Advanced Programming

C++ Basic Structure, Declarations & Definitions

This document serves as supplementary material for lecture 2 on advanced programming. It is designed to provide students with a foundational understanding of the key concepts for this lecture, such as expressions, operators, statements, iteration, functions, control flow, data types, declarations and definitions, and scope and namespaces.

1 Expressions and Operators

1.1 Expressions

In C++, an expression is a combination of variables, operators, and values that yield a result. Expressions can be as simple as a single value or as complex as a function call.

```cpp
int length = 10; // simple expression
int width = 20;
int perimeter = 2*(length + width); // some operations
int area = calc_area(length, width); // function call (more on functions later)
```

1.2 Operators

Operators are symbols that specify the type of operation to be performed on operands. C++ supports various operators, including arithmetic, relational, logical, and concise operators.

Arithmetic Operators

Valid operand types: Numeric (`int, short, long, float, double`)

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

Table 1: Arithmetic Operators

Important: The remainder operator (%) requires integer type literals!. Using the wrong types results in **compilation errors**, e.g:

```cpp
double x = 50.0;
int y = 6;
int remainder = x % y;
```

yields:

```
error: invalid operands of types ‘double’ and ‘int’ to binary ‘operator%’
```

Relational Operators

Valid operand types: Numeric (`int, short, long, float, double`)

Operator	Operation
==	Equal
!=	Not equal
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Table 2: Relational Operators

Logical Operators

Valid operand types: Boolean (`true, false`)

Operator	Operation
&&	And
||	Or
!	Not

Table 3: Logical Operators

Both Relational Operators and Logical Operators return `true` (1) or `false` (0)

Concise Operators

Special case of operator for simplifying binary arithmetic operations.

`a = a + b` → `a += b`

These operators are particularly useful for increments in loops (you will see more about loops below).

`a = a + 1` → `a += 1` → `++a`

2 Statements

A statement in C++ is the smallest standalone element of a program that expresses an action to be carried out. Statements can be expressions followed by a semicolon, declarations, or control statements.

```
int x = 5; // declaration statement
y = 10; // expression statement
if (x <= y) x = 20; // control statement
```

2.1 Control Statements

Control statements determine the flow of control in a program. Common control statements include Selection Statements (if - else), and Iteration Statements (while and for).

Selection (if - else)

The if - else statement allows branching based on conditions.

```
if (a < b)
    cout << "a is less than b" << endl;
else
    cout << "a is greater than b" << endl;
```

2.2 Iteration

Iteration statements allow repeating a block of code multiple times. The two primary iteration constructs in C++ are while loops and for loops.

While Loop

```cpp
int i = 0; // control variable
while (i < 10) { //termination criterion
    cout << "Current iteration: " << i << endl;
    ++i; // increment
}
```

While loops only require a boolean termination criterion. However, you must ensure that the termination criterion will be met at some point:

```cpp
bool continue = true;
int i = 0;
while (continue) { // termination criterion
    cout << "Current iteration: " << i << endl;
    ++i; // increment
}
```

This results in an infinite loop, since the termination criterion remains unchanged. A fix for this could be:

```cpp
bool continue = true;
int i = 0;
while (continue_loop) { // termination criterion
    cout << "Current iteration: " << i << endl;
    if (i>10){
      continue_loop = false; // update expression
    }
    ++i; // increment
}
```

This fix might look a bit complex, however, it can be useful when your termination criterion depends on multiple operations or inputs/outputs

For Loop

```cpp
for (int i = 0; i < 10; ++i) {
    cout << "Current iteration: " << i << endl;
}
```

This loop is equivalent to the while loops previously shown, however, the for loop syntax wraps all the conditions for the iteration in one place.

```cpp
for (int i = 0; i < 10; ++i)
```

For loops can also be range–based:

```cpp
for (int i : {0, 1, 2, 3, 4, 5}) // the initialiser is a list (using braces)
    cout << "Current iteration: " << i << endl;
```

For loops provide simplicity and readability, particularly when iteration conditions are known.

3 Functions

Functions in C++ allow code modularity and reusability. A function is defined to perform a specific task and can be called multiple times within a program.

```cpp
int add_int(int a, int b) {
    return a + b;
}
```

The general structure of a function is:

```
return_type  function_name(parameters) {
    // body of the function
    return return_value;
}
```

Functions are defined with a return type, a name, a parameter list, and a return value. However, some times you want to use a function to do something, but you don't require a return value (e.g. writing an output file or printing something on the screen). For those cases, you can use the return type `void`

```cpp
void exit_msg() {
    cout << "Good bye!" << endl;
}
```

4 Control Flow

Control flow refers to the order in which individual statements, instructions, or function calls are executed or evaluated. Let's recall our previous example of a while loop in a fully functional implementation:

```cpp
#include <iostream> // include directive
using namespace std; // adding a namespace to the global scope
int main() {
    // variable definition
    bool continue_loop = true;
    int i = 0;
    // loop
    while (continue_loop) { // termination criterion
        cout << "Current iteration: " << i << endl;
        // selection statement
        if (i>10){
          continue_loop = false; // update expression
        }
        ++i; // increment
    }
}
```

5 Data Types

C++ supports several data types, which can be broadly categorised into built-in types, derived types, and user-defined types.

Built-in Types

These data types have a specific size in memory. This size determines the range of values allowed for each type. Here's a code snippet you can use to check the size of built–in data types:

Built-in Types	Derived Types	User–defined
`int`	function	class
`short`	array	structure
`long`	pointer	union
`float`	reference	enum
`double`		typedef
`bool`		
`char`		
...		

```
#include <iostream>
using namespace std;
int main() {
    cout << "Size of int : " << sizeof(int) << " bytes."  << endl;
    cout << "Size of float : " << sizeof(float) << " bytes."  << endl;
    cout << "Size of double : " << sizeof(double) << " bytes."  << endl;
    cout << "Size of bool : " << sizeof(bool) << " bytes."  << endl;
    return 0;
}
```

For some data types, you can change the size or the range of values allowed. This is achieved by using type modifiers: signed, unsigned, short, long.

```
#include <iostream>
using namespace std;
int main() {
    cout << "Size of int : " << sizeof(int) << " bytes."  << endl;
    cout << "Size of signed int : " << sizeof(signed int) << " bytes."  << endl;
    cout << "Size of unsigned int : " << sizeof(unsigned int) << " bytes."  << endl;
    cout << "Size of short int : " << sizeof(short int) << " bytes."  << endl;
    cout << "Size of long int : " << sizeof(long int) << " bytes."  << endl;
    return 0;
}
```

Now let's see the changes in the allowed range of values (this is a bit more complex, but don't worry about that. We are just making use of specific libraries. You'll learn about that later)

```
#include <iostream>
#include <climits>
using namespace std;
int main() {
    cout << "Size of short int: " << sizeof(short int) << " bytes." << endl;
    cout << "Signed short min value: " << SHRT_MIN << endl;
    cout << "Signed short max value: " << SHRT_MAX << endl;
    cout << "Unsigned short min value: 0" << endl;
    cout << "Unsigned short max value: " << USHRT_MAX << endl;

    cout << "Size of int type: " << sizeof(int) << " bytes." << endl;
    cout << "Signed int min value: " << INT_MIN << endl;
    cout << "Signed int max value: " << INT_MAX << endl;
    cout << "Unsigned int min value: 0" << endl;
    cout << "Unsigned int max value: " << UINT_MAX << endl;

    cout << "Size of long int type: " << sizeof(long int) << " bytes." << endl;
    cout << "Signed long min value: " << LONG_MIN << endl;
    cout << "Signed long max value: " << LONG_MAX << endl;
    cout << "Unsigned long min value: 0" << endl;
    cout << "Unsigned long max value: " << ULONG_MAX << endl;
    return 0;
}
```

6 Declarations and Definitions

Declarations introduce names into a program and specify their types:

```
int a; // variable declaration
int add_int(int, int); // function declaration (without function body)
```

Definitions tell the program what the declared name contains and initialise objects.

```
int a = 7; // variable definition
int add_int(int a, int b) {
    return a + b; // function declaration (with function body)
}
```

Let's see an example of declarations and definitions:

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

int add_int(int, int); // function declaration

int main()
{
    int a = 1; // variable declaration and definition
    int b = 2; // variable declaration and definition
    int c = add_int(a, b); // variable declaration and initialisation
                           // with the value returned by function add_int.
                           // The program doesn't know the value yet,
                           // but it already knows the value will be of type int
                           // because the function was already declared
    cout << a << "+" << b << " = " << c << endl;
}

int add_int(int a, int b) {
    return a + b; // function definition
}
```

In this example, the function `add_int` is declared at the beginning of the program and is defined at the end of the program. Declaring the function at the beginning allows for its usage for the definition of variable `c`, regardless of not knowing what the function does. The relevant information for using this function for defining `c` is the return type of the function. This is usually referred as *abstraction* and provides flexibility for large programs through the use of header files.

7 Scope

Scope refers to the region of code where a name (variable, function, etc.) is visible. C++ supports several kinds of scopes: Global Scope, Local Scope, Class Scope, Namespace Scope.

- Global Scope: Names declared outside any function or class.
- Local Scope: Names declared within a function or a block.

Let's recall our function example with a little change:

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

int add_int(int, int); // function declaration

int a = 1; // variable in the global scope
int main()
{
    int b = 2; // variable in the scope of main
    for (int i = 0; i < 10; ++i){
        int c = i + add_int(a, b); // variable in the scope of the loop statement
        cout << i << "+" << a << "+" << b << " = " << c << endl;
    }
}

int add_int(int a, int b) {
  return a + b;
}
```

- Class Scope: Names declared within a class. Classes will be explained in future lectures.
- Namespace Scope: Names declared within a namespace.

8 Namespace

Namespaces prevent name conflicts by creating logical scopes. The typical structure of a namespace is:

```cpp
namespace MyNamespace {
    int x; // variable in namespace
    void myFunction() { // function in namespace
        // ...
    }
}
```

There are different ways to use names from a namespace:

- Calling the namespace (hint: you may have noticed the line `using namespace std` in the previous examples)

```cpp
using namespace MyNamespace;
x = 10;
myFunction();
```

- Using fully qualified names

```cpp
MyNamespace::x = 10;
MyNamespace::myFunction();
```

Let's see an example:

```cpp
using namespace std;
namespace string_vals {
    string a;
    string b;
    string plus_operation(string a, string b) {
        return a + b;
    }
}
namespace int_vals {
    int a;
    int b;
    int plus_operation(int a, int b) {
        return a + b;
    }
}
```

We have created two namespaces: `string_vals`, `intvals`. Both include variables with the same name (`a`, `b`) and a function with the same name (`plus_operation`). Now let's use them.

Option 1: `using namespace NamespaceName+`

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

namespace string_vals {
    string a;
    string b;
    string plus_operation(string a, string b) {
        return a + b;
    }
}
namespace int_vals {
    int a;
    int b;
    int plus_operation(int a, int b) {
        return a + b;
    }
}

using namespace string_vals; // define the namespace you want to use

int main()
{
    a = "A"; // variable definition should be consistent with the type in the namespace
    b = "B";
    cout << a << "+" << b << " = " << plus_operation(a, b) << endl;
}
```

We have chosen to use only one of the namespaces (`string_vals`). What if we use both?... Using both results in ambiguous references. This means that the program will be confused trying to find out which declaration of the variables and the function should be used. There is a way to avoid this issue...

Option 2: Using fully qualified names

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

namespace string_vals {
    string a;
    string b;
    string plus_operation(string a, string b) {
        return a + b;
    }
}
namespace int_vals {
    int a;
    int b;
    int plus_operation(int a, int b) {
        return a + b;
    }
}

int main()
{
    // using the string_vals namespace
    string_vals::a = "A";
    string_vals::b = "B";
    cout << string_vals::a << "+" << string_vals::b << " = " <<
    string_vals::plus_operation(string_vals::a, string_vals::b) << endl;
    // using the int_vals namespace
    int_vals::a = 1;
    int_vals::b = 2;
    cout << int_vals::a << "+" << int_vals::b << " = " <<
    int_vals::plus_operation(int_vals::a, int_vals::b) << endl;
}
```

In this case you explicitly refer to the namespace (`string_vals::`, `int_vals::`). This ensures that there's no ambiguity with names, allowing for using both namespaces at the same time.

References

- Stroustrup, B. (2013). The C++ Programming Language. Addison-Wesley.
- Meyers, S. (2005). Effective C++. Addison-Wesley.
- Josuttis, N. M. (2012). The C++ Standard Library: A Tutorial and Reference. Addison-Wesley.
- Sutter, H., & Alexandrescu, A. (2005). C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Addison-Wesley.
- ISO/IEC. (2023). Programming Languages - C++ Standard (N4928). Retrieved from ISO C++.