Technicalities

Raul P. Pelaez

November 13, 2024

Contents

1 Introduction **1**

2 `enum` **1**

3 The edge of static typing **2**
3.1 `std::variant` . . . 2
3.2 `std::any` . . . 4
3.3 `std::optional` . . . 4

4 Flushing: `std::endl` vs. `'\n'` **6**

5 Assertions **7**
5.1 `assert` . . . 7
5.2 `static_assert` . . . 8

6 `std::move` and `std::forward` **9**

7 Arguments to main **10**

8 Exercises **11**
8.1 Enumerating . . . 11
8.2 Parsing input . . . 12
8.3 Scarce resources . . . 14

1 Introduction

This lesson includes C++ technical details that, while not core concepts, are useful tools for your programming toolbox. This lesson is a bit of a grab bag containing what "was left behind" during the rest of the course.

2 `enum`

Enumerations are a way to define a new type that can only take a limited number of values. They are useful for code readability and reducing errors by constraining possible values. For example, if we have a function that takes a parameter that can only be one of three values, an enumeration helps ensure correct usage.

Source code

```
enum class Color { red, green, blue };
void draw(Color c) {
  switch(c) {
  case Color::red:
    std::cout << "Drawing in red\n";
    break;
  case Color::green:
    std::cout << "Drawing in green\n";
    break;
  case Color::blue:
    std::cout << "Drawing in blue\n";
    break;
  }
}
```

Advice

The Cycles game uses an enum to identify the possible moves a player can attempt (e.g., north, south, east, west):

Source code

```
enum class Direction { north = 0, east, south, west };
```

Note the `= 0` initializer, which sets the first value to 0 and increments the following values by 1. This can be useful for indexing or other operations.

Info

`enum` vs. `enum class`
Adding the `class` keyword to an `enum` declaration makes it a scoped enumeration, also known as an `enum class`. This prevents implicit conversions to integers and restricts the scope of the enumeration values. It is recommended to use `enum class` instead of `enum` for better type safety. Scoped means that one must use the enum's name to access its values, as in `Color::red` instead of just `red`.
No implicit conversions means that in order to use an enum value as an integer, one must explicitly cast it, as in `static_cast<int>(Color::red)`.

3 The edge of static typing

C++ is a statically typed language, which means that the type of every variable must be known at compile time and cannot change after definition. This allows the compiler to catch many errors before the program runs. We really really want to design our programs around this idea. However, C++ strives to gives us complete freedom (and thus responsibility) to do whatever we want.
We glimpsed into this when discussing `reinterpret_cast`, but there are other ways to relax the type system when we really need to. I will show you two of them, but remember, they are dangerous.

3.1 `std::variant`

`std::variant` represents a type-safe `union`. A `union` is a data structure that can hold, during its lifetime, objects of different types (from a predetermined list), taking up memory equal to the largest object

it can hold. Managing the active type in a `union` can be error-prone. `std::variant`, introduced in C++17, keeps track of the active type automatically, making it a safer alternative.

Source code

```
// A variable that can hold an int, OR a double, OR a string
std::variant<int, double, std::string> v;
// "Activate int mode"
v = 42;
std::cout << std::get<int>(v) << "\n";
// "Activate double mode"
v = 3.14;
std::cout << std::get<double>(v) << "\n";
// "Activate string mode"
v = "Hello";
std::cout << std::get<std::string>(v) << "\n";
// This will throw an exception
std::cout << std::get<int>(v) << "\n";
```

Advice

Note that variant is different from a tuple:

Source code

```
// A tuple that holds an int, AND a double, AND a string
std::tuple<int, double, std::string> t = std::make_tuple(42, 3.14,
↪  "Hello");
std::cout << std::get<0>(t) << "\n";
std::cout << std::get<1>(t) << "\n";
std::cout << std::get<2>(t) << "\n";
```

This tuple needs enough storage to hold all its elements at the same time, while the variant only needs enough storage for the largest element it can hold. The tuple will store all its elements all the time, while the variant will only store the active element.

Advanced

The older `union` construct is still available in C++, but you should generally consider it unsafe and error-prone. Just so you know, here's an example of a `union`:

Source code

```
union U {
  int i;
  double d;
  std::string s;
};
U u;
u.i = 42;
std::cout << u.i << "\n";
u.d = 3.14;
std::cout << u.d << "\n";
u.s = "Hello";
std::cout << u.s << "\n";
// This is undefined behavior but it will compile and run!
std::cout << u.i << "\n";
```

3.2 `std::any`

`std::any` is a type-safe container for single values of any type. Unlike `std::variant`, which requires a list of possible types, `std::any` can hold any type of object. Internally, `std::any` may use dynamic memory allocation (and `reinterpret_cast`), making it less efficient than `std::variant`. Use `std::any` sparingly, as its flexibility can reduce code readability.

Source code

```
std::any a; // No need to specify the possible types
a = 42;
std::cout << std::any_cast<int>(a) << "\n";
a = 3.14;
std::cout << std::any_cast<double>(a) << "\n";
a = "Hello";
std::cout << std::any_cast<const char*>(a) << "\n";
```

Advice

Using `std::any` is generally more expensive than using `std::variant`, which is more expensive than using regular variables. These constructs can reduce code readability and complicate maintenance. Avoid using `std::any` or `std::variant` if possible, as their flexibility may indicate a need to rethink the design.
Anecdotally, I do not remember ever using `std::any` in any of my projects. `std::variant` is more common (I have used it in really register-memory-constrained high-performance code before), but still not as much as regular variables.

3.3 `std::optional`

`std::optional` is a container that may or may not hold a value. It is useful for functions that may fail to return a value, allowing the caller to check if the value is present. `std::optional` is a safer alternative

to using null pointers or error codes. To showcase the need for it, see this function:

Source code

```
int findIndex(const std::vector<int>& vec, int value) {
  for (int i = 0; i < vec.size(); i++){
    if (v == value) {
      return i;
    }
  }
  return -1; // Not found
}
```

We had to use a magic number (-1) to indicate that the value was not found, while arguably this function should not even be able to return negative numbers (an index should always be positive, so an unsigned would be preferable). There are many ways to go around this issue. For instance, the STL algorithms normally return an iterator to the relevant element or the end iterator if the element is not found (see `std::find`). C APIs tend to have the actual output as a pointer argument and return a status code instead (see for instance BLAS or LAPACK). We could also write something like this:

Source code

```
std::tuple<bool, int> findIndex(const std::vector<int>& vec, int value) {
  for (int i = 0; i < vec.size(); i++){
    if (vec[i] == value) {
      return {true, i};
    }
  }
  return {false, 0}; // Not found
}
```

But this is not very readable and also non-standard. `std::optional` is a better alternative:

Source code

```
std::optional<int> findIndex(const std::vector<int>& vec, int value) {
  for (int i = 0; i < vec.size(); i++){
    if (vec[i] == value) {
      return i;
    }
  }
  return std::nullopt; // Not found
}
```

We would use this function like this:

Source code

```
std::vector v = {1, 2, 3, 4, 5};
auto index = findIndex(v, 3);
if (index.has_value()) {
  std::cout << "Found at index " << index.value() << "\n";
 } else {
  std::cout << "Not found\n";
 }
```

4 Flushing: `std::endl` vs. `'\n'`

Sending every single entity received by an output stream to the output device immediately would be inefficient (for instance, sending a network packet for every character in a book vs a single one with the whole book). Instead, the output is buffered, and the buffer is flushed when it is full or when the program ends. Flushing the buffer forces the output to be sent immediately with the current state of the buffer.
`std::endl` and `'\n'` both insert a newline character into the output stream. However, `std::endl` also flushes the output buffer.
Flushing the buffer can be useful when immediate output is required, but it can also slow down the program. In most cases, using `'\n'` is sufficient for adding newlines. However, in certain situations not flushing can create strange behavior, in particular when we are trying to debug a program based on its output (like messages to the CLI).
Try to compile this code with and without the offending line:

Source code

```
#include <variant>
#include <iostream>
int main(){
  std::variant<int, double, std::string> v;
  v = 42;
  std::cout << std::get<int>(v) << "\n";
  v = "Hello";
  std::cout << std::get<std::string>(v) << "\n";
  // This will throw an exception
  // std::cout << std::get<int>(v) << "\n";
}
```

Most probably, if you uncomment the bogus line, only the exception will be printed, even when the code up until that point is correct and runs. The reason is that std::cout is not getting the chance to flush the buffer before the program crashes. This is a common source of confusion when debugging. We can solve it by using `std::endl` instead of `'\n'` for this kind of debugging output.

Info

We can also use `std::flush` to flush the buffer without adding a newline character. These two lines are equivalent:

Source code

```
std::cout << std::endl;
std::cout << "\n" << std::flush;
```

5 Assertions

Assertions are used to enforce conditions and validate assumptions in code. For example, checking for null pointers before proceeding with a function can prevent runtime errors.

Source code

```
void f(int* x) {
  if (x == nullptr) {
    std::cerr << "Error: x is nullptr\n";
    return;
  }
  // Do something with x
}
```

Advice

There is a reason why almost every time I need to showcase some kind of error-checking, assertion, error condition, etc. I use a pointer argument. It is probably the number one source of errors in C and C++ code in the wild. Rockets have fallen from the sky, stock markets have crashed and hospitals and airports have been shut down because someone did not handle a pointer correctly.
That is to say, for the millionth time, if you are writing C++, use pointers as little as possible, and if you do, be paranoid when checking them.

5.1 `assert`

The `assert` macro checks conditions during runtime. If a condition in `assert` evaluates to false, the program terminates with an error message. This is useful for debugging but is often disabled in production by defining `NDEBUG`.

Source code

```
#include <cassert>

void divide(int a, int b) {
  assert(b != 0 && "Denominator cannot be zero.");
  std::cout << "Result: " << a / b << "\n";
}
```

Advice

CMake automatically defines `NDEBUG` when building in release mode. This disables `assert` statements, so they are not executed in production code. This is a good practice to avoid unnecessary overhead in release builds. You can ask CMake to build in debug mode by using the `CMAKE_BUILD_TYPE` variable:

Source code

```
# Configure in debug mode (adds -g, disables optimizations, and enables
↪  assertions)
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
# Configure in release mode (enables optimizations and disables
↪  assertions)
$ cmake -DCMAKE_BUILD_TYPE=Release ..
```

`assert` is a good development tool and helps with debugging, but it is not a replacement for proper error handling and input validation. It is best used for catching programming errors, not for handling user input or expected conditions.

5.2 `static_assert`

`static_assert` performs compile-time checks, ensuring conditions hold true before compilation completes. This is useful for constraints on template parameters, making the code more robust and error-free.

Source code

```
template<typename T>
void checkType() {
  static_assert(std::is_integral<T>::value, "T must be an integral type.");
}
```

Advice

Nowadays we would use C++20 concepts for this:

Source code

```
template<std::integral T>
void checkType() {
  // Do something
}
```

Also to check that some assumption required by our code is true, like the size of a certain type:

Source code

```
static_assert(sizeof(int) == 4, "int must be 4 bytes");
```

Note that you should not rely on stuff like an `int` being 4 bytes, that is why we have `std::int32_t` and friends. And if you need code that depends on the size of a type, you should try to use `sizeof` to make it generic.

Advanced

`static_assert` can be used for highly sophisticated compile-time checks. For example, say that you want to write a function that only accepts lambda functions that do not capture:

Source code

```
#include <type_traits>
int main(){
  auto lambda = []() { return 42; };
  // Good
  static_assert(std::is_convertible_v<decltype(lambda), int (*)()>,
                "Lambda must be captureless");
  auto lambda_with_capture = [&]() { return 42; };
  // Compile error
  static_assert(std::is_convertible_v<decltype(lambda_with_capture), int
  ↪  (*)()>,
                "Lambda must be captureless");
}
```

6 `std::move` and `std::forward`

`std::move` and `std::forward` are utilities for handling resource management in C++. `std::move` enables transfer of resources by converting an object into an rvalue reference, indicating it can be "moved from." It is often used with movable types like `std::vector` or `std::string`.

`std::forward` is primarily used in template code to preserve the value category of arguments. It enables both lvalue and rvalue references to be passed efficiently without redundant copies.

Example of `std::move`:

Source code

```
std::vector v1 = {1, 2, 3};
auto v2 = std::move(v1); // v1 is now empty
```

Example of `std::forward` in template functions:

Source code

```
template<typename T>
void wrapper(T&& arg) { // && here means "universal reference"
  process(std::forward<T>(arg));
}
//..
std::vector v = {1, 2, 3};
wrapper(v); // v is passed by lvalue reference
wrapper(std::move(v)); // v is passed by rvalue reference
wrapper(std::vector{1, 2, 3}); // temporary vector is passed by rvalue reference
```

Does not matter what the category of the argument to wrapper is, it will be forwarded to process with the same category. This is useful when you want to pass an argument to another function without copying it, but you do not know if the argument is an lvalue or an rvalue.

Info

rvalue references and universal references
The double ampersand && in T&& is a universal reference, which can bind to both lvalues and rvalues. `std::forward` preserves the value category of the argument, forwarding it to another function without unnecessary copies.
The value category refers to whether an object is an lvalue or an rvalue. An lvalue is an object with a name, while an rvalue is a temporary object or an object that can be moved from. The syntactic rules of && can be confusing, but the distinction is important for understanding move semantics and perfect forwarding.
Examples:

Source code

```
Widget&& var1 = someWidget;      // here, "&&" means rvalue reference
auto&& var2 = var1;              // here, "&&" does not mean rvalue
↪  reference
template<typename T>
void f(std::vector<T>&& param);  // here, "&&" means rvalue
template<typename T>
void f(T&& param);               // here, "&&" might mean either rvalue
↪  or lvalue reference
std::vector<int> v;
//...
auto &&val = v[0]; // val becomes an lvalue reference.
```

The heart of the matter here is that "&&" sometimes means rvalue reference, but sometimes it means either rvalue reference or lvalue reference.
A very nice writeup here. All the rules here.

7 Arguments to main

The `main` function can accept two arguments: an integer representing the number of arguments and an array of strings holding the arguments themselves. This allows command-line arguments to be passed to a C++ program.

Source code

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < argc; i++) {
    std::cout << "Argument " << i << ": " << argv[i] << "\n";
  }
}
```

Advice

Once you need to parse command-line arguments in a somewhat more sophisticated way, you should consider using a library like args or argparse.

8 Exercises

8.1 Enumerating

Goal

Create an enumeration called `Direction` with four values: north, east, south, and west. Write a function that takes a `Direction` as an argument and returns a string with the direction's name. If the direction is not recognized, the function should treat this as an error.

Learning goal

Practice error handling.
Practice the usage of `enum`.
Practice the usage of `std::optional`.

Milestone

Write an `enum class` called `Direction` with four values: north, east, south, and west.
We are going to write several versions of a function called `directionToString` that takes a `Direction` as an argument and returns a string with the direction's name.
The difference between the different versions is how they handle errors:

1. The first version should return an empty string if the direction is not recognized.
2. The second version should throw an exception if the direction is not recognized.
3. The third version should return an `std::optional<std::string>`, returning an empty optional if the direction is not recognized.
4. Can you think of another way to handle the error? (like defining it out of existence)

Write the three functions and demonstrate their usage and how you handle their returns/error conditions.
What is the best way to handle errors in this case?

hint

- Given that the core functionality (enum to string) is the same in all cases, it might be a good idea to separate the error handling code from the "main" functionality. For instance, you could write the subsequent versions as wrappers around the first one.
- Remember that the main goal when writing/extending/modifying code is to reduce complexity. Note that this does not mean AT ALL neglecting error handling or input sanitizing.

8.2 Parsing input

Goal

Imagine that you have a program that can take arguments of any kind from the CLI, acting upon each one depending on the type. For instance, if the argument is an integer, it should print "Integral received: [value]", if it is a floating point number, it should print "Floating point received: [value]", and if it is a string, it should print "Unrecognized type received (assuming string): [value]".
Complete such a program (start from the milestone) and make sure that you can run it with the expected output:

Source code

```
$ g++ -std=c++20 -o parse parse.cpp
$ ./parse int 42
      Integral received: 42
$ ./parse double 3.14
      Floating point received: 3.14
$ ./parse string Hello
      Unrecognized type received (assuming string): Hello
```

Learning goal

Practice with `std::any` and taking arguments from the CLI, remember string handling with `std::stringstream`.
Showcase C++20 concepts (template type constraints).

Milestone

Complete this code by filling in the comments. You may need to include some headers.

Source code

```cpp
#include <iostream>
template<std::convertible_to<std::string> T>
void do_something(T value){
  std::cerr << "Unrecognized type received (assuming string): ";
  std::cerr << value << std::endl;
}

template<std::integral T>
void do_something(T value){
  std::cout << "Integral received: " << value << std::endl;
}

template<std::floating_point T>
void do_something(T value){
  std::cout << "Floating point received: " << value << std::endl;
}

auto parse(std::string type, std::string value){
  /*Your code here*/ result;
  /*Your code here*/
  return result;
}

int main(int argc, char *argv[]){
  if(argc < 3){
    std::cout << "Usage: " << argv[0] << " [type] [value]" << std::endl;
    return 1;
  }
  auto result = parse(argv[1], argv[2]);
  if(result.has_value()){
    if(result.type() == typeid(int)){
      int i = /*Your code here*/;
      do_something(i);
    } else if(result.type() == typeid(double)){
      double d = /*Your code here*/;
      do_something(d);
    } else{
      std::string s = /*Your code here*/;
      do_something(s);
    }
  } else {
    std::cerr << "Failed to parse value" << std::endl;
  }
  return 0;
}
```

hint

- Remember about stringstream.
- Use std::any.

8.3 Scarce resources

Goal

Part of your application runs in a highly constrained environment, so much so in fact that in the particular function you are going to write there is allowance for just one single register. A register in this context represents a variable with a "small" size (`int`, `double`, `float`, ...).
Your function, though, really needs to use variables of several types in order to do its work.
It needs to call some initalization, which returns an integer value that must be used to gather some data, which in turn is used to perform a computation.

Advice

Although the different operations are not really complex in the snippet I provide (I needed to fit them in a few lines for showcasing), this particular contraption comes from a real situation I have encountered many times writing low level CUDA code. See here for instance. They use an `union` here, its old code.

Learning goal

Practice with `std::variant`.
Showcase C++20 concepts (template type constraints).
Showcase `static_assert` and `assert`.

Milestone

Complete the following code snippet so that it compiles and runs, modify only the lines marked with "Your code here".

Source code

```cpp
#include <algorithm>
#include <cassert>

int initialization(){
  // Some very complex initialization procedure
  return 42;
}

struct Data{
  int16_t data;
};

template<std::integral T>
Data get_data(T t){
  // Some very complex data retrieval procedure
  return Data{static_cast<int16_t>(t+98325)};
}

double perform_computation(Data d){
  return d.data * 1234.0;
}

template<std::floating_point T>
void check_result(T result){
  // This is a runtime check, ensures your variable has the correct type
  assert(result == (static_cast<int16_t>(42+98325))*1234.0);
}

void really_constrained_function(){
  /*Your type here*/ v;
  /*Your code here*/ = initialization();
  /*Your code here*/ = get_data(/*Your code here*/);
  /*Your code here*/ = perform_computation(/*Your code here*/);
  // This is a compile time check, ensures your variable takes only the
  ↪  size of the largest type (plus an index)
  static_assert(sizeof(v) == std::max({sizeof(int), sizeof(double),
  ↪  sizeof(Data)}) + sizeof(v.index()));
  check_result(/*Your code here*/);
}

int main(){
  really_constrained_function();
  return 0;
}
```

hint

- You might need to include some header.

Advanced milestone

It is possible that the `static_assert` fails in your system. Depending on the provided arguments to the variant, the size of it might not be the sum of the sizes of the types it can hold and be larger.
Can you explain why?