Modern C++ Programming

3. Basic Concepts II

- Entities and Control Flow

Federico Busato

University of Verona, Dept. of Computer Science
2021, v3.11

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 };

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)

`enum class` (scoped enum) data type is a *type safe* enumerator that is not implicitly convertible to `int`

```
enum class Color { BLACK, BLUE, GREEN };
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, GREEN, BLUE };

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

- `enum class` does not support other operations

```
enum       color_t { RED, GREEN, BLUE };
enum class Color   { RED, GREEN, BLUE };

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, GREEN, BLUE };
```

Enumerated Types Features

- `enum class` can be explicitly converted

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

- `enum class` should be always initialized

```
enum class Color { RED, GREEN, BLUE };

Color my_color;  // "my_color" may be outside RED, GREEN, BLUE!!
```

- `enum class` can contain alias

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

- `enum class` is automatically enumerated in increasing order

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

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

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

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

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

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

Color a{2};     // ok, equal to Color:BLUE
// Color b{4}; // compile error
```

Enumerated Types Features (C++20)

C++20 allows to introduce the enumerator identifiers into the local scope

```
enum class Color { RED, GREEN, BLUE };

switch (x) {
    using enum Color; // C++20
    case RED:
    case GREEN:
    case BLUE:
}
```

`struct`, `union`, and Bitfield

A structure `struct` allows aggregating different variables into a single unit

```
struct A {
    int   x;
    char  y;
    float z;
};
```

C++20 introduces *designated initializer list*

```
A a1{1, 0, 2};        // a.x == 1, a.y == 0, a.z == 2
A a2{.x = 1, .z = 2}; // designated initializer list

void f(A a) {}
f({.x = 1, .z = 2})   // designated initializer list
```

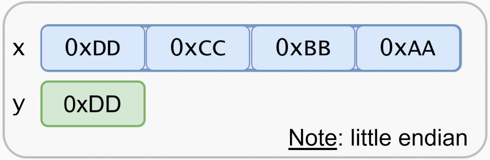
Union

A `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

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

```cpp
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

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

- C++11 The `using` keyword has the same semantic of `typedef` specifier (*alias-declaration*), but with a better syntax

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

- C++11 The `decltype` keyword 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;
```

C++11 The `auto` keyword 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` 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`) `auto` (as well as `decltype`) can be used for defining both function input C++20 and output types C++11/C++14

```
auto g(int x) -> int { return x * 2; } // C++11
// "-> int" is the deduction type
// a better way to express it is:
auto g2(int x) -> decltype(x * 2) { return x * 2; }

auto h(int x) { return x * 2; }         // C++14

void f(auto x) {}                       // C++20
// less expensive than template
//----------------------------------------------------
int x = g(3); // C++11

f(3);          // C++20
f(3.0);        // C++20
```

Control Flow

Assignment and Ternary Operator

- Assignment 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;
```

`if` Statement

- *Short-circuiting*:

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

- *Ternary operator*:
  `<cond> ? <expression1> : <expression2>`
  <expression1> and <expression2> must return a value of the same or convertible type

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

Loops

- `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

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:  @@@@
//         @@@@
```

Range-based for loop

C++17 extends the concept of **range-based 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;
```

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
}
```

Fallthrough:

```
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;
}
```

Control Flow with Initializing Statement

Control flow with **initializing statement** aims at simplifying complex actions before the condition evaluation and restrict the scope of a variable which is visible only in the control flow body

C++17 introduces `if` statement with initializer

```
if (int ret = x + y; ret < 10)
    cout << ret;
```

C++17 introduces `switch` statement with initializer

```
switch (auto i = f(); x) {
   case 1: return i + x;
```

C++20 introduces `range-for loop` statement with initializer

```
for (int i = 0; auto x : {'A', 'B', 'C'})
    cout << i++ << ":" << x;  // print: 1:A  2:B  3:C
```

When goto could be useful:

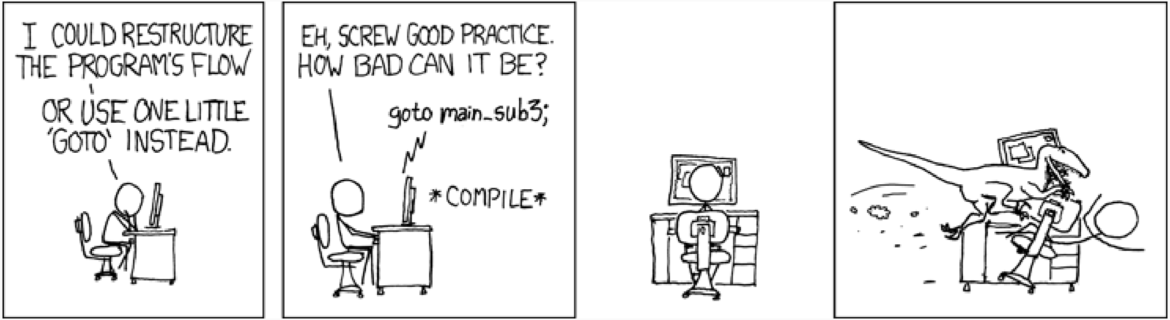
```
bool flag = true;
for (int i = 0; i < N && flag; i++) {
    for (int j = 0; j < M && flag; j++) {
        if (<condition>)
            flag = false;
    }
}
```

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