Introduction to Modern C++

RMIT UNIVERSITY

Lecture Objectives

- Most of the notes for this course were written with C++98 in mind. We are trying to modernise the course so this lecture and a lecture later in the course will look at modern C++.
- In this lecture we will explore some of the features of C++11 and C++14 that have modernised the language far beyond the C++98 standard.

Lecture Objectives (Continued)

- We will be covering the following new features:
  - Auto type deduction
  - Safe pointer types in modern C++
  - Rvalue references and std::move
  - nullptr and nullptr_t
  - Initializer lists and uniform initialization
  - Constexpr

Automatic type deduction

- C++ is well known for its complex type system and long type names.
- For example, this is a valid variable declaration:

```
   std::map<std::string,
std::list<std::list<string>>::iterator it =
map.begin();
```

- This variable declaration is so complex that your compiler may be confused by it. So it’s little wonder that us mere humans have trouble with it!

Automatic type deduction (cont)

- This is not the only reason to have automatic type deduction. Some assignments may not match on type but involve an implicit conversion. That conversion will take extra time.

- Here is an example of automatic type deduction:

```
 auto size = some_vector.size();
```

- An auto variable is not a variant type where we can assign any value to it. Under the hood, the compiler detects the type on the left hand size of the initialization and allocates that type to the variable on the right hand side.

Automatic type deduction (cont)

- You can even have functions that have a return type of auto:

```
auto get_instance(int type)
{
   //logic to decide which instance
   //to return. Note that all return
   //paths in the function but return
   //a variable of the same datatype.
}
```

- Note: you cannot have a function that accepts parameters of the auto type.

Memory management in Modern C++

- In C++, unlike many other object oriented languages, we can create objects on the stack. Say I have a class or struct (it doesn’t matter which) declared in main such as the following:

```
struct foo{
int bar;
public:
Foo() : bar(3){}
}
int main(void){
foo f;
}
```

f is allocated and initialized here and freed when we exit main(). This is not the case with pointers which need to be explicitly managed

Memory management in Modern C++

- You should already know that the generic tool in C and C++ to refer to a memory address is use a pointer.
- The traditional approach to allocating memory in C and C++ is to use malloc() or new.
- These functions allocate a chunk of memory by placing a request with the operating system kernel to return a segment of the size specified.
- There are some problems with this mechanism that the modern c++ header file <memory> is designed to simplify and replace pointers in many circumstances.

Memory management in Modern C++ (Cont)

- Pointers are ambigous because of their definition in the language.

- For example if you have a function that is defined as:

```
 void cleanup(stuff *mystuff);
```

- Is this function meant to take ownership of mystuff? Or is mystuff an alias to memory that is owned by another function? This is particularly important in c++ as each class is meant to manage its own memory.
- Actually from this prototype we don’t even know whether the parameter passed in is an array or not.

Memory management in Modern C++ (Cont)

- The solution to this problem is safe pointers and in particular the unique_ptr class.
- This class abstracts away explicit memory management so that when the unique_ptr is no longer in scope, the memory it points to is freed.
- There is a complication though and that’s the fact that unique_ptr cannot be copied (assigned or passed by value)

Memory management in Modern C++ (Cont)

- unique_ptr can be used to allocate a single object or an array of objects and the syntax for accessing the data held in a unique_ptr is consistent with how you use a pointer.

- For example:

```
std::unique_ptr<stuff> mystuff =
    std::make_unique<stuff>();
```

- The above creates a unique_ptr that points to a single element of type stuff which is initialized by the default constructor (whatever is in the brackets are the arguments to the constructor).

Memory management in Modern C++ (Cont)

- Retrieve the underlying pointer to pass around to another function:

```
mystuff.get()
```

- Retrieve the data pointed to by the pointer held by mystuff:

```
*mystuff
```

- In fact, you can use the * and -> operators on the unique_ptr as if this is a pointer.

Memory management in Modern C++ (Cont)

Remember how I said you can’t pass around a unique_ptr by value or assign it to another variable?

You can instead create a shared_ptr just like with unique_ptr but it is almost always the wrong solution.

What this means semantically is that each function that has a copy is an “owner” of this memory and the memory will only be freed when all shared_ptr objects that point to this memory cease to exist.

Passing by move rather than value

- We will see later in the course that it is possible to define a move constructor and by doing this, forbid passing by value. This is what the standard library developers have done when defining unique_ptr.
- Therefore in order to pass a move only type into a function, or to assign it, we need to pass by move. We do this by wrapping the variable in a call to std::move such as:
- You would call such a function as follows:

```
//not an actual function in the standard library
take_ownership(std::move(mystuff));
```

Rvalue References

It is common in many programming languages to distinguish between an Rvalue and an Lvalue as value types.

An good way to think about lvalues is that something must be an lvalue to be on the left hand of an assignment. Rvalues cannot be on the left hand of an assignment. You cannot take the address of an rvalue either.

For example: the integer literal 3. Does it make sense to assign another value to it, eg: 3=4? Likewise, can we take the address of 3?

We cannot get a reference to 3 either.

Rvalue References (Continued)

This was extended in C++11 by the concept of an rvalue reference. That is, a reference to a new thing.

We can define an rvalue reference with the && symbol.

This is not a reference to a reference but rather a reference that is disconnected from its original context – it represents a change in the ownership of an object.

In fact, std::move() is actually implemented by converting a reference passed in into a rvalue reference and therefore rather than copying the data simply involves arranging a couple of pointers. The original object however is no longer valid as the data now has a new owner.

nullptr and nullptr_t

- Historically the NULL pointer in c++ was more loosely typed than in C where in C it is defined as a void pointer that points to the 0th address – there is typically a #define such as:

```
#define NULL (void*)0
```

- It was common (still is to some extent) for 0 to be used instead of NULL. nullptr was introduced to narrow the definition to being an actual pointer to the NULL address.
- nullptr is an object of type nullptr_t for which the == operation is overloaded for comparison with pointers and it returns true if the pointer being compared is NULL.

Initializer Lists and Uniform Initialisation

- Before C++11 we had “initializer lists” but there was no class to represent them.
- This means it was a lot easier to initialize an array than to initialize stl container types
- Eg: int nums[] = {3,1,5,7,9};
- But to do the same for a vector:
- vector <int>v;
- v.push_back(10);
- v.push_back(20);
- So it means a lot more code was required to use the stl data structures over the basic array structure.

Initializer Lists and Uniform Initialisation (Continued)

- This has been solved by the introduction of the initializer_list class
- An initializer list object is created when we have two braces with a number of comma separated items inside it.
- When constructing an object from this we may either construct using a constructor that accepts an initializer list
- If that's not available, a constructor that accepts the exact arguments will be selected.

Eg: std::vector<int> ints{3,5,7,8,9};

- Will construct a vector of ints whose original values are as specified.

Initializer Lists and Uniform Initialization (Continued)

- There are some impacts to using an initializer_list for initialization.
- There must be an exact match for the constructor that is called – no conversion in allowed.
- Eg: int i{3.5}; //compiler error
- int i = 3.5; //allowed because there is an implicit conversion from double to int.
- int i(3.5);

Constexpr

- All expressions take time but it is a question of where that time occurs.
- C++11 introduced the concept of expressions that can be evaluated at compile time completely and thus take no runtime.
- These expressions must involve no runtime overheads and any functions they call must also be constexpr. A constexpr can even be a function. Thus, the following is legal:
- //if a and b are not known at compile time then it gracefully degrades to a runtime expression.
- constexpr int mult(int a, int b)
- {
    - return a * b;
- }

Constexpr (Continued)

- However the following expression in a class is illegal:
- constexpr std::string name = “fred”;
- This is because the above expression involves memory allocation and thus cannot be completely resolved at compile time (the constructor is not labelled constexpr).

Enum class

- Enumerations in C can be handy but they have some limitations:
  - Each enumeration value is in the global namespace and so it is common to have namespace collisions (enums with the same name in different enumerations resulting in compiler errors)
  - They are not type safe in that enumerations are basically integers under the hood and enumeration values from different enumerations can easily be mixed.

Enum class (continued)

- The solution to this is the enum class. They are typesafe and the declaration of them is little different to C enumerations:

- enum class mood
- {
-     SAD, HAPPY, BLUE
- };

- enum class color
- {
-     RED, GREEN, BLUE
- };

Enum class (continued)

- And we then each enumeration is in its own namespace – all elements of the mood enum class are preceded with mood:: so we have mood::HAPPY and mood::SAD.
- Likewise mood and color are different types: we can’t compare moods with colors and nor should we – they are totally unrelated concepts.