COSC 151: Intro to Programming: C++

Chapter 4
Expressions

Chapter 4: Expressions

- **Objectives**
  - We should be able to
    - Understand the difference between unary and binary operators
    - Understand the context of an expression to differentiate between certain overloaded operators
    - Identify some of the available operators
    - Understand when implicit conversions may occur
    - Use explicit conversions when necessary

C4: Operator Types

- **Unary operators**
  - Work on a single operand
  - Some examples:

```
int i = 0;
int* p = &i; // unary (address-of) operator
*p = 10; // unary (dereference) operator
```

- **Binary operators**
  - Work on two operands
  - Some examples

```
int x = 10, y = 10;
if(x == y)  // binary (equality) operator
{
}

if(x < y) // binary (less than) operator
{
}
```

C4: Overloaded Symbols

- **Some symbols are overloaded, and their meaning depends on the context**

```
int x = 10;
int *px = &x; // * here means pointer-to-int
*px = 11; // * here means dereference px
int z = x * 2; // here * means multiplication
int wow = *px * x; // *px means dereference px
                   // px * x means multiplication
```

C4: l-values and r-values

- An **l-value** is something that can appear on the left hand side of an expression
- An **r-value** is something that cannot have its address taken

```
// A function that returns the absolute
// value of it's parameter
int absolute_value(int);

// here x is an l-value
// while what is returned from
// absolute_value() is an r-value
int x = absolute_value(-4);
```

- This is an incredibly complex topic to understand don't worry if you don't “get it” right now

C4: Precendence and Associativity

- **C++ has a complex set of operator precedence rules**
  - Some rules aren't surprising

```
int x = 2 + 5 * 4; // 22 not 28
```

- **There is a full operator precedence table in section 4.12 of the book**
- **Parenthesis override precedence and associativity**
  - Whenever an expression gets complicated add parenthesis to ensure it is evaluated as expected

```
int circ = pi * (r * r);
```

C4: Order of Evaluation

- **Most operators don't define the order of evaluation of operands**

```
int r = f1() * f2();
```

- **`f1()` and `f2()` both must be evaluated before the multiplication can occur**
- **But we don't know which one will be done first (`f1()` or `f2()`)**
  - So we shouldn't try to infer

C4: Order of Evaluation

- **Four operators do guarantee order of evaluation of operands**
  - Logical And (`operator&&`)

```
a() && b(); // b() only evaluated if a() is true
```

  - Logical Or (`operator||`)

```
a() || b(); // b() only evaluated if a() is false
```

  - Conditional (ternary) operator (`?:`)

```
a() ? b() : c(); // b() only evaluated if a() is true
                 // c() only evaluated if a() is false
```

  - Comma (`operator,`)

```
a(), b();  // a() evaluated before b()
```

C4: Arithmetic Operators

- **Unary Plus (seldom seen)**

  ```
  + expr
  ```

- **Unary Minus**

  ```
  - expr
  ```

- **Multiplication**

  ```
  expr * expr
  ```

- **Division**

  ```
  expr / expr
  ```

- **Remainder (Modulo) (not for floating point types)**

  ```
  expr % expr
  ```

- **Addition**

  ```
  expr + expr
  ```

- **Subtraction**

  ```
  expr - expr
  ```

C4: Arithmetic Operators (contd)

- **Behavior of these operators is largely self explanatory**

```
int a = 21 + 6; // a == 27
int b = 21 - 6; // b == 15
int c = 21 * 6; // c == 126
int d = 21 / 6; // d == 3  (the whole value of the division)
int e = 21 % 6; // e == 3  (the remainder of the division)
int f = 21 + 2 * 3; // 126, multiplication has higher precendence
```

- **Be careful of arithmetic overflow**
  - Can lead to undefined behavior

C4: Logical and Relational Operators

- **Logical Not (!)**

```
!str.empty(); // true if str is not empty
```

- **Less Than (<), Less Than or Equal (<=)**

```
x < 10; // true if x is less than 10
x <= 10; // true if x is 10 or less
```

- **Greater Than (>), Greater Than or Equal (>=)**

```
x > 10; // true if x is greater than 10
x <= 10; // true if x is 10 or less
```

- **Equality (==), Inequality (!=)**

```
x == 10; // true if x is 10
x != 10; // true if x is not 10
```

- **Logical And (&&)**

```
x && y; // true if x and y are both true, evalutates y only if x is true
```

- **Logical Or (||)**

```
x || y; // true if either x or y are true, evaluates y only if x is false
```

C4: Logical Not and Bool Conversions

- **Arithmetic and pointer types support `bool` conversions**
  - We can use these conversions and logical not to test if a value is “true” or “false”

```
int x = ...;
int* px = ...;

if(x)  // true if x is not 0
{
   // ...
}

if(px)  // true if px is not nullptr
{
   // ...
}
```

C4: Assignment Operators

- **Assignment vs. Initialization**

```
int x = 0, y = 10; // initialization of x, y (not assignment)
std::vector<int> v;

x = 1; // assigns x the value of 1
x = {1.2}; // compiler error, narrowing conversion

v = {1, 2, 3, 4, 5}; // v now holds 5 elements, 1 thru 5

x = y = 3; // Assignment can “chain”

x -= 1; // shorthand for x = x - 1
x += 3; // shorthand for x = x + 3
x *= 2; // short for x = x * 2
x /= 4; // short for x = x / 4

if(x = y) // gives x the value of y, returns true if the result is non-zero

if(y == 3) // true only if y == 3
```

C4: Increment and Decrement

- **Used to increment (or decrement) a value by 1**
  - Prefix (operator before the operand) the result is the new value of the operand
  - Postfix (operator after the operand) the result is the previous value of the operand
  - **Generally, prefer the Prefix version**

```
int x = 0;
++x; // x is now 1
int y = x++; // y is 1, x is 2
--y; // y is 0
y = x--; // y is 2, x is 1
```

C4: Member Access Operators

- **These operators provide member data access**
  - The dot operator provides access to a member from an object of class type
  - The arrow operator is synonymous with (*p).m

```
std::string s;          // s of class-type (std::string)
std::string* ps = &s; // ps of pointer-to-class type

auto sz1 = s.size(); // use . operator for member access
auto sz2 = (*p).size(); // same, via pointer p
auto sz3 = p->size(); // use -> operator for member access
```

C4: Conditional Operator

- **The conditional (or ternary) operator is a short hand for if/else**

```
string final_grade = (grade < 60) ? “fail” : “pass”;

// Same as below...

string final_grade;

if(grade < 60)
{
   final_grade = “fail”;
}
else
{
   final_grade = “pass”;
}
```

C4: Implicit Conversions

- **Some conversions “just happen” without programmer knowledge**
  - This happen in many situations and the rules are complex.
  - For now, just be aware that these can (and do) occur

```
int rad = 3;
auto circ = 3.14 * (r * r); // the result of (r * r)
                            // is promoted to double
                            // circ is of type double
if(rad) // rad is converted to bool
{
}
```

C4: Explicit Conversions

- **Sometimes, programmers want to force a conversion to an explicit type**
  - We can do this with named casts
  - `static_cast`
    - Performs a well-defined type conversion
  - `const_cast`
    - Removes const (this is a “code smell”)
  - `reinterpret_cast`
    - Performs a low level reinterpretation of the bit pattern of the operands (usually not what you want)

C4: static_cast

- static_cast allows us to force a well defined conversion that wouldn't be used without explicitly performing it

```
int one = 1, two = 2;
double half = one / two;  // 1 and 2 are integers
                          // half gets the value of 0

// explicitly request a conversion to double
double half = static_cast<double>(one) / two;
```

C4: const_cast

- **const_cast allows us to cast away const.**
  - This is best avoided, as it can easily lead to undefined behavior

```
int x = 0;
const int& crx = x; // const reference to non-const x

const_cast<int&>(crx) = 4; // OK, but ugly, x is 4

const int pi = 3.14;
const int& crpi = pi;

const_cast<int&>(crpi) = 3.14159; // UNDEFINED BEHAVIOR
```

C4: `reinterpret_cast`

- **“*Leave me alone, I know what I'm doing*” – Kimi Räikkönen**
- **reinterpret_cast removes any type checking and allows any conversion requested.**
- **It's inherently unsafe, and should only be used when you know *exactly* what you're doing**

C4: Old-Style Casts

- **Legacy code has legacy “C-Style” casts.**

```
// old (c-style cats) type(expr) or (type) expr
double half = double(one) / (double) two;
```

- **The behavior of these casts are the same as `static_cast`, `const_cast` or `reinterpret_cast`, but removes the programmers control**
  - A C-style casts always succeeds, and the type system is completely bypassed

Final Thoughts

- Unary Operators work on a single operand
- Binary Operators work on two operands
- An l-value can appear on the left hand side of an expression
- An r-value cannot have it's address taken
- Use parenthesis to ensure evaluation order

Final Thouhgs (contd)

- **Increment and Decrement operators are shorthand for adding (or subtracting) one**
- **Be aware of type conversions**
  - To bool
  - Implicit conversions
- **Use explicit conversions (casts) only when necessary**