Variables, primitive types, and expressions in C++

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Variables

Variables - name, type and scope

Example - factorial calculation: $n! = 1 \times 2 \times ...(n-1) \times n$

```
...
int main() {
  int n;
  cout << "Value of n? "; cin >> n;
  int f = 1;
  for (int i = 1; i <= n; i++) f = f * i;
  cout << "n! = " << f << '\n';
  return 0;
}
```

A variable in C++ has a **name**, a **type** and a **scope**:

- The **name** identifies the variable;
- The **type** defines the domain of values that the variable can assume;
- The **scope** is the region of code where the variable can be used after its declaration.

Variables - name, type and scope (cont.)

```
...
int main() {
  int n;
  cout << "Value of n? "; cin >> n;
  int f = 1;
  for (int i = 1; i <= n; i++) f = f * i;
  cout << "n! = " << f << '\n';
  return 0;
}
```

In the example:

- Variables are named `n`, `f` and `i`;
- All variables have `int` type - this means that they can hold signed integer values with (usually) 32-bit precision (on 32/64 bit machines).
- The scope is limited to the `main` function where they are declared – they are called **local variables**.

Variables - rules for declaration and use

- The name of a variable can not be a keyword.
- The scope of a variable begins with its declaration. This means a variable can only be used after its declaration.
- A variable must be declared once and only once. Distinct variables in the same scope must have different names.
- A value assigned to a variable must be compatible with the type of the variable.

Variables - rules for declaration and use (cont.)

Example errors and corresponding compiler messages:

```
int if = 0; // if is a keyword
```

    error: expected unqualified-id before ‘if’

```
int a = 0;
int b = 1;
int a = 2; // re-declaration of a
```

    error: redeclaration of ‘int a’

```
a = 1; // use prior to declaration
int a = 2;
```

    error: ‘a’ was not declared in this scope

```
int a = "xyz"; // incompatible value
```

    error: invalid conversion from ‘const char*’

Instruction blocks and scope

An instruction block between `{` and `}` defines a closed scope. Variables defined in the block can not be used outside it.

```
...
if (a > b) {
  int tmp = a;
  a = b;
  b = tmp;
}
tmp = 1; // ERROR
```

```
error: use of undeclared identifier 'tmp'
tmp = 1;
```

Global variables

Global variables are declared outside a function:

```
int g = 10; // Global variable
int f(int n) {
  return n + n + g; // use of g
}
```

The use of global variables is **usually a bad idea**, as they tend to induce unstructured programming patterns, **except for the use of constants declared through the `const` modifier** or a few special cases, e.g., `std::cout`.

```
// The use of const in a variable declaration
// forbids assignments to it beyond
// the initialisation value.
const int g = 10;
```

Primitive types

Integer types

Type	Size (bytes)	Min. value	Max. value
`char`	1	-2^7 (-128)	$2^7 - 1$ (127)
`short`	2	-2^{15}	$2^{15} - 1$
`int`	4	-2^{31}	$2^{31} - 1$
`long`	8	-2^{63}	$2^{63} - 1$
`unsigned char`	1	0	$2^8 - 1$ (255)
`unsigned short`	2	0	$2^{16} - 1$
`unsigned int`	4	0	$2^{32} - 1$
`unsigned long`	8	0	$2^{64} - 1$

In addition to `int`, other traditional types for integer values are: `char`, `short` and `long`, along with their `unsigned` variants.

The size in bytes (and corresponding value range) is **dependent** on the architecture / compiler. Above, we depict the sizes typically employed in a 64-bit architecture (ex. Intel x86_64).

Integer types (const.)

- The `sizeof` operator can be used to indicate the size required for the representation of a type or expression, e.g.,

```
cout << sizeof(int) << " " << sizeof(long) << "\n";
```

```
4 8
```

- The `climits` header defines constants for the minimum and maximum values for each type, e.g., `INT_MIN` and `INT_MAX` for `int`.

```
#include <climits>
. . .
cout << INT_MIN  << " " << INT_MAX   << "\n"
     << LONG_MIN << " " << LONG_MAX  << "\n";
```

```
-2147483648 2147483647
-9223372036854775808 9223372036854775807
```

Integer constants

Decimal	`10 65 -1 1234 123u`
Character codes (as in ASCII)	`'\n'` (10) `'A'` (65) `'0'` (48)
Octal	`012` (10) `0101` (65)
Hexadecimal	`0x0A` (10) `0x41` (65)

The `u`/`U` suffix explicitly states that the constant is `unsigned int`, e.g., `123u`. Similarly, `L` or `l` are used for `long` constants, and `UL` or `ul` are used for `unsigned long`; they may be required for constants that overflow a 32-bit representation, e.g.,

```
long x          = 9223372036854775807L;  // 2^63 - 1
unsigned long y = 18446744073709551615UL; // 2^64 - 1
```

The `bool` type

The `bool` type is used to represent values `true` or `false`.

In the context of integer expressions, `true` evaluates to `1` and `false` evaluates to `0`.

Example use:

```c
bool is_hexadecimal_digit(char c) {
  if (c >= '0' && c <= '9')
    return true;
  if (c >= 'a' && c <= 'f')
    return true;
  if (c >= 'A' && c <= 'F')
    return true;
  return false;
}
```

Enumerations

Enumeration types are user-defined types (not primitive types) that define a domain of integer constants. For instance, the following code illustrates the definition of a `month` enumeration, and the declaration of a variable with that type:

```
enum month {
  JANUARY = 1,
  FEBRUARY, /* implicitly 2 */ MARCH, /* 3 */
  APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER,
  OCTOBER, NOVEMBER, DECEMBER /* 12 */
};
...
month m = DECEMBER;
```

If the type is omitted, then only the integer constants are defined:

```
enum { JANUARY=1, ..., DECEMBER };
. . .
int m = DECEMBER;
```

Floating point types

`float` and `double` are primitive types for floating point values:

- `float`: single-precision floating point, 32 bits in 64-bit architectures, values range from 10^{-38} a 10^{38};
- `double`: double-precision floating point, 64 bits in 64-bit architectures, values range from 10^{-308} a 10^{308};

Constants:

Decimal	0.01 -1.23 1230.0 123.5f
Scientific notation	1e-2 -123e-02 123e+1

Suffix `f` is be used to indicate that a constant is explicitly of `float` type (`double` is assumed otherwise).

The `void` type

`void` is the type for the empty set of values.

A variable can not be declared with the `void` type.

The `void` type **must** be used to state that a function returns no values, and **can optionally be used** to state that a function has no arguments,

```
void f(int x) {  // f has no return value;
  cout << x;
  return; // a return instruction may be used,
          // but without an associated value
}
int g(void) { // no arguments; void may be omitted
  return 123;
}
```

Use of `auto`

The `auto` keyword can be used to declare a variable whose type should be **deduced by the compiler** from its initialisation value.

Example:

```
auto x = 10;   // int
auto y = 10UL; // unsigned long
auto z = 1.2f; // float
auto w = 1.2;  // double
cout << sizeof(x) << ' ' << sizeof(y) << ' '
     << sizeof(z) << ' ' << sizeof(w) << "\n";
```

Output:

```
4 8 4 8
```

`auto` should be used sparingly, as it may obfuscate the meaning of a program. It is adequate to avoid writing complex/verbose type names, as we will see later in the semester.

Use of `typedef`

User-defined types can be defined as aliases of other types through `typedef`, e.g,

```
// Definition of types integer and real
typedef int integer;
typedef double real;
...
// Use of integer and real for variables
integer i = 0;
real    r = 2.5;
```

Expressions

Expression

An **expression** may be composed by constants, variables, and function calls combined through **operators**.

Examples:

```
y = (1.0 + a) * b * c / f(1e-02, 2, x - 2);
x *= a <= b && c > d ? a : b;
x++;
--x;
z ^= g(~x, x | y);
```

Assignment operator

General form:

```
a = b;
```

- `a`, called the **l-value**, identifies the target for the assignment
- `b`, called the **r-value**, is the value to be assigned

Although uncommon, assignments can be chained, e.g.:

```
i = j = k = 123;
```

→ Further reference

Arithmetic operators

Expression	Operation
`a + b`	Sum
`a - b`	Subtraction
`a * b`	Multiplication
`a / b`	Division
`a % b`	Modulo

`-` and `+` can also be used as unary operators, e.g., as in

```
+a
-a
- (+a * -b)
```

→ Further reference

Bitwise arithmetic

Expression	Operation
`a & b`	Bitwise AND
`a	b`	Bitwise OR
`a ^ b`	Bitwise XOR
`~a`	Bit inversion - NOT
`a << b`	Left shift of `a` by `b` bits.
`a >> b`	Right shift of `a` by `b` bits.

→ Further reference

Composed assignment operators

Expression	Equivalent to
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`
`a &= b`	`a = a & b`
`a |= b`	`a = a | b`
`a ^= b`	`a = a ^ b`
`a <<= b`	`a = a << b`
`a >>= b`	`a = a >> b`

→ Further reference

Comparison operators

Expression	Evaluates to `1` if ...
`a == b`	`a` is equal to `b`
`a != b`	`a` is not equal to `b`
`a < b`	`a` is lower than `b`
`a <= b`	`a` is lower or equal to `b`
`a > b`	`a` is higher than `b`
`a >= b`	`a` is higher or equal to `b`

→ Further reference

Logical operators and evaluation order

Expression	Evaluates to 1 if ...
`a && b`	`a` and `b` both differ from 0 (both are "true")
`a || b`	`a` or `b` differ from 0 (one of them is "true")
`!a`	`a` is 0 (is "false")

Expressions `a && b` e `a || b` are guaranteed to have a left-to-right evaluation order, and `b` is evaluated only if necessary:

- `a && b` evaluates expression `a` first and `b` is evaluated only if `a != 0`.
- `a || b` evaluates expression `a` first and `b` is evaluated only if `a == 0`.

In contrast, an expression like `a+b` has an undefined evaluation order, i.e., expression `a` is not guaranteed to be evaluated first (→ read more).

Note: `and`, `or` and `not` can also be used (as in Python) in place of `&&`, `||` and `!` respectively.

Ternary conditional operator `?:`

An expression of the form

```
a ? b : c
```

yields `b` if `a != 0`, and `c` otherwise.

For instance, in

```
x = y > 100 ? 1 : 2;
```

`x` is assigned to `1` if `y > 100`, and `2` otherwise.

Increment and decrement operators

General form:

```
++a      a++      --b      b--
```

These operators are useful to express increments and decrements concisely.

For instance

```
a++;
--b;
```

is equivalent to

```
a += 1;
b -= 1;
```

Increment and decrement operators (cont.)

What makes prefix and postfix variants different?

- Prefix operators `++a` and `--b` update the variable before evaluation, i.e., the expression's result reflects the update.
- Postfix operators `a++` and `b++` update the variable after evaluation, i.e., the expression's result does not reflect the update.

For instance, in

```
int a = 1;
int b = ++a + 1; // <=> ++a; int b = a + 1;
```

`a` is updated **before** the assignment to `b`, hence `b` is assigned value `3`.
On the contrary, in

```
int a = 1;
int b = a++ + 1; // <=> int b = a + 1; a++;
```

`a` is updated **after** the assignment to `b`, hence `b` is assigned value `2`.

Increment and decrement operators (cont.)

The use of `++` and `--` in conjunction with other operators is not recommended, as the code can easily become confusing.

Moreover, **undefined behavior** may result, as in

```
int a = 1;
int b = a + ++a;
```

Given that a left-to-right evaluation order for the sum operator is not guaranteed, `b` can be assigned above to `3` (`1+2`, `a` on the left evaluated first) or `4` (`2+2`, `++a` on the right evaluated first).

Operators - precedence and associativity

Precedence	Operators	Associativity
...		
3	`* / %`	Left
4	`+ -`	Left
5	`<< >>`	Left
6	`< > <= >=`	Left
7	`== !=`	Left
...		
14	`=`	Right

The table fragment above covers a subset of all C operators. C++ has quite a few more. As usual in programming languages: precedence determines the evaluation order; associativity determines the direction of evaluation for operators, disambiguating evaluation order for operators with equal precedence.

Operators - precedence and associativity (cont.)

`*` has precedence over `+` and `-`, so

```
a * b + c * d - e
```

is equivalent to

```
(a * b) + (c * d) - e
```

but not to

```
a  * (b + c) * (d - e)
```

Operators - precedence and associativity (cont.)

Left or right associativity determine the interpretation of expressions containing operators with equal precedence.

Since `*` and `/` associate left

```
a * b / c
```

is equivalent to

```
(a * b) / c
```

On the contrary, `=` associates right, hence

```
a = b = 10;
```

is equivalent to

```
a = (b = 10);
```

Namespaces

Namespaces and scope

We can declare variables (or functions, types, ...) with the same name in distinct namespaces, e.g.

```
namespace a {
  const int g = 10;
  int f(int n) { return n + g; } // g refers to a::g
}
namespace b {
  const int g = 1000000;
  int f(int n) { return n - g; } // g refers to b::g
}
```

Namespaces and scope (cont.)

Recall that `x` defined in namespace `n` needs to be referred to as `n::x` except if a `using namespace n;` directive is in context. Definitions may clash when employing `using`, e.g.

```
namespace a {
  const int g = 10;
  int f1(int n) { return n + g; }
}
namespace b {
  const int g = 1000000;
  int f1(int n) { return n - g; }
}
using namespace a; using namespace b;
// a::g or b::g ? a::f1 or b::f1 ?
int f2(int n) { return f1(n) * g; }
```

```
  error: call to 'f' is ambiguous
  error: reference to 'g' is ambiguous
```

Nested namespaces

Namespaces can be nested.

Example:

```
namespace a {
  namespace b {
    const int g = 1;
  }
  const int g = 1 + b::g; // 2
}
const int g = 1 + a::g + a::b::g; // 4
```