CSCI251/CSCI851 Autumn-2020
Advanced Programming (**S3a**)

Getting organised I:

Structs, unions, and randomness

Outline

- Abstract data types …
  - Structs.
  - Unions.
- Randomness…

Abstract data types

- It's quite common to have related elements of data, particularly when we are modelling some non-trivial entity.
  - Humans, for example, have a lot of characteristics, not just a name.
- Once we start dealing with multiple instances of something, we want to be able to both distinguish between the characteristics, and distinguish between the same characteristic for different instances.

- For example, let's say that we are writing a program related to a cat.
  - An individual cat might be described by a name, mass, tail length, and colouring.
- If there is only one cat we might reasonably and unambiguously use

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

- But what happens when we have lots of cats ...

- We could use arrays …

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

- … but it's tidier to have some way of connecting the data elements other than simply through the index.
  - If there is ever any reason to change an index value, sorting cats for example, we need to make sure the indices line up.
- 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, but there are a couple of related pre-class entities:
  - The `struct`:
    - This is a special kind of class so it's not overly exciting.
  - The `union`:
    - This is special kind of class too, but a more limited one.

The `struct` construct

- A C++ struct is a way to group related data elements together, in a similar way to a C++ or Java class.
  - **Aside for now**: A struct differs only from a class in that the access specifiers default to public whereas in classes they default to private.
  - Typical programming style has structs only being used in C++, in place of classes, if all the members are going to be public.
  - Everything being public breaches encapsulation so there should be a good reason for doing this…
- A struct declaration ends with a semi-colon (;).

```
struct Cat {
   string name;
   float mass;
   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
```

- 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?

- Linking data elements seems sensible but if we think about functions that only act on data within a struct, having linked functions seems reasonable too.
  - And sure, structs can have functions too.
    - Remember they differ from classes only in the default access specifiers.
- Structs are often used to declare simple datatypes, but don't have to be.

- The problem with putting member functions in Structs is that unless you remember to add in access specifiers people don’t have to use the interface you provide, the data is public by default ☹
- So, we would expect to use classes when we want to control the way people interact with the data.

```
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();
}
```

Static consts in structs …

- If you have a static const you can only declare and initialise it in a struct, or class, if it's of integral or enumeration type.
  - Or if it's initialised by a constant expression.
- Otherwise you have to initialise it outside…

```
struct Trial {
   static const double trial;
};
const double Trial::trial = 11.73;
```

Static consts in structs …

- From Stroustrup: http://www.stroustrup.com/bs_faq2.html
- “So why do these inconvenient restrictions exist? A class is typically declared in a header file and a header file is typically included into many translation units. However, to avoid complicated linker rules, C++ requires that every object has a unique definition. That rule would be broken if C++ allowed in-class definition of entities that needed to be stored in memory as objects.”
- The following is okay (constant expression):
- `static constexpr double trial=11.73;`

Another type of type: The `union`

- A union declaration is similar to a `struct` declaration, but the fields of a `union` all share the same memory.

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

- So assigning values to fields: ‘i’ or ‘f’ would write to the same memory location.
- You should use a union when you want a variable to be able to have values of different types, but not simultaneously.

Costing a construct …

- Structs and unions are special cases of class, and while we will return to this later, it's useful to note 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: Old school…

- Pre C++11 C++, and C, relied on the use of a C library function `rand`…
  - Uniform distribution in the range 0 to X.
    - With X a system dependent value, but at least 32767…

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

int main()
{
        srand(time(0));

        for ( int i=0; i < 20; i++)
                cout << rand() << endl;

        return 0;
}
```

Seeding the random generator with time.

Randomness? Seeding?

- When talking about generating randomness in programming we are typically meaning generating sequences of numbers using PRNGs: pseudorandom numbers generators.
  - It's pseudorandom because it is likely to look random and have difficult to find patterns, but the patterns will be there, it's deterministic.
  - Same input, same output.
  - Similar input, expect very different sequence.
- Seeding is setting a start point/initial point for our sequence.

- Limitations:
  - You often want a different range.
  - You often don't want an integer.
  - You often don't want a uniform distribution.
    - You want some values to appear more than others.
  - Even with the same seed, you aren't guaranteed to get the same sequences on different platform.
    - This is a problem if you are using generator and seed as a way of storing something like a game level.
  - There are also global state problems that make testing difficult since the state of `rand()` can be modified by calls from anywhere.

So classically …

- You would get integers in the range `rand` gives you and try to map them to whatever you need.
  - This can often be done in a way that gives a non-uniform distribution, even though a uniform distribution may be what you want.
- You could use assertions to test if the randomness matches what you expect, but there can be problems doing that because of calls from elsewhere if you have multiple threads.

Randomness: C++11

- Now you should use the random-number engine, with the library `random` included.

```
$ g++ rand-eng.cpp
```

```
default_random_engine randEng;
for ( int i = 0; i < 10; i++)
    cout << randEng() << endl;
```

- This is default seeded … the output is fixed.

- Before we used the default constructor, but we can use one with an integral seed too.

```
default_random_engine randEng(seed);
```

- Or set the seed later..

```
randEng.seed(seed);
```

- And we can see the ranges simply using …

```
randEng.min()        randEng.max();
```

Distributions and ranges …

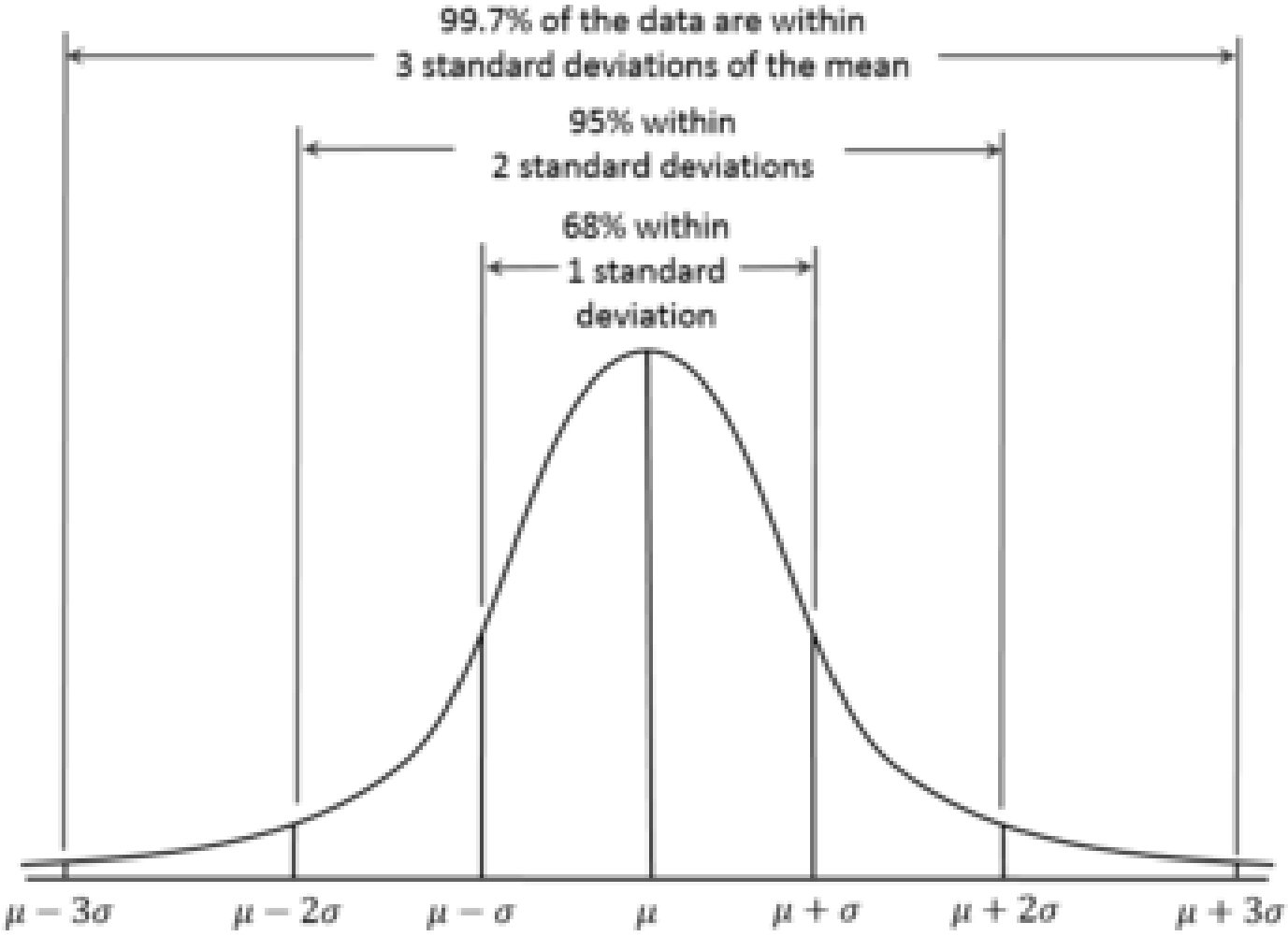
- Uniform between in some range…

```
uniform_int_distribution<unsigned> uniform(0,9);
default_random_engine randEng;
for ( int i = 0; i < 10; i++)
      cout << uniform(randEng) << endl;
```

- There are other distributions you might want to use, such as the normal distribution, where the constructor takes a mean and standard deviation:

```
normal_distribution<> normal(5,1.5);
```

Mean and standard deviation



- Taken from

https://en.wikipedia.org/wiki/Normal_distribution#Standard_normal_distribution

Integers with a normal distribution

- The textbook shows how you can use the `lround` function to round the values obtained to the nearest integer.

```
for ( int i = 0; i < 10; i++)
      cout << lround(normal(randEng)) << endl;
```

Some notes …

- Make sure the `randEng` construction isn't in a loop.
- If you have a local generator function, which sets up the range and distribution type, you should declare both the engine and distribution as being static…

```
static default_random_engine randEng;
static uniform_int_distribution<unsigned> uniform(0,9);
```

  - You may want different ranges …
- Be careful using `time(0)` as a seed.
  - It's predictable.
    - This may be a good thing, but typically if the randomness is to do with security you don't want predictability.
  - It's the same if you are running the same process at (roughly) the same time.

Mersenne Twister: `std::mt19937`

- Period before repeating : $2^{19937}-1$.

```
std::mt19937 mtg;
```

- The `mt19937` is a convenient typedef.

- Range: 0 to 4294967295.
- `mtg.min()` `mtg.max();`
- It's not a cryptographically secure PRNG.

- Here goes an example of a function for returning a random integer in a specified range.

```
int randint (int x) {
	static std::mt19937 mtg(time(0));
	return std::uniform_int_distribution<int> (0, x-1) (mtg);
}
```

- The generator is static and you are returning a value from the appropriate range.

```
for (int i=0;i< 10;i++)
	std::cout << randint(10+10*i) << std::endl;
```