Basic Language Constructs for C++03

January 21, 2019

Brian A. Malloy
1. Overview

- These slides review basic C++ language constructs up to, but not including, **classes**.
- In the review, we discuss both C++03 and C++11.
- In some cases, we compare and contrast the two versions.
- The slides are accompanied by videos that further elicitate the concepts found here.
2. References

- Any Intro C++ text
- The C++ ISO Standard
3. Data and Expressions

bool ⇒ true or false
3.1. Operators

- Expressions are composed of operators, variables, constants and parentheses
- Logical operators: `&&`, `||`, `!`
- Relational operators: `<`, `>`, `==`, `!=`, `<=`, `>=`
- However, an expression can be considered as a Boolean condition where 0 is false and all other values are true:
  ```
  int x = rand();
  if (x) ...
  ```
- Of course, the rules for mixed types still apply, so `2/4` evaluates to 0
3.2. Operators

- unary, binary, and ternary describe the number of operands that an operator uses.
- For example, \(-7\) is **unary** minus; i.e., one operand
- \(3 - 7\) is **binary** minus; i.e., two operands
- There is only one **ternary** operator and it’s very useful; for example, the following expression evaluates to the larger of the two operands: \((a > b) \ ? \ a : b\)
3.3. Prefix and Postfix Operators

- Prefix operators are evaluated in place.
- Postfix operators are evaluated at the end of the statement

```cpp
#include <iostream>
int main() {
    int i = 0, j = 0;
    std::cout << ++i << std::endl; // output is 1
    std::cout << j++ << std::endl; // output is 0
    std::cout << i << j << std::endl; // output is 11
    return 0;
}
```
3.4. Insertion/Extraction Operators

- They are binary, left associative operators that evaluate to the operator
- For example, the stream insertion operator, `operator ≪` evaluates to `operator ≪`, which is why the following expression works:

The expression:
```
cout ≪ x ≪ y ≪ endl;
```
is actually:
```
(((cout ≪ x) ≪ y) ≪ endl);
```
where `(cout ≪ x)` places the value of `x` into the output stream and evaluates to `cout ≪` so that the expression becomes:
```
((cout ≪ y) ≪ endl);
```
which places `y` into the output stream and evaluates to `(cout ≪ endl);`
3.5. constants and constant expressions

- **const**: named constants are preferable to `# define`, which is a C artifact
  - `const char STAR = '*'`;
  - `const unsigned MAX = 100`;

- **constexpr**: value known at compile time
  
  ```cpp
  constexpr int n1 = 10;
  std::array<int, n1> a1;  // fine
  constexpr int n2 = 10;
  int a2[n2];         // fine
  int n3 = 10;
  int a3[n3];       // warning
  int n = 10;
  std::array<int, n> a2;  // error
  ```
3.6. NULL, 0, and nullptr

- NULL and 0 are integers
- nullptr is a pointer of all types
- prefer nullptr

```cpp
void f(int i) { std::cout << "int" << std::endl; }
void f(char* c) { std::cout << "pointer" << std::endl; }

int main() {
    f(NULL); // error ambiguous call
    f(0); // error ambiguous call
    f(nullptr); // prints pointer
}
```
3.7. Mixed Type Expressions

- Are promoted or truncated:
  1. \(5/2 \Rightarrow 2\)
  2. \(\text{int}(2.3) \Rightarrow 2\)
  3. \(\text{float}(2/4) \Rightarrow 0.0\)
  4. \(4/8 \Rightarrow 0\)
  5. \(\text{float}(4)/8 \Rightarrow 0.5\)
  6. \(2.0/4 \Rightarrow t0.5\)

- Prefer C++ cast → easier to find in code
  \(\text{static
cast<float>}(5/10)\) evaluates to 0.0
3.8. Structured Data Types

- Arrays, like C, are passed by reference
- **unions**: obviated by inheritance
- **structs**: same as classes except for default protection:
  - Default protection of class is **private**
  - Default protection of struct is **public**
  - structs are useful for storing global data: I prefer Singleton
- Classes are covered in slides about classes
4. Control Structures

• selection: if, if/else, switch

• repetition: for, while, do/while

• In general, I much prefer clarity and readability to obfuscated, hacked, terse code. Thus, I prefer the use of brackets because they promote readability. The first example below is preferable to the second:

```c
int sum = 0;
for (unsigned i = 0; i < MAX; ++i) {
    sum += i;
}
```

```c
int sum = 0;
for (unsigned i = 0; i < MAX; ++i) sum += i;
```
4.1. switch

- If a `switch` value matches a `case` value, then it matches all cases until a `break` is encountered:

```cpp
int count = 0;
int index = 1;
switch (index) {
    case 0: ++count;
    case 1: ++count;
    case 2: ++count;
    case 3: ++count;
    case 4: ++count;
    case 5: ++count;
    default: ++count;
}
```
```cpp
cout << count << endl; // prints 6
```
4.2. switch/case/break is useful

- We may wish to match several values, so multiple case values w/out a break are like logical or. In the next example, we can match either upper or lower case letters:

```c
int count = 0;
char ch = 'b';
switch (ch) {
    case 'A': case 'a': ++count; break;
    case 'B': case 'b': ++count; break;
    case 'C': case 'c': ++count; break;
    default: cout << "Oops" << endl;;
}
```
4.3. Short-circuit Evaluation

- If evaluation of the first operand obviates evaluation of the second, then the second operand is not evaluated.

- Short-circuit evaluation can be useful. If `number` happens to be zero, then we won’t get a division by zero error in the following example:

```c
float sum = 0.0;
int number = rand();
if ( number != 0 && sum/number > 90.0) {
    ...;
}
```
4.4. for

- The scope of the loop control variable (LCV) (in this case i) is the loop body:

```cpp
for (int i = 0; i < MAX; ++i) {
    cout << i;
}
i is out of scope here
```

- The following hack would be more readable if the programmer used `while (true)`

```cpp
// Obfuscated code; great for job security!
i = 0;
for ( ; ; ) {
    if (i > MAX) break;
    cout << ++i;
}
```
• ranged for loops: We will discuss later w/ vectors
5. Functions

- Can be void or return a value.
- Each C++ program contains a function called `main`, which returns an integer.
- There are two acceptable forms of `main`:

```cpp
int main() {
    return 0;
}
```
```
int main(int argc, char* argv[]) {
    return 0;
}
```
and the return statement is optional
5.1. Parameter Transmission Modes

- The C language has **one** mode
- C++ has four modes:
  1. **value**: default; makes local copy
  2. **reference**: use &; pass the address
  3. **const reference**: for large objects
  4. **rvalue reference**: later: ref v ptr
```cpp
#include <iostream>
void f(int x) { ++x; }
void g(int& x) { ++x; }
int main() {
    int i = 0, j = 0;
    f(i);
    g(j);
    std::cout << i << j << std::endl; // output is 01
}
```
5.2. Arrays are passed by reference

```cpp
#include <iostream>
const int MAX = 3;
void f(int a[]) {
    for (int i = 0; i < MAX; ++i) {
        a[i] = i;
    }
}
int main() {
    int a[3];
    f(a);
    std::cout << a[2] << std::endl; // output is 2
}
```
5.3. Static Function Variables

- Initialized upon first entry to the function
- Usually stored in global data segment

```cpp
#include <iostream>
void f() {
    static int count = 0;
    int index = 0;
    std::cout << ++count << ++index << std::endl;
}
int main() {
    f();
    f();
}
```

************* output *************

11
21
5.4. Default Parameter Values

- If no value is passed to formal parameter, a default value is assigned, left to right.
- Thus, \( x \), on line 2, is assigned the ascii code for 'A', which is 65, on line 7:

```cpp
#include <iostream>
void f(int x = 0, char ch = 'Z') {
    std::cout << x <<", " << ch << std::endl;
}
int main() {
    f(17, 'B');
    f('A');
    f();
}
```

************ output ************
17, B
65, Z
0, Z
5.5. Function Overload

- Two functions with same name but different parameter types
- The function return value cannot be used to resolve overload

```cpp
#include <iostream>
void write(double x) {
    std::cout << "x is " << x << std::endl;
}
void write(int i) {
    std::cout << "i is " << i << std::endl;
}
int main() {
    double x = 2.5;
    write(7); // output: i is 7
    write(x); // output: x is 2.5
}
5.6. Command Line Parameters

- You can pass values into function `main`
- `argc` is number of parameters passed; `argv` is an array of C strings containing the values
- There’s always at least one parameter passed: the name of the executable

```cpp
#include <iostream>
int main(int argc, char* argv[]) {
    for (int i = 0; i < argc; ++i) {
        std::cout << argv[i] << 't';
    }
    std::cout << std::endl;
}
```

************ invocation ************
$ ./a.out 2 4 cat
************ output ************
./a.out 2 4 cat
6. Namespaces

- We can use the scope operator (colon → ::) to access all three instances of `number`:

```cpp
#include <iostream>
int number = 99;
namespace A {
    int number = 23;
}
int main() {
    int number = 0;
    std::cout << ::number << std::endl;
    std::cout << A::number << std::endl;
    std::cout << number << std::endl;
    return 0;
}
```