Week 10 Tutorial

NWEN 241
Systems Programming

Alvin Valera

Alvin.valera@ecs.vuw.ac.nz

std::string in C++

- C++ has a **`string`** class type that implements string datatype
- Not *exactly similar* to Java String class
  - Java strings are immutable reference types; C++ strings are mutable!
- Wide range of operators and member functions are available for variables declared as string type
- Include **`<string>`** header file
- Use **`standard`** namespace

```
#include <iostream>
#include <string>
using namespace std;

int main (void)
{
  string s1, s2; // empty
  s1 = "Hello";
  s2 = "Hello World !";

  cout << "String 1: " << s1 << "Length: "
  << s1.length() << endl;

  cout << "String 2: " << s2 << “Length: "
  << s2.length() << endl;

  return 0;
}
```

Character Array vs String Class in C++

String Operation	Character Array	String class
Copy string s2 into string s1.	`strcpy(s1, s2);`	`s1 = s2;`
Concatenates string s2 onto the end of string s1.	`strcat(s1, s2);`	`s1 + s2;`
Returns the length of string s1.	`strlen(s1);`	`s1.length();`
Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.	`strcmp(s1, s2);`	`==, <=, >, and >=` operators Or `s1.compare(s2)`
Returns a pointer to the first occurrence of character ch in string s1.	strchr(s1, ch);	`s1.find(ch)`
Returns a pointer to the first occurrence of string s2 in string s1.	strstr(s1, s2);	`s1.find(s2);`

Inheritance - Extending Classes

- Syntax of extending a single base class:

```
class subclass_name : access_mode baseclass_name {
      class_member_list
};
```

  - *subclass_name* is the identifier given to the sub class being declared
  - *access_mode* controls the access of inherited fields
  - *baseclass_name* is the identifier of the super class being extended

Let’s write some code

- Rewrite the class `GameCharacter` (discussed in W9 Tutorial) such that the members `level`, `health` and `experience` can be inherited.

- For the same class, rewrite the parameterized constructor to use an *initializer list*

Let’s write some code

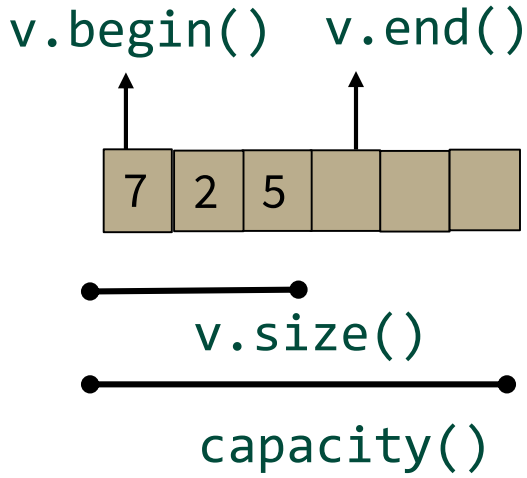
- Extend the class `GameCharacter` as follows:
  - Call the sub-class `Mage`, with a parameterized constructor to take in `name`
  - `Mage` should preserve the access specifiers of the parent
  - It should have a new private attribute called `mana` (magical energy)
  - It should have two public methods called `castSpell()` and `renegerateMana()`
  - It should override the `display()` method
  - Write a C++ program (with `main()` function) to use the new class

- Make the `display()` method in `GameCharacter` virtual
  - What happens if `Mage` does not implement `display()`?

- Make the `display()` method in `GameCharacter` pure virtual
  - What happens if Mage does not implement `display()`?

Vector

- One of the containers is **Vector.**
- Defined in `<vector>`
- Same as **dynamic arrays** with the ability to resize itself **automatically** when an element is inserted or deleted, with their storage being handled automatically by the container.



- Vector elements are placed in contiguous storage.
- The **capacity** of the vector is decided by the compiler (implementation dependent). It is generally bigger than the **size** of elements in it.
- This gives the ability to quickly insert an element to the end or remove the last one, just by keeping track of the number of elements.
- Vectors also have safety features that make them easier to use than arrays, automated bounds checking and memory management

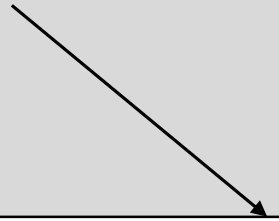
Example

```
#include <iostream>
#include <vector>
using namespace std;

int main(void)
{
    vector <int> v ={1,2,3,4,5};

    cout<<"Incrementing the vector by 1:"<<endl;

    for (vector<int>::iterator it = v.begin() ; it != v.end(); ++it){
        *it = *it + 1;
        cout << ' ' << *it;
    }
}
```



can also use auto – the auto keyword dynamically determines the data type of the assigned value

```
Output:
Incrementing the vector by 1:
2 3 4 5 6
```

Let’s write some code

- Write some code to demonstrate different ways to initialize a vector
  - Use iterator to go through the vector
  - What happens if we change the iterator type to `auto`?

Accessing members of a vector

```
for (vector<Entry>::iterator it = book.begin() ; it != book.end(); ++it)
{
      cout<< it->name <<" "<< it->number << endl;
}
```

for (*range_declaration* **:** *range_expression* **)***loop_statement*

a declaration of a named variable, whose type is the type of the element of the sequence represented by range_expression, or a reference to that type. Often uses the auto specifier for automatic type deduction.

any expression that represents a suitable sequence or a braced-init-list.

```
struct Entry { string name; int number; };
//book is a vector of Entries
```

Range-based for loop

```
void print_book(vector<Entry> & book)
{
    for (const auto& x : book)
         cout << x.name <<" "<<x.number << endl;
}
```

Let’s write some code

- Write some code to demonstrate different ways to initialize a vector
  - Use iterator to go through the vector
  - What happens if we change the iterator type to `auto`?
  - What happens if you access an element using [ ] that is out of bounds?
  - How about if you use `at()`?
- Rewrite above code to use new for-loop
- Modify above code as follows:
  - Insert a new element at the end of the vector
  - Insert a new element at the beginning

Let’s write some code

- What happens if you remove a vector element while traversing it with an iterator?