Common Errors in C++
## Contents

1 Common Syntax Errors ........................................ 1
   1.1 Use of undeclared identifier .......................... 1
   1.2 Identifier not found .................................. 2
   1.3 Missing ‘;’ before identifier ......................... 4
   1.4 Expected unqualified-id .............................. 5
   1.5 Expected ‘}’ to match this ‘{’ ...................... 5
   1.6 Errors with include and using statements .......... 6

2 Common Logic Errors ........................................ 8
   2.1 Using uninitialized variables ....................... 8
   2.2 Using a single equal sign to check equality ........ 9
   2.3 Overstepping array boundaries ...................... 10
   2.4 Integer overflow .................................. 11
   2.5 Errors with Quote Types ............................ 12
1 Common Syntax Errors

It is incredibly helpful to know how to read error messages. These will tell you exactly where and what is going wrong in your code. While there may be some extra information, there will always be at least one line we want to pay attention to:

```
filename.cpp:7:15: error: error message
```

The first number after the file name refers to the line in the code that is giving the error. The second number is the character\(^1\) where the error starts. In the above example, the error occurs on line 7, starting at character 15.

In most cases the error will be on the same line shown in the message. Sometimes, though, the error could be above the given line. This most often happens when a semicolon is forgotten on the line above, causing the line in the message to give an error. If you can’t find an error on the line given in the message, start working up the code line by line and checking each one for any mistakes.

1.1 Use of undeclared identifier

A variable was either not declared, or misspelled somewhere in the code.

Example of code causing error:

```
int main()
{
    cout << x;
    return 0;
}
```

\(^1\)Characters refer to the individual letters, space, and punctuation marks.
In this example, x was never declared before we attempted to use it. To fix this we simply need to declare x before using it:

```c++
int main()
{
    int x = 0;
    cout << x;
    return 0;
}
```

Misspellings are another common cause of this issue. Capitalization matters in C++, so it could be as simple as declaring a variable:

```c++
string myVariable;
```

then trying to use it as:

```c++
cout << myvariable;
```

The message for these types of errors will print the variable name it believes is undeclared before the rest of the message, giving a big hint as to what is wrong. If you know you declared the variable, make sure it is spelled and capitalized the same throughout your code.

1.2 Identifier not found

A function was not declared, or misspelled somewhere in the code.

Just like variables, functions in C++ need to be declared before they can be used. A function declaration is simply the function header placed at the top of the code, usually just below any include statements and the using namespace directive.

Example of code causing error:
In this example, `menu` is called in `main` and defined below. It hasn’t been declared yet, though, and therefore will give an error.

To fix this, we must declare the `menu` function, like so:

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

void menu();

int main()
{
    menu();
    return 0;
}

void menu()
{
    //...
}
```

Note that when declaring a function, it should match the header of the function definition exactly, followed by a semicolon. If there are any differences between the function declaration and definition header, another type of error will occur.
The messages for these types of errors will print out the function name causing the problem, just like with undeclared identifiers mentioned above, so it will again point out the exact location of the problematic code.

*This message may also say undeclared identifier, depending on the compiler.

1.3 Missing ‘;’ before identifier

There is a missing semicolon somewhere in the code.

This is a very common, and easy to make, mistake. Everything in C++ gets a semicolon at the end except include statements, and anything that is immediately followed by curly braces, such as function definitions, if/else if/else statements, and loops.

Example of code causing error:

```cpp
int main()
{
    cout << "Hello World!"
    return 0;
}
```

In the above example, there is a missing semicolon at the end of “Hello World”. To fix this, we simply need to add in the missing semicolon:

```cpp
int main()
{
    cout << "Hello World!";
    return 0;
}
```

These error messages will usually give the expression preceding where the missing semicolon should be, or even sometimes point to exactly where it’s missing from.

*This message may also say expected ‘;’ after expression, depending on the compiler.
1.4 **Expected unqualified-id**

There is probably an extra semicolon somewhere in the code.

These errors usually occur when there is an extra semicolon somewhere in the code, such as after a function header, a loop header, or an if/else if/else header.

Example of code causing error:

```c
int main();
{
    //...
}
```

In this example, the semicolon after main() is causing the error. To fix this error, simply remove the extra semicolon:

```c
int main()
{
    //...
}
```

Semicolons only go after complete statements. If you’re unsure, think about what comes immediately after the line in question. If it’s an opening curly brace, then there should not be a semicolon.

*This message may also state missing function header, depending on the compiler.*

1.5 **Expected ‘}’ to match this ‘{’**

This could be either a missing curly brace, or an extra curly brace.

This is another easy mistake to make. Sometimes this error message will show us exactly where the missing curly brace is, but if your code has multiple opening and closing curly braces, it can be tougher to track down.

Go through your code, starting with the top and working down, and make sure every time you open a curly brace that it has a matching closing
curly brace.

This can be easier to notice if you place your curly braces on their own lines, so it’s more noticeable how they match up.

Example of code causing error:

```cpp
int main()
{
    int x = 5;
    if(x == 5)
    {
        cout << x;
        return 0;
    }
}
```

In this example, our if statement opens a curly brace but doesn’t close it off.
To fix this, we simply add in the missing curly brace:

```cpp
int main()
{
    int x = 5;
    if(x == 5)
    {
        cout << x;
    }
    return 0;
}
```

1.6 Errors with include and using statements

Other common syntax errors occur when there is a missing include statement, or a missing using namespace std directive.

A missing include statement will look similar to an undeclared identifier error. If you’re using certain keywords, and you’re positive that there aren’t
any spelling errors, then you’re probably missing an include statement.

Common include statements are:

```
#include <iostream>
#include <string>  // include <cstdlib>
#include <cmath>  // include <algorithm>
```

A missing using namespace std error will usually print out some extra information. It generally appears as an undeclared identifier error, with a message asking ‘did you mean’. This message is usually followed by the path for where the keyword is installed.

Example of code causing error:

```
#include <iostream>

int main()
{
    cout << "Hello World";
    return 0;
}
```

The error message for the above code will look something similar to:
```
error: use of undeclared identifier ‘cout’; did you mean ‘std::cout’?
```

We can fix this by either adding the using namespace std directive before main, like so:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World!";
    return 0;
}
```
Or, if we wanted to exclude the using namespace directive, we could add the namespace and scope operator to cout, like so:

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello World!";
    return 0;
}
```

## 2 Common Logic Errors

Logic errors don’t usually show up as an error message, but instead cause your code to act unexpectedly.

While these won’t show error messages, and will let you compile successfully, some compilers will show warning messages. These start with the word warning: and won’t stop the code from compiling, but they will give you a clue that something might not be quite right.

### 2.1 Using uninitialized variables

Uninitialized variables mean we have declared a variable without a value, then tried to use it somewhere.

In C++, when we declare a variable without giving it a value, the compiler will go ahead and give the variable a value as a placeholder. While many assume the value would be 0, the compiler in fact can set it to any value it feels like, so it may not act like you’d expect.

Example code with uninitialized variable:
int main()
{
    int x;
    cout << x;
    return 0;
}

This will print out a random number that the compiler gave to the variable x. To fix this kind of logic error, always be sure to initialize your variables before using them.

Example code with initialized variable:

```cpp
int main()
{
    int x = 0;
    cout << x;
    return 0;
}
```

### 2.2 Using a single equal sign to check equality

In C++, a single equal sign (=) will always set the variable on the left of the equal sign to the value on the right of the equal sign. If we need to check if a variable is the same as a value (be it a number, letter, boolean, string, etc.) then we need to use double equal signs (==).

Example with logic error:

```cpp
int main()
{
    string goAgain = "Yes";
    while(goAgain = "Yes")
    {
        cout << "Continue?" << endl;
        cin >> goAgain;
    }
    return 0;
}
```
This example will end up in an infinite loop because our while condition
(goAgain = "Yes"). By using the single equal sign, we are constantly setting
the variable goAgain to "Yes".

To check if a variable is the same as a value, we want to use the double
equal sign:

```cpp
int main()
{
    string goAgain = "Yes";
    while(goAgain == "Yes")
    {
        cout << "Continue?" << endl;
        cin >> goAgain;
    }
    return 0;
}
```

2.3 Overstepping array boundaries

These logic errors occur when we attempt to access an element that doesn’t
exist in an array.

When we declare an array, we declare it with a size in square brackets.
Overstepping array boundaries happens when we try to get an element with
an index larger than the size we declared the array with.

Example code with logic error:

```cpp
int main()
{
    int myNumbers[5] = {63, 1, 22, 13, 91};
    cout << myNumbers[13] << endl;
    return 0;
}
```

In the above example we declared an array with a size of 5, meaning our
indices are 0 - 4. 13 is definitely outside of that range, so we end up with
a logic error. What generally happens in this instance is it will print out a
random number. While it won’t stop your code from running, it will make it run in unintended ways.

In the above example it is fairly easy to spot when we go out of bounds of the array. This error most commonly occurs in a less easy to spot way: loops.

Example code with logic error:

```cpp
int main()
{
    int myNumbers[5] = {63, 1, 22, 13, 91};
    for(int i = 0; i <= 5; i++)
    {
        cout << myNumbers[i] << endl;
    }
    return 0;
}
```

The above example logic error is harder to spot because it looks like it would act expectantly. However, our for loop condition (i <= 5) will cause a problem. Keep in mind that array indices start counting at 0, so our indices are are 0 - 4. By including the equal sign in our condition, i will eventually equal 5, which is not a valid index for our array.

### 2.4 Integer overflow

When we declare a variable, we give it a data type, such as int, float, double, string, char, etc. In doing so, the computer will make room in memory. The amount of room in memory devoted to a variable is dependent on the datatype, and the computer itself\(^2\).

An integer overflow occurs when we try to save to an int variable a number that is too big, or too small, to fit in an integer datatype.

Example code with logic error:

\(^2\)This range in size can vary, so it’s best to search for datatype limits based on your own system.
int main()
{
    int x = 9999999999;
    cout << x;
    return 0;
}

On my system, the above code will print 1410065407. This is because the number 9999999999 is too big to store in an integer datatype variable. This causes what is called an overflow.

While it may seem obvious when setting a single variable that a number may be too large for an integer, these kinds of logic errors most commonly occur when performing arithmetic.

To avoid this problem if you know a number will be very large when declared, or if a number can possible get very large from doing arithmetic, we should declare it as a long integer. This is done as follows:

long int x = 9999999999;

This long keyword can also be used with any other number data type, such as floats or doubles.

2.5 Errors with Quote Types
In C++, the type of quote used matters. Single quotes (’ ’) are used for characters\(^3\). Double quotes (” ”) are used for strings.

While you can use double quotes for characters without issue, you cannot go the other way around and use single quotes for strings.

This kind of error, like the above mentioned logic errors, won’t stop the code from compiling. Instead, it give a warning and act in a way we don’t expect.

Example code with logic error:

\(^3\)Characters refer to the individual letters, numbers, spaces, and/or punctuation marks.
As we’re using single quotes for a full string in the example above, the compiler will give us a warning. Warnings won’t stop us from compiling, but they give a hint that something in our code isn’t quite right. This warning will look along the lines of:

```
warning: multi-character character constant
or
warning: character constant too long for its type
```

A syntax error can also occur when misusing quotation marks. In C++, opening and closing quotation marks have to match.

Example code with syntax and logic error:

```
int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

This example will give both a warning and an error message:

```
warning: missing terminating ' " ' character
error: expected expression
```