Week 2

Fundamental Types, Classes, Enumerations

Agenda

- Week 2-1 – Fundamental Types
  - Integer Types, Char Types, Floating Point Types
  - Initialization Methods, Type Inference, Alignment
- Week 2-2 – Pointers, References, Arrays
  - Generic Pointers, L/Rvalue References, Range Based For
- Week 2-3 – Classes & Scoped Enumerations
  - Move Operators, Class Variables, Unions, Enumerations

Week 2-1

Fundamental Types

- In C++, a fundamental type is used to distinguish
  - How to interpret a bit string in a region of memory (is it an int or a double or a char)
  - And to define what operations are applicable to that same region of memory
- The types used in C++ are composed of some number of bytes where larger types consist of more bytes and thus take up more memory to represent
- Two sets of types will be discussed here in more detail:
  - Integer Types – (int, char)
  - Floating Point Types (float, double)

Integer Types

- Integer types consist of
  - Standard integers which could be either signed or unsigned (positive and negative or just positive numbers)
  - Booleans (true or false)
  - Character types
- As an aside with types that are multi-byte (consists of more than one byte), there is a notion of ordering known as endian-ness. Systems that store the most significant byte first are known as big-endian and the opposite is known as little-endian. Most intel and amd based platforms are little-endian.

Standard Integers

- There are effectively 5 signed (positive and negative) integer types in C++17:
  - signed char – typically is stored in 1 byte. Not necessarily the same thing as ‘char’
  - short int – or more simply short, generally occupies at least as much memory as a signed char (usually at least 2 bytes)
  - int – the usual integer type, generally occupies 4 bytes but at least as much as a short int
  - long int – or more simply long, generally occupies at least as much memory as an int
  - long long int – or more simply long long occupies at least as much memory as an long int (typically 8 bytes)

Standard Integer Ranges

Type	Min	Max
signed char	-128	127
short int	<= -32,768	>= 32,767
int	-2,147,483,648	2,147,483,647
long int	<= -2,147,483,648	>= 2,147,483,647
long long int	<= -9,223,372,036,854,775,808	>= 9,223,372,036,854,775,807

The range of the standard signed integer types in 2's compliment

Unsigned Standard Integers

- For each signed integer type there is a complimentary unsigned version of it:
  - unsigned char, unsigned short int, unsigned int, unsigned long int, unsigned long long int
- The different between these and the signed integers is that they only represent positive numbers there by effectively extending the number of positive numbers they can represent

Type	Min	Max - 32 bit
unsigned char	0	255
unsigned short int	0	>=65,535
unsigned int	0	4,294,967,295
unsigned long int	0	>= 4,294,967,295
unsigned long long int	0	>= 18,446,744,073,709,551,615

Denoting Integer Types

```
int a, b, c, d;

a = 91;   // decimal
b = 0133; // octal - note the leading 0
c = 0x5b; // hexadecimal - note the leading 0x
d = 0X5B; // hexadecimal - note the leading 0X
```

```
long int a, b, c, d;

a = 91L;   // long: L or l
b = 0133L; // octal - long: L or l
c = 0x5bL; // hexadecimal - long: L or l
d = 0X5BL; // hexadecimal - long: L or
```

```
long long int a, b, c, d;

a = 91LL;   // long: LL or ll
b = 0133LL; // octal - long: LL or ll
c = 0x5bLL; // hexadecimal - long: LL or ll
d = 0X5BLL; // hexadecimal - long: LL or ll
```

Denoting Integer Types

```
unsigned int g;
unsigned long int h;
unsigned long long int k;

g = 0x5bU;  // unsigned int: U or u
h = 0X5BUL; // unsigned long: (U or u) and (L or l) any order
k = 456789012345ULL; // unsigned long long: (U or u, LL or ll) any order
```

Character Types

- There are effectively 6 character types in C++17:
  - char – typically is stored in 1 byte. The usual char type used to represent basic characters. Depending on the platform char will be equivalent to signed or unsigned char
  - Locales – these sets of char types are used to represent characters that go beyond the basic ones (alpha-numeric, basic symbols … etc). Non-English languages can be represented with these types
  - wchar_t – is a wide character type, is implementation dependent as to how much memory it consists of but is defined to be large enough to hold the largest character set supported by the implementation
  - char16_t – is a wide character type used to hold 16 bit character sets like UTF-16
  - char32_t – is a wide character type used to hold 32 bit character sets like UTF-32

Denoting Character Types

```
wchar_t k, m, n, p;
k = L'[';    // character - note the leading L
m = L'\133'; // octal - note the leading L'\
n = L'\x5b'; // hexadecimal - note the leading L'\x
p = L'\X5B'; // hexadecimal - note the leading L'\X

char16_t k, m, n, p;
k = u'[';    // character - note the leading u
m = u'\133'; // octal - note the leading u'\
n = u'\x5b'; // hexadecimal - note the leading u'\x
p = u'\X5B'; // hexadecimal - note the leading u'\

char32_t k, m, n, p;
k = U'[';    // character - note the leading U
m = U'\133'; // octal - note the leading U'\
n = U'\x5b'; // hexadecimal - note the leading U'\x
p = U'\X5B'; // hexadecimal - note the leading U'\X
```

Floating Point Types

- There are effectively 3 floating point types in C++17:
  - float – single precision floating point number, typically 4 bytes
  - double – double precision floating point number, typically 8 bytes
  - long double – double precision floating point number with possibly more precision than double, at least 8 bytes

Type	Size	Significant Digits	Min Exponent	Max Exponent
double	8 bytes	15	-307	308
float	4 bytes	6	-37	38

Type Limits References

- https://en.cppreference.com/w/cpp/language/types
- https://docs.microsoft.com/en-us/cpp/c-language/limits-on-floating-point-constants?view=vs-2019
- https://docs.microsoft.com/en-us/cpp/c-language/cpp-integer-limits?view=vs-2019

Initialization Methods

- There are a few variations of initialization for variables present in C++ and some which is already familiar:
  - C-style assignment – int x = 7;
  - C-style with narrowing – int x = 7.2;
  - Universal form – int x { 5 } – This is the preferred method
  - Universal form (redundant) – int x = { 5 }
- When using the universal form it denies things like type narrowing as well as reporting errors in case of type mismatches. It's a more type safe init method.

Initialization Methods

- Array types can also make use of a similar initialization method called aggregate initialization that uses an initializer-list
  - A initializer-list a comma separated list of values – { 1, 2, 3, 4 …. } which can be used to init arrays
- When using an empty initializer-list, all the items in the array will be initialized to zero

Initialization Methods

- Below are some of the variants of this initialization method for arrays

```
Type identifier [ c ] = { initializer-list };
Type identifier [ c ] { initializer-list };
Type identifier [   ] = { initializer-list };
Type identifier [   ] { initializer-list };
Type* identifier = new Type[ n ] { initializer-list };
```

Type Inference

- Using the auto keyword we can have the compiler infer what the type of a particular variable is being on the right operand at initialization
  - auto x = 3.6; // The type of x will be based on 3.6 so some sort of floating point type (likely double)
- This is particularly useful when dealing with long named types to shorten them
- Auto however can not be used with arrays to determine their type as of yet

Type Alignment

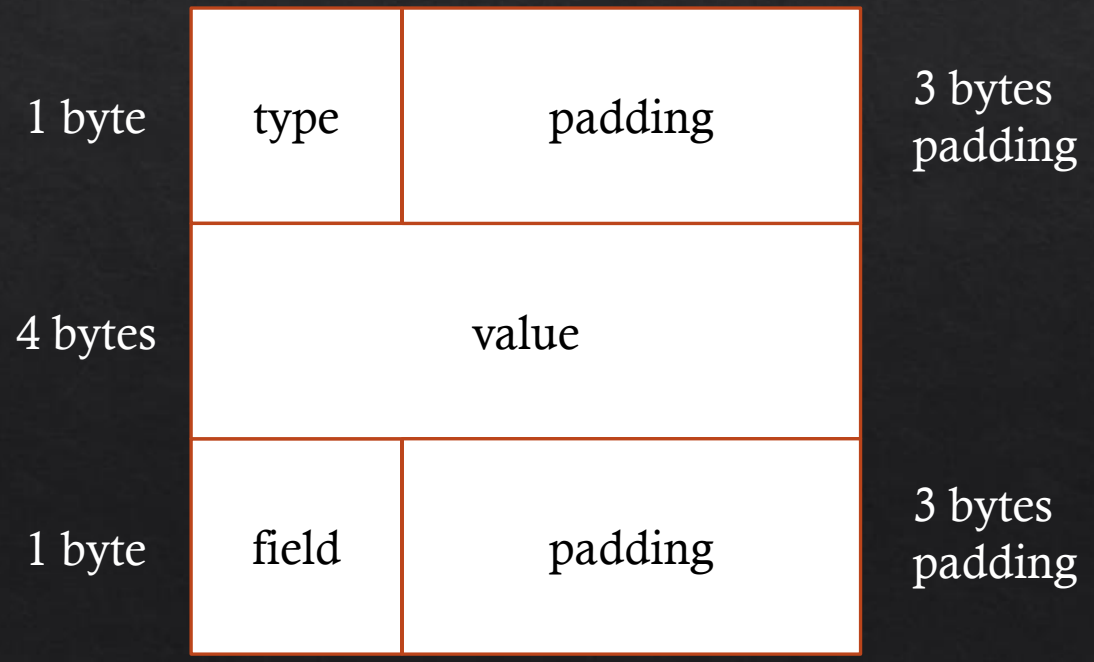
- Type alignment refers to the position of our data structures inside the memory
- We sometimes need to not only have enough memory for our objects but also to have them properly aligned for performance or to simply work sometimes on certain platforms
- Consider this struct:

```
struct sample {
  char type; // 1 byte
  int value; // 4 bytes
  char field; // 1 byte
};
```

Type Alignment

- Integers typically want to be aligned on the ‘word’ boundaries of 4 bytes so a more realistic view of this struct in memory might be like this. Note the padding. Effectively because the struct is arranged in the order we had it will require padding to make it fit the boundaries well.

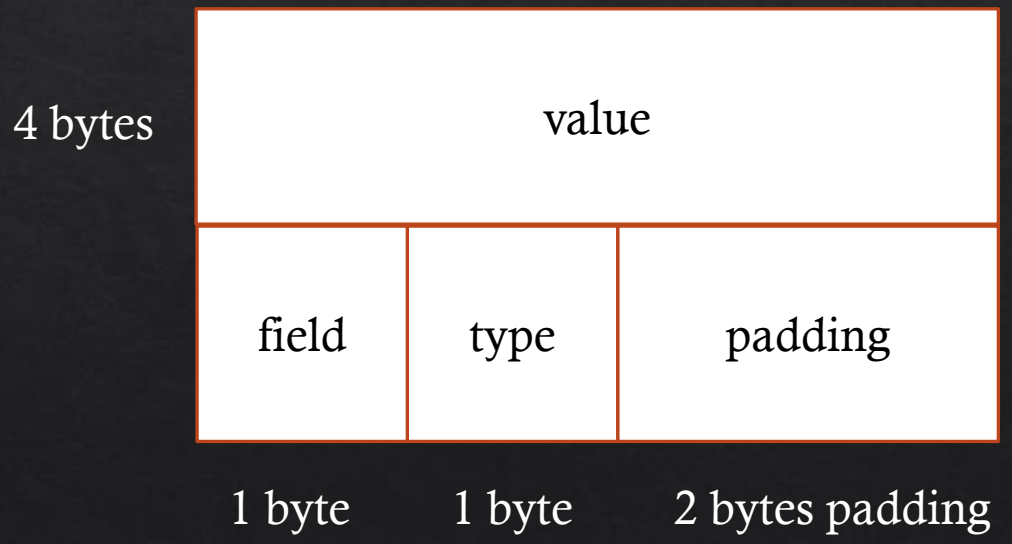
```
struct sample {
  char type; // 1 byte
  int value; // 4 bytes
  char field; // 1 byte
};
```



Type Alignment

- If we were to rearrange the struct into this ordering (the value integer first) of its members instead we would have a more efficient layout (we still need some padding however).

```
struct sample {
  int value; // 4 bytes
  char field; // 1 byte
  char type;  // 1 byte
};
```



Type Alignment

- Using the alignof function you can check on how different types on a given system need to be aligned – alignof(int), alignof(double)
- And following that you can then manually configure alignment for say a given struct with the alignas function
- Generally alignment can be left alone unless there's a strong case to need to optimize the objects in play or it's required for operation

```
struct alignas(8) sample
{
  int value; // 4 bytes
  char field; // 1 byte
  char type;  // 1 byte
};
```

Now aligned on 8 byte boundaries

Week 2-2

Pointers Types

- A pointer is a type that holds an address as its value. Every type has a associated pointer type to go along with it. In order to get the value at the address, dereferencing is needed
  - int – int*, double – double*, char – char*
- Different pointer types are not compatible with each other would need to be casted to interoperate
- Pointers can point to something or to nothing (nullptr). Generally pointers should be initialized to nullptr and not left dangling undefined. Trying to dereference a nullptr typically leads to a run time error.

Pointers Types

- Like with fundamental types, pointers can also be typedef'd to less the amount of typing:

```
typedef long long int* lliptr

lliptr myptr = nullptr;
```

Note that the pointer portion is already part of the typedef so there's no need for an extra *


```
typedef long long int lli;

lli* myptr2 = nullptr;
```

Note the *

Void Type

- The void type is an incomplete type (it's missing something – consider an abstract base class). C++ disallows the creation of objects that have a void type.
- We typically use void to denote the return type of functions that don't return anything
- Additionally we can also use the void type for generic pointers

Generic (void) Pointers

- Generic pointers are different from the other typed pointers discussed so far. It doesn't hold any information regarding the type of an object, just the address
- With that in mind generic pointers can't be dereferenced similarly to nullptr (as they lack type information)
- The void keyword is used to identify a pointers as a generic pointer:

```
int x = 5;
void* ptr = nullptr;
ptr = &x;
```

- In order to make use of a void pointer you will likely need to cast it as a some typed pointer using constrained casts:

```
int* myintptr = static_cast<int*>(ptr)
```

Generic (void) Pointers

- Generic pointers can also be used as a go around between different pointer types through casting:

```
char x = 'A';
int* i = nullptr ;
char* c = &x;
i = static_cast<int*>(static_cast<void*>(c));
```

References

- A reference is an alias to an existing object
- In C++17 there are two categories of references:
  - Lvalue references – these kinds of references refer to an accessible region of memory that has identity, originally meant to be the left-hand side of an assignment, signified by a single &
  - Rvalue references – the complement to an lvalue, signified by double &&, it identifies:
    - an object near the end of its lifetime
    - a temporary object or subobject
    - a value not associated with an object / identity

Standard Library w/ References

- There are two functions we'll be making use of when dealing with Rvalue references and Lvalue references from the standard library:
  - std::ref() – returns a lvalue reference to its argument, we'll be revisiting this in the second half of the course when we deal with some of the standard template library containers and functions
  - std::move() – returns a rvalue reference to its argument, we'll see some practical use of this shortly

Range Based For-Loops

- A ranged based for loop is a nice improvement on a regular for loop in that it allows you to traverse through an array without need its size:

```
for (int i = 0; i < sizeof(x) / sizeof(int); ++i) {
  cout << "reg-item: " << x[i] << endl;
}
```

Notice the difference

```
for (int& item : x) {
  cout << "range-item: " << item << endl;
}
```

Range Based For-Loops

- Rather than having a specific type we have the auto keyword infer the type of the items in the array

```
for (auto& item : x) {
  cout << "range-item: " << item << endl;
}
```

Notice the difference

- Range based for loops can be applied to any of the collection types in the standard

Week 2-3

Move Operators

- Sometimes there cases where we will work with objects that will no longer be needed once it has been copied or assigned. In these cases rather than make use of the usual copy-assignment and copy-constructor mechanisms which will copy everything over by allocating new memory, we can alternatively make use of move operators
- Move operators much like their naming will more simply move resources by copying the addresses of the source, this eliminates the need for actual copying
- The signatures for a move constructor and move assignment operator are as follows:
  - Class-name(Class-name&&); - Playdoh(Playdoh&& src)
  - Class-name& operator=(Class-name&&); - Playdoh& operator=(Playdoh&& src)

Move Operators

- The && symbols will denote that these functions work with Rvalues. They take Rvalue references to the source object (object to be moved) and swap the addresses of it and the current object:
  - Class-name(Class-name&&); - Playdoh(Playdoh&& src)
  - Class-name& operator=(Class-name&&); - Playdoh& operator=(Playdoh&& src)
- When we want to activate these functions (vs the usual copy mechanisms) we can use the std::move function to do so.

Anonymous Classes

- Anonymous classes are classes that don't have an identifier attached to their definition
- They're used in the cases where you don't need to reference their typing and may only need to use a small number of instances in a closed scope
- They use an instance identifier following their definition to name the instances
- Generally they're used as on demand classes without needing to fully declare/define one out as a separate module.

Anonymous Classes

Note the lack of name

```
class { // anon class
  double efficiency;

  public:
  void setEfficiency(double e) { efficiency = e; }
  int fuelCost(int distance) {
    return distance * efficiency;
  }
} engine;
```

We have a single instance of this unnamed class called ‘engine’

Class Variables/Functions

- A class variable is a data member that holds the same information for every instance of that class.
- Class functions compliment these variables by being able to access them at any time. These functions must only access class variables and not other instance data members
- To establish a variable as a class variable you use the static keyword
- The use of class functions allow for those class variables to be private

Structs & Unions

- Structs & Unions are like classes that have weaker encapsulation. They are public by default.
- Structs are arranged sequentially but not necessarily contiguously in memory
- Unions on the other hand are arranged in parallel in memory meaning that the members of a union share the same address in memory
- Thus an object of a union can only hold the value of one its members at a time

Enumerations

- Enumerations or enums present a mechanism to allow for using descriptive names in place of constant values. This can improve the readability of code
- The underlying type behind an enum is an int
- Enums come in two types:
  - Plain Enumerations
  - Scoped Enumerations

Plain Enums

- A plain num has the syntax of:
  - enum Type { symbolic_constant_1, symbolic_constant_2, … };
  - enum Traffic_light { red, yellow, green };
  - In this case red, yellow, green would map to 0, 1, 2
- We can then use names like a class and initialize objects based on them:
  - Type identifier;
  - Traffic_light x = red;

Scoped Enums

- A scoped enum has the syntax of (the main diff is the class):
  - `enum class Type { symbolic_constant_1, symbolic_constant_2, ... };`
  - `enum class Traffic_light { red, yellow, green };`
- The difference between scoped enums and plain ones that that they will abide by the scope of the enums and thus need to be resolved with their scope (scope resolution)
- It additionally denies the use of implicit conversion to integers
- Generally scoped enums can be considered safer enums

Underlying Enum Types

- You can assign your own values to enums to override the default starting point of 0 , 1, 2, 3…

```
enum class Traffic_light {
  red = 1,
  yellow = 3,
  green = 6,
  orange = 12
};
```