13. Enumerators, `struct` and `union`

13.1. Basics of `enum`

An enumerator (`enum`) creates a data type that can take the value in a set of named integral constants. By default, the first will be `0`, the second `1`, etc.

```
#include <iostream>
int main() {
  enum level_t { LOW, MEDIUM, HIGH };
  level_t danger = HIGH;  // note: HIGH == 2
  std::cout << danger << '\n';
}
```

The integer values that represent the constants can also be set and controlled explicitly. They are numbered in increasing order:

```
enum battery_t { FULL = 100, ADEQUATE = 50, EMPTY = 0 };
enum level_t { LOW = -100, MEDIUM, HIGH };  // MEDIUM == -99, HIGH == -98

enum day_t { MON = 1, TUE, WED, THU, FRI, SAT, SUN};
// MON == 1, TUE == 2, WED == 3, THU == 4, FRI == 5, SAT == 6, SUN == 7
```

13.1. Basics of **enum**

The advantage of **enum** is that the code is readable easier to maintain variables that can take any of a given set of variables :

```
#include <iostream>
int main() {
  enum level_t { LOW, MEDIUM, HIGH };
  level_t humidity = LOW;
  std::string s;
  switch (humidity) {
    case LOW: s = "low"; break;
    case MEDIUM: s = "medium"; break;
    case HIGH: s = "high"; break;
  }
  std::cout << "The humidity is " << s << '\n';
}
```

13.1. Basics of **enum**

Since the values are mapped to integers, this might lead to problems:

```
#include <iostream>
int main() {
  enum level_t { LOW, MEDIUM, HIGH };
  enum battery_t { FULL, ADEQUATE, EMPTY };  // LOW cannot be reused here
  level_t status1 = LOW;      // multiple types' values are basically integers,
  battery_t status2 = EMPTY;  // the following will lead to a warning only:
  std::cout << ( status1 == status2 ) << '\n';
}
```

Typically, **enum**s are used when values are unlikely going to change, such as week days, months, colors, card values.

13.2. Scoped enumeration `enum class`

Since C++11, a *scoped* enumeration (`enum class`) data type is a type-safe enumerator (not a class) that is not implicitly convertible to an integer.

```
enum class Level { LOW, MEDIUM, HIGH };
Level status1 = Level::LOW;  // "status1 = LOW" would lead to error

enum class Battery { FULL, ADEQUATE, EMPTY };
Battery status2 = Battery::EMPTY;  // "status2 = EMPTY" would lead to error

// the following will lead to an "invalid operands" error:
std::cout << ( status1 == status2 ) << '\n';
```

The scoped enumeration's underlying data type can be set explicitly:

```
enum class Choice : int8_t { YES, MAYBE, NO, UNKNOWN };
```

13.2. Scoped enumeration `enum class`

A scoped enumeration (`enum class`) cannot be compared to, or use the constants as, integers. `enum class Level { LOW, MEDIUM, HIGH };` will cause `Level l = Level::MEDIUM; std::cout << l;` to result in an error. `l` is of type `Level`, which `std::cout <<` doesn't have a method.

The need for scope might also make code less terse and longer:

```
enum class Level { LOW, MED, HIGH };
Level humidity = Level::LOW;
std::string s;
switch (humidity) {
  case Level::LOW: s = "low"; break;
  case Level::MEDIUM: s = "med"; break;
  case Level::HIGH: s = "high"; break;
}
```

Since C++20, `using enum` can import enumerators in the local scope:

```
enum class Level { LOW, MED, HIGH };
Level humidity = Level::LOW;
std::string s;
switch (humidity) {
  using enum Level;
  case LOW: s = "low"; break;
  case MEDIUM: s = "med"; break;
  case HIGH: s = "high"; break;
}
```

13.3. `typedef` and `struct`

Structures (preceded by `struct`) historically combine variables of different types, similar to how arrays combine variables of the same type.
In C++, structures are semantically *very* similar to classes, but by default do not set attributes or methods to private (plus a bit more).

```
struct anonExamEntry {
 long studentId = 0;  // initialization here
 float grade = 1.0;   // possible since C++11
};

anonExamEntry entry1;
entry1.studentId = 17017491;
entry1.grade = 2.3;
```

13.3. **typedef** and **struct**

typedef is a keyword that is used to assign a new name to any existing data-type:

```
typedef int integer; integer i = 9;
```

In C, new variables of a particular structure type would need the keyword **struct** to be added each time:

```
struct examEntry {
  long studentId;
  float grade;
};
struct examEntry entry;  // "anonExamEntry entry;" not possible in C
```

Which is why **typedef** is used to provide a new name instead:

```
typedef struct examEntry examEntry;  // "struct examentry" = "examEntry"
```

13.4. **union** and **std::variant**

A **union** is a special class type that can hold only one of its non-static attributes.
It is at least as big as needed to store the largest attribute, but is usually not larger.
Unions can have non-virtual methods, but cannot be involved in inheritance.

```
#include <iostream>

union Number {   // a Number has:
  long l;        // either a long,
  unsigned u;    // or an unsigned,
  double f;      // or a double
};
```

union.cpp

```
int main() {
  Number n;
  n.l = 7l;
  std::cout << n.l << '\n';
  n.f = 8.9;
  std::cout << n.f << '\n';
  std::cout << sizeof(n) << '\n';
}
```

13.4. **union** and **std::variant**

Since C++17, STL includes **std::variant**, which replaces many uses of unions and union-like classes:

variant.cpp

```
#include <iostream>
#include <variant>

int main() {
  std::variant<char, std::string> s;  // s stores a char or a string
  s = 'a';
  std::visit([](auto x){ std::cout << x << '\n';}, s);  // visit since C++17
  s = "string!";
  std::visit([](auto x){ std::cout << x << '\n';}, s);  // (more info here)
}
```

13.5. The Visitor Pattern

Visitor allows adding new virtual functions anytime to a family of classes, without modifying these.
Instead, a special visitor class implements all of the appropriate specializations of the virtual function.
This visitor is given the instance reference, and implements the goal through double dispatch (which we can use **`std::visit`** for).

13.5. The Visitor Pattern

Example: Different user interface events have different properties (position, key, size, etc.), so are represented by different classes.
A Visitor handles all these event types, instead of needing to declare / implement the handling in the Events. Other Visitors for Events can be added later.

Visitor.cpp

```
// Define different event types
struct ClickEvent { int x, y; };
struct KeyEvent { char key; };
struct ResizeEvent { int width, height; };

// An Event is a variant of all possible events
using Event = std::variant<ClickEvent, KeyEvent, ResizeEvent>;
```

13.5. The Visitor Pattern

Visitor.cpp

```
void handleEvent( const Event & event ) {  // Handle Events with a Visitor
  std::visit( [](auto && e) {  // handle in lambda function to std::visit
      using T = std::decay_t<decltype(e)>;  // see explanations in source file
      if constexpr (std::is_same_v<T, ClickEvent>) {
        std::cout << "Click at: " << e.x << ',' << e.y;
      } else if constexpr (std::is_same_v<T, KeyEvent>) {
        std::cout << "Key pressed: " << e.key;
      } else if constexpr (std::is_same_v<T, ResizeEvent>) {
        std::cout << "Window now: " << e.width << 'x' << e.height;
      } else std::cout << "Unknown event."
      std::cout << '\n';
    }, event);
}
int main() {
  // Create different events, and afterwards handle them through the Visitor:
  Event e[] = { ClickEvent{10, 20}, KeyEvent{'A'}, ResizeEvent{800, 600} };
  for (auto event : e) handleEvent(event);
}
```