• Q. How could you change your lab 1 code to replace a lowercase letter with a shifted uppercase letter?

A. There are many ways to do this, but the best is by simply adding a constant offset of ‘A’ at the end of the cipher command instead of ‘a’. I
have included an adaptation of my code which does this (by default).

```
#include <string>
#include <iostream>
#include <algorithm>
#include <locale>

typedef unsigned long long int AlphabetSizeType;
typedef char Char;
template <class T>
class AbstractCipherFunctional {
protected:
virtual T encipher(const T& clearData) = 0;
public:
virtual ~AbstractCipherFunctional() {}
inline T operator ()(const T& clearData) {
return encipher(clearData);
}
1};
template <class T>
struct ShiftCipherConfiguration {
public:
const T CIPHER_OFFSET;
const T OFFSET_CODE;
const T OFFSET_CODE2;
const AlphabetSizeType ALPHABET_SIZE;
ShiftCipherConfiguration(
T cipherOffset = T('j'),
T offsetCode = T('a'),
T offsetCode2 = T('A'),
AlphabetSizeType alphabetSize = 26
) :
CIPHER_OFFSET(cipherOffset - offsetCode),
OFFSET_CODE(offsetCode),
OFFSET_CODE2(offsetCode2),
ALPHABET_SIZE(alphabetSize) {
/* empty */
}
};
```

```
template <class T>
class ShiftCipherFunctional : public AbstractCipherFunctional<T> {
private:
const ShiftCipherConfiguration<T>& _config;
protected:
// This is the line that does all the actual work
T encipher(const T& clearData) {
return (
( clearData - _config.OFFSET_CODE+ _config.CIPHER_OFFSET)
% _config.ALPHABET_SIZE
2) + _config.OFFSET_CODE2;
}
public:
ShiftCipherFunctional(
const ShiftCipherConfiguration<T>& config
) :
_config(config) {
}
~ShiftCipherFunctional() {}
};
template <class Character = Char, template <class> class CipherFunctional =
ShiftCipherFunctional >
class EncipheringStream {
private:
CipherFunctional<Character> _cipherFunctional;
std::basic_string<Character> _cipherData;
public:
EncipheringStream(const std::basic_string<Character>& clearData,
CipherFunctional<Character>& cipherFunctional)
:
_cipherFunctional(cipherFunctional)
{
_cipherData.resize(clearData.length());
std::transform(
clearData.begin(),
clearData.end(),
_cipherData.begin(),
_cipherFunctional
);
}
std::basic_string<Character> get_enciphered_data() const {
3return _cipherData;
}
```

```
};
template <class Character = Char, template <class> class CipherFunctional =
ShiftCipherFunctional >
inline std::ostream& operator<<(std::ostream& out, const EncipheringStream<Character,
CipherFunctional>& estr) {
return out << estr.get_enciphered_data();
}
template <class Character = Char, template <class> class CipherFunctional =
ShiftCipherFunctional >
inline std::wostream& operator<<(std::wostream& out, const EncipheringStream<Character,
CipherFunctional>& estr) {
return out << estr.get_enciphered_data();
}
int main(int argc, char *argv[]) {
std::basic_string<Char> clearData = "abcdefghijklmnopqrstuvwxyz";
ShiftCipherConfiguration<Char> defaultConfig('j', 'a', 'A');
ShiftCipherFunctional<Char> cipher(defaultConfig);
EncipheringStream<Char> streamCipher(clearData, cipher);
std::cout << streamCipher << std::endl;
return EXIT_SUCCESS;
}
```

The code returns the following:

JKLMNOPQRSTUVWXYZABCDEFGHI

• Q. How would you change your lab 1 code to decrypt a shift-encrypted string?

A. By doing math! Specifically modular arithmetic. I want to be able to reuse my cipher code to return the decrypted string. This is easy. Let c be the clear text, e the cipher text, o 1 the first offset code, o 2 the second offset code, c o the cipher offset code ('j' by default in my code), and a the alphabet size.

The cipher equation is

$(c + o_1 + c_o) \bmod a + o_2 = e.$

Thus, switching c and e (we want to reverse the cipher):

4

$(1)(e + o_1 + c_o) \bmod a = (c - o_2) \quad (2)$

$(e + o_1 + c_o) \bmod a = (c - o_2) \bmod a \quad (3)$

$(e + o_1 + o_2 + c_o) \bmod a = c \bmod a \quad (4)$

$(e + o_1 + o_2 + c_o) \bmod a = c \bmod a \quad (5)$

We actually need to subtract 1 from the e + o 1 + o 2 + c o term because of the way the characters are encoded. Thus, adding this little code snippet to the main function from the previous question solves the problem.

```
std::basic_string<Char> out = streamCipher.get_enciphered_data();
char reverseCode =
```

(defaultConfig.ALPHABET_SIZE + defaultConfig.CIPHER_OFFSET +
defaultConfig.OFFSET_CODE + defaultConfig.OFFSET_CODE2 - 1)
% defaultConfig.ALPHABET_SIZE;
ShiftCipherConfiguration<Char> reverseConfig(reverseCode, 'A', 'a');
ShiftCipherFunctional<Char> decipher(reverseConfig);
EncipheringStream<Char> streamDecipher(out, decipher);
std::cout << streamDecipher << std::endl;
Which prints out
abcdefghijklmnopqrstuvwxyz

• **Q. Why does the add node function**
void add_node(node*& head_ptr, const int& payload)
pass in the head ptr by reference? Give me a scenario where this is needed.
A. For the same reason you ever pass an object by reference! You want to access (and perhaps alter) the literal object, not a copy of that object.
In this case we want to be able to manipulate the head ptr variable directly. This is obviously necessary if we want to add an element which
satisfies
payload < head_ptr->data
In this case we want to make the pointer change targets (to a new node). If we were to pass in head ptr by value then we would only be able to
change the target of the copy.

• **Q. For floating point numbers, which of these is most likely to be computationally accurate? Why?**
50.375
0.25
0.1
0.3
0.5
0.15
It actually does depend on how your machine is encoding the number, but the vanilla answer is this:
The numbers which can be constructed as a sum of (possibly negative) powers of 2, such that the magnitude of the difference between the largest
and smallest required power of 2 is less than the number of bits reserved for the mantissa (significand) in the floating point object your machine
employs will be most computationally accurate. If the difference between the largest and smallest required power of 2 is greater than the number
of bits reserved for the mantissa, round off error will occur and calculations using such a number will suffer some measure of degraded accuracy.
Additionally, if the magnitude of the number to be described is not in the range $2^{-N}, 2^{N} - 1$ where N is the number of bits reserved for the

exponent, then the number can not be described at all (overflow or underflow has occurred). Translated from pedantic nerd into English, that comes out as the following: if the number can be neatly written in binary, and the number is
not too large or too small for the machine to describe, then the result will be accurate. Thus we need to convert the above numbers into binary
to find which ones require too many bits to describe.
0.375 = 0.011
0.25 = 0.01
0.1 = 0.0001100110011...
0.3 = 0.01001100110011...
0.5 = 0.1
0.15 = 0.001001100110011001100110011001...
That is, 0.375, 0.25, and 0.5 are all exactly described in a floating point system, while 0.1, 0.3, and 0.15 are all approximated.
• Q. Write me a function that reverses the data in an int array. Would this be as easy to do in a singly linked list?
A. This is actually pretty easy to do.
The snarky answer is this

```
void reverse(int_array& arr) {
std::reverse(arr.data, arr.data+arr.count);
}
```

The more interesting answer is this:

```
6int_array* reverse(const int_array& arr) {
int_array* rarr = new int_array();
int* start = arr.data;
int* end = arr.data + arr.count;
int* target = rarr->data;
rarr->data = std::reverse_copy(start, end, target);
rarr->capacity = arr.capacity;
rarr->count = arr.count;
return rarr;
}
```

The cute answer is this:

```
void reverse(int_array& arr) {
int* start = arr.data;
int* end = arr.data + arr.count - 1;
int index = 0;
int middle = arr.count >> 1;
while (index < middle) {
std::iter_swap(start + index, end - index)
++index;
}
}
```

The official answer (not using standard library) is this:

```
int_array* reverse(const int_array& arr) {
int_array* rarr = new int_array();
init(rarr);
int* start = arr.data;
int* end = start + arr.count - 1;
for(int* itter i = end; i != start; --i ) {
add(*rarr, *itter);
}
return rarr;
}
```

• Q. Given an int array and a slot index, write me a loop that shifts the end elements in the int array data right to open a gap at that index.

A. Since you made me use a loop I will do this the boring way.

```
7void shift_right(int_array& arr, size_t index) {
if( arr.count + 1 >= arr.capacity ) {
resize(arr);
}
for( size_t i = arr.count - 1; index <= i; --i) {
arr.data[i + 1] = arr.data[i];
}
++arr.count;
}
```

• Q. Given an int array and a slot index, write me a loop that shifts the end elements in the int array data left to overwrite that index.

A. Since you made me use a loop I will do this the boring way (again).

```
void shift_left(int_array& arr, size_t index) {
for( size_t i = index; i < arr.count - 1; ++i) {
arr.data[i] = arr.data[i + 1];
}
--arr.count;
}
```

• Q. How can I tell if two linked lists are shallow copies of each other?

A. They will have head ptr values pointing to the same location in memory.

```
bool is_shallow_copy(const node* const& headA, const node* const& headB) {
return headA == headB;
}
```

• Q. What is wrong with this code for adding to an int array? How could you fix it?

```
if (arr.capacity == arr.count) {
resize(arr);
```

}
arr.data[++arr.count] = payload;
A. There are actually three things wrong with this code, two technical, one stylistic.
1. It is treating arr.count as an index. It is not an index. You need to subtract one from ++arr.count to get this to behave as intended. (works in this case)
2. It is not accounting for the fact that the array is about to grow by one in its resize check.
3. It needlessly attempts to do two things at once. Assign then increment. The “cute” trick is dumb, and dangerous if C++ macros are used.
This should be better:
if (arr.capacity <= arr.count + 1) {
resize(arr);
}
arr.data[arr.count] = payload;
++arr.count;

• **Q. What will this code do?**
for (unsigned int k = arr.count - 1; k >= 0; --k)
cout << arr.data[k];
A. It will print out (in reverse) the elements of the arr.data array and then segfault (barring exceptional conditions). Unsigned ints are always
greater than or equal to zero. Thus the loop will keep running until it “wraps” the k iterator. At which point the computer will attempt to
dereference the arr.data + 2^(8sizeof(unsigned int)) - 1 memory location, which is (in all likelihood) a memory access violation. It is possible that
arr.data contains 2^(8sizeof(unsigned int)) entries, in which case the loop will simply run forever.

• **Q. Write me a working function that uses an unsigned int in a loop to print the int array** backwards.
A. Sure.
for (unsigned int k = arr.count - 1; k != 0; --k) {
std::cout << arr.data[k] << std::endl;
}
std::cout << arr.data[0] << std::endl;
Done. That said, there is a somewhat more elegant way:
for (unsigned int k = 0; k < arr.count; ++k) {
std::cout << arr.data[arr.count - 1 - k] << std::endl;
}
• **Q. Why do you get a warning when assigning between ints and unsigned ints? Is this a good idea or a bad one?**
A. You get a warning because there is the potential for data loss / corruption. Unsigned ints use two’s compliment to store negative numbers.
Thus you will get a stupidly large value if you do this:
9unsigned int a = static_cast<int>(-1);

The problem can also (obviously) go in the other direction.
int a = static_cast<int>(very large number here)
resulting in a negative value of a. Usually not what you want. Is it a good idea? Not normally. Unless you need to work with the binary for some
reason or need to do some kind of crazy math then I doubt this is the behavior you want.

• Q. When I pass a pointer by value, what happens to the data it points to?
A. Nothing. all the work is being done on a copy

• Q. When should I pass a pointer by reference?
A. Same rules for a general object apply. You pass by reference when you want to access the literal portion of memory the object occupies. If you
want to change what the pointer points to, you need to pass by reference.

• Q. When should I pass a pointer by constant reference?
Any time we wish to protect the data we're pointing to. For example a print function doesn't want to change the original. Copy would be another good one. Basically any function that needs read permission only

• Q. Why do I have to pass a size along with an array of ints?
A. In C / C++ arrays are just pointers pointing to the start of a block of memory. Thus, even on the stack, they do not encode enough information
(by themselves) to tell calling function / method how long the array actually is. Thus, you need to pass in a length.

• Q. Will this function
void lobo(int& num, unsigned int size)
10let us change the data for num? Will it let us change the address of num in memory? If so, which of these changes will be permanent after the function runs? Why?
• A. Yes, lobo will allow us to change num. No, lobo will not allow us to change the address of num in memory. Any changes made to num by
lobo will be persistent after lobo runs. This is because num is passed in by reference, thus the literal section of memory that num occupies will
be given to lobo for manipulation.

• Q. Will this function
void zobo(int num, unsigned int size)
let us change the data for num? Will it let us change the address of num in memory? If so, which of these changes will be permanent after the function runs? Why?
A. Yes, zobo will allow us to change num. No, zobo will not allow us to change the address of num in memory. Any changes made to num by
zobo will not be persistent after zobo runs. This is because num is not passed in by reference, thus it is only a copy of the argument which zobo

manipulates.

- **Q.Will this function**
  void bobo(int* array, unsigned int size)
  let us change the data in an array? Will it let us change the address of the array in memory. Of these changes will be permanent after the function runs? Why?

A. Yes, bobo will let you change the data in an array. It will let us change the address of a shallow copy of the array in memory. This copy will cease to exist on function termination. Any changes made to the data pointed to by the array pointer will be persistent. The reason bobo will let you change the data in an array is simple: the copy of the array pointer points to the array as well. Thus we do have a way to get direct access to the array memory.

- **Q. Will this function**
  void flobo(const int*& array, unsigned int size)
  let us change the data in the array? Will it let us change the address of the array? If so, which of these changes will be permanent after the function runs? Why?

A. No, flobo will not let you change the data in the array. Yes it will let you change the address of the array, and that change will be persistent.
You can not change the data because the pointer points to constant data. Thus *array = whatever is not allowed. On the other hand, array = whatever is allowed, and because the array is passed in by reference then that change will be persistent.

- **Q. Will this function**
  void globo(int*& array, unsigned int size)
  let us change the data in an array? Will it let us change the size of the array? Will it let us change the address? If so, which of these changes will be permanent after the function runs? Why?

A. Yes, globo will let us change the data in the array. Yes it will let us change the size of the array. Yes it will let you change the address of the array (and indeed this is required if you want to change the size of the array). All changes made to array will be persistent after the function runs. int*& passes in an array by reference. Arrays are just pointers. Thus you can change the data (by following the pointer to the target and changing the target). You can also run array = whatever to change the location it points to (be careful not to leak memory), and if you run array = new int[someOtherSize] then you will change the size of the array. That said, this function has no way of returning that new size (size is passed in by value), so this is likely a bad idea.

• Q. What is the main behavioral difference between variables on the local memory (stack) and variables on the heap?
A. Variables on the heap are preserved after the termination of the scope in which they were defined. Stack variables are not.

• Q. How do I copy the last 7 elements of a 20-element array to the last 7 elements of a 45-element array using the copy command?
A. With science!

```
const size_t A_SIZE = 20;
const size_t B_SIZE = 45;
const size_t C = 7;
int* a = new int[A_SIZE];
int* b = new int[B_SIZE];
// insert code which initializes a and b
std::copy(a + A_SIZE - C - 1, a + A_SIZE, b + B_SIZE - C - 1);
```

• Q. How can I tell if a pointer is bad?
A. You can't really tell whether a pointer is pointing to a valid address in memory. The only way you have to protect yourself from running down pointers into whatever they are pointing at is a nullptr, or null flag.

• Q. Why is this code dangerous?

```
string& get_a_string() {
string answer;
cout << "Tell me a word: " << endl;
cin >> answer;
12
Will return false otherwise.return answer;u
}
int main() {
string user_input = get_a_string();
cout << "You entered " << user_input << endl;
}
```

A. The code is dangerous because it pretends that stack variables are guaranteed to survive past the scope in which they were declared. It simply
is not so.

• Q. Why is insertion sort O(n 2) at the worst? What is the best case for insertion sort?
A. You have n numbers to sort. The worst case for insertion sort is that the numbers are in reverse sorted order. Term 0 requires no moves. Term
1 requires one swap (3 moves). Term 2 requires 4 moves. Term 3 requires 5 moves. And so on, up to Term n - 1, which requires n + 1 moves. So
we have,

$$\sum_{i=1}^{n-1}(i+2) = n \cdot \frac{3+n-1}{2} = \frac{n^2+n}{2} \quad (6)$$

Thus, worst case we have $O(n^2)$.

The best case is that the array is already sorted, in which case we only need to compare $n-1$ integers. Thus $O(n)$.

• Q. What can an array do faster than a list (and why)?

A. An array supports random access. Thus it is really good at accessing data quickly (each memory location is implicitly calculable from any other memory location in the array). On the other hand, you need to search through every element in a list to find the element you are looking for; no random access possible. Arrays also are usually better at inserting elements at the end of the array than lists are. This is because arrays can reserve extra memory for multiple elements at a time, while lists need to allocate memory element by element. If you are bold and willing to write a deque then this property extends to inserting elements at the beginning of the array.

• Q. What can a list do faster than an array (and why)?

A. A list can insert elements in the middle of the array much faster than an array can. This is because lists do not use continuous sections of memory, and as such you only need to break one link and form two more to run an insert. On the other hand, arrays require you to move some or all of the elements in the array in order to insert an element.

• Q. What error happens if I write a loop with a condition like this to look for an element in an int array data:

```
unsigned int k = 0;
while (arr.data[k] < target && k < arr.count)
++k;
```

13A. You will get a segfault if target is larger than any element in [arr.data, arr.data + arr.count). If not, the code will “work,” in that it will make k satisify arr.data[k] >= target;.

The segfault results from the fact that the conditions are out of order. Any decent C / C++ compiler generates code which checks conditions in a lazy manor. Thus if the code instead read:

```
unsigned int k = 0;
while ( k < arr.count && arr.data[k] < target )
```

++k;
then all would be well (the segfault resulting check would not occur).

• Q. What happens in the following code, and is it a problem?
node* head_list_1 = nullptr;
add_node(head_list_1, 9);
add_node(head_list_1, -78);
add_node(head_list_1, 200);
node* head_list_2 = list_head_1;
add_node(list_head_2, 100);
print_list(list_head_1);
remove_node(list_head_2, -78);
print_list(list_head_1);
A. I will just comment the code.
node* head_list_1 = nullptr; // declare new node pointer and set it to nullptr
add_node(head_list_1, 9); // add 9 to the list
add_node(head_list_1, -78); // add -78 to the list
add_node(head_list_1, 200); // add 200 to the list
node* head_list_2 = list_head_1; // declare a new node pointer and make it a shallow copy of list_head_1
add_node(list_head_2, 100); // add 100 to the list (both pointers point to same list)
print_list(list_head_1); // print the list [-78, 9, 100, 200] (there is only one list)
remove_node(list_head_2, -78); // remove -78 from the above list
print_list(list_head_1); // prints out list containint [9, 100, 200]
This behavior is not a problem if you did intend a shallow copy. It is a bug if you needed a deep copy.

14• Q. Tell me why it's better to double the int array capacity when we resize. (Hint: plot out how many items you copy as your count increases
when your resize function doubles the capacity, and how many items you copy when your resize just adds 20 more slots to the capacity. Comparing these
plots should help.)
A. Resizing is a very expensive operation as we have to allocate a brand new chunk of memory which is larger than the current one and then proceed to copy all the current data values over to the new chunk. Doubling the size of an array means resize has to run significantly fewer times than for adding 20 assuming we're adding the same number of items to each array. The more we add to the array the more efficient doubling gets.

• Q. Suppose my algorithm is quadratic, and I triple the size of its input. How much longer will it take to run?
A. If the algorithm requires O n 2 , then tripling the size of the input will require time proportional to (2n) 2 = 8n. Thus it will take 8 times as
long to run.

• Q. Suppose my algorithm is logarithmic, O(log 2 (n)) and I quadruple the size of its input. How much longer will it take to run?
A. As before, the calculation is simple. Time proportional to log 2 (4n) = log 2 (4) + log 2 (n) = 2 + log 2 (n). That is, if t(n) = α log 2 (n), then
t(4n) = 2α + t(n) → Δt = 2α.

Notes:

By Value:

```
void aMethod(aParameter p) { }
```

When passing by value, a temporary COPY is made in memory taking up space and making any changes to the variable *useless* outside of the aMethod scope. Once control returns to the calling procedure, any changes to the variable will not be recognized, and you will still have the original variable.

By Reference:

```
void aMethod(aParameter& p) { }
```

NOTE: The & can be placed against the parameter type, against the parameter name, or there can be a space on either side.

This essentially passes a pointer but not the same kind of pointer you would get when creating the following:

Variable* v;

That is why it is called by reference. Since that is what you have, p now acts as if it were the parameter just like in the by value way, but any changes affect the original and stay changed outside of the aMethod scope.