



Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

Programming in Modern C++

Module M48: C++11 and beyond: General Features: Part 3

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

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

- Introduced following **C++11** general features:
 - Initializer List
 - Uniform Initialization
 - Range for Statement

NPTEL



Module Objectives

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- Introducing following **C++11** general features:
 - constexpr (+ **C++14**)
 - noexcept
 - nullptr
 - Inline namespace
 - static_assert
 - User-defined Literals (+ **C++14**)
 - Digit Separators and Binary Literals (+ **C++14**)
 - Raw String Literals
 - Unicode Support
 - Memory Alignment
 - Attributes (+ **C++14**)



Module Outline

Module M48

Partha Pratim
Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

- 1 **constexpr**: Evaluate constant expressions at compile-time
- 2 **noexcept**: To prevent Exception Propagation
- 3 **nullptr**: null Pointer Literal
- 4 **Inline namespaces**: Efficient Version Management
- 5 **static_assert**: Compile-time Assertions
- 6 **User-defined Literals**: UDTs closer to Built-in Types
- 7 **Digit Separators and Binary Literals**
- 8 **Raw String Literals**
- 9 **Unicode Support**
- 10 **Memory Alignment**
- 11 **Attributes**
- 12 **Module Summary**



constexpr: Evaluate constant expressions at compile-time

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

Sources:

- [constexpr](#), [isocpp.org](#)
- [Demystifying constexpr](#), 2016
- [An Overview of the New C++ \(C++11/14\)](#), Scott Meyers Training Courses
- [Understanding constexpr specifier in C++](#), [geeksforgeeks.org](#)
- [Difference between 'constexpr' and 'const'](#), [stackoverflow.com](#), 2013
- C++20 consteval specifier

constexpr: Evaluate constant expressions at compile-time



constexpr

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

- The `constexpr` mechanism
 - provides more *general constant expressions*
 - allows constant expressions involving *user-defined types*
 - provides a way to guarantee that an *initialization is done at compile time*

```
enum Flags { good = 0, fail = 1, bad = 2, eof = 4 };  
constexpr int operator|(Flags f1, Flags f2)  
{ return Flags(int(f1) | int(f2)); }
```

```
void f(Flags x) {  
    switch (x) {  
        case bad:      /* ... */ break;  
        case eof:      /* ... */ break;  
        case bad|eof:  /* ... */ break;  
        default:       /* ... */ break;  
    }  
}
```



constexpr

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- Here `constexpr` says that the function must be of a simple form so that it *can be evaluated at compile time if given constant expressions arguments*
- In addition to be able to evaluate expressions at compile time, we want to be able to require expressions to be evaluated at compile time
- `constexpr` in front of a variable definition does that (and implies `const`):

```
constexpr int x1 = bad|eof;    // okay

void f(Flags f3) {
    constexpr int x2 = bad|f3; // error: cannot evaluate at compile time
    int x3 = bad|f3;           // okay
}
```

- Recall the use of `constexpr` in `std::initializer_list` in **Module 47**



constexpr

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- Typically we want the compile-time evaluation guarantee for *global or namespace objects*, often for objects we want to place in *read-only storage*
- This also works for objects for which the constructors are simple enough to be **constexpr** and expressions involving such objects:

```
struct Point {  
    int x, y;  
    constexpr Point(int xx, int yy) : x(xx), y(yy) { }  
};
```

```
constexpr Point origo(0,0);  
constexpr int z = origo.x;  
constexpr Point a[] = { Point(0,0), Point(1,1), Point(2,2) };  
constexpr int x = a[1].x;    // x becomes 1
```

- Note that the constructor can still be used in the usual way with non-constant parameters too



constexpr and const

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

- Please note that `constexpr` is not a general purpose replacement for `const` (or vice versa)
 - `const`'s primary function
 - ▷ is to express the idea that an object is not modified through an interface (even though the object may very well be modified through other interfaces)
 - ▷ It just so happens that declaring an object `const` provides excellent optimization opportunities for the compiler
 - ▷ In particular, if an object is declared `const` and its address is not taken, a compiler is often able to evaluate its initializer at compile time (though that's not guaranteed) and keep that object in its tables rather than emitting it into the generated code
 - `constexpr`'s primary function
 - ▷ is to extend the range of what can be computed at compile time, making such computation type safe and also usable in compile-time contexts (such as to initialize enumerator or integral template parameters)
 - ▷ Objects declared `constexpr` have their initializer evaluated at compile time
 - ▷ they are basically values kept in the compiler's tables and only emitted into the generated code if needed



constexpr and const

- `constexpr` needs compile-time constant for initialization whereas `const` treats the initialized value as constant in run-time

```
#include <iostream>

constexpr int m = 100;           // Okay: m is 100: compile-time constant
                                // const will also work

void f(int n) {
    constexpr int c1 = m + 1;    // Okay: c1 is 101: compile-time constant

    // constexpr int c2 = n + 1; // Error: n is not compile-time constant
    const int c2 = n + 1;        // Okay: but value of c2 cannot be changed

    // constexpr int c3 = c2 + 1; // Error: c2 is not compile-time constant
    const int c3 = c2 + 1;        // Okay: but value of c3 cannot be changed

    std::cout << c1 << ' ' << c2 << ' ' << c3 << std::endl; // 101 11 12
}

int main() { f(10); }
```



Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- `static` specifies the lifetime of the variable
- A `static constexpr` variable has to be set at compilation, because its lifetime is the the whole program
- Without the `static` keyword, the compiler is not bound to set the value at compilation, and could decide to set it later
- The most powerful thing about constant expressions, is that they enable us to do meta-programming without resorting to templates. So we can write a compile time function that computes factorial in a straightforward way:

```
constexpr unsigned int factorial(unsigned int n) {  
    return (n <= 1) ? 1 : (n * factorial(n - 1));  
}
```

```
static constexpr auto magic_value = factorial(5);
```



- `constexpr` cannot be used for all functions. For example:

```
constexpr int add_vectors_size(const vector<int>& a, const vector<int>& b)
{ return a.size() + b.size(); } // a.size() is not compile time constant
```

gives compilation error on `a.size()` as size of a vector is not compile time constant

- However, the following works fine:

```
#include <iostream>
#include <array> // Fixed size array
using namespace std;

template<size_t N1, size_t N2>
constexpr int add_arrays_size(const array<int, N1>& a, const array<int, N2>& b)
{ return a.size() + b.size(); } // a.size() is compile time constant

int main() { array<int, 10> p; array<int, 20> q;
    static constexpr auto n = add_arrays_size(p, q);
    cout << n << endl; // 30
}
```



constexpr (C++14)

Post-Recording

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- In **C++11**, to make a function **constexpr** can mean rewriting it. Consider:

```
constexpr int my_charcmp(char c1, char c2) { return (c1==c2)? 0: (c1<c2)? -1: 1; }
```
- That is useful for characters. Can we extend it to strings? That would require iteration over the characters of the string, which **C++11** did not allow in **constexpr** functions, so the **C++11** version that supports strings would have to be recursive
- **C++14** allows more things inside the body of **constexpr** functions, notably:
 - local variables (not **static** or **thread_local**, and no *uninitialized variables*)
 - mutating objects whose lifetime began with the constant expression evaluation
 - **if**, **switch**, **for**, **while**, **do-while** (not **goto**)
- So in **C++14**, the above function generalized to strings can use a normal loop directly:

```
constexpr int my_strcmp(const char* str1, const char* str2) { int i = 0;
    for( ; str1[i] && str2[i] && str1[i] == str2[i]; ++i) { }
    if(str1[i] == str2[i]) return 0;
    if(str1[i] < str2[i]) return -1;
    return 1;
}
```
- **C++14** also removes the **C++11** rule that **constexpr** member functions are implicitly **const**



noexcept: To prevent Exception Propagation

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

noexcept: To prevent Exception Propagation

Sources:

- [noexcept to prevent exception propagation](http://isocpp.org), isocpp.org
- [An Overview of the New C++ \(C++11/14\)](#), Scott Meyers Training Courses



noexcept

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

- If a function cannot throw an exception or if the program is not written to handle exceptions thrown by a function, that function can be declared **noexcept**:

```
extern "C" double sqrt(double) noexcept; // will never throw
```

```
// Not prepared to handle memory exhaustion
```

```
vector<double> my_computation(const vector<double>& v) noexcept {  
    vector<double> res(v.size()); // might throw  
    for(int i; i<v.size(); ++i) res[i] = sqrt(v[i]);  
    return res;  
}
```

- If a function declared **noexcept** throws (so that the exception tries to escape the **noexcept** function)
 - the program is terminated by a call to **std::terminate()**
 - the call of **terminate()** cannot rely on objects being in well-defined states, that is, there is
 - ▷ no guarantee that destructors have been invoked
 - ▷ no guaranteed stack unwinding, and
 - ▷ no possibility for resuming the program as if no problem had been encountered
 - This is deliberate and makes **noexcept** a simple, crude, and very efficient mechanism
 - ▷ much more efficient than the old dynamic **throw()** exception specification mechanism



noexcept

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- It is possible to make a function *conditionally noexcept*. For example, an algorithm can be specified to be *noexcept* iff the operations it uses on a template argument are *noexcept*:

```
template<class T>
// can throw if f(v.at(0)) can
void do_f(vector<T>& v) noexcept(noexcept(f(v.at(0)))) {
    for(int i; i<v.size(); ++i)
        v.at(i) = f(v.at(i));
}
```

- Here, we use *noexcept* as an operator:
 - noexcept(f(v.at(0)))* is true if *f(v.at(0))* cannot throw, that is,
 - if the *f()* and *at()* used are *noexcept*
- The *noexcept()* operator is a
 - constant expression* and
 - does not evaluate its operand*



noexcept

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- The general form of a `noexcept` declaration is
 - `noexcept(expression)`
 - *plain noexcept* is simply a shorthand for `noexcept(true)`
 - All declarations of a function must have compatible `noexcept` specifications
- A destructor should not throw
 - a generated destructor is implicitly `noexcept` (independently of what code is in its body) if all of the members of its class have `noexcept` destructors (which they too will have by default)
- It is typically a bad idea to have a *move operation throw*
 - declare those `noexcept` wherever possible
 - A *generated copy or move operation* is implicitly `noexcept` if all of the copy or move operations it uses on members of its class have `noexcept` destructors
- `noexcept` is *widely and systematically* used in the standard library to *improve performance and clarify requirements*



nullptr: null Pointer Literal

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

nullptr: null Pointer Literal

Sources:

- [nullptr – a null pointer literal](http://isocpp.org), isocpp.org
- [An Overview of the New C++ \(C++11/14\)](#), Scott Meyers Training Courses
- [NULL](http://cppreference.com), cppreference.com (C)
- [NULL](http://cppreference.com), cppreference.com (C++)

Programming in Modern C++

Partha Pratim Das

M48.18



nullptr

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- `nullptr` is a literal denoting the null pointer
 - Literal of type `std::nullptr_t` in `<stddef>`
 - Convertible to any pointer type and to `bool`, but nothing else
 - It is not an integer and cannot be used as an integral value
- `nullptr` is provided to replace the macro `NULL`
 - C Implementations (`<stddef.h>`, and others)

```
#define NULL 0 // C++ compatible
```

```
#define NULL (10*2 - 20) // C++ incompatible
```

```
#define NULL ((void*)0) // C++ incompatible
```
 - C++ Implementations (`<stddef>`, and others)

```
#define NULL 0 // C++03. May be 0L in some compiler
```

```
#define NULL nullptr // C++11
```
- `NULL` or `0` causes confusion in following cases that `nullptr` can resolve:
 - **Function Overload Resolution**
 - **Forwarding Templates**



nullptr: Function Overload Resolution & Forwarding Template

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

// Simple Examples

```
int* q = nullptr; // q is null
char* p = nullptr; // p is null
char* p1 = 0;      // 0 still works, p1 is null and p == p1
char* p2 = NULL;   // p2 is null
if (p) ...         // compiles but fails
if (p == p1) ...   // compiles and succeeds
if (q == p2) ...   // error: comparison between distinct pointer types int* and char*

void g(int);
g(nullptr);        // error: nullptr is not an int. cannot convert std::nullptr_t to int
int i = nullptr;   // error: nullptr is not an int. cannot convert std::nullptr_t to int

void f(int); void f(int*); // Function overload resolution
f(0);                   // call f(int)
f(nullptr);              // call f(int*)
f(NULL);                 // error: call of overloaded f(NULL) is ambiguous for f(int) and f(int*)

void h(int*);           // h(0) and h(nullptr) are okay
template<typename F, typename P> // Forwarding template
void logAndCall(F func, P param) {
    ... func(param); // make log entry ..., then invoke func on param
}
logAndCall(h, 0);        // error: P deduced as int, and h(int) invalid
logAndCall(h, NULL);     // error: P deduced as long int, and h(long int) invalid
logAndCall(h, nullptr);  // P deduced as std::nullptr_t, and h(std::nullptr_t) is okay
```

Programming in Modern C++



Inline namespaces: Efficient Version Management

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

Inline namespaces: Efficient Version Management

Sources:

- [Inline namespaces](#), isocpp.org
- [An Overview of the New C++ \(C++11/14\)](#), Scott Meyers Training Courses
- [Inline namespaces and usage of the "using" directive inside namespaces](#), geeksforgeeks.org, 2021



Inline namespaces

Module M48

Partha Pratim Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

- The `inline namespace` mechanism is intended to support library evolution by providing a mechanism that supports a form of versioning. Consider:

```
// file V99.h:
inline namespace V99 {
    void f(int);    // does something better than the V98 version
    void f(double); // new feature
    // ...
}

// file V98.h:
namespace V98 {
    void f(int);    // does something
    // ...
}

// file Mine.h:
namespace Mine {
    #include "V99.h"
    #include "V98.h"
}
```



Inline namespaces

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- We here have a namespace `Mine` with both the latest release (`V99`) and the previous one (`V98`). If we want to be specific, we can:

```
#include "Mine.h"
using namespace Mine;
// ...
V98::f(1); // old version
V99::f(1); // new version
f(1);      // default version
```

- The point is that the `inline` specifier makes the declarations from the nested namespace appear exactly as if they had been declared in the enclosing namespace.
- This is a very *static* and *implementer-oriented facility* in that the inline specifier has to be placed by the designer of the namespaces – thus making the choice for all users
 - It is not possible for a user of `Mine` to say:
 - ▷ *I want the default to be V98 rather than V99*



Inline namespaces:

namespace in C++03 vs. inline namespace in C++11

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

// C++03: nested namespaces

```
#include <iostream>
using namespace std;
```

```
namespace ns1 { int v1 = 2;
    namespace ns2 { int v2 = 3;
        namespace ns3 { int v3 = 5;
            }
        }
    }

int main() { // Fully qualified names must
    cout << ns1::v1 << ' ';
    cout << ns1::ns2::v2 << ' ';

    cout << ns1::ns2::ns3::v3 << endl;
}
2 3 5
```

- **Note:** If the outermost namespace (`ns1`) is `inline`, then the symbols within `ns1` are available in the global namespace. For example, `ns1::ns2::ns3::v3` can be accessed as `v3` in `main()` besides as `ns1::ns2::v3` and `ns1::v3`. This property is used in **Version Control**

// C++11: inline namespaces

```
#include <iostream>
using namespace std;
```

```
// inline namespace ns1; for global access
namespace ns1 { int v1 = 2;
    inline namespace ns2 { int v2 = 3;
        inline namespace ns3 { int v3 = 5;
            }
        }
    }

int main() { // Qualified by enclosing namespace
    cout << ns1::v1 << ' ';
    cout << ns1::v2 << ' ';
    cout << ns1::ns2::v3 << ' ';
    cout << ns1::v3 << endl;
}
2 3 5 5
```




Inline namespaces:

inline namespace effects by using namespace in C++03

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

```
// C++03: nested namespaces
#include <iostream>
using namespace std;

namespace ns1 { int v1 = 2;
    namespace ns2 { int v2 = 3;
        namespace ns3 { int v3 = 5;
            }
        }
    }

int main() { // Fully qualified names must
    cout << ns1::v1 << ' ';
    cout << ns1::ns2::v2 << ' ';

    cout << ns1::ns2::ns3::v3 << endl;
}
```

2 3 5

- **Note:** With `using namespace ns1` before `main()` the symbols within `ns1` will be in the global namespace. Like, `ns1::ns2::ns3::v3` can be accessed as `ns1::ns2::v3`, `ns1::v3`, and `v3`
- However, `using namespace ns1` belongs to the *application space*. Hence, **the choice of putting it belongs to the user and default version cannot be forced**. `inline namespace` addresses this *default enforcement* for **Version Control**

```
// C++03: inline namespace effect by using
#include <iostream>
using namespace std;

namespace ns1 { int v1 = 2;
    namespace ns2 { int v2 = 3;
        namespace ns3 { int v3 = 5;
            }
        using namespace ns3;
    }
    using namespace ns2;
} // using namespace ns1; for global access

int main() { // Qualified by using namespaces
    cout << ns1::v1 << ' ';
    cout << ns1::v2 << ' ';
    cout << ns1::ns2::v3 << ' ';
    cout << ns1::v3 << endl;
}
```

2 3 5 5



static_assert: Compile-time Assertions

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

static_assert: Compile-time Assertions

Sources:

- [static_assert: Compile-time Assertions](https://ericniebler.com/2014/07/12/static-assert/), [isocpp.org](https://ericniebler.com/2014/07/12/static-assert/)



- A static (compile time) assertion consists of a constant expression and a string literal:
`static_assert(expression, string);`
- The compiler evaluates the expression and writes the string as an error message if the expression is false (that is, if the assertion failed). For example: (More example in **Module 51**)

```
static_assert(sizeof(long)>=8, "64-bit code generation required for this library");  
struct S { X m1; Y m2; };  
static_assert(sizeof(S)==sizeof(X)+sizeof(Y), "unexpected padding in S");
```
- A `static_assert` can be useful to make assumptions about a program and its treatment by a compiler explicit. Note that since `static_assert` is evaluated at compile time, it cannot be used to check assumptions that depends on run-time values:

```
int f(int* p, int n) {  
    static_assert(p==0, "p is not null"); // error: static_assert expression  
                                         // not a constant expression  
    // ...  
}
```
- Instead, we should use a normal `assert(p==0 && "p is not null");` or test and throw an exception in case of failure



User-defined Literals: UDTs closer to Built-in Types

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

User-defined Literals: UDTs closer to Built-in Types

Sources:

- [User-defined literals](#), isocpp.org
- [User Defined Literals in C++](#), geeksforgeeks.org, 2018
- [User-defined literals](#), microsoft.com, 2021
- [An Overview of the New C++ \(C++11/14\)](#), Scott Meyers Training Courses



User-defined Literals

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

- C++ has always provided literals for a variety of built-in types:

```
123      // int
1.2      // double
1.2F     // float
'a'      // char
1ULL     // unsigned long long
0xD0     // hexadecimal unsigned
"as"     // string
```

- However, in C++03 there are no literals for user-defined types. This violates the principle that UDTs should be supported as well as built-in types are. Common requests include:

```
"Hi!"s           // std::string, not
                  // "zero-terminated array of char"
1.2i             // imaginary
123.4567891234df // decimal floating point (IBM)
101010111000101b // binary
123s            // seconds
123.56km        // not miles! (units)
1234567890123456789012345678901234567890x // extended-precision
```



User-defined Literals: Literal Operators

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- C++11 supports *user-defined literals* through the notion of *literal operators* that *map literals with a given suffix into a desired type*. For example:

```
constexpr complex<double> operator "" _i(long double d) { // imaginary literal
    return complex<double>{ 0.0, static_cast<double>(d) }; // complex is a literal type
} // Note the use of constexpr to enable compile-time evaluation
```
- Literal operator has the syntax: `<ReturnType> operator "" <Suffix> (<Parameters>)`:
 - `ReturnType` can be anything including `void`
 - `Suffix` must start with an underscore (`_`). Only the Standard Library is allowed to define literals without the underscore. Suffixes will tend to be short (like `_s` for `string`, `_i` for `imaginary`, `_m` for `meter`, and `_x` for `extended`), so different uses could easily clash. Use namespaces to prevent clashes:
 - `Parameters` can be any one of four kinds of literals
 - ▷ *Integer literal*: `unsigned long long int`
 - ▷ *Floating-point literal*: `long double`
 - ▷ *String literal*: `(const char*, size_t)`, `(const wchar_t*, size_t)`, `(const char16_t*, size_t)`, `(const char32_t*, size_t)`, or `(char const*)`,
 - ▷ *Character literal*: `char`, `wchar_t`, `char16_t`, or `char32_t`



User-defined Literals: `std::string` Literals

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

```
#include <iostream>
#include <string>
using namespace std;

std::string operator"" _s(const char* p, size_t n) { // std::string literal
    return string(p, n); // requires free store allocation
}

template<class T> void f(const T& a) {
    cout << a << endl;
}

int main() {
    f("Hello");           // pass pointer to char* => const char (&)[6]
    f("Hello"_s);         // pass (5-character) std::string object
    f("Hello\n"_s);       // pass (6-character) std::string object
}
```



User-defined Literals: `std::complex` Literals

Module M48

Partha Pratim Das

Objectives & Outlines

`constexpr`

`noexcept`

`nullptr`

Inline namespaces

`static_assert`

User-defined Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

```
#include <iostream>
#include <complex>
using namespace std;

// Note the use of constexpr to enable compile-time evaluation
constexpr complex<double> operator "" _i(long double d) { // imaginary literal
    return complex<double>{ 0.0, static_cast<double>(d) }; // complex is a literal type
}

int main() {
    auto z = 3.0 + 4.0_i;           // complex(3.0, 4.0)
    auto y = 2.3 + 5.0_i;         // complex(2.3, 5.0)
    cout << "z + y = " << z+y << endl; // z + y = (5.3,9)
    cout << "z * y = " << z*y << endl; // z * y = (-13.1,24.2)
    cout << "abs(z) = " << abs(z) << endl; // abs(z) = 5
}
```




User-defined Literals: Metric Weight Literals

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

```
#include<iostream>
#include<iomanip>
using namespace std;

// user defined literals: kg, g, and mg
long double operator"" _kg(long double x) { // KiloGram: to gram
    return x * 1000;
}
long double operator"" _g(long double x) { // Gram
    return x;
}
long double operator"" _mg(long double x) { // MiliGram: to gram
    return x / 1000;
}

int main() {
    long double weight = 3.6_kg;
    cout << weight << endl; // 3600
    cout << setprecision(8) << (weight + 2.3_mg) << endl; // 3600.0023
    cout << (32.3_kg / 2.0_g) << endl; // 16150
    cout << (32.3_mg * 2.0) << endl; // 0.0646
}
```

Programming in Modern C++



User-defined Literals: Date Literals

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

```
#include <iostream>
#include <cstring>
using namespace std;
// User-defined Date class
class Date { int date, month, year;
public: Date(int d = 1, int m = 1, int y = 0): date(d), month(m), year(y) { }
    friend ostream& operator<<(ostream& os, const Date& d) {
        os << d.date << "/" << d.month << "/" << d.year; return os;
    }
};
// Literal operator for Date
Date operator "" _ad(const char* s, size_t) { // representation of date as "dd/mm/yyyy" format
    // parsing s into dd, mm, yyyy as int
    char *str = strdup(s); // copy needed as s is const char* - strtok cannot work on s
    char *date_str = strtok(str, "/"); int date = atoi(date_str);
    date_str = strtok(NULL, "/");      int month = atoi(date_str);
    date_str = strtok(NULL, "/");      int year = atoi(date_str);
    free(str);

    return Date{ date, month, year }; // Date is a literal type
}
int main() {
    auto myDate = "08/02/2022"_ad; // Date object created from literal
    cout << myDate << endl;
}
```

Programming in Modern C++



User-defined Literals

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- The basic (implementation) idea is
 - After parsing what could be a *literal*, the compiler always checks for a *suffix*
 - The user-defined literal mechanism simply *allows the user to specify a new suffix* and what is to be done with the literal before it
 - *It is not possible to redefine the meaning of a built-in literal suffix or augment the syntax of literals*
 - A literal operator can request to get its (preceding) literal passed
 - ▷ as *cooked* (with the value it would have had if the new suffix had not been defined) or
 - ▷ as *uncooked* (as a string) by simply requesting a single `const char*` argument:

```
Bignum operator"" x(const char* p) {  
    return Bignum(p);  
}  
  
void f(Bignum);  
f(1234567890123456789012345678901234567890x);
```

Here the C-style string `"12345678901234567890123456789012345678901234567890"` is passed to `operator"" x()`. Note that we did not explicitly put those digits into a string



Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- C++11 added user-defined literals, but did not use them in the standard library

- Now some very useful and popular ones work in C++14 for `std::` types:

```
auto a_string = "hello there"s;    // type std::string
auto a_minute = 60s;               // type std::chrono::duration = 60 seconds
auto a_day    = 24h;               // type std::chrono::duration = 24 hours
```

- Note `s` means *string* when used on a *string literal*, and *seconds* when used on an *integer literal*, without ambiguity



Digit Separators and Binary Literals

Module M48

Partha Pratim
Das

Objectives &
Outlines

`constexpr`

`noexcept`

`nullptr`

Inline namespaces

`static_assert`

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

Digit Separators and Binary Literals

Sources:

- [Digit Separators](https://ericniebler.com/2015/06/01/digit-separators/), [isocpp.org](https://ericniebler.com/2015/06/01/digit-separators/)
- [Binary Literals](https://ericniebler.com/2015/06/01/digit-separators/), [isocpp.org](https://ericniebler.com/2015/06/01/digit-separators/)



- **Digit Separator**

- In **C++14**, the single-quote character ' can be used anywhere within a numeric literal for aesthetic readability. It does not affect the numeric value

```
auto million = 1'000'000;  
auto pi = 3.14159'26535'89793;
```

- **Binary Literals**

- **C++14** supports binary literals:

```
auto a1 = 42;           // ... decimal  
auto a2 = 0x2A;         // ... hexadecimal  
auto a3 = 0b101010;     // ... binary
```

- This works well in combination with the new ' digit separators, for example, to separate nybbles or bytes:

```
auto a = 0b100'0001;    // ASCII 'A'
```



Raw String Literals

Module M48

Partha Pratim
Das

Objectives &
Outlines

`constexpr`

`noexcept`

`nullptr`

Inline namespaces

`static_assert`

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

Raw String Literals

Sources:

- [Raw string literals](http://isocpp.org), isocpp.org
- [An Overview of the New C++ \(C++11/14\)](#), Scott Meyers Training Courses



Raw String Literals

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- String literals where *special* characters are *not special*:

- For example, escaped characters and double quotes:

```
std::string noNewlines(R"(\n\n)");  
std::string cmd(R"(ls /home/docs | grep ".pdf")");
```

- For example, newlines:

```
std::string withNewlines(R"  
    Line 1 of the string...  
    Line 2...  
    Line 3)");
```

- Rawness* may be added to any string encoding:

```
LR"(Raw Wide string literal \t (without a tab))"  
u8R"(Raw UTF-8 string literal \n (without a newline))"  
uR"(Raw UTF-16 string literal \\ (with two backslashes))"  
UR"(Raw UTF-32 string literal 2620 (w/o a skull & crossbones))"
```

- Raw text delimiters may be customized:

- Useful when `)` is in raw text, for example, in regular expressions:

```
std::regex re1(R"! ("operator"|"operator->")!"); // "operator()"|"operator->"  
std::regex re2(R"xyzzzy("\([A-Za-z_] \w *\)")xyzzzy"); // "(identifier)" \
```




Unicode Support

Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

Unicode Support

Sources:

- [Unicode Support in the Standard Library](#), isocpp.org, 2013
- [An Overview of the New C++ \(C++11/14\)](#), Scott Meyers Training Courses
- [A Modern C++ and Unicode primer](#), cpptutor, 2021



Module M48

Partha Pratim
Das

Objectives &
Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

- For unicode support, C++11 adds two new character types:

```
char16_t           // 16-bit character (if available); akin to uint_least16_t
char32_t           // 32-bit character (if available); akin to uint_least32_t
```

- Literals of these types prefixed with u/U, are UCS-encoded:

```
u'x'               // 'x' as a char16_t using UCS-2
U'x'               // 'x' as a char32_t using UCS-4/UTF-32
```

- C++98 character types still exist, of course:

```
'x'               // 'x' as a char
L'x'              // 'x' as a wchar_t
```

- There are corresponding string literals:

```
u"UTF-16 string literal"    // => char16_ts in UTF-16
U"UTF-32 string literal"    // => char32_ts in UTF-32/UCS-4
"Ordinary/narrow string literal" // "ordinary/narrow" => chars
L"Wide string literal"      // "wide" => wchar_ts
```

- UTF-8 string literals are also supported:

```
u8"UTF-8 string literal"    // => chars in UTF-8
```



- Code points can be specified via `\unnnn` and `\Unnnnnnnnn`:

```
u8"G clef: \U0001D11E"           // 🎵
u"Thai character Khomut: \u0E5B" // คอ
U"Skull and crossbones: \u2620"  // ☠
```

- There are `std::basic_string` typedefs for all character types:

```
std::string s1;    // std::basic_string<char>
std::wstring s2;   // std::basic_string<wchar_t>
std::u16string s3; // std::basic_string<char16_t>
std::u32string s4; // std::basic_string<char32_t>
```

- C++98 guarantees only two `codecvt` facets for conversions among encodings:

- `char` \leftrightarrow `char` (`std::codecvt<char, char, std::mbstate_t>`)
 - ▷ “Degenerate” – no conversion performed
- `wchar_t` \leftrightarrow `char` (`std::codecvt<wchar_t, char, std::mbstate_t>`)

- C++11 adds more facets for conversions among encodings:

- `UTF-16` \leftrightarrow `UTF-8` (`std::codecvt<char16_t, char, std::mbstate_t>`)
- `UTF-32` \leftrightarrow `UTF-8` (`std::codecvt<char32_t, char, std::mbstate_t>`)
- `UTF-8` \leftrightarrow `UCS-2`, `UTF-8` \leftrightarrow `UCS-4` (`std::codecvt_utf8`)
- `UTF-16` \leftrightarrow `UCS-2`, `UTF-16` \leftrightarrow `UCS-4` (`std::codecvt_utf16`)
- `UTF-8` \leftrightarrow `UTF-16` (`std::codecvt_utf8_utf16`)

▷ Behaves like `std::codecvt<char16_t, char, std::mbstate_t>`



Memory Alignment

Module M48

Partha Pratim Das

Objectives &
Outlines

`constexpr`

`noexcept`

`nullptr`

Inline namespaces

`static_assert`

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

Memory Alignment

Sources:

- [Alignment](#), [isocpp.org](#)
- [alignof operator](#), [cppreference](#)
- [alignas specifier](#), [cppreference](#)



- C++11 introduces keywords, `alignof` and `alignas`, to support control of memory alignment
- The `alignof` keyword can get a platform-dependent value of type `std::size_t` to query the alignment of the platform
- In addition, `alignas` customizes the alignment of the structure

```
#include <iostream>
#include <cstdint> // max_align_t
struct Storage { // alignof = 8, sizeof = 24
    char a; int b; double c; long long d;
};
struct alignas(std::max_align_t) AlignasStorage { // alignof = 16, sizeof = 32
    char a; int b; double c; long long d;
};
int main() {
    std::cout << alignof(Storage) << ' ' << sizeof(Storage) << std::endl;
    std::cout << alignof(AlignasStorage) << ' ' << sizeof(AlignasStorage) << std::endl;
}
```
- `std::max_align_t` requires the same alignment for each scalar type, so it has almost no difference in maximum scalars
- The result on most platforms is `long double`, so the alignment requirement for `AlignasStorage` we get here is 8 or 16



Attributes

Module M48

Partha Pratim
Das

Objectives &
Outlines

`constexpr`

`noexcept`

`nullptr`

Inline namespaces

`static_assert`

User-defined
Literals

Digit Separators
/ Binary Literals

Raw String
Literals

Unicode Support

Memory
Alignment

Attributes

Module Summary

NPTEL

Attributes

Sources:

- [Attributes](#), isocpp.org



- **Attributes** is a new standard syntax for adding optional and/or vendor specific information into source code (for example, `__attribute__`, `__declspec`, and `#pragma`)
- **C++11** attributes differ from existing syntaxes by being applicable essentially everywhere in code and always relating to the immediately preceding syntactic entity. For example:

```
void f [ [ noreturn ] ] () { // attribute is placed in [ [ ... ] ]. f() never returns
    throw "error"; // OK
}

struct foo* f [ [ carries_dependency ] ] (int i); // hint to optimizer
int* g(int* x, int* y [ [ carries_dependency ] ] );
```
- `[[noreturn]]` and `[[carries_dependency]]` are the two attributes defined in the standard
- The use of attributes should only control things that do not affect the meaning of a program but might help detect errors (`[[noreturn]]`) or help optimizers (`[[carries_dependency]]`)
- One planned use for attributes is improved support for OpenMP. For example:

```
for [ [ omp::parallel() ] ] (int i=0; i<v.size(); ++i) { } // ... This may be risky as
// semantics of a parallel loop are decidedly not the same as a sequential loop
```
- In **C++14**, the `deprecated` attribute allows marking an entity deprecated, which is still legal to use but puts users on notice that use is discouraged and may cause a compilation warning
- Applicable to the declaration of a class, typedef-name, variable, non-static data member, function, enumeration, or template specialization



Module Summary

Module M48

Partha Pratim Das

Objectives & Outlines

constexpr

noexcept

nullptr

Inline namespaces

static_assert

User-defined Literals

Digit Separators / Binary Literals

Raw String Literals

Unicode Support

Memory Alignment

Attributes

Module Summary

- Introduced following **C++11** general features:
 - constexpr (+ **C++14**)
 - noexcept
 - nullptr
 - Inline namespace
 - static_assert
 - User-defined Literals (+ **C++14**)
 - Digit Separators and Binary Literals (+ **C++14**)
 - Raw String Literals
 - Unicode Support
 - Memory Alignment
 - Attributes (+ **C++14**)