Advanced Programming in C++

2014

Mixed Exercises

These exercises cover different aspects of C++. Their purpose is to support you in your studies to prepare for the computer examination. Some exercises cover basic aspects of C++, while some are more difficult and advanced, and in some case obscure.

There are also some larger exercises given separately, which are to be preferred. Most of the programming exercises given here are supposed to lead to rather small solutions, while most of the exercises given separately, in most cases, lead to more comprehensive coding.

The exercises given here are divided into different topics, with respect to their main feature. Some exercises may therefore actually cover several topics. The first clauses, i.e. Basic constructs and Statements are prerequisites, but should be looked into if you don't have pre-knowledge of C++.

- Basic language constructs
- Statements
- Class
- Operator verloading
- Derived classes
- Exceptions

Basic language constructs

This clause covers such things as declarations, definitions, simple data types, compound data types, built-in operators, expressions, and statements.

1. Which of the following declarations are *definitions* and which are not? Give a corresponding definition for those that are *not* a definition and a corresponding declaration for those that *are*.

```
char c;
string s;
auto count{ 1 };
const int MAX{ 4711 };
extern double d;
const char* msg{ ”Hello!” };
const char* Direction[]{ ”up”, ”down”, ”left”, ”right” };
struct Time { int h, m, s; };
int sec(Time* p) { return p->s; }
double sqrt(double);
template<typename T> T abs(T a) { return (a < 0) ? -a : a; }
typedef std::vector<double> data;
using char_ptr = char*;
struct Node;
enum Season { Spring, Summer, Autumn, Winter };
namespace N { int i; }
```

2. Which entities in exercise 1 have a size, i.e. to which may the operator **sizeof** be applied?

3. Write declarations for the following entities and initialize each of them:

a pointer to **char**
an array with 10 **int**
a reference to an array with 10 **int**
a pointer to an array with 10 elements of type string
a pointer to a pointer to **char**
a constant **int**
a pointer to a constant **int**
a constant pointer to an **int**

4. Use alias declarations (**using**) to define the following types, in the given order:

unsigned char
const unsigned char
pointer to **int**
pointer to pointer to **char**
pointer to array with element of type **char**
array with 10 pointer to **int**
pointer to an array with 10 pointer to **int**
array 20 of array 10 with pointer to **int**

5. For each of the following declarations: Is it legal? If it is, what does it declare?

```
int a(int i);
int a(int);
int a(int (i));
int a(int (*b)());
int a(int b());
int a(int ());
```

6. What does the following mean? How can it be used?

```
using x = int(&)(int, int);
```

7. Which size does the character array msg have? Which length does the string "Hello world!" have?

```
char msg[]{ "Hello world!" };
```

8. Which value, in principle, does the operator **sizeof** give for the following variables?

```
char        str1[100]{ "String" };
char        str2[]{ "String" };
const char* str3{ "String" };
string      str4{ "String" };
```

Why must the third declaration be **const**?

9. Fully parenthesize each of the following expressions:

```
a = b+c*d<<2&8
a&&077!=3
a==b || a==c && c<5
c = x!=0
0<=i<7
f(1,2)+3
a = -1+ +b-- -5
a= b==c++
a = b = c = 0
a[4][2] *= *b? c: *d*2
a-b, c = d
*p++
*--p
++a--
(int*)p->m
*p.m
*a[i]
```

How will the seventh expression above be parsed, if one removes all spaces? Will it be a legal expression? Will it be semantically the same as the expression with spaces?

Note: This illustrates the *maximum munch principle* used by the parser.

10. Write declarations for the following types, in the order given. Show both alias declarations (**using**) and **typedef**s.

a function which takes two arguments, one of type pointer to **char** and one of type reference to **int**, not returning anything
a pointer to such a function
a function which takes such a function pointer as argument
a function which returns such a pointer

11. Given the declarations:

```
unsigned u = 0;
bool     b = false;
double   d = 0.0;
```

What results will the following *expressions* give, and what value will the three *variables* have afterwards if, of course, the expression in question is legal?

`u--`	and thereafter	`++u`
`b++`	and thereafter, again	`b++`
`++b`		
`b--`		
`--b`		
`d++`		

The operand to an increment operator must be a modifiable *lvalue*, but what is the result of postfix ++/-- and prefix ++/--, respectively? Is it a *lvalue* or a *rvalue*?

Statements

1. Given the following **for** statement where line is a character array with length length.

```
for (int i = 0; i < length; ++i)
  if (line[i] == ' ')
    ++space_count;
```

Rewrite it as a corresponding **while** statement.

2. Rewrite the **for** statement in exercise 1 above, so it steps through `line` using a pointer instead of indexing.

Note 1: These dual constructs for accessing array elements shows two well-known *idioms* in C and C++; the *"array indexing idiom"* and the *"pointer idiom"*.

Note 2: Using pointer arithmetic to step through an array is a source optimization technique frequently used in C programs, by tradition. A modern C or C++ compiler may produce better code if you refrain from this!

3. Rewrite the for statement in exercise 1 above, using *range access* functions.

4. Rewrite the for statement in exercise 1 above, using a *range based* **for** statement.

Class

1. One can define a class, say Integer, with almost the exact behaviour as **int**, without overloading every operator which is defined for **int**, by defining **operator int**() for Integer. Why do we get into trouble in the following example?

```
struct Ten {
  operator Integer() const { return Integer(10); }
};

Integer i(10);  // Constructor Integer(int) is supposed to be defined
const char* hex = "0123456789ABCDEF";
char d = hex[i];

Ten ten;
d = hex[ten];
```

2. Define a class `Integer` which acts as **int** but for which only the following operations are allowed: `+` (both unary and binary), `-` (both unary and binary), `*`, `/`, `%`, `++` and `--` (both in prefix and postfix versions, with the same semantics as for **int**). How can you make these operations also legal for mixed expressions with **int** and `Integer`?

Overload **operator** << and **operator** >> for `Integer`. Could be a good idea to implement **operator**<< first of all operators.

Operator overloading

1. Define a type for storing a persons name (first and last name as two separate strings) and address (street, post code and city as three separate strings). Overload the **operator**>> and **operator**<< to read and write, respectively, such data, given a certain format (e.g first and last name in that order on the first line and the three parts of the address on three successive lines). Write a test program which copies a stream of names and addresses.

2. Construct a class named ifcharstream with **operator**[] overloaded for reading characters from a text file randomly. A file name is to be given when an ifcharstream object is created, and a file stream for both reading and writing is to be opened and bound to the given file.

3. Use istream::seekg for positioning in the file.

Derived classes

1. The following class is given:

```
struct Base
{
  void id() const { cout << "base\n"; }
};
```

1) Derive a direct subclasses to Base called Derived and declare a member function id() also in Derived, to print out "Derived". Write a program which declares ordinary object of Bae and Derived and also declare Base and Derived pointers and let then point to dynamically allocates objects of type Base and Derived. Call id() for the different cases and especially note the printout for the case when a derived object is referred to by a base* pointer.

2) Copy the program from step 1. Declare id() to be virtual in both Base and Derived and add a virtual destructor to Base, since this is now a polymorphic class hierarchy. Run the same test suite as in step 1. There should be at least some difference.

3) Copy the program from step 2. Remove id() from Derived, leaving Derived empty. Modify Base::id() to print the “name” of the class by applying **typeid** to the object in question and calling member function name() for the type_info object the **typeid** expression creates. Run same test as prevously.

4) Copy the program from step 3. Replace the **typid** expression and call to name() with a call to demangle_name(), which can be found from the course exercises home page.

2. The following abstract class for implementing callbacks is given:

```
struct callback {
  virtual void operator()() = 0;
};
```

Define, by deriving from the class callback, and overriding av operator() a class template which can be used for binding callback functions of type **void**(*)(). A simple program which uses these classes may look as:

```
#include <iostream>
using namespace std;

struct callback { /* As above */ };

template<…> … ;       // The template class to be defined

void hello() { cout << "hello world!\n"; }

void greet(… msg) {  // Function making callback via parameter
  msg();
}

int main() {
  greet(…hello…);
}
```

Define also a class template for callbacks to a *member function* in an object. The member function is supposed to have the type **void**(*)().

Exceptions

1. Write a program with a recursive function, which calls itself to a certain call depth, e.g 10, and then throws an exception. The function shall as argument have the call depth when the exception is to be thrown. Let main() catch the exception and print at which call depth the exception was thrown. The call depth when the exception is to be thrown shall be given on the command line when the program is started. A run can look as follows, if the name of the program is throwup:

```
%unix throwup 10
exception thrown at call depth 10.
```