C++ Strings and Containers

CSE 1320

Alex Dillhoff

University of Texas at Arlington

Strings

The C++ standard library provides a `string` class which has many convenient functions.

For example, it is simple to perform basic operations like concatentation:

```
string fname {"Amos"};
string lname {"Burton"};
string fullname {fname + " " + lname};
```

Strings

It is even possible to concatenate onto an existing string with +=.

```
string address {"500 UTA"};
address += " Blvd.";
```

Strings

There are many class functions which provide a much more usable interface than what we saw in C.

```
string name = "amos Nagata";
// Get the last name
string lname = name.substr(5, 6);
// replace the first 4 characters
// with "naomi"
name.replace(0, 4, "naomi");
// capitalize the first character
name[0] = toupper(name[0]);
```

Strings

String comparison is even simpler with the overloaded == operator.

```
bool checkPassword(const string& pw1, const string& pw2) {
    return pw1 == pw2;
}
```

Strings

There are many other features of `strings`, including formatting.

These can be referenced documentation found online.

Containers

Computers are powerful for their ability to manipulate a large collection of objects very quickly.

Thus, a useful programming language will offer efficient ways of representing such a collection.

C++ provides many different containers used to store both basic and user-defined types.

Containers

The most common container used is the `vector`.

A `vector` can be created for any type.

```
vector<Ship> ships = {
    {"Rocinante", 1234},
    {"Nebuchadnezzar", 4321}
};
```

Containers

A `vector` can be initialized a number of ways.

```
vector<int> v1 = {1, 2, 3, 4}; // size 4
vector<string> v2; // size 0
vector<double> v3(32,1.0); // size 32, first element is 1.0
```

Containers

Elements can easily be added to a `vector` using `push_back()`.

```
    Ship s = {"Columbia", 2003};
    ships.push_back(s);
```

Range Checking

It is possible to use exceptions to safely guard against errors that would otherwise cause a catastrophic error when using containers.

```
try {
    Ship s = ships[5]; // out of range index
} catch (out_of_range) {
    // handle error
}
```

Linked Lists

The C++ Standard Library provides an implementation of a doubly-linked list called `list`.

As a container, its initialization is very similar to that of `vector`.

```
list<Ship> ships = {
    {"Rocinante", 1234},
    {"Nebuchadnezzar", 4321},
    {"Columbia", 2003}
};
```

Linked Lists

Accessing individual elements is done through iterating.

```
for (const auto& s: ships) {
    cout << s << endl;
}
```

Linked Lists

It is possible to insert and erase elements of a `list`.

```
list<Ships>::iterator s1_itr; // *s1_itr refers to a Ship
list<Ships>::iterator s2_itr; // *s2_itr refers to a Ship
Ship new_ship {"Challenger", 2020};
ships.insert(s1_itr, new_ship);
ships.erase(s2_itr);
```

Hash Map

C++ also provides an implementation of an `unordered_map`.

This abstracts all of the core functions such as rehashing away from the developer.

Hash Map

An `unordered_map` is initialized as follows.

```
unordered_map<string, int> users {
    {"Naomi", 1},
    {"Amos", 2},
    {"James", 3},
    {"Chrisjen", 4}
};
```

Hash Map

Given a key, the value can easily be retrieved using array indexing notation.

```
int id = users["Naomi"];
```