Idiomatic Modern C++ for Linux

Week 4: Advanced Type Concepts

Today's agenda

- type deduction
- type aliases
- const
- constexpr variables and functions
- conversions and type casting

The ‘auto’ keyword

- All objects need to be of an explicit type, however, some literals have an implicitly determined type depending on their format.
- The ***auto*** keyword can be used in place of a type specifier to deduce the type of object from its initializer.
- auto can also be used to deduce the return type of a function, in the function definition itself. (cursed)
- Can be a handy tool when it increases readability

The ‘auto’ keyword

```
// Default literal values
auto d {4.6}; // default floating point type is a double, NOT float
auto i {46}; // int
auto b {true}; // bool
auto c {'^'}; // char
auto cstr = "hi, I'm a C string!"; // const char[20]
```

auto dos and don’ts

Do use auto:

- when it increases the readability of your code

```cpp
std::unique_ptr<std::vector<int>> my_vec_ptr =
    std::make_unique<std::vector<int>>(1, 3, 5, 7);
```

```cpp
auto my_vec_ptr = std::make_unique<std::vector<int>>(1, 3, 5, 7);
```

Don’t use auto:

- when it obfuscates what’s going on with your code
- as the return type for a function, generally speaking

```cpp
auto result = ProcessSomeData(arg);
```

```cpp
// you can use 'auto' to deduce a return type,
// but there's not a lot of good reasons to do this.
// if I were a developer using an API that had an 'auto' return type,
// and needed to dig into the code to figure out a bug I was encountering,
// I'd probably not be too happy having the return type
// obscured down further in the function body
auto mul (int a, int b) {
    return a * b;
}
```

Literal suffixes

- C++ lets you specify the type of a literal value with ***literal suffixes***
- e.g. “12.4f” is a ***float* literal**, “12463159ul” is an ***unsigned long* literal**

Literal suffixes

```
// Literal suffixes
auto f {6.12f}; // 'f' indicates a float literal.
auto f2 {8.14F}; // all of these work for upper or lowercase
auto u {192436U}; // unsigned int
auto l {167204296121}; // long int
auto ll {7961l}; // long long int
auto ull {98235454llu}; // long long unsigned int

// std::string literal.
// only works with the std::string_literals namespace
auto str {"hi, I'm a std::string"s };
```

Introducing 2 new stl containers

- std::pair – ties two elements together
- Access elements by pair.first and pair.second

- std::unordered_map – stl hashmap implementation.
- Closest analogue to python dictionaries
- Key type must be hashable
- Elements are NOT sorted by key, unlike std::map
- O(1) insertion, access and deletion

Type aliases

- Type aliasing lets you create a shorthand for an existing type.
- This can be really convenient when you're using the same verbose type in multiple places and want to cut down on how much code you're writing.
- A type alias can also make the intent of your code much clearer

typedef – C style type aliases

- 'typedef' is C's way of declaring type aliases.
- Like many C features, it still exists in C++ for backwards compatibility
- Some type definitions can be harder to read
- The ‘using’ keyword has largely replaced typedef and should be preferred

```cpp
typedef std::pair<int, int> Point2i;
```

```cpp
// with typedef, FuncType is buried in the typedef and harder to read
typedef float (*FuncType)(double, std::string);
```

```cpp
using Point2i = std::pair<int, int>;
```

```cpp
// using makes FuncType much clearer
using FuncType = float(*)(double, std::string);
```

Nested type aliases

```
std::unordered_map<std::string, std::pair<double, double>> house_locations {
    {"living_room", {2.3, 5.6}},
    {"bathroom", {4.1, 4.62}},
    {"bedroom", {8.4, 9.172}},
};
```

```
using Point2d = std::pair<double, double>;
using StringToPointMap = std::unordered_map<std::string, Point2d>;
```

```
StringToPointMap house_locations {
    {"living_room", {2.3, 5.6}},
    {"bathroom", {4.1, 4.62}},
    {"bedroom", {8.4, 9.172}},
};
```

- you can also create composite type aliases using other type aliases.

about const

- Variables can be declared immutable with the 'const' keyword.
- Values can be determined either at compile-time or runtime, depending on if the value is a constant expression or returned from a function (that isn’t constexpr).
- Values declared const are stored in the executable.

```
const double PI = 3.1415926;
//PI = 3.15; // won't compile, PI declared const

const int random_number = rand(); // determined at runtime

const MyCoolStruct s {100, "gecs"};
```

Const pointers and pointers to const

```
// pointer to a const int.
// The pointer can point elsewhere,
// but the data it points to can't be modified
const int* ptr2const = &random_number;
ptr2const = &val2;
//*ptr2const = 4;
```

```
// const pointer to an int.
// the pointer can't point elsewhere,
// but the data it points to can be modified
int* const constptr = &val;
// constptr = &val2;
*constptr = 5;
```

Const references

- Const references let you refer to some original value by its location in memory, but prevents mutation and unnecessary copying
- Most of the time, we'll end up using const references when accessing values or passing to a function

```cpp
// When iterating through containers,
// it's best practice to use const references
// if you don't need to modify the underlying data.
std::unordered_map<std::string, unsigned int> some_hashmap {
    {"foo", 81},
    {"abc", 123},
    {"I love strings", 9999999}
};

// range based for loop, using an unordered map iterator
for (const auto& it : some_hashmap) {
    cout << it.first << ", " << it.second << endl;
}
```

```cpp
void my_cool_function(const MyCoolStruct& my_struct) {
    my_struct.n = 45; // won't compile, my_struct is a const reference
}
```

Bonus – Structured bindings

- C++17 introduced **structured bindings**
- Syntactic sugar to make value unpacking easier, and the “cool” for–loops even cooler

```cpp
// structured binding syntax - new in C++17
for (const auto& [s, n] : some_hashmap) {
    cout << s << ", " << n << endl;
}
```

```cpp
const MyCoolStruct s {100, "gecs"};
auto [hundred, gecs] = s;
```

Introducing constexpr

- The ***constexpr*** keyword declares that an expression can be evaluated at compile-time.
- This makes it possible to compute expressions and store them *before your program even runs*
- Save CPU cycles by offloading computation to the compiler
- Constexpr can apply to
  - Variables
  - Functions
  - Templates (template metaprogramming)
  - Conditional compilation

Constexpr variable rules

You can initialize a variable as constexpr iff:

- The declaration is also a definition
- It’s a literal type
- It’s initialized by the declaration
- The full expression of its initialization is constexpr

Constexpr variable examples

```cpp
constexpr double TWO_PI = 6.28318530718;
```

```cpp
// Struct aggregates
// and classes with only constexpr constructors and member methods
// can also be evaluated at compile-time.
constexpr MyCustomPoint2iStruct p1 {12, 4};
```

Constexpr function rules

- Non-void return type
- Define with constexpr before the function definition
- No non-const-qualified types in variable definitions, unless they’re initialized with constexpr
- Only calls other constexpr functions
- Must produce a constexpr when called with a constexpr

Constexpr function examples

- In order to run at compile time, all args must be known at compile time

```cpp
// fib can evaluate either at compile time or runtime.
constexpr long fib(long n) {
    return (n < 2) ? 1 : fib(n-1) + fib(n-2);
}
```

```cpp
using namespace std::chrono;

auto constexpr_start = high_resolution_clock::now();
// 24 is a constant literal, so fib will evaluate during compile-time here.
constexpr long compiletime_fib_result = fib(24);
auto constexpr_stop = high_resolution_clock::now();
auto constexpr_duration = duration_cast<nanoseconds> (constexpr_stop - constexpr_start);

auto nonconstexpr_start = high_resolution_clock::now();
// a was not declared constexpr, so fib will execute at runtime here
long a {24};
const long runtime_fib_result = fib(a);
auto nonconstexpr_stop = high_resolution_clock::now();
auto nonconstexpr_duration = duration_cast<nanoseconds> (nonconstexpr_stop - nonconstexpr_start);
```

```
nanoseconds took to calculate fib(24) (75025) during compile-time: 29
nanoseconds took to calculate fib(24) (75025) during runtime: 217467
```

Constexpr function examples

- Functions that can evaluate at compile–time are also great for initializing objects whose size ***must be known*** at compile time.

```
constexpr int mul (int a, int b) {
    return a * b;
}
```

```
constexpr std::array<int, mul(3, 9)> my_arr {1, 2, 3, 4, 5};
```

A word on consteval (C++20 onwards)

- constexpr tells the compiler that an expression *may* evaluate at compile-time, but doesn't enforce it.
- ***consteval*** enforces compile-time evaluation for a function, if for whatever reason you require a function to only ever evaluate at compile-time and not runtime

Conversions – implicit conversion

- Implicit conversion is done automatically by the compiler in contexts where the **expected** type differs from the **actual** type
- Doesn’t change the underlying data
- Implicitly involves a temporary copy

Conversions – implicit conversion

- Data is represented by different objects in different ways
- The bits from n are copied into f, but the underlying bit representation of float differs from int
- the code is certainly not gonna display "51"
- Be careful with **memcpy** as it can result in undefined behavior

```
int n{51};
float f;
std::memcpy(&f, &n, sizeof(float));
cout << "float memcpy'd over to an int: " << f << endl;
```

```
float memcpy'd over to an int: 7.14662e-44
```

Implicit conversion examples

```cpp
// implicitly converts integer literal 3 to a double
double d{3};
// implicitly converts integer literal 6 to a doubles
d = 6;
```

```cpp
std::function<void(const long)>some_other_fn([](const long n) {});

int a = 4;
some_other_fn(a); // int implicitly converted to long
```

```cpp
std::function<float()>some_fn([] {
    // double literal implicitly converted to float
    return 3.00000001;
});

float f = some_fn();
```

```cpp
// integer values converted to a boolean
if (17) {
    cout << "integers that are nonzero are implicitly \"true\""<<endl;
}
if (0) {
    cout << "this code shouldn't be reached" << endl;
}
```

Implicit conversion can fail

- A conversion will fail if there’s no known way to convert a type to another

```
// won't compile. can't convert const char to int
int bad_convert{"26"};
```

Explicit conversion: C-style casting

- The old C syntax is still present in C++ for backwards compatibility
- There's also "function style" casting syntax present in C++
- C style casting should be avoided in favor of static_cast
  - It may perform either a static_cast, const_cast or reinterpret_cast, possibly introducing undefined behavior
  - Harder to read and search for in code
- Only one use case not covered by C++ casts: can convert a derived object to an inaccessible base class (e.g. private inheritance)

```cpp
// old way, C Style cast
double a = (double)42;
// alternate "function style" cast
double b = double(42);
```

Explicit conversion: static_cast

- Explicitly convert a value of one type to a value of another type
- Does compile–time check for if the conversion can be made
- Intentionally less powerful than C style casting, so can’t reinterpret a type or cast away constness

```cpp
// Modern way, static_cast (preferred way in C++)
double c = static_cast<double>(42);
```

Explicit conversion: reinterpret_cast

- Used to reinterpret the underlying *bit pattern* of some type to another type
- Primarily used for pointer conversion
- Can be extremely dangerous and introduce undefined behavior if used incorrectly
- Don’t use unless you can’t use static_cast or const_cast

```
// Reinterpret cast is primarily used to convert pointer types.
int n {51};

// We create a pointer to a float, and store the address of d (an integer),
// but converted as if it were a pointer to a float.
// You'll notice this yields the same result as our memcpy example from earlier.
// Reinterpret cast can introduce undefined behavior and should be avoided whenever possible.
float* f = reinterpret_cast<float*>(&n);
cout << "pointer to int reinterpret_casted as a pointer to float: " << *f << endl;
```

```
pointer to int reinterpret_casted as a pointer to float: 7.14662e-44
```

reinterpret_cast used in production

- Now that I’ve just told you to never use reinterpret_cast, here’s an example of fast square root inverse not unlike the one used in Quake
- Does some wacky pointer reinterpretation combined with one iteration of least squares error minimization

```
float fast_inv_sqrt(float x, int num_gauss_newton_iterations = 1)
{
    float y = x; // current guess y = sqrt(x)

    // i points to current guess y, interpreted as uint32
    uint32_t* i = reinterpret_cast<uint32_t*>(&y);
    constexpr uint32_t exp_mask = 0x7F800000; // 0xFF<<23
    constexpr uint32_t magic_number = 0x5f000000; // 190<<23

    // initial guess using magic number
    *i = magic_number - ((*i >> 1) & exp_mask);

    // refine guess using some Newton iterations
    for (size_t i = 0; i < num_gauss_newton_iterations; ++i)
    {
        y = (x * y * y + 1) / (2 * x * y);
    }

    return y;
}
```

Numeric promotion

- C++ is designed to be portable, and often it's more efficient to process a wider type
- We can just write functions that take an int and double, and "promote" more narrow types, without explicitly defining a function with 'short', 'char', etc

```cpp
short s{12};
print_integer(s); // numeric promotion of short to int
print_integer('a'); // numeric promotion of char to int

// "true" will convert to 1 when promoted to an int
// whereas "false" converts to 0
print_integer(true);

print_double(8.6f); // float literal promoted to double
print_double(96.523124321);
```

Narrowing conversions

- It’s best to explicitly static_cast to the narrower type if you need to
- Be careful about loss of precision or loss of information

```cpp
double d{3.1415};
// should be a compiler warning with stricter linting
// implicit narrowing conversion
print_integer(d);

// explicitly narrowing, so no compiler warning
print_integer(static_cast<int>(d));
```

Constexpr and narrowing conversions

- Some constexpr conversions aren't considered narrowing by the compiler.
- Since constexpr asserts that a value is known at compile time, we won't get compiler errors so long as the source value is preserved.

```cpp
int n1{ 5 };
unsigned int u1 { n1 };
```

```
[ 12%] Building CXX object CMakeFiles/conversion.dir/src/conversion.cpp.o
/code-examples/W4-advanced-type-concepts/src/conversion.cpp: In function 'void narrowin
g_conversions()':
/code-examples/W4-advanced-type-concepts/src/conversion.cpp:131:23: warning: narrowing
conversion of 'n1' from 'int' to 'unsigned int' [-Wnarrowing]
  131 |     unsigned int u1 { n1 };
      |                       ^~
[ 25%] Linking CXX executable conversion
[ 25%] Built target conversion
```

```cpp
constexpr int n1{ 5 };
unsigned int u1 { n1 };
```

```
-- Generating done (0.0s)
-- Build files have been written to: /code-examples/W4-advanced-type-concepts/build
[ 25%] Built target conversion
```

Constexpr and narrowing conversions

- Strangely, narrowing a constexpr floating point type is not technically considered narrowing by the compiler, even when there's loss of precision

```cpp
double my_extremely_precise_number {0.426421124085756};
float totally_not_narrowing{my_extremely_precise_number};
cout << totally_not_narrowing << endl;
```

```
 12%] Building CXX object CMakeFiles/conversion.dir/src/conversion.cpp.o
'code-examples/W4-advanced-type-concepts/src/conversion.cpp: In function 'void narrowin
_conversions()':
'code-examples/W4-advanced-type-concepts/src/conversion.cpp:141:33: warning: narrowing
conversion of 'my_extremely_precise_number' from 'double' to 'float' [-Wnarrowing]
  141 |     float totally_not_narrowing{my_extremely_precise_number};
      |                                 ^~~~~~~~~~~~~~~~~~~~~~~~~~~
 25%] Linking CXX executable conversion
```

```cpp
constexpr double my_extremely_precise_number {0.426421124085756};
float totally_not_narrowing{my_extremely_precise_number};
cout << totally_not_narrowing << endl;
```

```
-- Build files have been written to: /code-examples/W4-advanced-type-concepts/build
[ 12%] Building CXX object CMakeFiles/conversion.dir/src/conversion.cpp.o
[ 25%] Linking CXX executable conversion
```

Arithmetic conversion

- Certain operators require both operands to have the same type.
- if we invoke one of them with operands of different types, one of the operands will be implicitly converted to match using a set of rules called the usual arithmetic conversions.

Arithmetic conversion

```
// The above code should print out 'i, d' (for integer and double)
// but what about the following expression?
auto sum = my_int + my_double;
cout << typeid(sum).name() << ' ' << sum << endl;
```

```
// Short is not on the priority list for operator+,
// so both operands get promoted to int.
short l {9};
short r {10};

cout << typeid(l + r).name() << endl;
```

Arithmetic conversion

- Things start to get weird when we throw signedness into the mix.
- –3 is promoted to an unsigned integer that ends up being larger than 5

```
cout << std::boolalpha << (-3 < 5u) << endl;
```

```
false
```

Additional Resources

- https://medium.com/@sofiasondh/c-const-vs-constexpr-the-comparison-183f9dd92deb
- https://rmminusr.com/cursed-reinterpret-cast/
- https://en.cppreference.com/w/cpp/language/usual_arithmetic_conversions.html