Maciej Sobieraj

Lecture 3

Outline

1. **Accessing different kind of data**

Conversion

- What is a conversion?
  - **A conversion is the act of changing the nature of the data without (if possible) changing the value**.
- You can avoid conversion by coding the literal in a clearer way, e.g. like this:
  - **long** data = 1L;
- The suffix '*L*' (or '*l*' – they're interchangeable) says that the literal is explicitly defined as *long*.

Conversion

- All possible conversions are divided into two classes:
  - **automatic conversions** which are performed somewhat behind our backs
    - **implicit conversions**
  - **explicit conversions** which are performed at the developer's command and expressed using special language
    - **typecasting**

Implicit conversions

```
int Int = 1;
short Short = 2;
long Long = 3;
float Float = 4.0;
double Double = 5.0;

int f(int x) {
      return x;
}

// example no. 1
Int = Int + Short;
// example no. 2
if(Double)
 Double--;
// example no. 3
Float = 1;
// example no. 4
f(Float);
// example no. 5
float g(void) {
      return -1;
}
```

- Some of the contexts where implicit conversions play an important role:
  - a value is used as **part of a complex expression** built of many values of different types (example 1)
  - a value plays **the role of a logical condition** within instructions like *if*, *while*, *do*, etc (example 2)
  - a value is **subject to assignment** and is used to:
    - change the value of a variable (example 3)
    - set the value of a formal parameter (example 4)
    - specify the return value of a function (example 5)

Explicit conversions

- The C++ language give us two ways to specify explicit conversions:
  - the so-called **C-style casting**
    - (new_type_name) expression_of_old_type
  - the so-called **functional notation**, which is a native C++ syntax convention
    - new_type_name(expression_of_old_type)

```
#include <iostream>
using namespace std;
int main(void) {
   float f = 3.21;
   double d = 1.23;
   int k = int(f) + (int)d;
   cout << k << endl;
   return 0;
}
```

Conversions – gains and losses

- **the length of the memory representation remains the same or increases**
  - we can be confident then that the original value will be preserved. We can expand it with zero bits to fill the target memory space and the sign bit may be moved to its new position, but the value itself will not change.

```
#include <iostream>
using namespace std;
int main(void) {
        short s = 32767;
        int i = s;
        if(i == s)
                cout << "equal" << endl;
        else
                cout << "not equal" << endl;
        return 0;
}
```

Conversions – gains and losses

```
#include  <iostream>
using namespace std;
int main(void) {
      int i = 2147483647;
short s = i;

    if(i == s)
     cout << "equal" << endl;
    else
        cout << "not equal" << endl;
    return 0;
}
```

Conversions – gains and losses

```
#include  <iostream>
using namespace std;
int main(void) {
        float f = 1234.5678;
        double d = f;

        if(d == f)
                cout << "equal" << endl;
        else
                cout << "not equal" << endl;
        return 0;
}
```

Conversions – gains and losses

```
#include  <iostream>
using namespace std;
int main(void) {
        double d = 123456.789012;
        float f = d;

        if(d == f)
                cout << "equal" << endl;
        else
                cout << "not equal" << endl;
        return 0;
}
```

Conversions – gains and losses

```
#include <iostream>
using namespace std;
int main(void) {
        float f = 123.456;
        float g = 1e100;
        int i = f;
        int j = g;

        cout << i << endl;
        cout << j << endl;
        return 0;
}
```

Promotions

- **A promotion involves the conversion of data taking part in an evaluation to the safest type**.
- Formally, all the promotions are conducted according to the following set of rules:
  - data of type *char* or *short int* will be converted to type *int* (this is called **an integer promotion**);
  - data of type *float* undergoes a conversion to type *double* (**floating point promotion**);
  - if there's any value of type *double* in the expression, the other data will be converted to a *double*;
  - if there's any value of type *long int* in the expression, the other data will be converted to *long int*;

Promotions

```
#include <iostream>
using namespace std;
int main(void) {
        int Int = 2;
        char Char = 3;
        short Short = 4;
        float Float = 5.6;

        Int = Short + Char + Float;
        cout << Int << endl;
        return 0;
}
```

Promotions

- We can predict that the following implicit conversions will take place:
  - promotions go first, resulting in the following conversions:
    - **int**(Short) + **int**(Char)
  - the sum of *Short* and *Char* as well as the *Float* variable will be converted to *double*, that is:
    - **double**((**int**(Short) + **int**(Char)) + **double**(Float))
  - the final sum will be calculated as a *double*, and then conversion into an *int* type takes place:
    - **int**(**double**((**int**(Short) + **int**(Char)) + **double**(Float)))

What is a string?

- A **string** (in the C++ language sense) is **a set of characters**

```
#include <string>

string PetName;
```

Initializing a string

- **A string can be initialized** in a way that is identical to the one used for other regular types

```
string PetName = "Lassie";
```

- There's also another way to initialize string variables, more suited to the style of object programming.

```
string PetName("Lassie");
```

Initializing a string

- Both forms (assigning and functional) are permissable.

```
string IsHome = PetName;
string HasReturned(PetName);
```

String operators: +

- **The string type has its own operators**.
- One of the most important and most frequently used is the **+** operator

```
#include  <iostream>
#include  <string>

using namespace std;

int main(void) {
      string TheGood = "Jekyll", TheBad = "Hyde";
      cout << TheGood + " & " + TheBad << endl;
      cout << TheBad + " & " + TheGood << endl;
      return 0;
}
```

String operators: +

- he + (concatenation) operator has one important limitation. **It cannot concatenate literals**.

```
#include  <iostream>
#include  <string>

using namespace std;

int main(void) {
      string String;
      String = "A" + "B";
      String = String + "C";
      String = "B" + String;
      cout << String << endl;
      return 0;
}
```

String operators: +=

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
        string TheQuestion = "To be ";
        TheQuestion += "or not to be";
        cout << TheQuestion << endl;
        return 0;
}
```

Inputting strings

- The *cin* stream treats spaces (to be precise, not only regular spaces but also all so-called **white characters**) as delimiters

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string LineOfTypes;
	cin >> LineOfTypes;
	cout << LineOfTypes << endl;
	return 0;
}
```

Inputting strings

- If you want **to input a whole line of text** and treat the white characters just like any other character, you have to use the ***getline*** function.

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string LineOfTypes;
	getline(cin,LineOfTypes);
	cout << LineOfTypes << endl;
	return 0;
}
```

Comparing strings

```
#include  <iostream>
#include  <string>

using namespace std;

int main(void) {
    string secret = "abracadabra";
    string password;
    cout << "Enter password:" << endl;
     getline(cin,password);
     if(secret == password)
          cout << "Access granted" << endl;
     else
          cout << "Sorry";
    return 0;
}
```

Comparing strings

- All the operators designed to compare : > < >= <= !=.
- You can check if one of the strings is greater/lesser than the other, but remember that these comparisons are carried out in alphabetical order where:
  - 'a' is greater than 'A' (sic)
  - 'z' is greater than 'a',
  - but, 'a' is greater than '1'.

Comparing strings

```
#include  <iostream>
#include  <string>

using namespace std;

int main(void) {
    string str1, str2;
    cout << "Enter 2 lines of text:" << endl;
     getline(cin,str1);
     getline(cin,str2);
     cout << "You've entered:'" << endl;
     if(str1 == str2)
          cout << "\"" << str1 << "\" == \"" << str2 << "\"" << endl;
     else if(str1 > str2)
          cout << "\"" << str1 << "\" > \"" << str2 << "\"" << endl;
     else
          cout << "\"" << str2 << "\" > \"" << str1 << "\"" << endl;
    return 0;
}
```

Comparing strings

- If we want a particular method (member function) to process data embedded within an object, we **activate the member function for the object**. It looks like this:
  - object.member_function();

Comparing strings

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
    string secret = "abracadabra";
    string password;
    cout << "Enter password:" << endl;
     getline(cin,password);
     if(secret.compare(password) == 0)
          cout << "Access granted" << endl;
     else
          cout << "Sorry";
    return 0;
}
```

- password.compare(secret)

Comparing strings

- The function can also diagnose all of the possible relations between two strings. Here's how it works:
  - str1.compare(str2) == 0 when str1 == str2
  - str1.compare(str2) > 0 when str1 > str2
  - str1.compare(str2) < 0 when str1 < str2

Comparing strings

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
    string str1, str2;
    cout << "Enter 2 lines of text:" << endl;
     getline(cin,str1);
     getline(cin,str2);
     cout << "You've entered:'" << endl;
     if(str1.compare(str2) == 0)
          cout << "\"" << str1 << "\" == \"" << str2 << "\"" << endl;
     else if(str1.compare(str2) > 0)
          cout << "\"" << str1 << "\" > \"" << str2 << "\"" << endl;
     else
          cout << "\"" << str2 << "\" < \"" << str1 << "\"" << endl;
    return 0;
}
```

Substrings

- he strings allow themselves to be processed in a more precise way when only selected parts of them are taken into consideration.
- A part of a string is called **a substring**.
- If we want to create a new string consisting of characters taken from another (or even the same) string's substring, we can use a member function called ***substr***, and its simplified, informal prototype looks like this:
  - newstr = oldstr.substr(substring_start_position, length_of_substring)

Substrings

- Both parameters have default values. This enables us to use the function in a more flexible way. So:
  - *s.substr(1,2)* describes a substring of the s string, starting at its second character and ending at its third character (inclusively)
  - *s.substr(1)* describes a substring starting at the second character of the *s* string and containing all of the remaining characters of *s*, including the last one; the omitted *length_of_substring* parameter defaults to covering all the remaining characters in the *s* string
  - *s.substr()* is just a copy of the whole s string (the *substring_start_position* parameters defaults to 0)

Substrings

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string str1, str2;
	str1 = "ABCDEF";
	str2 = str1.substr(1,1) + str1.substr(4) + str1.substr();
	cout << str2 << endl;
	return 0;
}
```

The length of a string

- The size of string is provided by two twin member functions. Their names are different, but their behaviours are identical. We can say that these functions are **synonyms**.
- Their informal prototypes look like these:
  - int string_size = S.size();
  - int string_length = S.length();

The length of a string

- Try to predict its output.

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string str = "12345";
	int pos = 1;
	cout << str.substr(pos).substr(pos).substr(pos).size() << endl;
	return 0;
}
```

More detailed string comparison

- S.compare(substr_start, substr_length, other_string)
- S.compare(substr_start, substr_length, other_string, other_substr_start, other_substr_length)

- string S = "ABC";
- cout << S.compare(1,1,"B");

- string S = "ABC";
- cout << S.compare(1,1,"ABC",1,1);

More detailed string comparison

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string S = "ABC";
	cout << S.compare(1,1,"BC") + S.compare(2,1,S,2,2) << endl;
	return 0;
}
```

Finding strings inside strings

- Strings can search for a substring or for a single character. For this purpose, we need to use one of the variants of the *find* member function. Two of them are particularly useful:
  - int where_it_begins = S.find(another_string, start_here);
  - int where_it_is = S.find(any_character, start_here);
- If the search fails, both functions return a special value denoted as **string::npos**

Finding strings inside strings

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string greeting = "My name is Bond, James Bond.";
	string we_need_him = "James";
	if(greeting.find(we_need_him) != string::npos)
		cout << "OMG! He's here!" << endl;
	else
		cout << "It's not him." << endl;
	int comma = greeting.find(',');
	if(comma != string::npos)
		cout << "Curious. He used a comma." << endl;
	return 0;
}
```

How big is the string actually?

- For example, you can ask any string for the size of the currently allocated buffers. The answer comes from the member function called *capacity*.
  - **int** currently_used = S.capacity();
- Every string can grow, but there's a limit to its extension - a value defined for all the strings in the implementation. You can find it out by using function called *max_size* use it:
  - **int** not_more_than = S.max_size();

How big is the string actually?

```
#include <iostream>
#include <string>

using namespace std;

void printInfo(string &s) {
	cout << "length = " << s.length() << endl;
	cout << "capacity = " << s.capacity() << endl;
	cout << "max size = " << s.max_size() << endl;
	cout << "---------" << endl;
}

int main(void) {
	string TheString = "content";
	printInfo(TheString);
	for(int i = 0; i < 10; i++)
		TheString += TheString;
	printInfo(TheString);
	return 0;
}
```

How to control the content of the string

- We can empty the string, **completely removing all the characters currently stored inside it**.
  - Emptying the string is done by the member function called *clear*.
- Changing the size of the string is carried out by the member function called *resize*.
  - You can specify a character to be used to fill the newly allocated space
- You can also check if a particular string is **empty**

How to control the content of the string

```
#include <iostream>
#include <string>

using namespace std;

void PrintInfo(string &s) {
	cout << "content =\"" << s << "\" ";
	cout << "capacity = " << s.capacity() << endl;
	cout << "is empty? " << (s.empty() ? "yes" : "no") << endl;
	cout << "---------" << endl;
}

int main(void) {
	string TheString = "content";
	PrintInfo(TheString);
	TheString.resize(50,'?');
	PrintInfo(TheString);
	TheString.resize(4);
	PrintInfo(TheString);
	TheString.clear();
	PrintInfo(TheString);

	return 0;
}
```

How to control the content of the string

- Strings are able to present their content **as if it were an actual array**.

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string TheString = "content";
	for(int i = 0; i < TheString.length(); i++)
		TheString[i] = TheString[i]  - 'a' + 'A';
	cout << TheString << endl;
	return 0;
}
```

Appending a (sub)string

- Function *append*. It's designed to **append one string to another**
  - string str1 = "content"; str2 = "appendix"; str1.append(str2);
  - // str1 contains "contentappendix" now
- *append* function is able to append not only a string, but also a substring of the string, like this:
  - string str1 = "content"; str2 = "appendix";
  - str1.append(str2,0,3);
  - // str1 contains "contentapp" now

Appending a (sub)string

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string TheString = "content";
	string NewString;
	NewString.append(TheString);
	NewString.append(TheString,0,3);
	NewString.append(2,'!');
	cout << NewString << endl;
	return 0;
}
```

Appending a character

- If you want to append just one character to a string, you can do it by using the *append* function, but there's a more efficient way, by using the *push_back* member function.

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string TheString;
	for(char c = 'A'; c <= 'Z'; c++)
		TheString.push_back(c);
	cout << TheString << endl;
	return 0;
}
```

Inserting a (sub)string or a character

- Inserting a string into a string is like distending its contents from within.
  - string quote = "to be "; quote.append(quote); quote.insert(6, "or not ");
  - cout << quote << endl;

Inserting a (sub)string or a character

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string quote = "Whyserious?", anyword = "monsoon";
	quote.insert(3,2,' ').insert(4,anyword,3,2);
	cout << quote << endl;
	return 0;
}
```

Assigning a (sub)string or a character

- The *assign* member function does a job which is very similar to the insert's job, but **does not retain** the previous string content, and instead just **replaces it with a new one**.

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
        string sky;
        sky.assign(80,'*');
        cout << sky << endl;
        return 0;
}
```

Replacing a (sub)string

- The *replace* member function is more subtle. It can **replace a part of the string with another string or another string's substring**.

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string ToDo = "I'll think about that in one hour";
	string Schedule = "today yesterday tomorrow";

	ToDo.replace(22, 12, Schedule, 16, 8);
	cout << ToDo << endl;
	return 0;
}
```

Erasing a (sub)string

- We can also **remove a part of a string**, making the string shorter than before.
  - TheString.erase();
    - erases all the characters from the string and leaves it empty.

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
    string WhereAreWe = "I've got a feeling we're not in Kansas anymore";

    WhereAreWe.erase(38, 8).erase(25, 4);
    cout << WhereAreWe << endl;
    return 0;
}
```

Exchanging the contents of two strings

```
#include <iostream>
#include <string>

using namespace std;

int main(void) {
	string Drink = "A martini";
	string Needs = "Shaken, not stirred";

	cout << Drink << ". " << Needs << "." << endl;
	Drink.swap(Needs);
	cout << Drink << ". " << Needs << "." << endl;
	return 0;
}
```

Name spaces are all around us

- **The name space is a space in which a particular name has an unambiguous and clear meaning**.
  - home_name_space :: entity_name
  - **using namespace** std;

```
#include <iostream>

int main(void) {
        cout << "Play it, Sam" << endl;
        return 0;
}
```

Introducing the namespace

- We've qualified all the ambiguous names with a prefix consisting of the home name space (*std*) and a special operator written as "**::**".
- The operator's official name is "**scope resolution operator**".

```
#include <iostream>

int main(void) {
    std::cout << "Play As time goes by" << std::endl;
    return 0;
}
```

Defining a name space

- Defining a name space looks like this:
  - **namespace** the_name_of_the_space {
  - }

```
#include <iostream>

using namespace std;

namespace Hogwarts {
	int Troll = 1;
}

namespace Mordor {
	int Troll = 2 ;
}

int main(void) {
	cout << Hogwarts::Troll << " " << Mordor::Troll << endl;
	return 0;
}
```

Defining a name space

- Note that the using *namespace* statements must not lead to a situation where an identifier could be considered to have originated from more than one name space.

```
#include <iostream>

using namespace std;

namespace Hogwarts {
    int Troll = 1;
}

namespace Mordor {
    int Troll = 2 ;
}

using namespace Hogwarts;

int main(void) {
    cout << Troll << " " << Mordor::Troll << endl;
    return 0;
}
```

Defining a name space

- If the *using namespace* statement is **placed inside a block**, its scope ends in the same place where the block ends.

```
#include <iostream>
using namespace std;
namespace Hogwarts {
    int Troll = 1;
}
namespace Mordor {
    int Troll = 2 ;
}
int main(void) {
    {
        using namespace Hogwarts;
        cout << Troll << " " ;
    }
    {
        using namespace Mordor;
        cout << Troll << endl;
    }
    return 0;
}
```

Expanding a name space

- Note that the first appearance of a name space is called “**an original name space**”. Any name space with the same name (identifier) that appears after the original name space is called “**an extension name space**”.

Expanding a name space

```
#include <iostream>
using namespace std;
namespace Hogwarts {
	int Troll = 1;
}
namespace Mordor {
	int Troll = 2 ;
}
namespace Hogwarts {
	float Wizard = -0.5;
}
namespace Mordor {
	float Wizard = 0.5;
}
int main(void) {
	cout << Hogwarts::Troll << " " << Hogwarts::Wizard << endl;
	cout << Mordor::Troll << " " << Mordor::Wizard << endl;
	return 0;
}
```

Using an entity

- The statement that allows us to **selectively decide which entities should be used** and which should remain hidden inside the space.

```
#include <iostream>
using namespace std;
namespace Hogwarts {
	int Troll = 1;
	float Wizard = -0.5;
}
namespace Mordor {
	int Troll = 2 ;
	float Wizard = 0.5;
}
using Mordor::Troll;
using Hogwarts::Wizard;
int main(void) {
	cout << Hogwarts::Troll << " " << Wizard << endl;
	cout << Troll << " " << Mordor::Wizard << endl;
	return 0;
}
```

An unnamed name space

- We may define a name space without a name (**an anonymous namespace**).
- This kind of namespace is **implicitly and automatically used** in a source file where its definition is visible.

An unnamed name space

```
#include <iostream>
using namespace std;
namespace {
        int Troll = 1;
        float Wizard = -0.5;
}
namespace Mordor {
        int Troll = 2 ;
        float Wizard = 0.5;
}
int main(void) {
        cout << Troll << " " << Wizard << endl;
        cout << Mordor::Troll << " " << Mordor::Wizard << endl;
        return 0;
}
```

Renaming a name space

- **namespace** new_name = old_name;

- The new name of the name space may be used together with the old one.

```
#include <iostream>
using namespace std;
namespace What_A_Wonderful_Place_For_A_Young_Sorcerer {
      int Troll = 1;
      float Wizard = -0.5;
}
namespace Mordor {
      int Troll = 2 ;
      float Wizard = 0.5;
}
namespace Hogwarts = What_A_Wonderful_Place_For_A_Young_Sorcerer;
int main(void) {
      cout << Hogwarts::Troll << " " <<
      What_A_Wonderful_Place_For_A_Young_Sorcerer::Wizard << endl;
      cout << Mordor::Troll << " " << Mordor::Wizard << endl;
      return 0;
}
```