

Modern C++ Programming

4. BASIC CONCEPTS III

ENTITIES AND CONTROL FLOW

Federico Busato

2023-12-21

- 1** Entities
- 2** Declaration and Definition
- 3** Enumerators
- 4** struct, Bitfield, and union

5 Control Flow

- `if` Statement
- `for` and `while` Loops
- Range-based `for` Loop
- `switch`
- `goto`
- Avoid Unused Variable Warning [`maybe_unused`]

Entities

Entities

A C++ program is set of language-specific *keywords* (`for`, `if`, `new`, `true`, etc.), *identifiers* (symbols for variables, functions, structures, namespaces, etc.), *expressions* defined as sequence of operators, and *literals* (constant value tokens)

C++ Entity

An **entity** is a value, object, reference, function, enumerator, type, class member, template, or namespace

Identifiers and *user-defined operators* are the names used to refer to *entities*

Entities also captures the result(s) of an *expression*

Preprocessor macros are not C++ entities

Declaration and Definition

Declaration/Definition

Declaration/Prototype

A **declaration** (or *prototype*) introduces an *entity* with an *identifier* describing its type and properties

A *declaration* is what the compiler and the linker needs to accept references (usage) to that identifier

Entities can be declared multiple times. All declarations are the same

Definition/Implementation

An entity **definition** is the implementation of a declaration. It defines the properties and the behavior of the entity

For each entity, only a single *definition* is allowed

Declaration/Definition Function Example

```
void f(int a, char* b); // function declaration

void f(int a, char*) { // function definition
    ...                // "b" can be omitted if not used
}

void f(int a, char* b); // function declaration
                        // multiple declarations is valid

f(3, "abc");           // usage
```

```
void g(); // function declaration

g();      // linking error "g" is not defined
```


Declaration/Definition struct Example

A declaration without a concrete implementation is an incomplete type (as `void`)

```
struct A;    // declaration 1
struct A;    // declaration 2 (ok)

struct B {   // declaration and definition
    int b;
// A x;    // compile error incomplete type
    A* y;   // ok, pointer to incomplete type
};

struct A {   // definition
    char c;
}
```

Enumerators

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
bool    b      = (color == fruit); // print 'true'!!
// and, most importantly, does the match between a color and
// a fruit make any sense?
```

Strongly Typed Enumerator - enum class

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;
```

```
// bool b = (color == fruit) compile error we are trying to match colors with fruits
```

```
// BUT, they are different things entirely
```

```
// int a1 = Color::GREEN; compile error
```

```
// int a2 = Color::RED + Color::GREEN; compile error
```

```
int a3 = (int) Color::GREEN; // ok, explicit conversion
```

enum/enum class Features

- enum/enum class can be compared

```
enum class Color { RED, GREEN, BLUE };  
cout << (Color::RED < Color::GREEN); // print true
```

- enum/enum class are automatically enumerated in increasing order

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

- enum/enum class can contain alias

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

- C++11 enum/enum class allows to set the underlying type

```
enum class Color : int8_t { RED, GREEN, BLUE };
```

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

```
enum class Color { RED, GREEN, BLUE };  
Color a{2}; // ok, equal to Color:BLUE
```

- C++17 `enum/enum class` support *attributes*

```
enum class Color { RED, GREEN, [[deprecated]] BLUE };  
auto x = Color::BLUE; // compiler warning
```

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

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

enum/enum class - Common Errors

- enum/enum class should be always initialized

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

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

- C++17 Cast from *out-of-range values* respect to the *underlying type* of enum/enum class leads to undefined behavior

```
enum Color : uint8_t { RED, GREEN, BLUE };
```

```
Color value = 256; // undefined behavior
```


- C++17 `constexpr` expressions don't allow *out-of-range values* for (only) `enum` without explicit *underlying type*

```
enum      Color      { RED };
enum      Fruit : int { APPLE };
enum class Device    { PC };

// constexpr Color a1 = (Color) -1; compile error
const     Color a2 = (Color) -1; // ok
constexpr Fruit a3 = (Fruit) -1; // ok
constexpr Device a4 = (Device) -1; // ok
```

struct, **Bitfield**, and union

A `struct` (*structure*) aggregates different variables into a single unit

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

It is possible to declare one or more variables after the definition of a `struct`

```
struct A {  
    int x;  
} a, b;
```

Enumerators can be declared within a `struct` without a name

```
struct A {  
    enum {X, Y}  
};  
A::X;
```

It is possible to declare a `struct` in a local scope (with some restrictions), e.g. function scope

```
int f() {  
    struct A {  
        int x;  
    } a;  
    return a.x;  
}
```

Contrary to C, C++ does not allow *anonymous* `struct` (i.e. without a name)

```
struct { // anonymous struct, compile error  
    int x;  
};
```

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
```

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

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

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

x 0xDD 0xCC 0xBB 0xAA

y 0xDD

Note: little endian

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

A a;
a.x = 1023; // bits: 00..0000011111111111
a.y = 0;    // bits: 00..0000011000000000
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`

Contrary to `struct`, C++ allows *anonymous union* (i.e. without a name)

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

Control Flow

if Statement

The `if` statement executes the first branch if the specified condition is evaluated to `true`, the second branch otherwise

- *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
```

for and while 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 Features and Jump Statements

- 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 **safer**

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

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

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

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 concept of **range-based loop** for *structure binding*

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

The `switch` statement evaluates an expression (`int`, `char`, `enum class`, `enum`) and executes the statement associated with the matching case value

```
char x = ...
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
}
```

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:

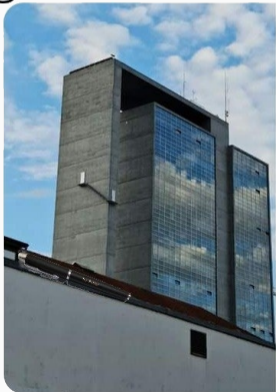
```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        if (<condition>)
            goto LABEL;
    }
}
LABEL: ;
```

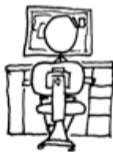
Best solution:

```
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;  
}
```

Junior: what's wrong
with goto command?

goto command:





Most compilers issue a warning when a variable is unused. C++17 introduces `[[maybe_unused]]` attribute to prevent this warning in a portable way

```
int f(int value) {  
    [[maybe_unused]] int x = value;  
    #if defined(ENABLE_SQUARE_PATH)  
        return x * x;  
    #else  
        // static_cast<void>(x); // before C++17  
        return 0;  
    #endif  
}
```

```
template<typename T>
int f([[maybe_unused]] T value) {
    if constexpr (sizeof(value) >= 4)
        return 1;
    else
        return 2;
}
```

```
template<typename T>
int g([[maybe_unused]] T value) {
    using R = decltype(value);
    return R{};
}
```

// MSVC issues a warning if `[[maybe_unused]]` is not applied