CSCI251/CSCI851 Spring-2018
Advanced Programming (**S3b**)

Getting organised II:

Structs, unions, and randomness

Outline

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

Abstract data types …

- Prior to classes there are a couple of (abstract data type) types that it's useful to introduce:
  - 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 interesting special.
    - We will get structs out of the way first.

Structs

- A C++ structure is a way to group logically related items together, some might say in a similar way to a C++ or Java class.
  - Some might say similarly, but a struct is the same as a class except that the access specifiers default to public.
  - Typical programming style has structs only being used, in place of classes, if all the members are going to be public.
  - Everything being public breaches encapsulation unless it's all interface so there should be a good reason for doing this…
- Struct declaration end with a semi-colon (;).

```
struct StudentType {
    int id;
    bool isGrad;
};
```

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

```
StudentType 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 arbitrarily nested structs…

```
struct AddressType {
  string city;
  int zip;
};

struct StudentType {
  int id;
  bool isGrad;
  Address addr;
};
```

- To access nested structure fields use more dots…

```
StudentType s;
s.addr.zip = 53706;
```

Can structs have member functions?

- Sure, 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.
  - And 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.”

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 class …

- Structs and unions are special cases of class, and while we can 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.

- Limitations:
  - You often want a different range.
  - You often don't want an integer.
  - You often don't want a uniform distribution either.
    - That is, you want some values to appear more than others.
- 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.

Randomness: C++11

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

```
$ CC -std=c++11 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);
```

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);
```

- Be careful using `time(0)` as a seed.
  - It's predictable.
  - It's the same if you are running the same process at roughly the same time.