Expressions and Declarations in C++ Programming

To perform a calculation in the C++ program you need an expression. An expression is a statement that has both a value and a type. In the C++ program, a declaration is statement that defines a variable or it's a "holding tank" for some sort of value like a number or character.

Expressions

Expressions take one of the following forms:

- objName       // for a simple object
- operator expression // for unary operators
- expr1 operator expr2 // for binary operators
- expr1 ? expr2 : expr3 // for the ternary operator
- funcName([argument list]); // for function calls

Literal expressions

A literal is a form of constant expression. The various types of literals are defined in the following table.

<table>
<thead>
<tr>
<th>Example</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>1L</td>
<td>long int</td>
</tr>
<tr>
<td>1LL</td>
<td>long long int</td>
</tr>
<tr>
<td>1.0</td>
<td>double</td>
</tr>
<tr>
<td>1.0F</td>
<td>float</td>
</tr>
<tr>
<td>'1'</td>
<td>char</td>
</tr>
<tr>
<td>&quot;a string&quot;</td>
<td>char* (automatically terminated with a null character)</td>
</tr>
<tr>
<td>L&quot;a string&quot;</td>
<td>wchar_t*</td>
</tr>
</tbody>
</table>
| u8"this is a UTF-8 string with a UTF-8" | char8_t* }
character: \u2018"

u"this is a UTF-16 string with a UTF-16 character: \u2018"

U"this is a UTF-32 string with a UTF-32 character: \U00002018"

true, false

0b101

binary (C++ 2014 standard)

**Declarations**

Declarations use both intrinsic and user-defined types. The intrinsic types are:

```cpp
[<signed | unsigned>] char
[<signed | unsigned>] wchar_t
[<signed | unsigned>] [<short | long | long long>] int
float
[long] double
bool
```

Declarations have one of the following forms:

```cpp
[<extern|static>][<const>] type var[=expression]; // variable
[<extern|static>][<const>] type array[size][={list}]; // array
[const] type object[(argument list)]; // object
[const] type object [= {argument list}]; // alternative
[const] type * [const] ptr[=pointer expression]; // pointer
type & refName = object; // reference
type fnName([argument list]); // function
```

The keyword `auto` can be used if C++ can determine the type of variable itself:

```cpp
auto var = 1L; // the type of var is long int
```

The keyword `decltype` extracts the type of an expression. This type can then be used wherever a type name is used. For example, the following example uses `decltype` to declare a second variable with same type as an existing variable:

```cpp
decltype(var1) var2; // the type of var2 is the same as var1
```

A function definition has the following format:

```cpp
// simple function
[<inline|constexpr>] type fnName(const argument list) {...}
```
// member function defined outside of class
[inline] type Class::func(argument list) [const] {...}
// constructor/destructors may also be defined outside of class
Class::Class([argument list]) {...} 
Class::~Class() {...}
// constructors/destructor may be deleted or defaulted
// in lieu of definition
Class::Class([argument list]) = <delete|default>;
Class::~Class() = <delete|default>;

An overloaded operator looks like a function definition. Most overloaded
operators may be written either as member or simple functions. When written
as a member function, *this is the assumed first argument to the operator:

MyClass& operator+(const MyClass& m1, const MyClass& m2); // simple
MyClass& MyClass::operator+(const MyClass& m2); // member;

Users may also define their own types using the class or struct keywords:

<struct | class> ClassName [ : [virtual] [public] BaseClass]
{
    <public|protected>:
    // constructor
    ClassName([arg list]) <[: member(val),...] {...} | ;>
    ClassName() [= <delete|default>];
    // destructor
    [virtual] ~ClassName() <{...} | [=<delete|default>>;
    // public data members
    type dataMemberName [= initialValue];
    // public member functions
    type memberFunctionName([arg list]) [{...}]
    // const member function
    type memberFunctionName([arg list]) const [{...}]
    // virtual member functions
    virtual type memberFunctionName([arg list]) [{...}];
    // pure virtual member functions
    virtual type memberFunctionName([arg list]) = 0;
    // function that must override a base class function
type memberFunctionName([arg list]) override;
    // a function that cannot be overridden in a subclass

type memberFunctionName([arg list]) final;
};

In addition, a constructor with a single argument may be flagged
as explicit meaning that it will not be used in an implicit conversion from
one type to another. Flagging a constructor as default means "use the
default C++ constructor definition". Flagging a constructor
as delete removes the default C++ constructor definition.

C++ supports two types of enumerated types. The following old enumeration
type does not create a new type:

enum STATE { DC, // gets 0
    ALABAMA, // gets 1
    ALASKA, // gets 2
    ARKANSAS, // gets 3
    // ...and so on
    };

int n = ALASKA; // ALASKA is of type int

By default an individual entry is of type int but this can be changed in the
C++ 2011 standard:

enum ALPHABET:char { A = 'a', // gets 'a'
    B, // gets 'b'
    C, // gets 'c'
    // ...and so on
    };

char c = A; // A is of type char

C++ 2011 allows a second format that does create a new type:

// the following enumeration defines a new type STATE
enum class STATE { DC, // gets 0
    ALABAMA, // gets 1
    ALASKA, // gets 2
    ARKANSAS, // gets 3
    // ...and so on
    };

STATE s = STATE::ALASKA; // now STATE is a new type

// the following uses a different underlying type
enum class ALPHABET:char { A = 'a', // gets 'a'
B, // gets 'b'
C, // gets 'c'
    // ...and so on
};

ALPHABET c = ALPHABET::A; // A is of type ALPHABET

Template declarations have a slightly different format:

// type T is provided by the programmer at use
template <class T, {...}> type FunctionName([arg list])
template <class T, {...}> class ClassName { {...} };

---

Operators in C++ Programming

All operators in C++ perform some defined function. This table shows the operator, precedence (which determines who goes first), cardinality, and associativity in the C++ program.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Cardinality</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>() [] -&gt; .</td>
<td>unary</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ + - ++ — &amp; * (cast)</td>
<td>unary</td>
<td>left to right</td>
</tr>
<tr>
<td>sizeof</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* / %</td>
<td>binary</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>binary</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>binary</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;= &gt;=</td>
<td>binary</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>binary</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>binary</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>binary</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>binary</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>binary</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>ternary</td>
<td>right to left</td>
</tr>
<tr>
<td>= *= /= %= += -= &amp;= ^=</td>
<td>=</td>
<td>binary</td>
</tr>
<tr>
<td>&lt;&lt;= &gt;&gt;=</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Flow Control in C++ Programming

The following C++ structures direct the flow of control through the program. If you're an experienced programmer, the function of these structures will be familiar from other languages.

**IF**

The following command evaluates `booleanExpression`. If it evaluates to `true`, then control passes to `expressions1`. If not, then control passes to the optional `expressions2`.

```
if (booleanExpression)
{
    expressions1;
}
[else
{
    expressions2;
}
]
```

**WHILE**

The following command evaluates `booleanExpression`. If this evaluates to `true`, then control passes to `expressions`. At the end of the block, control passes back to `booleanExpression` and repeats the process.

```
while (booleanExpression)
{
    expressions;
}
```

**DO…WHILE**

The following command executes `expressions`. It then evaluates `booleanExpression`. If this evaluates to `true`, control returns to the top of the loop and repeats the process.

```
do
{
```
expressions;
} while(booleanExpression);

FOR

The following command executes initCommand which may be an expression or a variable declaration. It then evaluates boolExpression. If this evaluates to true, then control passes to expressions1. If boolExpression is false, then control passes to the first statement after the closed brace of the for loop. Once expressions completes, control passes to the expression contained in loopExpression before returning to boolExpression to repeat the process. If initCommand declares a new variable, it goes out of scope as soon as control passes outside of the loop.

for (initCommand; boolExpression; loopExpression)
{
    expressions;
}

FOR (EACH)

The 2011 standard introduces a second form of for loop sometimes known as a "for each" because of its similarity to the foreach found in some other languages. In this form, the variable declared in declaration takes the value of the first member of list and executes the expressions block. When complete, the declared variable takes the second value of list and executes expressions again. This process is repeated for each value in list.

for (declaration: list)
{
    expressions;
}

SWITCH

The following command evaluates integerExpression and compares the result to each of the cases listed. If the value is found to equal one of the constant integral values, val1, val2, etc., control passes to the corresponding set of expressions and continues until control encounters abreak. If expression does not equal any of the values, control passes to the expressionsN following default.

switch(integerExpression)
{ 
    case val1:
        expressions1;
        break;
    case val2:
        expressions2;
        break;
    [default:
        expressionsN;
    ]
}

**BREAK, CONTINUE, GOTO**

A `continue` passes control to the end of the closed brace of any of the looping controls. This causes the loop to continue with the next iteration. For example, the following loop processes prime numbers between 1 and 20:

```c++
for(int i = 0; i < 20; i++)
{
    // if the number is not prime...
    if (!isPrime(i))
    {
        // ...skip over to the next value of i
        continue;
    }
    // proceed on processing
}
```

A `break` passes control to the first statement after the closed brace of any of the looping commands. This causes execution to exit the loop immediately. For example, the following reads characters until and end-of-file is encountered:

```c++
while(true)
{
    // read a line from input object
    input >> line;
    // if a failure or end-of-file occurs...
    if (cin.eof() || cin.fail())
    {
```
// ...then exit the loop
break;
}
// process the line

A goto label passes control to the label provided. The break example above could have been written as follows:

while(true)
{
    // read a line from input object
    input >> line;
    // if a failure or end-of-file occurs...
    if (cin.eof() || cin.fail())
    {
        // ...then exit the loop
        goto exitLabel;
    }
    // process the line
}
exitLabel:
    // control continues here