You can find code from the last live lecture on Canvas Modules (Kennslustund06.zip).
You don't need to start there but you can use some of the classes already defined there, like the DataClass.

It is possible to typecast a reference or pointer to any data type into a char* or void*. When this is done the data will not change at all, but the memory location will simply be referenced and the underlying data read byte by byte through the pointer (char* or void*).
This can be helpful when you want to send anything into ofstream.write() or in networking when building network packages to write. The char* or void* can then be typecast back into the original data type, but then obviously the programmer must make sure that actual data is correct and will fit into the size of the type or class used.
Try all of these and understand how they work and where the data ends up:

```
int some_integer = 25984;
//note that it's the reference being typecast, not the integer itself
char *bytes = (char *)(&some_integer);
output_file.write(bytes, sizeof(int));

DataClass some_data(93478, 'k');
char *bytes = (char *)(&some_data);
output_file.write(bytes, sizeof(DataClass)); // or
output_file.write((char *)(&some_data), sizeof(DataClass));

char *bytes_copy = new char[sizeof(DataClass)];
memcpy(bytes_copy, (void *)(&some_data), sizeof(DataClass));

DataClass *some_data_pointer = new DataClass(93478, 'k');
//Here we typecast the pointer.
//No need for an extra reference since the pointer is already a memory location
char *bytes = (char *)some_data_pointer;
output_file.write(bytes, sizeof(DataClass)); // or
output_file.write((char *)some_data_pointer, sizeof(DataClass));

DataClass some_data;
input_file.read((char *)(&some_data), sizeof(DataClass));

DataClass some_data;
char *bytes = new char[sizeof(DataClass)];
input_file.read(bytes, sizeof(DataClass));
memcpy((void *)(&some_data), bytes, sizeof(DataClass));
```

Take special note if your class includes **pointers** as instance variables, as only the pointer will be read, not the underlying data. In these cases a programmer needs to write functions to serialize or pack the actual data into an array of bytes, similar to what we have been doing in previous assignments.

```
Class DataClass{
public:
   DataClass(int i, char *str);
   int my_number;
   char *my_string;
};
DataClass some_data(93478, "some string here");
char *bytes = (char *)(&some_data);
output_file.write(some_data, sizeof(DataClass));
```

Here the data written will only include the memory location of the string, not the actual characters. This obviously doesn't help, since the memory location of a string when read in some other run of the program (or a different program) will probably not be the same.

Check out the solutions ZIP for **exercises 6**. There are even more versions of char* and void* typecasts, memcpy, read and write.

It is not necessary to use any of this in assignments, but it is definitely an extra choice to add to your repository of C++ programming methods. Knowing this may also help a programmer easily solve problems in unexpected places later on.

Connecting to skel.ru.is
In PA3 and following assignments I will expect students to check that their programs compile and work on a common remote linux server. This means that if we can't compile your code on our own machines, we will compile it on skel. If we can't compile it on **skel** we will simply not compile it, which is no fun!
It is best to try to make your code work on both your own machine and on skel. That maximizes the odds of it working on any machine, and it will probably be the best standard C++ as well.

For now compile your code with the compile argument **-std=c++11**. This is a modern standard, including any new C++ additions that we may need to use for now, but also solid and included in pretty much every compiler.

Connecting onto skel to work there:
ssh ru_user_name@skel.ru.is
Use official RU password

mkdir cpp_course
cd cpp_course
mkdir PA3

Copying files between local directory (own machine) and home directory on skel
(do this on in local directory):
scp my_files.cpp ru_user_name@skel.ru.is:~/cpp_course/PA3/
scp ru_user_name@skel.ru.is:~/cpp_course/PA3/my_outputs.txt ./

Bitstrings!
If you are only implementing version A in PA3, then it should be enough to store your bitstrings as strings with the characters 0 and 1. If you're going for version B though, you will need to pack them into actual bytes. Either way you will want to make some version of the following...
...special practice exercise on the next page (same as last two weeks)...

Here's one that I will not give a solution for until after **PA3** *is handed in, but it's worth the time to implement it and hang on to. It may not be the fastest way to do this (some students may later wish to optimize) but getting it to work properly is key to building bit strings in character arrays where the data may need to be read on a tighter level than in whole bytes*.

Make the class BitString (or whatever you want to call it).
Here is a suggestion for operations, but this class will not be handed in, but you may want to use it as part of a later assignment, so make it in a way that you understand and feel works for the way you may use it.

BitString::BitString()
You may want to have parameters, initial number of bytes in the array, etc. You may also wish to make a copy constructor and maybe both a default constructor and one with parameters, or one with parameters with default values, whichever works for your implementation.

BitString::~BitString()
Destructor to clean up the byte/bit data.

void BitString::setNextBit(bool value)
Sets the next bit in the bitstring.
use &, |, << and >> to actually set the next bit. Don't use a whole byte to represent a single bit.
It can help to keep track of a current_bit or bitstring_position variable or something like that.

bool BitString::getNextBit()
Gets the next bit in the bitstring.
use &, |, << and >> to actually get the value of the next bit. Then return a boolean value representing 1 or 0.
It can help to keep track of a current_bit or bitstring_position variable or something like that.

void BitString::goToStart()
Set the get/set position to the start of the string. Can be helpful when you start to read from the bitstring after having added to it. Students can of course have separate set_pos and get_pos variables if that works for them.

char* getBytes() (or unsigned char if that's what you're using)
Returns a pointer to the actual array of bytes holding the data. This can be used to write it to a file.

int getLength()
Return the length of the bit string (in bits)

bool isAtEnd()
Returns True if the last bit has been read (current_position is at the end). This can be helpful when reading back from the bit string.