Module M55

Partha Pratim Das

Programming in Modern C++

Module M55: C++11 and beyond: Non-class Types and Template Features

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Introduced several class features in C++11 with examples
- Explained how these features enhance OOP, generic programming, readability, type-safety, and performance in C++11

Module Objectives

- To introduce several features in C++11 for non-class types and templates
- To familiarize with enum class and fixed width integer
- To familiarize with variadic templates

Module Outline

Other (non-class) Types

Sources:

- `enum class`
  - enum class, isocpp.org
  - An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
  - Closer to Perfection: Get to Know C++11 Scoped and Based Enum Types
  - enum to string in modern C++11 / C++14 / C++17 and future C++20, stackoverflow.com
- Integer Types
  - Fixed width integer types, isocpp.org
  - long long – a longer integer, isocpp.org
  - Extended integer types, isocpp.org
- Generalized `unions`
  - Generalized unions, isocpp.org
- Generalized PODs
  - Generalized PODs, isocpp.org

Other (non-class) Types

Other (non-class) Types

- There have been several additions to non-class types in C++11. They include:
  - `enum class`: These solve several problems for `enum` in C++03
  - **Integer Types**: These include:
    - ▷ Fixed width integer types (as enhancements to integer types with size that is standard-defined). This comes from C99 feature
    - ▷ `long long` – a longer integer of 64 bits
    - ▷ Extended precision in integer types
  - **Generalized `unions`**: That allows rules for using `union` members with ctor / dtor / copy ops as enhancement over C++03
  - **Generalized PODs**: That defines rules for enhanced PODs in C++11
- Important features to learn
  - `enum` class
  - Fixed width integer, and
  - `long long`

enum class

- `enum classes` (also called: *new* `enums`, *strong* `enums`, *scoped* `enums`) address 3 problems with C++03 enumerations:
  - C++03 `enums` *implicitly convert to an integer*, causing errors when someone does not want an enumeration to act as an integer
  - C++03 `enums` *export their enumerators to the surrounding scope*, causing name clashes
  - The *underlying type of an* `enum` *cannot be specified* in C++03, causing confusion, compatibility problems, and makes forward declaration impossible
- `enum classes` (*strong* `enum`) are strongly typed and scoped:

```
enum Alert { green, yellow, orange, red }; // C++03 enum
enum class Color { red, blue };   // scoped and strongly typed enum
                                  // no export of enumerator names into enclosing scope
                                  // no implicit conversion to int
enum class TrafficLight { red, yellow, green };
Alert a = 7;              // error (as ever in C++03)
Color c = 7;              // error: no int->Color conversion
int a2 = red;             // okay: Alert->int conversion
int a3 = Alert::red;      // error in C++03; okay in C++11
int a4 = blue;            // error: blue not in scope
int a5 = Color::blue;     // error: not Color->int conversion
Color a6 = Color::blue;   // okay
```

enum class: Scopes

- Consider enum in C++03:

```
enum Color { Bronze, Silver, Gold };
enum Bullion { Silver, Gold };
enum Metal { Silver, Gold, Platinum };
enum CreditCard { Silver, Gold, Platinum };
```

- Silver and Gold clash in names between Color, Bullion, Metal and CreditCard. In C++11, we can use scoped enum:

```
enum class Color { Bronze, Silver, Gold };
enum class Bullion { Silver, Gold };
enum class Metal { Silver, Gold, Platinum };
enum class CreditCard { Silver, Gold, Platinum };
```

- No clash of names as enumerators of a scoped enum use a qualified name with enclosing scope:

```
Color col1 = Bronze; // error, Bronze not in scope

Color col2 = Color::Bronze;                          // OKay
if ((col2 == Color::Silver) || (col2 == Color::Gold)) // OKay
//...
```

enum class: Underlying Type

- Specification of underlying type (optional) now permitted provided every value fits the type:

```
enum Color: unsigned int { red, green, blue };
enum Weather: std::uint8_t { sunny, rainy, cloudy, foggy };

enum Status: std::uint8_t { pending, ready, unknown = 9999 }; // error! unknown does not fit size

enum Color { red, green, blue };   // okay -- type specification is optional as in C++03
```

- *Strongly typed* enums:
  - No implicit conversion to int
    - No comparing scoped enums values with ints
    - No comparing scoped enums objects of different types.
    - Explicit cast to int (or types convertible from int) okay
  - Values scoped to enum type
  - Underlying type defaults to int

```
enum class Elevation: char { low, high }; // underlying type = char
enum class Voltage { low, high };         // underlying type = int
Elevation e = low;                        // error! no low in scope
Elevation e = Elevation::low;             // okay
int x = Voltage::high;                    // error! no conversion to int
if (e) ...                                // error! no conversion to bool
if (e == Voltage::high) ...               // error! no conversion from Elevation to Voltage
```

enum class: Forward-Declaration

- enums of *known size* may be *forward-declared*:

```
enum Color;                                  // as in C++03, error!: size unknown
enum Weather: std::uint8_t;                  // okay
enum class Elevation;                        // okay, underlying type implicitly int
double atmosphericPressure(Elevation e);     // okay
```

Fixed Width Integer Types

- Size of integral types in C++03 are implementation-defined:
  - `sizeof(unsigned char) =` 1 byte: This is standard defined
  - `sizeof(char)`, `sizeof(short)`, `sizeof(int)`, etc.: unspecified
  - The following order is only guaranteed:
    `sizeof(unsigned char) <= sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)`
- C++11 provides fixed width integer types in `<cstdint>` for `N = { 8, 16, 32, 64 }`:
  - `int<N>_t` (`uint<N>_t`): For example, `int8_t` (`uint8_t`)
    - signed (unsigned) integer type with width of exactly `N` bits with no padding bits
    - signed integer type to use 2's complement for negative values
  - `int_fast<N>_t` (`uint_fast<N>_t`): For example, `int_fast8_t` (`uint_fast8_t`)
    - fastest signed (unsigned) integer type with width of at least `N` bits
  - `int_least<N>_t` (`uint_least<N>_t`): For example, `int_least8_t` (`uint_least8_t`)
    - smallest signed (unsigned) integer type with width of at least `N` bits
  - `intmax_t` (`uintmax_t`):
    - maximum-width signed (unsigned) integer type

Extended Size & Precision of integers

- What is the difference between the `int` types: `int8_t`, `int_least8_t`, and `int_fast8_t`?
  - Suppose we have a C compiler for a 36-bit system, with `sizeof(char)` = 9 bits, `sizeof(short)` = 18 bits, `sizeof(int)` = 36 bits, and `sizeof(long)` = 72 bits. Then
    - `int8_t` does not exist, because there is no way to satisfy the constraint of having *exactly 8 value bits with no padding*
    - `int_least8_t` is a `typedef` of `char`. NOT of `short` or `int`, because the standard requires the *smallest type with at least 8 bits*
    - `int_fast8_t` can be anything. It is likely to be a `typedef` of `int` if the *native* size is considered to be *fast*
- C++11 provides support for `long long` – a longer integer
  - An integer that's at least 64 bits long. For example:

    ```
    long long x = 9223372036854775807LL;
    ```
  - No, there are no `long long long`s nor can `long` be spelled `short long long`
- C++11 provides support for extended integer (precision) types with a set of rules

Generalized unions

- In C++03, a *member with a user-defined ctor, dtor, or assignment cannot be a member of a* **union**:

```
union U {
    int m1;
    complex<double> m2; // error (silly): complex has constructor
    string m3;          // error (not silly): string has an invariant maintained by ctor, copy, & dtor
};
```

- Obviously, it is illegal to write one member and then read another

```
U u;             // which constructor, if any?
u.m1 = 1;        // assign to int member
string s = u.m3; // disaster: read from string member
```

- C++11 allows a member of types with ctor and dtor. It also adds a restriction to make the more flexible unions less error-prone by encouraging the building of discriminated unions
- *Union member types are restricted*:
  - *No virtual functions*, *No references*, *and No bases* (as ever)
  - *If a union has a member with a user-defined ctor, copy, or dtor then that special function is deleted; that is, it cannot be used for an object of the union type*. This is new. For example:

```
union U1 {
    int m1;
    complex<double> m2; // okay
};
```

```
union U2 {
    int m1;
    string m3;  // okay
};
```

  - This may look error-prone, but the new restriction helps

Generalized unions

- Consider:

```
U1 u;              // okay
u.m2 = { 1, 2 }; // okay: assign to the complex member
U2 u2;             // error: the string destructor caused the U2 destructor to be deleted
U2 u3 = u2;        // error: the string copy constructor caused the U2 copy constructor to be deleted
```

- Basically, U2 is useless unless it is in a *discriminated unions*, such as:

```
class Widget { private: // Three alternative implementations represented as a union
    enum class Tag { point, number, text } type;    // discriminant
    union { point p;  /* point has constructor */ int i;
            string s; // string has default ctor, copy operations, and dtor
    }; // ...
    widget& operator=(const widget& w) { // necessary because of the string variant
        if (type==Tag::text && w.type==Tag::text) { s = w.s; // usual string assignment
            return *this;
        }
        if (type==Tag::text) s.~string();        // destroy (explicitly!)
        switch (w.type) {
            case Tag::point: p = w.p; break;     // normal copy
            case Tag::number: i = w.i; break;
            case Tag::text: new(&s)(w.s); break; // placement new
        }
        type = w.type; return *this;
    }
};
```

Generalized PODs

- A POD (*Plain Old Data*) is something that can be manipulated like a C struct, for example, *bitwise copyable* with `memcpy()`, *bitwise initializable* with `memset()`, etc.
- In C++03 a POD is decided by a set of restrictions on the features used in the definition of a `struct`:

```
struct S { int a; };                                        // Is a POD
struct SS { int a; SS(int aa): a(abs(aa)) { assert(a>=0); } }; // Not a POD in C++03; a POD in C++11
struct SSS { virtual void f(); /* ... */ };                 // Definitely not POD
```

- In C++11, `S` and `SS` are *standard layout types* (a superset of *POD types*) where the ctor does not affect the layout (so `memcpy()` would be fine), only the initialization rules do (`memset()` would be bad)
- However, `SSS` will still have the `vptr` and will not be anything like *plain old data*. C++11 defines:
  - POD Types: Check by `is_pod<T>::value` of type `bool` (deprecated in C++20)
  - Trivially Copyable Types: Check by `is_trivially_copyable<T>::value` of type `bool`
  - Trivial Types: Check by `is_trivial<T>::value` of type `bool`, and
  - Standard-Layout Types: Check by `is_standard_layout<T>::value` of type `bool`

  to deal with various technical aspects of what used to be PODs. POD is defined recursively:
  - *If all members and bases are PODs, then it is a POD*
  - *Naturally: No virtual functions, No virtual bases, No references, and No multiple access specifiers*
- In C++11, PODs is that adding or subtracting constructors do not affect layout or performance

Sources:

- **Extern templates**
  - Extern templates, isocpp.org
- **Template aliases**
  - Template aliases; Type alias, alias template (since C++11), isocpp.org
  - Type alias, alias template (since C++11)
  - Alias Templates and Template Parameters, 2021
- **Variadic templates**
  - Variadic templates, isocpp.org
  - Variadic templates in C++, Eli Bendersky, 2014
- **Local types as template arguments**
  - Local types as template arguments, isocpp.org
- **Right-angle brackets (Nested Template Closer)**
  - Right-angle brackets, isocpp.org
- **Variable templates (C++14)**
  - Variable templates, isocpp.org

Templates

Templates

- There have been several additions to templates in C++11. They include:
  - **Extern templates**: Used to suppress multiple instantiations
  - **Template aliases**: Used to make a template *just like another template*
  - **Variadic templates**: These are templates with variable number of parameters that are useful in various contexts like writing a type-safe `printf` or defining a `tuple`
  - **Local types as template arguments**: Uses for local as well as unnamed types as template arguments
  - **Right-angle brackets (Nested Template Closer)**: Fixes "`>>`" issue of C++03 for nested templates
  - **Variable templates** (C++14): Variables can now be directly templatized
- Important features to learn:
  - Variadic templates, and
  - Nested template closer

Extern Templates

- A template specialization can be explicitly declared as a way to suppress multiple instantiations. For example:

```
#include "MyVector.h"

// Suppresses implicit instantiation below --
// MyVector<int> will be explicitly instantiated elsewhere
extern template class MyVector<int>;

void foo(MyVector<int>& v) {
    // use the vector in here
}
```

- The *elsewhere* might look something like this:

```
#include "MyVector.h"
template class MyVector<int>; // Make MyVector available to clients
                              // For example, of the shared library
```

- This is basically a way of avoiding significant redundant work by the compiler and linker

Template aliases

- We can make a template *like another template* with a few of template arguments bound:

```
template<class T>
using Vec = std::vector<T, My_alloc<T>>;  // standard vector using my allocator
Vec<int> fib = { 1, 2, 3, 5, 8, 13 };     // allocates elements using My_alloc
vector<int, My_alloc<int>> verbose = fib; // verbose and fib are of the same type
```

- using is used to get a linear notation where *name is followed by what it refers to*. Also, *we can alias a set of specializations but we cannot specialize an alias*:

```
// int_exact_trait<N>::type is a type with exactly N bits
template<int> struct int_exact_traits  { typedef int type; };
template<> struct int_exact_traits<8>  { typedef char type; };
template<> struct int_exact_traits<16> { typedef char[2] type; };
// ... define alias for convenient notation
template<int N> using int_exact = typename int_exact_traits<N>::type;
int_exact<8> a = 7; // int_exact<8> is an int with 8 bits
```

- Type aliases can also be used as a different syntax for ordinary type aliases:

```
typedef void (*PFD)(double);          // C style
using PF = void (*)(double);          // using plus C-style type
using P = auto (*)(double) -> void; // using plus suffix return type
```

Template aliases: Example: Matrix

- Consider template class Matrix:

```
template <typename T, int Line, int Col>
class Matrix { ... };
```

- Matrix has 3 parameters. The type parameter T, and the non-type parameters Line, and Col
- For readability, we want to have two special matrices: a Square and a Vector. A Square's number of lines and columns should be equal. A Vector's line size should be one.

```
template <typename T, int Line>
using Square = Matrix<T, Line, Line>; // #1

template <typename T, int Line>
using Vector = Matrix<T, Line, 1>;    // #2
```

- using declares a type alias (#1 & #2). While the primary template Matrix can be parametrized in the three dimensions T, Line, and Col, the type aliases Square and Vector reduce the parametrization to the two dimensions T and Line
- Template alias creates names for partially bound templates. Using Square and Vector is easy:

```
Matrix<int, 5, 3> ma;
Square<double, 4> sq; // Matrix<double, 4, 4>
Vector<char, 5> vec;  // Matrix<char, 5, 1>
```

Variadic templates: printf

- Let us start by implementing `printf` – the most well-known *variadic function*. Consider:

```
const char* pi = "pi";
const char* m = "The value of %s is about %g (unless you live in %s)\n";
printf(m, pi, 3.14159, "Indiana"); // int printf(const char *format, ...) in C
```

- The simplest case of `printf()` is when there are *no arguments except the format string*:

```
void printf(const char* s) {
    while (s && *s) {
        if (*s==’%’ && *++s!=’%’) // make sure that there was not meant to be more args (%% for %)
            throw std::runtime_error("invalid format: missing arguments"); // from <exception>
        std::cout << *s++;
    }
}
```

- That done, we must handle `printf()` with more arguments (*recursive*):

```
template<typename T, typename... Args>                  // note the "..."
void printf(const char* s, T value, Args... args) { // recursive function. note the "..."
    while (s && *s) {
        if (*s==’%’ && *++s!=’%’) {      // a format specifier (ignore which one it is)
            std::cout << value;          // use first non-format argument
            return printf(++s, args...); // "peel off" first argument: recursive call
        }
        std::cout << *s++;
    }
    throw std::runtime_error("extra arguments provided to printf");
}
```

Variadic templates: printf

- The code *peels off* the first non-format arg. and then calls itself recursively. When there is no more non-format arg., it calls the first `printf()` – *functional programming at compile time*
- The `Args...` defines what is called a *parameter pack* – a sequence of (type/value) pairs to `peel off` arguments starting with the first:
  - When `printf()` is called with one argument, the first `printf(const char*)` is chosen
  - When `printf()` is called with two or more arguments, the second `printf(const char* s, T value, Args... args)` is chosen, with the first argument as `s`, the second as `value`, and the rest (if any) bundled into the parameter pack args for later use
  - In the call `printf(++s, args...)` the parameter pack args is expanded so that the next argument can now be selected as `value`
  - This carries on until `args` is empty so that the first `printf()` is called
- For generic functional programming, we declare and use a simple variadic template function:

```
template<class ... Types>
    void f(Types ... args); // variadic template function. That is, a function that can take
                            // an arbitrary number of arguments of arbitrary types
f();        // OK: args contains no arguments
f(1);       // OK: args contains one argument: int
f(2, 1.0);  // OK: args contains two arguments: int and double
```

Variadic templates: adder

- Let us implement a function that adds all of its arguments together:

```
template<typename T> T adder(T v) { cout << __PRETTY_FUNCTION__ << endl; return v; }

template<typename T, typename... Args> // template parameter pack: typename... Args
T adder(T first, Args... args)         // function parameter pack: Args... args
{ cout << __PRETTY_FUNCTION__ << endl; return first + adder(args...); }
```

- And we could call and trace it as: `long sum = adder(1, 2, 3, 8, 7); // 21`

```
T adder(T, Args ...) [with T = int; Args = int, int, int, int] // __PRETTY_FUNCTION__ is a trace
T adder(T, Args ...) [with T = int; Args = int, int, int]      // expansion macro in gcc
T adder(T, Args ...) [with T = int; Args = int, int]
T adder(T, Args ...) [with T = int; Args = int]
T adder(T) [with T = int]
```

- We could also call as:

```
std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
std::string ssum = adder(s1, s2, s3, s4); // "x"+"aa"+"bb"+"yy" = "xaabbyy"
```

- `adder` will accept any number of arguments, and will compile properly as long as it can apply the `operator+` to them following template and overload resolution rules
- Variadic templates are like recursive code with a base case (`adder(T v)`) and a general case which *recurses* as in `adder(args...)`
- In `adder` - the first argument is peeled off the template parameter pack into type `T` (argument first). So with each call, the parameter pack shortens by one parameter to hit the base case

Variadic templates: Example: power_square

- Let us consider another function for practice:

```
#include <iostream>

template <typename T>T square(T t) { return t * t; } // A function that 'squares' a number
template <typename T> // Our base case just returns the value
double power_sum(T t) { cout << __PRETTY_FUNCTION__ << endl; return t; }
template <typename T, typename... Rest> // Our new recursive case
double power_sum(T t, Rest... rest) { cout << __PRETTY_FUNCTION__ << endl;
    return t + power_sum(square(rest)...);
}
int main() {
    int result = power_sum(2, 4, 6);
    // 2 + power_sum(square(rest)...);
    // 2 + power_sum(square(4), square(6));
    // 2 + (square(4) + power_sum(square(rest)...))
    // 2 + (square(4) + power_sum(square(square(6)));
    // 2 + (square(4) + (square(square(6)))
    std::cout << result;
}
double power_sum(T, Rest ...) [with T = int; Rest = int, int]
double power_sum(T, Rest ...) [with T = int; Rest = int]
double power_sum(T) [with T = int]
1314
```

Variadic templates: Example: count

- Consider:

```
#include <iostream>
template<typename... Types> // declare list struct
struct Count;               // walking template. Putting { } is optional

template<> struct Count<> { // recognize end of list
    const static int value = 0;
};

template<typename T, typename... Rest> // walk list
struct Count<T, Rest...> {
    const static int value = 1 + Count<Rest...>::value;
};

int main() {
    auto count1 = Count<int, double, char>::value; // count1 = 3
    auto count2 = Count<int>::value;               // count2 = 1
    auto count3 = Count<>::value;                  // count3 = 0
    std::cout << count1 << std::endl;
    std::cout << count2 << std::endl;
    std::cout << count3 << std::endl;
}
```

- A simple way to count the number of arguments

Local types as template arguments

- In C++03, local and unnamed types could not be used as template arguments. C++11 relaxes:

```
void f(vector<X>& v) {
    struct Less { bool operator()(const X& a, const X& b) { return a.v<b.v; } };
    sort(v.begin(), v.end(), Less()); // C++03: error: Less is local
                                      // C++11: okay
}
```

- In C++11, we also have the alternative of using a lambda expression:

```
void f(vector<X>& v) {
    sort(v.begin(), v.end(), [] (const X& a, const X& b)  return a.v < b.v; ); // C++11
}
```

- It is worth remembering that naming action can be quite useful for documentation and an encouragement to good design. Also, non-local (necessarily named) entities can be reused.
- C++11 also allows values of unnamed types to be used as template arguments:

```
template<typename T> void foo(T const& t) { }
enum X { x };
enum { y };
int main() {
    foo(x);        // C++03: okay; C++11: okay
    foo(y);        // C++03: error; C++11: okay
    enum Z { z };
    foo(z);        // C++03: error; C++11: okay
}
```

“>>” as Nested Template Closer

- >> now closes a nested template when possible:

```
std::vector<std::list<int>> vi1;  // fine in C++11, error in C++03
```

- The C++03 *extra space* approach remains valid:

```
std::vector<std::list<int> > vi2; // fine in C++11 and C++03
```

- For a shift operation, use parentheses:
  - That is, “>>” now treated like “>” during template parsing:

```
constexpr int n = ... ;                            // n, m are compile-
constexpr int m = ... ;                            // time constants
constexpr std::list<std::array<int, n  >> 2 >> L1; // error in C++03: 2 shifts
                                                   // error in C++11: 1st ">>"
                                                   // closes both templates
std::list<std::array<int, (n>>2) >> L2;            // fine in C++11,
                                                   // error in C++03 (2 shifts)
```

Variable templates (C++14)

- A variable template may be introduced by a template declaration at namespace scope, where declaration declares a variable

```
#include <iostream>

template<typename T> T n = T(5); // variable template with default value: C++14

int main() { n<int> = 10; // instantiating variable template
    std::cout << n<int> << " ";    // instantiated value: 10
    std::cout << n<double> << " "; // default value: 5
}
```

- It can be constant too:

```
#include <iostream>
// variable template with constanr value
// math constant with precision dictated by actual type
template<typename T> constexpr T pi = T(3.1415926535897932384 6); // C++14
auto area_of_circle_with_radius = [](auto r) { return pi<decltype(r)> * r * r; }; // C++14
// template<class T> T area_of_circle_with_radius(T r) { return pi<T> * r * r; } // C++11

int main() { double r1 =  2.0; int r2 = 2;
    std::cout << area_of_circle_with_radius(r1) << std::endl; // for double: 12.5664
    std::cout << area_of_circle_with_radius(r2) << std::endl; // for int: 12
}
```

Module Summary

- Introduced several features in C++11 for non-class types and templates with examples
- Familiarized with important non-class types like enum class and fixed width integer
- Familiarized with important templates like variadic templates