Ch 7: Expressions

CSCI 330

Overview: Operators

- Operators define computation in C++
- Operators support:
  - built-in types (e.g., arithmetic, logical, comparison)
  - Custom behavior via operator overloading

Arithmetic Operators (Unary & Binary)

- Binary:

  +, -, *, /, %

- Unary:

  -, +, * (dereference), & (address-of), ! (logical NOT)

Logical & Comparison Operators

- Logical: %%, ||, ! -- short-circuiting (right hand side evaluated if needed)
- Comparison: ==, !=, <, <=, >, >=

Assignment and Compound Assignment

- Assignment: =
- Compound: +=, -=, *=, /=, etc.
- Evaluate RHS first, then assign to LHS

Incremental and Decrement Operators

- Pre-increment: ++x → increment, then use
- Post-increment: x++ → use, then increment
- Can be **overloaded** separately

Member Access Operators

. (direct member)

-> (pointer dereference)

[] (indexing)

() (function call)

Ternary Conditional Operator

- Syntax: condition ? expr1 : expr2
- Must be type-compatible

```
int a = (flag) ? 1: 0;
```

Comma Operator

int x = (std::puts(“Log”), 42);

- Evaluates both operands: x = std::puts(“Log”)
- if true, returns 42

Operator Overloading

- Type operator+(const Type&) const;

- Prefer non-member friend for summitry
- Overload common: +, ==, [], <<

The <new> Header

- Provides:
  - Operator new
  - Operator delete
  - Global new_handler
- Included by default in <iostream>
- Use this only when building memory managers, allocators, or performance-critical systems.

Buckets and Free Store Control

- Free store is dynamically allocated memory (new, delete)
- Buckets are memory chunks grouped for allocator reuse
- Replaceable operator new/delete enable custom allocators

Uisng Our Heap/Custom Allocators

- Override Operator new/delete for types

```
void* operator new(std::size_t size);
void operator delete(void* ptr);
```

Placement New

- Constructs object at pre-allocated memory location

```
void* raw = std::malloc(sizeof(MyType));
MyType* p = new (raw) MyType();
```

- Must manually call destuctor!

Operator Precedence and Associativity

- Precedence table defines binding order
- Associativity defines direction
  - Assignment: right-to-left
  - Arithmetic: left-to-right

Evaluation Order (C++17 Guaranteed)

- Sequencing rules:

  ```
  f(g(), h());     // g() and h() are sequenced before f(), guaranteed
  ```

- Undefined Behavior (avoid):

  ```
  x = x++;      //still undefined, not guaranteed
  ```

User_Defined Literals

- Add suffixes to literals for custom types

```
constexpr int operator”” _hex(const char* str);
auto size = 42_kb;
```

Type Conversions

- Implicit: automatic by compiler

```
double x = 3;    //implicit
```

- Explicit: using static_cast

```
int y = static_cast<int>(x);    //explicit
```

C-Style Casts

- Legacy: (int)x
- Dangerous: can be any of static/reinterpret/const cast
- Prefer modern C++ casts:

```
static_cast<int>(x)
```

User_Defined Type Conversions

- Conversion Operator:

  operator std::string() const;

- Mark explicit to prevent implicit

Constant Expressions

- constexpr: evaluated at compile time
- used in:
  - Array sizes
  - Templates
  - Enums

```
constexpr int squar(intx) {return x * x;}
```

Why constexpr Matters

- enables
  - compile-time evaluation
  - safer code
  - more optimization
- safer than #define
- more powerful than const

Volatile Expressions

- Use volatile to prevent compiler reordering:

```
volatile int* p = . . . ;
int x = *p;
```

- Common in:
  - Hardware programming
  - Signal handlers
  - Multithreaded code (with caution)