Associative containers: maps

Vectors are useful when we want to store a bunch of objects and access them one by one.

But this is not suitable for certain applications!

For example, consider the problem of calculating the frequency that each word appears in a text (this kind of problem comes up a lot in computational linguistics!)

This *can* be done only with vectors, but it's not too nice! See `Program14.cpp`

Associative containers: maps

Vectors are *sequential* containers: they represent a sequence of elements that you can access in order.

In contrast, maps are *associative* containers. They have a `key` and a `value` type, and they associate objects of `key` type with objects of `value` type.

This is very convenient in our example, because we seek to associate a word (the key) with a number (the value).

Associative containers: maps

```
#include <map>

std::map<std::string, int> wordCounts;

// Adding an element to a map
wordCounts["someWord"] = 10;

// Reading an element of the map
int counts = wordCounts["someWord"];
// Now count is 10

// If the word isn't found, the result is 0
counts = wordCounts["noSuchWord"];

// We can check whether a word appears or not:
bool appears = wordCounts.count("someWord") == 1;
```

Associative containers: maps

Iterating over a map is more complicated!

```
std::map<std::string, int> wordCounts;
for (int i = 0; i < wordCounts.size(); i++) {
  wordCounts[i]... // we can't do this anymore!
}
```

Instead of using integers for indices, we must use *iterators*!

See `Program15.cpp` for an exercise and an example of use!

Avoiding copies: references and pointers

Imagine we want to create a function that removes all the duplicate elements from a vector.

```
void removeDuplicates(std::vector<int> vec) {
  // We do something with vec
}
int main() {
  ...
  removeDuplicates(someVector);
}
```

But actually, this won't work!
When we call a function in C++, it gets a *copy* of the arguments we give it!
So any changes we make to `vec` inside this function only affect the copy!

Avoiding copies: references and pointers

A solution:

```
vector<int> removeDuplicates(vector<int> vec) {
  // We do something with vec
  return vec;
}
int main() {
  ...
  someVector = removeDuplicates(someVector);
}
```

Exercise: see `Program16.cpp` and fill in the gaps to implement `removeDuplicates` like this!

This does work but...

We are copying the vector around twice!

Avoiding copies: references and pointers

Instead of copying the vector when we call `removeDuplicates`, we'd prefer to have the function have access to the original version.

We can do this via *references*

Variables and function parameters can be references. This means they are simply an alias for some other variable.

```
vector removeDuplicates(std::vector<int> &vec) {
    ...
}
std::vector<int> myVector;
removeDuplicates(myVector);
```

See `Program17.cpp` for some examples.

Avoiding copies: references and pointers

Pointers are a bit like references, but more explicit to use (but they can do more things!)

A pointer represents a location in your computer's memory that contains some data.

```
int* pointer; // We don't have to initialize this!
int a;

pointer = &a; // Now pointer "points to" a
*pointer = 10; // This changes a!
```

See `Program18.cpp` for more examples!
A word of caution: pointers are *hard*. It's very easy to use them in wrong ways. We will avoid them as much as possible!

Object lifetime

So far, all our data has been stored in local variables.
Local variables exist only for as long as the local block:

```
void someFunction() {
  std::vector<int> v; // v is created here

  return; // v is destroyed here
}
```

A block is the body of a function, a `if` statement or any kind of loop:

```
if (someCondition) {
  std::vector<int> v; // v is created here
}
// here v doesn't exist anymore!
```

Object lifetime

Normally, when some variable is destroyed, it doesn't matter because we can't access it anymore.

But with pointers, we can circumvent this limitation!

```
std::vector<int>* someFunction() {
  std::vector<int> vec;
  return &vec; // vec is destroyed here!
}
std::vector<int>* v = someFunction();
// What happens if we access an element of \texttt{
```

See `Program19.cpp` for a running example!

Object lifetime

We can create objects that are not tied to a certain block!

This is done with the `new` keyword, which creates an object in a "central storage" and gives us a pointer to it.

```
std::vector<int>* someFunction() {
    std::vector<int>* vec = new std::vector<int>();
    return vec;
}
std::vector<int>* v = someFunction();
```

But since the resulting object will not be deleted at the end of the function, we are responsible for deleting it when we are done!

See `Program20.cpp` for an example.

An introduction to classes

Classes are a way of organizing code.

A class is some data plus some functions that operate on said data.

An example: a bank account.

- It has some data (the name of the owner, the balance)
- It has some methods (to deposit money, to withdraw it)
- The balance should never be allowed to be negative!

An introduction to classes

```
class BankAccount {
  public:
    // Constructor
    BankAccount(std::string owner) {
      this->owner = owner; }

    // Methods
    virtual void deposit(int amount) {
      this->balance = this->balance + amount; }
    ...

  protected:
    // Attributes
    int balance;
    std::string owner;
}
```

An introduction to classes

Once we have defined a class, we can create *instances* using `new`.

```
BankAccount* acc = new BankAccount("John Doe");
acc->deposit(100);
acc->summary();
```

Every instance of `BankAccount` keeps their own copies of `balance` and `owner`.

See `Program21.cpp`

Inheritance

Suppose our bank has regular clients and VIP clients. VIP clients have credit, so their balances are allowed to be negative!

We could create a completely different class `VIPAccount` and copy all the code of `BankAccount`, modifying the method `withdraw`

But then we would have two different types! We wouldn't be able to store all our clients in a single `std::vector`, for example.

Through *inheritance*, we can make `VIPAccount` share the code of `BankAccount`, and the types will be compatible!

Inheritance

```
class VIPAccount : public BankAccount {
  public:
  VIPAccount(std::string owner)
    : BankAccount(owner) {
  }

  virtual bool withdraw(int amount) {
    ...
  }
}
```

`VIPAccount` inherits all the methods and variables of BankAccount, but *overrides* some of them!

See `Program22.cpp`

Putting it all together

With all this, we can make a dungeon-crawler game!

Take a look at `Program23.cpp`, try to understand the code, ask questions, change things, see what happens!