Modern C++ Programming

3. Basic Concepts II

Federico Busato

University of Verona, Dept. of Computer Science
2020, v3.04

Table of Context

Enumerators

Enumerated Types

Enumerator

An **enumerator** (`enum`) is a data type that groups a set of named integral constants

```
enum color_t { BLACK, BLUE, GREEN = 2 };

color_t color = BLUE;
cout << (color == BLACK); // print false
```

The problem:

```
enum color_t { BLACK, BLUE, GREEN };
enum fruit_t { APPLE, CHERRY };

color_t color = BLACK;    // int: 0
fruit_t fruit = APPLE;    // int: 0
cout << (color == fruit); // print 'true'!!
// and, most importantly, does the match between a color and
// a fruit make any sense?
```

Enumerated Types (Strongly Typed)

enum class

C++11 introduces a *type safe* enumerator `enum class` (scoped enum) data type that are not implicitly convertible to `int`

```
enum class Color { BLACK, BLUE, GREEN = 2 };
enum class Fruit { APPLE, CHERRY };

Color color = Color::BLUE;
Fruit fruit = Fruit::APPLE;

// cout << (color == fruit); // compile error
//     we are trying to match colors with fruits
//     BUT, they are different things entirely

// int a = Color::GREEN;  // compile error
```

- `enum class` can be compared

```
enum class Color { RED = 1, GREEN = 2, BLUE = 3 };

cout << (Color::RED < Color::GREEN); // print true
```

- `enum class` does not support other operations

```
enum       color_t { RED = 1, GREEN = 2, BLUE = 3 };
enum class Color   { RED = 1, GREEN = 2, BLUE = 3 };

int v = RED + GREEN; // ok
// int v = Color::RED + Color::GREEN; // compile error
```

- The size of `enum class` can be set

```
#include <cstdint>
enum class Color : int8_t { RED = 1, GREEN = 2, BLUE = 3 };
```

- `enum class` can be explicitly converted

```
int a = (int) Color::GREEN; // ok
```

- `enum class` should be always initialized

```
enum class Color { RED = 1, GREEN = 2, BLUE = 3 };

Color my_color;  // "my_color" may be 0!!
```

- `enum class` can contain alias

```
enum class Device { PC = 0, COMPUTER = 0, PRINTER = 1 };
```

- `enum class` is automatically enumerated

```
enum class Color { RED, GREEN = -1, BLUE, BLACK };
//                (0)   (-1)       (0)   (1)
Color::RED == Color::BLUE; // true
```

- Cast from *out-of-range values* to `enum class` leads to undefined behavior (C++17)

```cpp
enum Color { RED = 0, GREEN = 1, BLUE = 2 };

int main() {
    Color value = (int) 3; // undefined behavior
}
```

- C++17 `enum class` supports *direct-list-initialization*

```cpp
enum class Color { RED = 0, GREEN = 1, BLUE = 2 };

int main() {
    Color a{2};          // ok, equal to Color:BLUE
//  Color b{4};          // compile error
//  Color c = {2};       // compile error
    Color d = Color{2}; // ok, equal to Color:BLUE
}
```

Union and Bitfield

Union 1/2

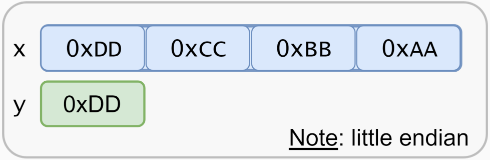
Union

A **union** (`union`) is a special data type that allows to store different data types in the same memory location

- The **`union`** is only as big as necessary to hold its *largest* data member
- The **`union`** is a kind of *"overlapping"* storage

```
union A {
  int  x;
  char y;
};
```

```
A a;
A.x = 0xAABBCCDD
```



```
union A {
    int  x;
    char y;
};  // sizeof(A): 4

A a;
a.x = 1023;    // bits: 00..000001111111111
a.y = 0;       // bits: 00..000001100000000
cout << a.x;  // print 512 + 256 = 768
```

NOTE: Little-Endian encoding maps the bytes of a value in memory in the reverse order. `y` maps to the last byte of `x`

C++17 introduces `std::variant` to represent a type-safe union

Bitfield

Bitfield

A **bitfield** is a variable of a structure with a predefined bit width.

A bitfield can hold bits instead bytes

```
struct S1 {
    int b1 : 10;  // range [0, 1023]
    int b2 : 10;  // range [0, 1023]
    int b3 : 8;   // range [0, 255]
}; // sizeof(S1): 4 bytes

struct S2 {
    int b1 : 10;
    int    : 0;   // reset: force the next field
    int b2 : 10;  //        to start at bit 32
}; // sizeof(S1): 8 bytes
```

`using`, `decltype`, and `auto`

using and decltype

- In C++11, the `using` keyword has the same semantics of `typedef` specifier (alias-declaration), but with better syntax

```
typedef int distance_t;  // equal to:
using distance_t = int;
```

- In C++11, `decltype` captures the type of an object or an expression

```
int a = 3;
decltype(a)        b = 5;    // 'b' is int
decltype(2.0f)     c = 3.0f; // 'c' is float
decltype(a + 2.0f) d = 3.0f; // 'd' is float
decltype(f(a))     e = ...;  // 'e' depends on f(a)

using T = decltype(a);       // T is int
T value = 3;
```

`auto` Keyword

The `auto` keyword (C++11) specifies that the type of the variable will be automatically deduced by the compiler (from its initializer)

```
auto a = 1 + 2;   // 1 is int, 2 is int, 1 + 2 is int!
//    -> 'a' is "int"
auto b = 1 + 2.0; // 1 is int, 2.0 is double. 1 + 2.0 is double
//    -> 'b' is "double"
```

`auto` keyword can be very useful for maintainability

```
for (auto i = k; i < size; i++)
    ...
```

On the other hand, it may make the code less readable if excessively used because of type hiding

Example: `auto x = 0;` in general makes no sense (`x` is `int`)

Math Operators

Math Operators (Precedence) 1/3

Precedence	Operator	Description	Associativity
1	`a++` `a--`	Suffix/postfix increment and decrement	Left-to-right
2	`++a` `--a`	Prefix increment and decrement	Right-to-left
3	`a*b` `a/b` `a%b`	Multiplication, division, and remainder	Left-to-right
4	`a+b` `a-b`	Addition and subtraction	Left-to-right
5	$\ll$ $\gg$	Bitwise left shift and right shift	Left-to-right
6	`<` `<=` `>` `>=`	Relational operators	Left-to-right
7	`==` `!=`	Equality operators	Left-to-right
8	&	Bitwise AND	Left-to-right
9	ˆ	Bitwise XOR	Left-to-right
10	|	Bitwise OR	Left-to-right
11	&&	Logical AND	Left-to-right
12	||	Logical OR	Left-to-right

In general:

- **Unary** operators have higher precedence than **binary operators**
- **Standard math operators** (`+, *, etc.`) have higher precedence than **comparison**, **bitwise**, and **logic** operators
- **Comparison** operators have higher precedence than **bitwise** and **logic operators**
- **Bitwise** operators have higher precedence than **logic** operators

Full table

`en.cppreference.com/w/cpp/language/operator_precedence`

Examples:

```
a + b * 4;                 // a + (b * 4)

a * b / c % d;             // ((a * b) / c) % d

a + b < 3 >> 4;            // (a + b) < (3 >> 4)

a && b && c || d;          // (a && b && c) || d

a | b & c || e && d;       // ((a | (b & c)) || (e && d)
```

Important: sometimes parenthesis can make expression worldly... but they can help!

Undefined Behavior

Expressions with undefined (implementation-defined) behavior:

```
int i = 0;
i = ++i + 2;       // until C++11: undefined behavior
                   // since C++11:  i = 3
i = 0;
i = i++ + 2;       // until C++17: undefined behavior
                   // since C++17: i = 3

f(i = 2, i = 1);   // until C++17: undefined behavior
                   // since C++17:  i = 2
i = 0;
a[i] = i++;        // until C++17: undefined behavior
                   // since C++17: a[1] = 1

f(++i, ++i);       // undefined behavior
i = ++i + i++;     // undefined behavior

n = ++i + i;       // undefined behavior
```

Statements and Control Flow

Assignment and Ternary Operator

- Assignment special cases:

```
int a;
int b = a = 3;  // (a = 3) return value 3
if (b = 4)      // it is not an error, but BAD programming
```

- *Structure Binding* declaration: C++17

```
struct A {
    int x = 1;
    int y = 2;
} a;

auto [x1, y1] = a;
cout << x1 << " " << y1;
```

- Ternary operator:

```
<cond> ? <expression1> : <expression2>
```

<expression1> and <expression2> must return a value of the same type

```
int value = (a == b) ? a : (b == c ? b : 3); // nested
```

- *Short-circuiting*:

```
if (<true expression> || array[-1] == 0)
... // no error!! even though index is -1
    // left-to-right evaluation
```

- C++17 `if` statement with *initializer*:

```
void f(int x, int y) {
    if (int ret = x + y; ret < 10)
        cout << "a";
}
```

It aims at simplifying complex statement before the condition evaluation. Available also for `switch` statements

Loops

C++ provides three kinds of loop:

- `for`

```
for ([init]; [cond]; [increment]) {
    ...
}
```

  To use when number of iterations is known

- `while`

```
while (cond) {
    ...
}
```

  To use when number of iterations is not known

- `do while`

```
do {
...
} while (cond);
```

  To use when number of iterations is not known, but there is at least one iteration

for Loop

- C++ allows "in loop" definitions:

```
for (int i = 0, k = 0; i < 10; i++, k += 2)
    ...
```

- Infinite loop:

```
for (;;)    // also while(true);
   ...
```

- Jump statements (break, continue, return):

```
for (int i = 0; i < 10; i++) {
    if (<condition>)
        break;    // exit from the loop
    if (<condition>)
        continue; // continue with a new iteration and exec. i++
    return;       // exit from the function
}
```

C++11 introduces the **range-based `for` loop** to simplify the verbosity of traditional `for` loop constructs. They are equivalent to the `for` loop operating over a range of values, but more **safe**

The range-based for loop avoids the user to specify start, end, and increment of the loop

```
for (int v : { 3, 2, 1 }) // INITIALIZER LIST
    cout << v << " ";     // print: 3 2 1

for (auto c : "abcd")    // RAW STRING
    cout << c << " ";    // print: a b c d

int values[] = { 3, 2, 1 };
for (int v : values)     // ARRAY OF VALUES
    cout << v << " ";    // print: 3 2 1
```

Range-based for loop can be applied in three cases:

- Fixed-size array `int array[3]`, `"abcd"`
- Branch Initializer List `{1, 2, 3}`
- Any object with `begin()` and `end()` methods

```
std::vector vec{1, 2, 3, 4};

for (auto x : vec) {
   cout << x << ", ";
// print:  "1, 2, 3, 4"
```

```
int matrix[2][4];
for (auto& row : matrix) {
    for (auto element : row)
        cout << ".";
    cout << "\n";
}
// print:  !!!!
//         !!!!
```

C++17 extends the concepts of **range loop** for *structure binding*

```
struct A {
    int x;
    int y;
};

A array[10] = { {1,2}, {5,6}, {7,1} };
for (auto [x1, y1] : array)
    cout << x1 << "," << y1 << "  ";  // print: 1,2  5,6  7,1
```

C++ `switch` can be defined over `int`, `char`, `enum class`, `enum`, etc.

```
char x = ...
int y;
switch (x) {
    case 'a': y = 1; break;
    default:  return -1;
}
return y;
```

```
MyEnum x
int y = 0;
switch (x) {
   case MyEnum::A:        // fallthrough
   case MyEnum::B:        // fallthrough
   case MyEnum::C: return 0;
   default: return -1;
}
```

C++17 `[[fallthrough]]` attribute

```
char x = ...
switch (x) {
    case 'a': x++;
              [[fallthrough]]; // C++17: avoid warning
    case 'b': return 0;
    default:  return -1;
}
```

Switch scope:

```
int x = 1;
switch (1) {
    case 0: int x;      // nearest scope
    case 1: cout << x;  // undefined!!
    case 2: { int y; }  // ok
//  case 3: cout << y;  // compile error
//  case 4: int x;      // compile error
}
```

When goto could be useful:

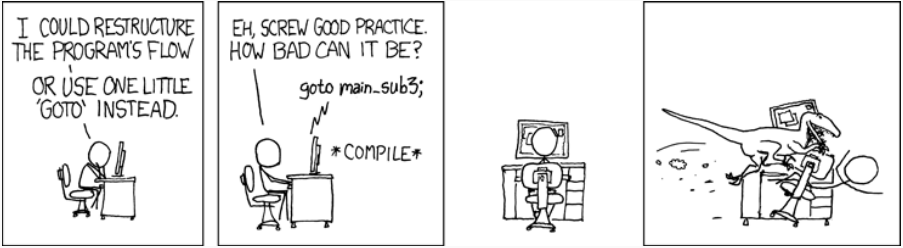
```
bool flag = true;
for (int i = 0; i < N && flag; i++) {
    for (int j = 0; j < M && flag; j++) {
        if (<condition>)
            flag = false;
    }
}
```

become:

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        if (<condition>)
            goto LABEL;
    }
}
LABEL: ;
```

Best solution:

```
bool my_function(int M, int M) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            if (<condition>)
                return false;
        }
    }
    return true;
}
```


I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE