Idiomatic Modern C++ for Linux

Week 3: Operators, Control Flow, Bit Manipulation

Today’s agenda

- Operators
- Branching
- Control flow
- Bit manipulation

Operators – Assignment

```
int a, b;
a = 10;                 // a:10, b:?
b = 4;                  // a:10, b:4
a = b;                  // a:4,  b:4
b = 7;                  // a:4,  b:7
cout << "a: " << a << ", b: " << b << endl;
```

Operators – Assignment

```
int x;
// assignment operations are expressions that can be evaluated.
// for the fundamental types, this evaluates to the one assigned in the operation.
int y = 2 * (x = 5);

cout << "y = " << y << endl;
```

Operators – Arithmetic

Basic arithmetic operations:

- addition
- subtraction
- multiplication
- division
- modulo

```
unsigned short x = 9000 + 1;
int y = 4 - 86;
float z = 56.71 * 100;
double w = x / z;
bool is_even = (x % 2) == 0;
```

Operators – Compound Assignment

- C++ provides some nice shortcuts for inplace variable modification

```
int x{5};
x += 4;  // equivalent to x = x + 4
x-=2;    //                x = x - 5
x*= 780; //                x = x * 780;
x /= 3;  //                x = x / 3;
```

Operators – Increment and Decrement

```
x++; // suffix increment
--x; // prefix increment

int prefix_x = 4;
int suffix_x = prefix_x;

int prefix_y = ++prefix_x; // prefix_x = 4, prefix_y = 4
int suffix_y = suffix_x++; // prefix_x = 4, prefix_y = 3
```

- ++x – increment x, then return x
- --x – decrement x, then return x
- x++ – copy x, increment x, then return the copy
- x-- – copy x, decrement x, return the copy

Operators – Value Comparison

```
<< a << " equals " << b <<": " << (a == b) << endl
<< a << " does not equal "<< b <<": " << (a != b) << endl
<< a << " greater than "<< b <<": "<< (a > b) << endl
<< a << " greater than or equal to "<< b <<": " << (a >= b) << endl
<< a << " less than "<< b <<": "<< (a < b) << endl;
```

```
1 equals 2: 0
1 does not equal 2: 1
1 greater than 2: 0
1 greater than or equal to 2: 0
1 less than 2: 1
```

Operators – sizeof

- The ***sizeof*** operator returns the size, in bytes, of an object or type
- The ***size_t*** type is an integral type return by ***sizeof*** that can store the maximum size of a theoretically possible object of any type (including arrays)
- size_t is mostly used when iterating through arrays
- size_t is not less than 16 bytes in width

```
template<typename T>
inline size_t get_size_in_bytes(const T& obj)
{
    return sizeof(obj);
}
```

```
cout << get_size_in_bytes(4)<<endl;
cout << get_size_in_bytes(4000)<<endl;
cout << get_size_in_bytes("abcdefgh")<<endl;
cout << get_size_in_bytes(std::string{"abcdefgh"})<<endl;
```

```
4
4
9
32
```

Control Flow – If / else statements

Basic structure of an if statement:

```
if (condition) {
    statement;
}
```

Where condition is some expression that evaluates to true or false

else if: check this condition if the previous condition was false

else: if no other conditions were met

```
int num {};
cout << "enter a number" << endl;
cin >> num;

if (num > 0) {
    cout << num << " is positive"<< endl;
}
else if (num < 0) {
    cout << num << " is negative"<< endl;
}
else {
    cout << num << " is zero"<< endl;
}
```

Control Flow – If without brackets

```
bool some_condition = false;

if (some_condition)
    cout << "some condition has been met" << endl;
    cout << "this code will always run. This isn't python!" << endl;

/*
equivalent to
if (some_condition) {
    cout << "some condition has been met"<<endl;
}
cout << "this code will always run. This isn't python!" << endl;
*/
```

- EITHER all statements inside the if expression must be bracketed, OR the line immediately following the conditional will be within the scope of the if expression.
- Exercise caution when using if with no brackets. It’s nice syntactic sugar, but can trip up if not careful

Control Flow – Multiple ifs, no else

```
int x = 4;
int y = 7;

if (x >= 4) {
    cout << "we'll reach this block of code " << endl;
}
if (y < 8) {
    cout << "and this one too, because we didn't use else if" << endl;
}
```

Control Flow – Multiple conditionals

You can use the **logical** operators to chain and check for multiple conditions.

e.g. instead of

```
if (condition) {
    if (other_condition) {
        statement;
    }
}
```

You can do

```
if (condition && other_condition) {
    statement;
}
```

```cpp
void multiple_conditionals()
{
    int x = 4;
    int y = 2;

    // Logical and: &&
    // Logical or: ||
    // Logical not: !

    // Since the first condition fails, the second one will not be checked
    if (x != 4 && y < 3) {
        cout << "this condition" << endl;
    }
    // First condition met, but second condition fails
    if (x >= 4 && y > 4) {
        cout << "that condition" << endl;
    }
    // Both conditions met, so we'll actually execute this branch
    if (x == 4 && y == 2) {
        cout << "the other condition" << endl;
    }
    // Although the first condition fails, the second condition is true so we'll branch
    if (x == 7 || y > 1) {
        cout << "the OTHER other condition" << endl;
    }
    // You can chain multiple conditionals, use parens, etc
    if (!(x < 3) && ((y >= 2) || (x < 6))) {
        cout << "the other other Other condition" << endl;
    }
}
```

Control Flow – Assignment vs equality operator

- Be careful with the "=" vs "==" operators.
- Since assignment expression returns a value, the following will compile but it's probably not what you want

```
int problems = 99;

if (problems = 100) {
    cout << "oh no, more problems" << endl;
}
```

Control Flow – Ternary operator

- The ternary operator is some syntactic sugar for conditional variable assignment
- ***result = (condition) ? first : second;***
- Equivalent to

```
if (x > y) {
    max_value = x;
} else {
    max_value = y;
}
```

```
int x = 10, y = 19;
int max_val = (x > y) ? x : y;
```

Control Flow – Switch Statements

- Pattern matching for integral types
- Syntax:

```
switch(integral_type)
{
    case some_number:
        statement;
        break;
}
```

```
void switch_statements(unsigned int num_cars)
{
    switch(num_cars)
    {
        case 0:
            cout << "lucky you, you get to bike and take transit everywhere!" << endl;
        case 1:
            cout << "yeah that's fair"
                 "especially if you have to commute a long way for work" << endl;
            break;
        case 2:
            cout << "ok I guess like maybe if you're a couple "
                 "or you need a truck for heavy lifting or something?" << endl;
            break;
        default:
            cout << "yeah you're part of the problem" << endl;
    }
}
```

Control Flow – Fallthrough Switch Statements

```
switch (2)
{
    case 1:
        cout << "this statement won't evaluate if num < 2" << endl;
    case 2:
        cout <<" but this one will "<< endl;
    case 3:
        cout <<" and this will " << endl;
    case 4:
        cout <<" this one too " << endl;
    case 5:
        cout <<" and this one " << endl;
}
```

- The absence of a ***break;*** at the end of your case will result in switch fallthrough
- Can be useful when designating one response for multiple cases, but be careful!

Control Flow – While loops

```
while (condition) {
    statement;
}
```

Where condition is some expression that evaluates to true or false

```
int i = 0;
while (i < 5) {
    cout << i << endl;
    i++;
}
```

Control Flow – do–while loops

```
do
{
    statement;
} while (condition)
```

Where condition is some expression that evaluates to true or false

NOTE that condition is assumed false from the start, so the block of code will execute at least once.

```
i = 10;
do {
    cout << i << endl;
    i++;
}
while (i < 5);
```

A brief aside to introduce std::vector

- Meet our first stl container!
- std::vector – dynamic array that’s part of the standard template library (stl)
- Storage handled by the container automatically: internally uses a contiguous dynamically allocated array
- If the array needs to grow in size, this may imply automatically reallocating a new larger array and copying the existing elements into it (expensive!)
- Pre-C++17, the type needed to be specified when instantiating, i.e. std::vector<int>, std::vector<bool>
- Nowadays, thanks to class template argument deduction, we can usually do
  - std::vector v {elements};
- (you might still wish to specify the type for readability in some cases)

Control Flow – Traditional for loops

```cpp
std::vector arr {-95, 612, 779, 124, -7};

for (size_t i = 0; i < arr.size(); i++)
{
    cout << arr[i] << endl;
}
```

```cpp
size_t j;

for (j = arr.size(); j > 0; j--)
{
    cout << arr[j] << endl;
}
```

for (init-statement; condition; end-expression)
**   statement;**

Note – size_t is an unsigned integral type that's the result of the **sizeof** operator (returns a type's size in bytes)
stores the maximum size of a theoretically possible object of any type.
Commonly used for array indexing / loop counting

Control Flow – Range-based for loops

```
for (int num : arr) // range-based for loop.
{
    cout << num << endl;
}
```

for (type item : container)
**   statement;**

Nice for iterating through stl types without needing to worry about the index

Control Flow – Const Reference range-based for loops

```cpp
// range-based for loop, with const references (avoid copy)
for (const int& num : arr)
{
    cout << num << endl;
}
```

- The range based for-loop from before was passing the elements ***by value*** **instead of *by reference***
- This means that *every element would be unnecessarily* ***copied*** when we access it inside the for loop
- We can avoid this by **passing by reference instead (int&)**
- **const** – prevents num from being mutated, since we are accessing the reference itself instead of a copy
- We'll look more at references in a future section, but it's worth introducing the concept here

Control Flow – infinite for loops

```
/*
fun fact: none of the for loop statements need to be filled in.
*/
for (;;) {
    cout << "infinite for loop ";
}
```

Control Flow – don’t use goto

- goto – unconditonal branch
- Consists of a *statement label* and *goto statement*
- Statement labels have ***function scope***.
- There’s no reason to use this in modern C++. Using unconditional jumps leads to spaghetti code and impossible to follow logic.
- Avoid using it whenever possible.

```cpp
int x{17};

// initialize a pointer to an int, but don't give it an address
int* y;

if (do_skip)
    goto hell;

// assign y to the address of x;
y = &x;

hell:
// if y remains a null pointer, this will cause the program to crash.
int z = *y;
std::cout<<z<<std::endl;
```

```cpp
loopforever: // statement label where the goto statement will branch to
cout << "you're stuck here forever!"<<endl;
goto loopforever; // this is the goto statement itself
```

Bit manipulation and bitwise operators

```cpp
std::bitset<4> x {0b1100};
// that's right, operator<< is used for both stream insertion AND left shift
cout << (x >> 1) << endl; // shift x right by 1, yielding 0110
cout << (x << 1) << endl; // shift x left by 1, yielding 1000
cout << ~x << endl; // flip all bits in x

std::bitset<4> y{0b0110};
cout << (x & y) << endl; // each bit is set when both corresponding bits in x and y are 1
cout << (x | y) << endl; // each bit is set when either corresponding bit in x and y is 1
cout << (x ^ y) << endl; // each bit is set when corresponding bits in x and y differ
```

- You can manipulate std::bitset (new with C++17) and other integral types with the **bitwise** operators
- ***Bitwise*** and / or / not: not to be confused with ***Logical*** and / or / not

Additional Resources

- https://gcc.gnu.org/onlinedocs/gcc-6.3.0/cpp/Macro-Pitfalls.html#Macro-Pitfalls
- https://en.cppreference.com/w/cpp/language/class_template_argument_deduction.html
-