty pointers to nullptr as much as possible!)
- To access something pointed to by a pointer (dereference), we use * or [ ]

```
int num = 99;
int* numPtr = nullptr; //pointer to nullptr
numPtr = &num
cout << *numPtr; //numPtr pointing at num, and accesses its value
```

- **References cannot refer to a different variable after initialization**
- Assignment to a reference changes the value of the object referred to (not the reference itself)

```
int num = 100;
int num2 = 200;
int& numRef = num; //numRef refers to num
numRef = num2; //numRef still referring to num, but changed num's
value to 200
```

ACTIVITY

1. Need a pointer refresher? Watch optional pointer videos Week 2 > Review videos 1 > Pointer basics 1,2,3 review
2. Answer the questions on the “Pointers and References practice” file on The Learning Hub.

   - The Learning Hub > Content > Week 2

Agenda

1. string and stringstream
2. C++ Vectors
3. new & delete

COMP 3522

string AND stringstream

C++'s std::string class (lower case s)

```
#include <string>

string s1; // Statically allocates a string object!
Invokes default constructor
string s2 = "Hello"; // This does too
string s3{"world!"}; // So does this
cout << s1 << " " << s2 << " " << s3 << endl;
```

*** In C++, the string object needn't terminate with \0**

The std::string class

- Member functions include:
  - size() returns the number of characters
  - length() returns the number of characters (same thing!)
  - c_str() returns a non-modifiable standard C char array

```
string line;
cin >> line;
cout << line.size();
cout << line.length();
const char * c_line = line.c_str();
```

http://en.cppreference.com/w/cpp/string/basic_string

More about the std::string

- We can use **relational operators** (>, <, >=, ==, etc.) to perform lexicographical comparisons (unlike Java which required compareTo or an overridden equals method)
- We can use **square brackets [ ]** to access chars in a std::string
- We can also use the **at(size_type pos)** member function to acquire a reference to the char at the specified index

```
string s = "hello";
cout << s[0]; //prints h
cout << s.at(1); //prints e
```

Classes in C++ (a short aside, more later!)

```
string first; // calls default constructor

string second = first; // calls copy constructor

first = second; // calls assignment operator
```

The getline function

- Defined in <string>
- Reads a line of characters from an input stream and puts the characters in the specified string (tosses the newline!)
- Returns the original input stream

```
string input;
getline(cin, input); // returns cin
```

getline (even more information!)

```
getline(inputstream, input, delimiter)
```

Keeps extracting characters until:

1. EOF (sets EOF bit)
2. Delimiter or newline is extracted (and tossed!)
3. So many characters have been extracted that it exceeds the number storable in line (sets the failbit)

getline (failures)

```
string input;
getline(cin, input);
```

cin user input	string input
Hello World	Hello World
Hello\nworld\n	Hello
\nWorld\n	EMPTY
Hello*	Hello (eofbit set)
Hello\n*	Hello
*	No change, eofbit and failbit are set

A C++ standard idiom

- To process a stream line by line, try:

```
string line;
while (getline (cin, line))
{
    /* process your line */
}
```

The istringstream class

- Great for **reading** and **manipulating strings**
- Defined in <sstream>
- Actual type is **basic_istringstream<char>**

```
#include <sstream>
string input{“   123abc”};
istringstream iss{input};
int n;
iss >> n;
cout << n << endl;
```

More istringstream

```
istringstream iss;
int n;
iss.str(“   123abc”);
iss >> n;
cout << n << endl;
```

Output:

123

More istringstream

```
istringstream iss;
int n;
string aString
iss.str("   123abc");
iss >> n >> aString;
cout << n << endl;
cout << aString << endl;
```

Output:

```
123
abc
```

More istringstream

```
istringstream iss;
int n;
string aString
iss.str("   123a b c");
iss >> n >> aString;
cout << n << endl;
cout << aString << endl;
```

Output:

```
123
a
```

More istringstream

```
istringstream iss;
iss.str(“   123a b c”);
while(!iss.eof())
{
     string newString;
     iss >> newString;
     cout << newString << endl;
}
```

Output:

123a

b

c

Even more istringstream

```
string line;
int n, sum{0};
istringstream iss; //create new iss
while (getline(cin, line)) {
    iss.clear(); //clear iss of failbits
    iss.str(line); //load line string into re-used iss
    if (iss >> n) {
        sum += n;
    }
}
```

One more istringstream example

```
string line;
int n, sum{0};
while (getline(cin, line)) {
     istringstream iss{line}; //load line string into new iss
     if (iss >> n) {
          sum += n;
     }
}
```

istringstream.cpp

THE C++ VECTOR

The C++ vector (think ArrayList)

- In **<vector>**
- A sequence container that **can change size** (like Java's ArrayList)
- Part of the STL (which we will cover in a few weeks)
- But for now it's very useful, even without knowing how to use its iterators
- http://www.cplusplus.com/reference/vector/vector/
- http://en.cppreference.com/w/cpp/container/vector

The C++ vector (think ArrayList)

- There are some very useful member functions:
  - **push_back(const T& value)** appends the given value to the end
  - **size()** //returns number of elements in vector
  - **operator[size_type pos]** returns a reference to the element at pos
  - **at(size_type pos)** returns a reference to the element at pos. Differs from operator[] by doing bounds check and throws exception
  - **erase(iterator pos)** removes element at iterator position
  - **clear()** removes all elements in vector

- We can use the for-each loop with the vector (it's called the **ranged-for** in C++)

The C++ vector (think ArrayList)

```
vector <int> intVector;
intVector.push_back(5);
intVector.push_back(10);
intVector.push_back(15);
intVector.erase(intVector.begin()+1) //erases 10 at
index 1
//classic for loop
for(int i=0; i<intVector.size(); i++)
{
   cout << intVector[i];
}
```

The C++ vector (think ArrayList)

```
vector <int> intVector;
intVector.push_back(5);
intVector.push_back(10);
intVector.push_back(15);
intVector.erase(intVector.begin()+1) //erases 10
```

```
//classic for loop
for(int i=0; i<intVector.size(); i++)
{
   cout << intVector[i];
}
```

```
//for each loop
for(int value: intVector)
{
   cout << value;
}
```

vector.cpp

new & delete

Dynamic memory management

- Refers to **manual** memory management
- Allows us to obtain more memory when required and release it when not necessary
- C does not inherently have any techniques to allocate memory dynamically for dynamic memory – we have to use library functions

Recall dynamic memory management in C

- There are 4 library functions defined under **stdlib.h** for dynamic memory allocation in C:

1. **malloc()** Allocates requested size of bytes and returns a pointer first byte of allocated space
2. **calloc()** Allocates space for an array elements, initializes to zero and then returns a pointer to memory
3. **realloc()** Changes the size of previously allocated space
4. **free()** Deallocates the previously allocated space

In C++, it's much easier

- We have two operators in C++ for allocating memory dynamically:
  1. **new**
  2. **new[]**

- The new operator returns a pointer to the memory that was just allocated

The new operator

```
int * my_pointer = nullptr;
my_pointer = new int { 3522 };
```

We say that my_pointer refers to a **data object** (not the same an an instance of a class)

We can also do this

```
int * my_pointer = new int;
*my_pointer = 3522;
```

Or

```
int * my_pointer = new int{3522};
```

What about new[]

```
int * my_pointer;
my_pointer = new int [5];

for (int i = 0; i < 5; ++i) {
     my_pointer[i] = i;
}
```

What's the difference?

```
int i;
int iArray[10];
```

- Memory is automatically allocated and deallocated
- Memory deallocated when function returns/completes

```
int * i = new int;
int * iArray = new int[10]
```

- Programmers' responsibility to deallocate memory when no longer needed
- **Memory leaks** occur if memory not deallocated. Memory exists even after function returns/completes

The delete keyword

- We must remember to free the allocated memory
- If we don't, we get a **memory leak**
- There is no garbage collector in C++
- We must remember to deallocate the memory

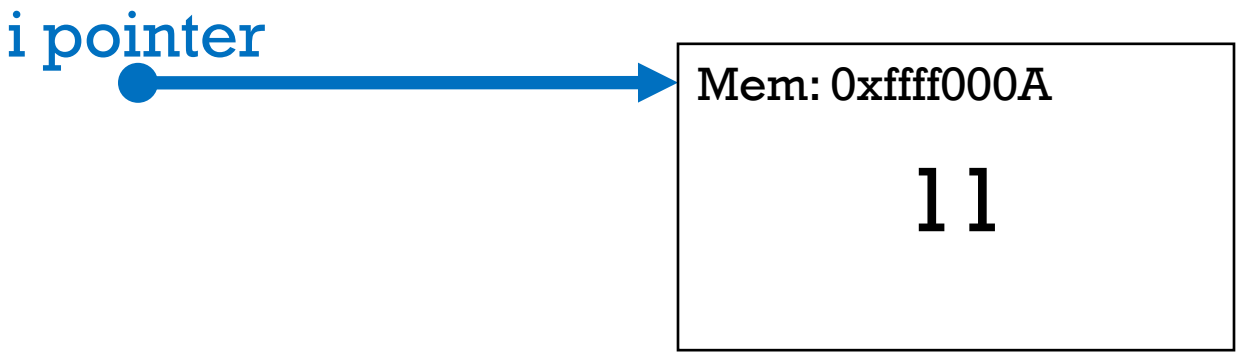
- We can do this with the **delete** and **delete[]** operators

The delete keyword

```
int *i = new int;
int *iArray = new int[10]

...//some code

delete i; //free allocated memory
delete[] iArray; // freed block of allocated memory
```

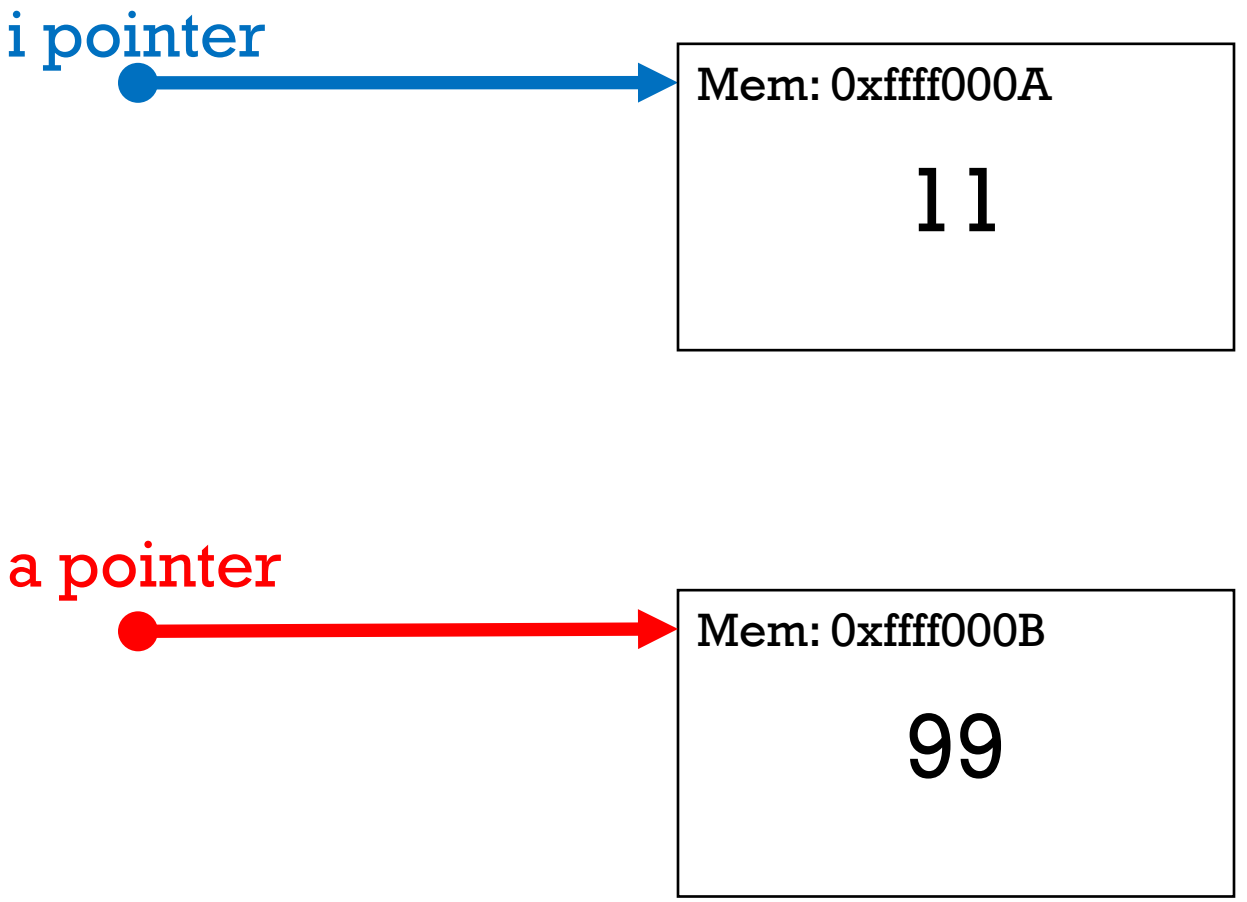
Memory leak example

```
int *i = new int{11};
```



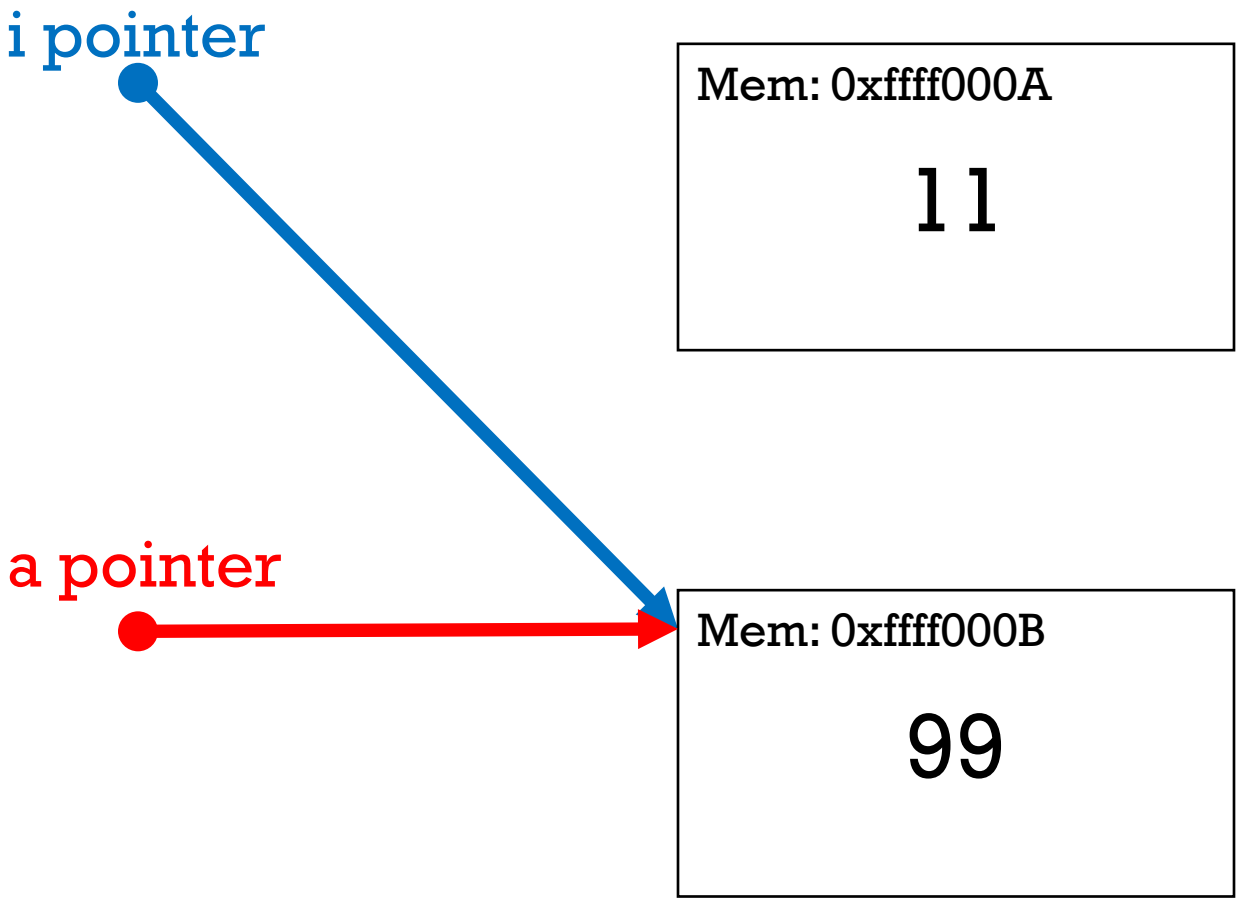
Memory leak example

```
int *i = new int{11};
int *a = new int{99};
```



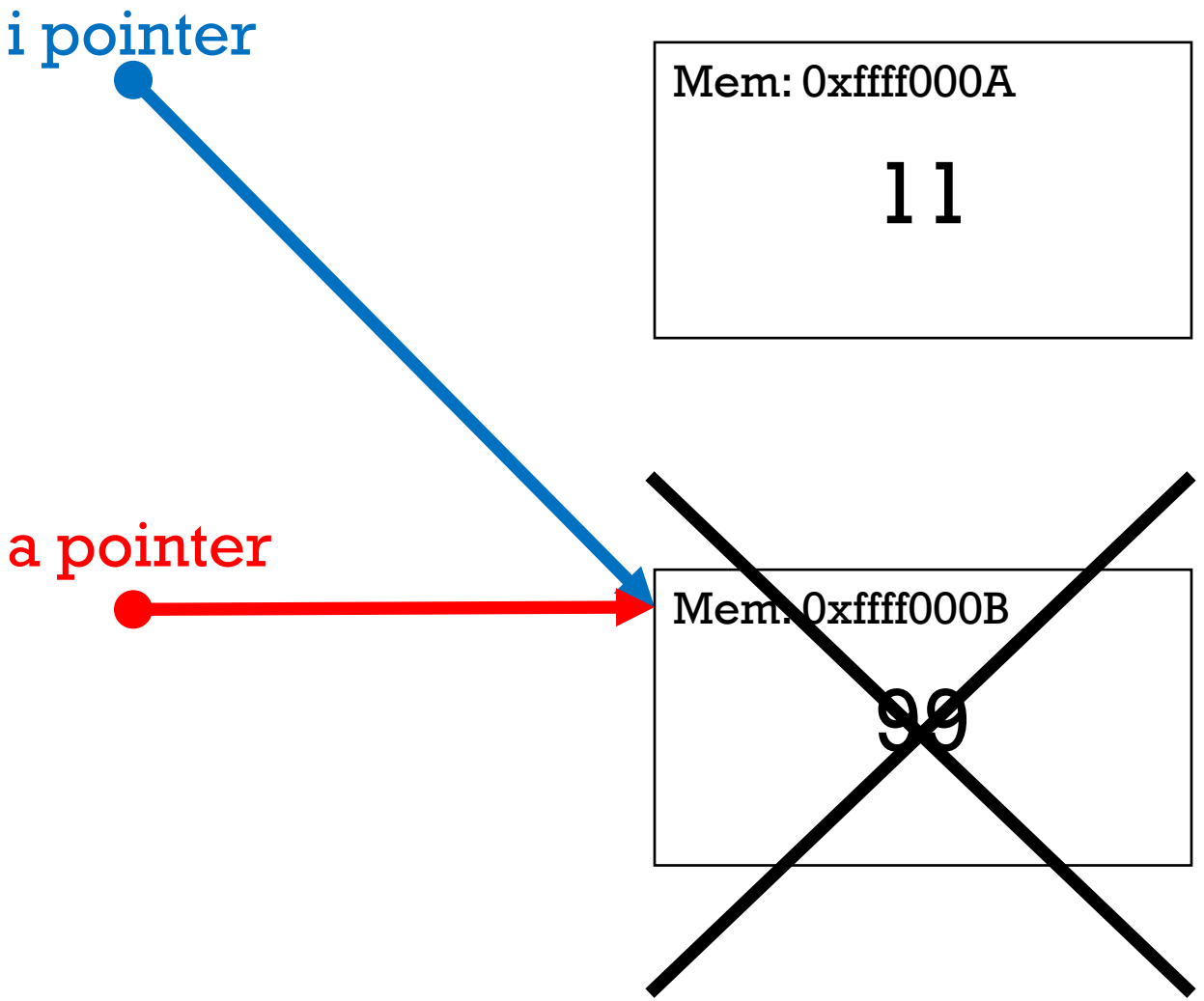
Memory leak example

```
int *i = new int{11};
int *a = new int{99};
i = a; //creates a memory leak
```



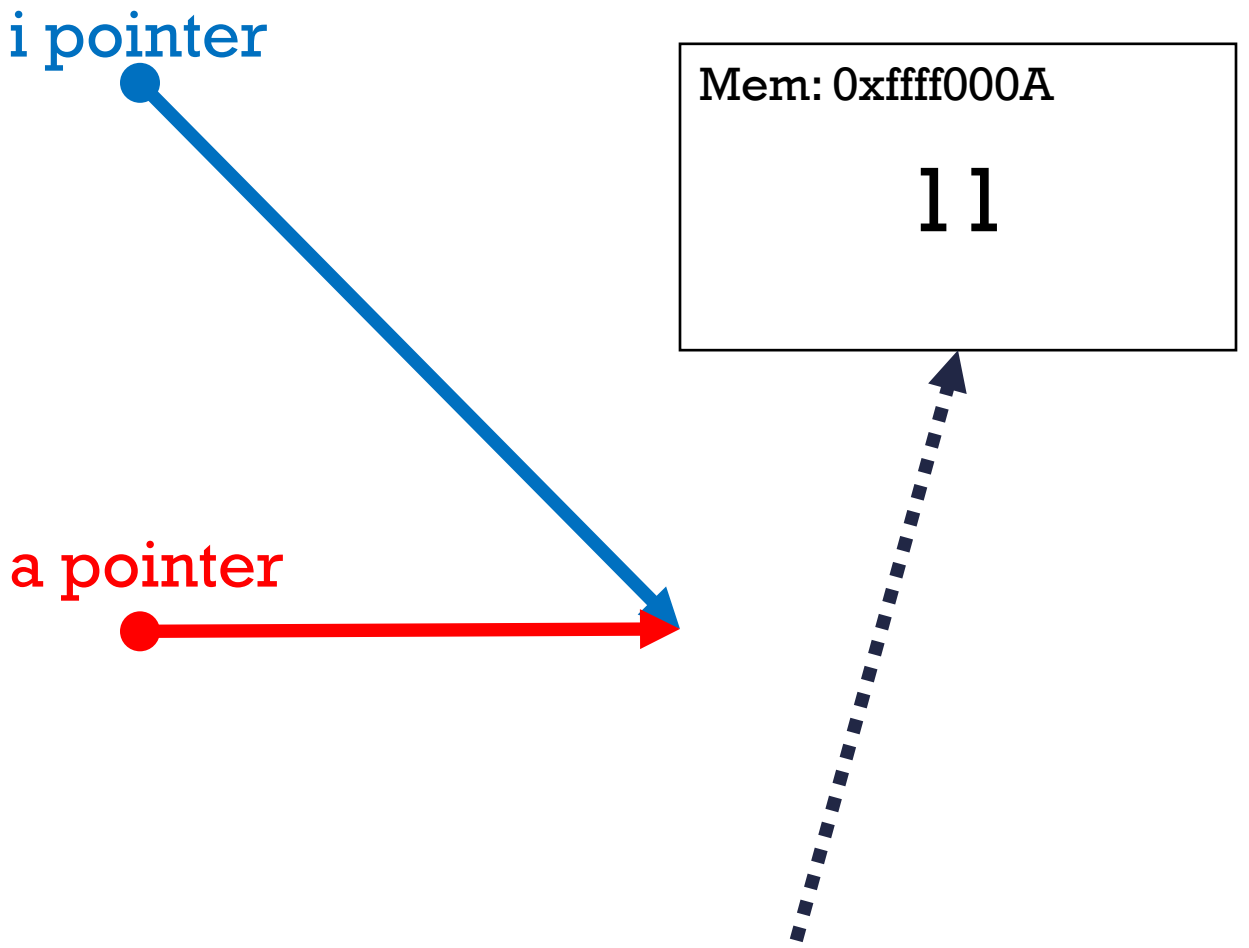
Memory leak example

```
int *i = new int{11};
int *a = new int{99};
i = a; //creates a memory leak

delete i; //free allocated memory
```



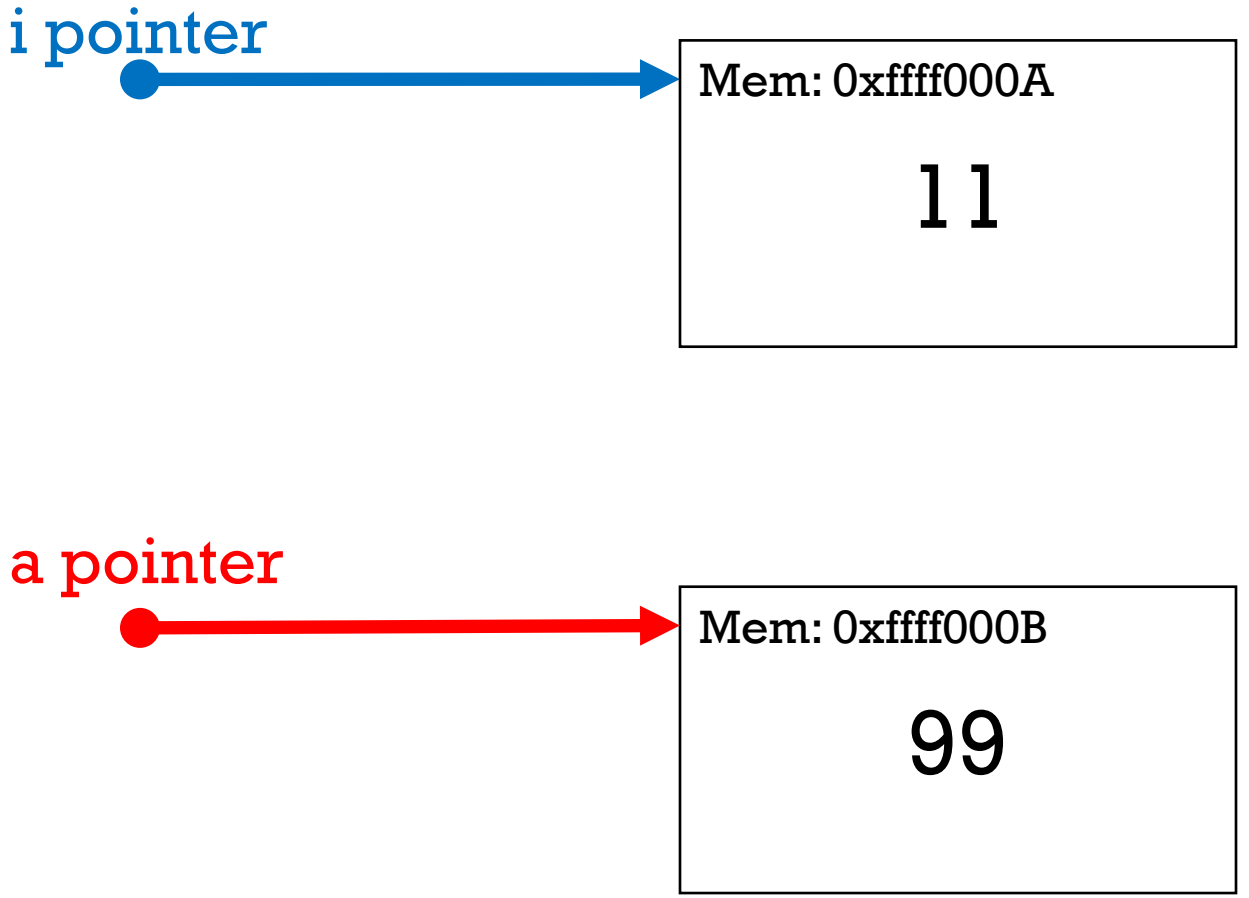
Memory leak example

```
int *i = new int{11};
int *a = new int{99};
i = a; //creates a memory leak

delete i; //free allocated memory
```



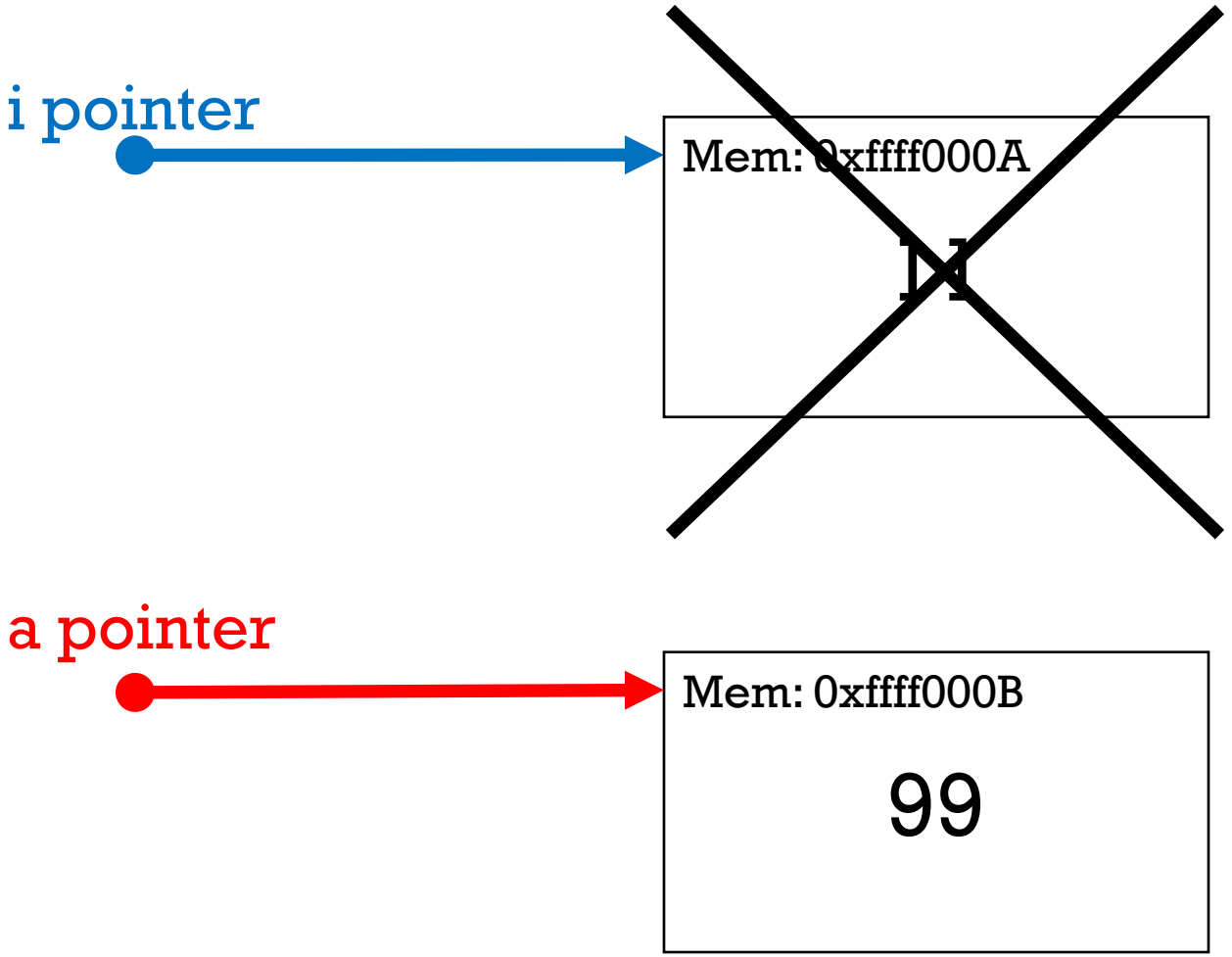
Memory leak solution

```
int *i = new int{11};
int *a = new int{99};
```



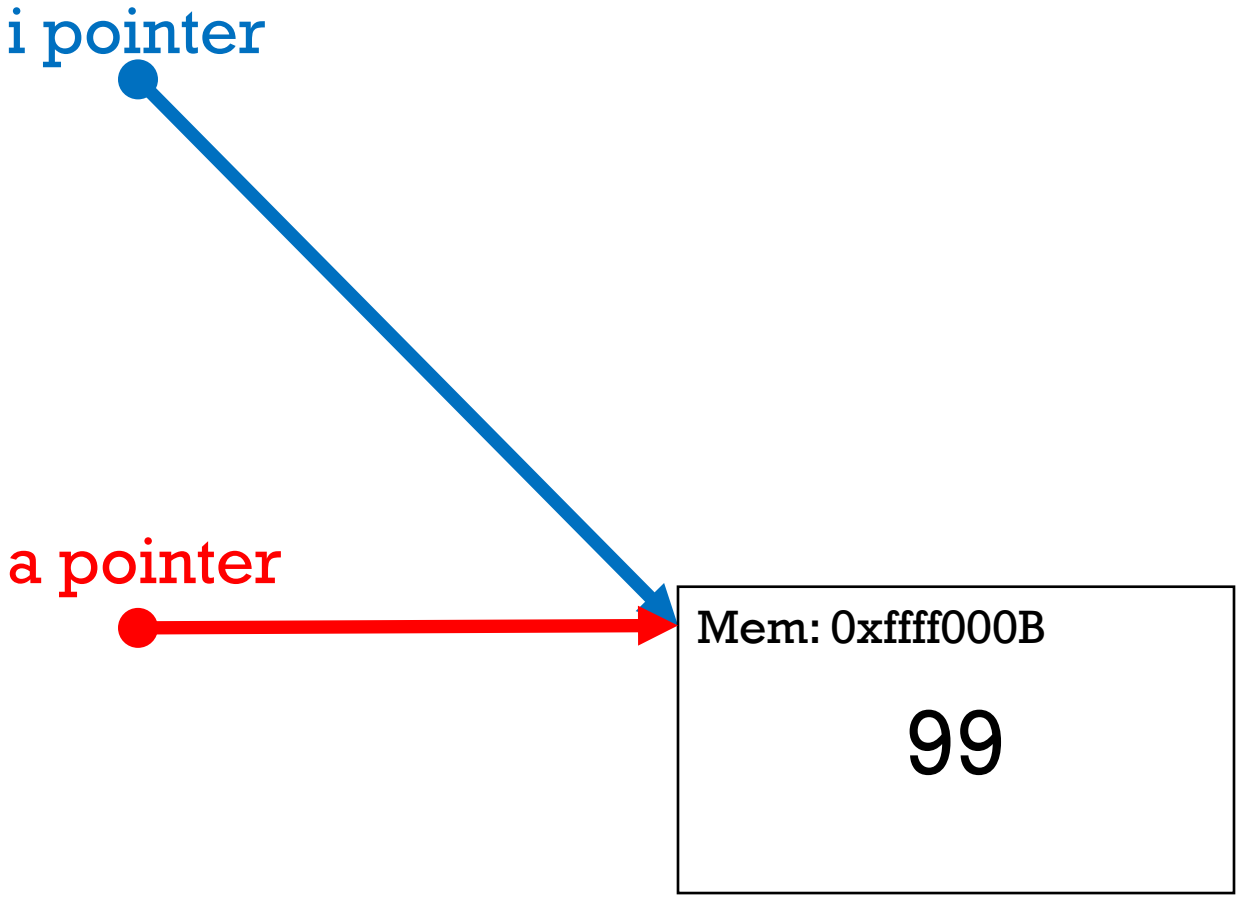
Memory leak solution

```
int *i = new int{1 1};
int *a = new int{99};
delete i; //deletes memory i pointing at
```



Memory leak solution

```
int *i = new int{1 1};

int *a = new int{99};

delete i; //deletes memory i pointing at

i = a; //i can now safely point to something else
```



Memory leak solution

```
int *i = new int{1 1};

int *a = new int{99};

delete i; //deletes memory i pointing at

i = a; //i can now safely point to something else

delete i;
```