CSCI251: Advanced Programming

Lecturer: Dr. Shixun Huang

struct, union, Randomness and File Handle

Spring 2023

Abstract data types

- It’s quite common to have related elements of data, particularly when we are modelling some non-trivial entity.
  - cat, for example, have a lot of characteristics, not just a name.
  - An individual cat might be described by a tail length, and colouring.
  - we might use (for one), but how about many?

```
string name;
float tailLength;
string colouring;
```

Abstract data types

- We could use arrays …

```
const int numberCats = 5;
string name[numberCats];
float mass[numberCats];
float tailLength[numberCats];
string colouring[numberCats];
```

- So, we define our own abstract data types, that typically contain multiple related data elements.

Abstract data types …

- We will typically use classes to provide this encapsulation
- there are a couple of related pre-class entities:
  - The `struct`:
  - The `union`:

The `struct` construct

- A C++ struct is a way to group related data elements
  - A struct access specifiers default to public whereas in classes they default to private.
  - Use structs if all the members are public.
  - Everything being public breaches encapsulation

- A struct declaration ends with a semi-colon (;).

```
struct Cat {
string name;
float tailLength;
string colouring;
};
```

```
struct Student {
    string name;
    int id;
};
```

- The structure name `Student` is a new type, so you can declare variables of that type, for example:

```
Student s1, s2;
```

- To access the individual fields of a structure, we use the dot operator:

```
s1.id = 123;
s2.id = s1.id + 1;
s1 = s2;    // copies fields of s2 to s1
```

Practice 0

- It’s possible to have instances of structs inside structs…

```
struct Address {
   string city;
   int postCode;
};

struct Student {
   string name;
   int id;
   Address addr;
};
```

- To access nested structure fields use more dots…

```
Student s;
s.addr.postCode = 2500;
```

Can structs have member functions?

- Yes, structs can have functions too.
  - Remember they differ from classes only in the default access specifiers.

- people don't have to use the interface (member functions) as the data is public by default

- So, use classes if we want to control the interface

```
struct Test {
   string name;
   int number;

   void setTest(string, int);
   void showTest();
};

void Test::setTest(string TestName, int TestNumber) {
   name = TestName;
   number = TestNumber;
}


void Test::showTest() {
  cout<<"Test string " << name << endl;
  cout<<"Number for this  "<< number << endl;
}
```

```
int main()
{
     Test myTest;
     myTest.setTest("Bob", 19);
     myTest.showTest();
}
```

```
int main()
{
     Test myTest;
     myTest.name="Bobby";
     myTest.number=15;
     myTest.showTest();
}
```

UNIVERSITY OF WOLLONGONG AUSTRALIA

Static consts in structs …

- If you have a static const
  - declare and initialise it in a struct

```
struct Trial {
 static const int trial = 11;
};
```

  - Or initialise it outside…

```
struct Trial {
 static const int trial;
};
const int Trial::trial = 11;
```

Trial t1;
cout<<Trial::trial<<" and "<<t1.trial<<endl;

Practice 1

Another type of type: The `union`

- the fields of a `union` all share the same memory.

```
union mytype {
    int i;
    float f;
};
```

```
union var{
char c[4];
int i;
};
```

- So assigning values to fields: 'i' or 'f' would write to the same memory location.
- use a union when ONE variable has values of different types, but not simultaneously.

Practice 2

Costing a construct …

- Structs and unions : their sizes
- A struct is just the concatenation of the data members, not so with the union.

```
struct adt {
    int i;
    float f;
};
```

```
union mytype {
    int i;
    float f;
};
```

```
cout << sizeof(int) << " " << sizeof(float) << endl;
cout << sizeof(adt) << endl;
cout << sizeof(mytype) << endl;
```

```
4  4
8
4
```

Randomness

Randomness is useful

What is a probability distribution?

A probability distribution is a statistical function that describes all the possible values and likelihoods that a random variable can take within a given range.

It is a mathematical description of a random phenomenon in terms of its sample space and the probabilities of events (subsets of the sample space).

Examples of probability distributions

Uniform - parameterised by min and max

Normal - parameterised by mean (μ) and standard deviation (σ)

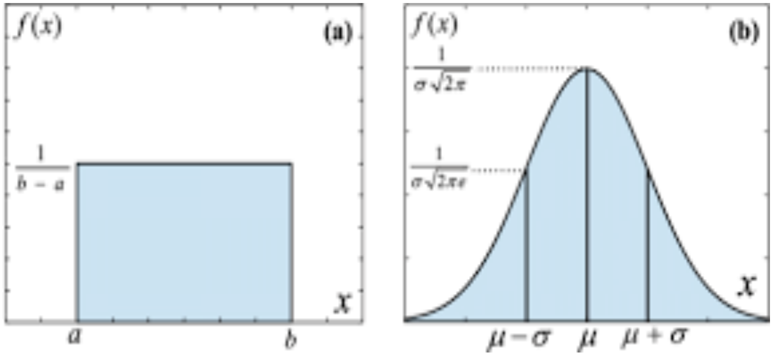


Figure: Uniform and Normal probability distributions

Randomness is useful

You must: `#include <random>` in order to use the random generation facilities in C++

Two more essential “ingredients”:

1. Random engine (i.e. source of randomness)
2. Probability distribution for the problem at hand

For most simple tasks `default_random_engine` provides sufficient randomness

```
default_random_engine defEng;
```

Random is truly **pseudorandom**; there is a pattern but it repeats after a very long time

To obtain repeatable experiment we seed the engine

```
default_random_engine defEng(seed);
```

To make it start at different points we seed with computer time

```
default_random_engine defEng(time(0));
```

Example Code

```
#include <iostream>
#include <random>
#include <ctime>
int main(){
std::default_random_engine rndEng(time(0));// randomness
std::uniform_int_distribution<> uniform1(0,9);//uniform variate
std::uniform_real_distribution<> uniform2(0,1);// uniform variate
std::normal_distribution<> normal1(2.5, 0.6);// normal variate
std::normal_distribution<double> normal2(7.0, 1.2);//normal variate
for (std::size_t sample = 0; sample < 5; ++sample){
  std::cout << uniform1(rndEng)<< " :: "
  << uniform2(rndEng) << " :: "
  << normal1(rndEng) << " :: "
  << lround(normal2(rndEng)) //integer normal variate
  <<std::endl;
}
return 0;
}
```

Example output from code

```
4   ::  0.353855        ::    2.17768    ::  9
1   ::  0.100106        ::    2.53111    ::  4
3   ::  0.611911        ::    2.27688    ::  11
9   ::  0.758766        ::    3.23352    ::  5
5   ::  0.0855263       ::    2.26473    ::  6
```

Other engines and distributions

```
//Engines
// 64-bit unsigned Merseme twister generator
std::mt19937_64 mtEng64;
// 32-bit unsigned Merseme twister generator
std::mt19937     mtEng32;

// Distributions
//Floating point valued lambda; lam defaults to 1.0
std::exponential_distribution<double> e(1.0);

// gamma distribution with alpha shape (a) and beta scale (b);
// both defaults to 1.0
std::gamma_distribution<double> g(a,b);
```

Further reading:

C++ Primer, 5th Ed., Stanley B. Lippman, Josée and Barbara E. Moo, page 882

The C++ Standard Library, A Tutorial and Reference 2nd Ed., Nicolai M. Josuttis, Chapter 17, page 907.

Practice 3

Sampling from given discrete distribution

- Given a discrete distribution governing an event (e.g. weather)

```
{ "Sunny": 0.4, "Rainy": 0.2, "Windy": 0.1, "Cloudy":0.3 }
```

- Notice that the probabilities sum to one (1)
- Suppose we want to sample (draw) events from this distribution; For example what will the weather be over the next 5 days?
- We want something like:

  Day 1 - Rainy
  Day 2 - Rainy
  Day 3 - Cloudy
  Day 4 - Windy
  Day 5 - Sunny

- Form the cumulative distribution from the discrete distribution

```
{0 – 0.4; 0.4 – 0.6; 0.6 – 0.7; 0.7 – 1.0}
```

Notice how it ended in 1.0

- We now use the uniform distribution to generate random variates that tells us which event happened at each sample

Sampling from given discrete distribution

Given a discrete distribution,

`{ "Sunny": 0.4, "Rainy": 0.2, "Windy": 0.1, "Cloudy":0.3 }`

Generate the cumulative distribution

`{0 – 0.4; 0.4 – 0.6; 0.6 – 0.7; 0.7 – 1.0}`

```
0.0 – 0.4  => Sunny
0.4 – 0.6  => Rainy
0.6 – 0.7  => Windy
0.7 – 1.0  => Cloudy
```

Generate uniform random variate between 0.0 and 1.0 using

```
std::uniform_real_distribution<> unifr1(0.0,1.0);
```

Map the outcome of the uniform distribution to the intervals of the cumulative distribution

Form example an outcome of 0.65 maps to `Windy`; 0.25 maps to `Sunny`, 0.4 maps to `Rainy`, etc.

how about a different order?

Handling Files

Outline

- Text file streams.
  - Errors in opening files.
  - Errors in reading files.
- Character input.
- Buffering.
- Binary I/O.

Text File Streams

Note that files are viewed here as sequences of bytes with an end-of-file character at the end.

```
#include <fstream>
using namespace std;
#include <iostream>
#include <sstream>

    ifstream inData;                    //declare an input file
    ofstream outData;                   //declare an output file
    string firstName, lastName;

    inData.open("names.txt");          // open input file
    outData.open("marks.txt");           // open output file
    inData >> firstName >> lastName;   // read from a file stream
    outData << 85.6;                     // write into a file stream
    inData.close();                     // close the input file
    outData.close();                    // close the output file
    return 0;
```

Unbounded file streams …

- On the previous slide we opened an input stream, and we opened an output stream.
- We could have used `fstream`, an unbounded file stream allowing both reading and writing.

```
fstream fstrm;
fstrm.open(filename, mode);
```

- Modes, for this and the i-o versions, partially:

```
in, out, app, ate, trunc, binary
```

- They are part of `fstream::`
- So: std::fstream file(filename, std::fstream::in);

Constant	Explanation
`app`	seek to the end of stream before each write
`binary`	open in binary mode
`in`	open for reading
`out`	open for writing
`trunc`	discard the contents of the stream when opening
`ate`	seek to the end of stream immediately after open

There are constraints on these flags.

- Some examples:
  - `out` is for `fstream` or `ofstream`.
  - `trunc` can only be set if `out` is.
  - `app` and `trunc` are mutually exclusive.

Errors in opening files …

- Don't assume a file stream has been opened successfully.
  - Incorrect file name:

```
inFile.open("names.txl");
```

  - Incorrect file opening mode:

```
ifstream inFile;
inFile.open("names.txt", ios::trunc);
```

  - Not enough room on the hard drive.
  - Hardware failure.
- Always check the status of a stream after `open`.

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

int main()
{
    ifstream inData;   // declare an input file stream
    char fileName[] = "exams.txt";
    string lastName, mark;

    inData.open( fileName );     // open input file
    if (!inData)               // check if opened successfully
    {
      cerr << "Error opening : " << filename << endl;
      return -1;       // exit with an error code
    }
    inData >> lastName >> mark;

    inData.close();   // Close the input file
    return 0;
}
```

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Errors in reading files …

- So the file seemed to open okay but …
- … being pessimistic, what goes wrong next.
  - The program may not have data to read as it hits the end of file.
  - The data may be invalid: … an alphabetic character instead of a digit character; a control character instead of an alphabetic one; etc.
  - The data may not be physically accessed from the disk due to its damage or network failure.

- Comprehensive error checking is needed, and appropriate error recovery.
  - C++ provides status flags and functions to detect possible errors.

1. The flag `eof` indicates that the end of file is reached.

```
if( inData.eof() )  {  Error recovery action }
```

2. The flag `fail` indicates a failure due to invalid data.

```
if( inData.fail() ) {  Error recovery action }
```

3. The flag `bad` indicates a hardware problem.

```
if( inData.bad() ) {  Error recovery action }
```

4. The function `good()` returns true if no any error has been detected.

Example

- A text file has content:

```
1 2 3
```

- Assuming appropriate headers etc ...

```
while( ! inData.eof() )
  {
       inData >> number;
       cout << number << " ";
  }
```

- Produces output ... ?

Practice 4

Error recovery …

- Once the stream is in the error state, it will stay that way until you take specific action:
  - All subsequent operations will do nothing, or loop forever no matter what they are or what is in the input.
  - You have to clear the stream by calling `clear()` to recover the stream from the fail state.

```
inFile >> newNumber;
if( inFile.fail() )
{
    inFile.clear();
    inFile.ignore(100, '\n');
}
```

Recovers the file stream from the error state. However, further reading of data may be useless as the wrong characters are still in the stream buffer.

Discard 100 characters (or until the end-of-line indicator) from the stream buffer.

Example (be careful about the format)

```
string nameFirst;
string nameLast;

  do inFile >> nameFirst >> nameLast;
  while( inFile.good() );
  cout<<nameFirst<<"+++"<< nameLast<<"+++";

  if(inFile.fail())  return -1;
  if(inFile.bad())   return -2;
  if(inFile.eof()) return 0;
```

A text file has content:

```
 1  2  3
```

Practice 5

Character input …

- How do we read **all** characters from the input stream; including blanks, tabs, and new-lines?
  - The extraction operator (>>) doesn't read white space or characters.
- You can use `get` functions to read a character:

```
ifstream inFile;
char nextChar;
nextChar = inFile.get();
```

- You can use `getline` to read a line of characters from a text file.

```
char lineBuffer[bufSize];
infile.getline( lineBuffer, bufSize );
```

- Be careful.

test.txt

```
1.0
Motor Oil
```

```
float price;
char productName[20] ;
char fileName[] = "test.txt";
ifstream inData;

inData.open( fileName );

inData >> price;

inData.getline(productName, 20);

cout << price << endl;
cout << productName << endl;
```

what is the output?

Practice 6

UNIVERSITY OF WOLLONGONG AUSTRALIA

- So, we need to clear the buffer …

```
inData >> price;
inData.ignore( 20, '\n' );
inData.getline(productName, 20);
```

You can also try to use

```
float price;
price = inData.get();
```

But you will get an unpredicted input. Check ASCII Chart.

A different form and a delimiter

- It's often unreasonable to forecast the input length.

- A more general form:

```
    istream& getline ( istream &is , string
&str , char delim );
   //get letters from istream, and save into
str, until met delim
```

- This has a default endpoint of \n, that's a new line.
- Example:
  - `#include <iostream>`
  - `string something;`
  - `getline(inData, something );`

Common Escape Sequences

	Escape Sequence	Description
\n	Newline	Cursor moves to the beginning of the next line
\t	Tab	Cursor moves to the next tab stop
\b	Backspace	Cursor moves one space to the left
\r	Return	Cursor moves to the beginning of the current line (not the next line)
\\	Backslash	Backslash is printed
\'	Single quotation	Single quotation mark is printed
\"	Double quotation	Double quotation mark is printed

Character output
Output formatting

- Use `put()` in a similar way to `get()`.

- You can use output manipulators if you want to format your output: #include <iomanip>

```
Cout << setw(10);
cout << setfill('-');
cout << setprecision(2);
```

See example

Practice 7

Buffering

- When you direct data into a file, it is not sent there immediately.
  - It is physically written to the device only when the buffer is full.
- When you read data from a file, you input it from the input buffer.
  - When the buffer becomes empty it is refilled with a new block of data.
- **Why buffer?**
  - It reduces the number of accesses to the external device.
- Only `cerr`, standard error, is never buffered.

std::cerr

- an object of class ostream
- C stream stderr or standard error stream
- Example

```
if (!inData)     cerr << "Error opening : " << filename << endl;
```

- To do buffering:
  - cout: is buffered. (for example, wait until the new line)
  - cerr: is not buffered. (for example, directly printing)
  - redirects output to a file, cerr still prints on the screen

Text and Binary Files

- Text files:
  - composed of characters
  - data types have to be formatted/converted into a sequence of characters
  - usually sequential access

- Binary files
  - binary numbers (bits)
  - usually random access

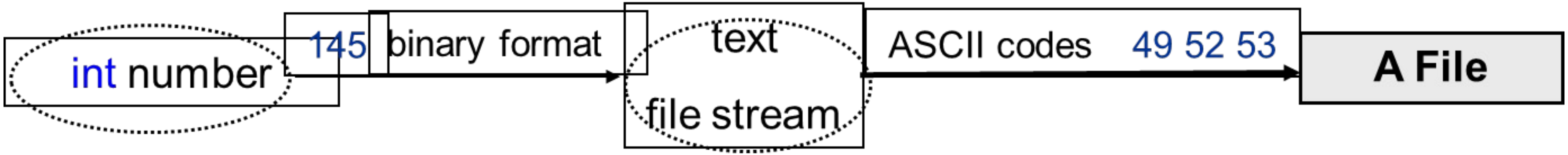
Text Files Input/Output

- Text File Input/Output functions also carry out data type conversion:

Example:

```
int number;
outFile << number ;
```

The **<<** operator converts `int` into a sequence of ASCII codes and writes them into a file.

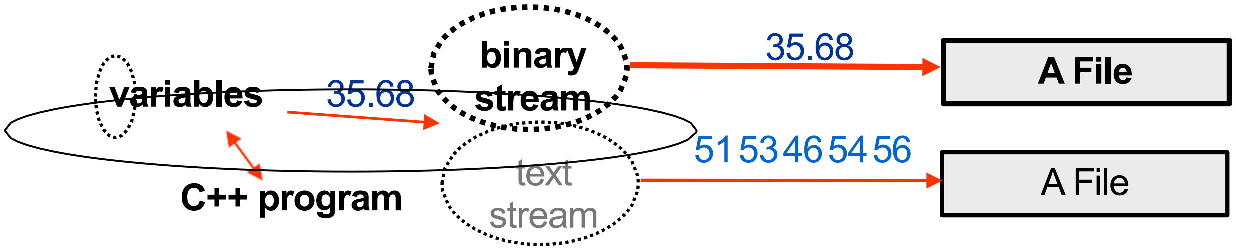



```
inFile >> number;
```

The **>>** operator converts a sequence of ASCII characters into an integer number and stores it into a variable.

Binary File Stream Concept

- A Binary File Stream is an interface between a program and a physical file that **does not perform any type conversion**.



- File Streams can be Text or Binary, but physical files do not have any special marker to indicate their type.
- A file name or its extension does not affect the file type.