This document is part of the HTML publication "**An Introduction to the Imperative Part of C++**"

The original version was produced by **Rob Miller** at **Imperial College London**, September 1996.

Version 1.1 (modified by **David Clark** at **Imperial College London**, September 1997)

Version 1.2 (modified by **Bob White** at **Imperial College London**, September 1998)

Version 1.3, 1.4, 2.0, ..., 2.15 (modified by **William Knottenbelt** at **Imperial College London**, September 1999-September 2016)

2. Variables, Types and Expressions

2.1 Identifiers

As we have seen, C++ programs can be written using many English words. It is useful to think of words found in a program as being one of three types:

1. Reserved Words. These are words such as `if`, `int` and `else`, which have a predefined meaning that cannot be changed. Here's a more complete list.
2. Library Identifiers. These words are supplied default meanings by the programming environment, and should only have their meanings changed if the programmer has strong reasons for doing so. Examples are `cin`, `cout` and `sqrt` (square root).
3. Programmer-supplied Identifiers. These words are "created" by the programmer, and are typically variable names, such as `year_now` and `another_age`.

An identifier cannot be any sequence of symbols. A valid identifier must start with a letter of the alphabet or an underscore ("_") and must consist only of letters, digits, and underscores.

(BACK TO COURSE CONTENTS)

2.2 Data Types

Integers

C++ requires that all variables used in a program be given a data type. We have already seen the data type `int`. Variables of this type are used to represent integers (whole numbers). Declaring a variable to be of type `int` signals to the compiler that it must associate enough memory with the variable's identifier to store an integer value or integer values as the program executes. But there is a (system dependent) limit on the largest and smallest integers that can be stored. Hence C++ also supports the data types `short int` and `long int` which represent, respectively, a smaller and a larger range of integer values than `int`. Adding the prefix `unsigned` to any of these types means that you wish to represent non-negative integers only. For example, the declaration

```
unsigned short int year_now, age_now, another_year, another_age;
```

reserves memory for representing four relatively small non-negative integers.

Some rules have to be observed when writing integer values in programs:

1. Decimal points cannot be used; although 26 and 26.0 have the same value, "`26.0`" is not of type "`int`".
2. Commas cannot be used in integers, so that (for example) 23,897 has to be written as "`23897`".
3. Special prefixes can be used to control the base of the number system used to interpret the value of the integer. A leading "`0`" indicates that the number following is an octal (base 8) number. For example the compiler will interpret "`011`" as the octal number 011, with decimal value 9. Similarly "`0x`" can be used to specify a hexademical (base 16) number and, at least in C++14, "`0b`" can be used to specify a binary (base 2) number.

(BACK TO COURSE CONTENTS)

Real numbers

Variables of type "`float`" are used to store real numbers. Plus and minus signs for data of type "`float`" are treated exactly as with integers, and trailing zeros to the right of the decimal point are ignored. Hence "`+523.5`", "`523.5`" and "`523.500`" all represent the same value. The computer also accepts real numbers in *floating-point* form (or "scientific notation"). Hence 523.5 could be written as "`5.235e+02`" (i.e. 5.235 x 10 x 10), and -0.0034 as "`-3.4e-03`". In addition to "`float`", C++ supports the types "`double`" and "`long double`", which give increasingly precise representation of real numbers, but at the cost of more computer memory.

Type Casting

Sometimes it is important to guarantee that a value is stored as a real number, even if it is in fact a whole number. A common example is where an arithmetic expression involves division. When applied to two values of type `int`, the division operator "`/`" signifies integer division, so that (for example) `7/2` evaluates to 3. In this case, if we want an answer of 3.5, we can simply add a decimal point and zero to one or both numbers - "`7.0/2`", "`7/2.0`" and "`7.0/2.0`" all give the desired result. However, if both the numerator and the divisor are variables, this trick is not possible. Instead, we have to use a type cast. For example, we can convert "`7`" to a value of type `double` using the expression "`static_cast<double>(7)`". Hence in the expression

```
answer = static_cast<double>(numerator) / denominator
```

the "`/`" will always be interpreted as real-number division, even when both "`numerator`" and "`denominator`" have integer values. Other type names can also be used for type casting. For example, "`static_cast<int>(14.35)`" has an integer value of 14.

(BACK TO COURSE CONTENTS)

Characters

Variables of type "`char`" are used to store character data. In standard C++, data of type "`char`" can only be a single character (which could be a blank space). These characters come from an available character set which can differ from computer to computer. However, it always includes upper and lower case letters of the alphabet, the digits `0`, ... , `9`, and some special symbols such as `#`, `£`, `!`, `+`, `-`, etc. Perhaps the most common collection of characters is the ASCII character set (see for example Savitch, Appendix 3 or just click here).

Character constants of type "`char`" must be enclosed in single quotation marks when used in a program, otherwise they will be misinterpreted and may cause a compilation error or

unexpected program behaviour. For example, "`'A'`" is a character constant, but "`A`" will be interpreted as a program variable. Similarly, "`'9'`" is a character, but "`9`" is an integer.

There is, however, an important (and perhaps somewhat confusing) technical point concerning data of type "`char`". Characters are represented as integers inside the computer. Hence the data type "`char`" is simply a subset of the data type "`int`". We can even do arithmetic with characters. For example, the following expression is evaluated as true on any computer using the ASCII character set:

```
'9' - '0' == 57 - 48 == 9
```

The ASCII code for the character '9' is decimal 57 (hexadecimal 39) and the ASCII code for the character '0' is decimal 48 (hexadecimal 30) so this equation is stating that

57(dec) - 48(dec) == 39(hex) - 30(hex) == 9

It is often regarded as better to use the ASCII codes in their hexadecimal form.

However, declaring a variable to be of type "`char`" rather than type "`int`" makes an important difference as regards the type of input the program expects, and the format of the output it produces. For example, the program

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

int main()
{
        int number;
        char character;

        cout << "Type in a character:\n";
        cin >> character;

        number = character;

        cout << "The character '" << character;
        cout << "' is represented as the number ";
        cout << number << " in the computer.\n";

        return 0;
}
```

Program 2.2.1

produces output such as

```
Type in a character:
9
The character '9' is represented as the number 57 in the computer.
```

We could modify the above program to print out the whole ASCII table of characters using a "for loop". The "for loop" is an example of a *repetition statement* - we will discuss these in more detail later. The general syntax is:

```
for (initialisation; repetition_condition ; update) {
        Statement1;
        ...
        ...
        StatementN;
}
```

C++ executes such statements as follows: (1) it executes the *initialisation* statement. (2) it checks to see if *repetition_condition* is true. If it isn't, it finishes with the "for loop" completely. But if it is, it executes each of the statements *Statement1* ... *StatementN* in turn, and then executes the expression *update*. After this, it goes back to the beginning of step (2) again.

We can also 'manipulate' the output to produce the hexadecimal code. Hence to print out the ASCII table, the program above can be modified to:

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

int main()
{
        int number;
        char character;

        for (number = 32 ; number <= 126 ; number = number + 1) {

                character = number;
                cout << "The character '" << character;
                cout << "' is represented as the number ";
                cout << dec << number << " decimal or "
                  <<hex<<number<< " hex.\n";
        }

        return 0;
}
```

Program 2.2.2

which produces the output:

```
The character ' ' is represented as the number 32 decimal or 20 hex.
The character '!' is represented as the number 33 decimal or 21 hex.
...
...
The character '}' is represented as the number 125 decimal or 7D hex.
The character '~' is represented as the number 126 decimal or 7E hex.
```

(BACK TO COURSE CONTENTS)

Strings

Our example programs have made extensive use of the type "`string`" in their output. As we have seen, in C++ a string constant must be enclosed in double quotation marks. Hence we have seen output statements such as

```
cout << "' is represented as the number ";
```

in programs. In fact, "`string`" is not a fundamental data type such as "`int`", "`float`" or "`char`". Instead, strings are represented as arrays of characters, so we will return to subject of strings later, when we discuss arrays in general.

User Defined Data Types

Later in the course we will study the topic of data types in much more detail. We will see how the programmer may define his or her own data types. This facility provides a powerful

programming tool when complex structures of data need to be represented and manipulated by a C++ program.

(BACK TO COURSE CONTENTS)

2.3 Some Tips on Formatting Real Number Output

When program output contains values of type "`float`", "`double`" or "`long double`", we may wish to restrict the precision with which these values are displayed on the screen, or specify whether the value should be displayed in fixed or floating point form. The following example program uses the library identifier "`sqrt`" to refer to the square root function, a standard definition of which is given in the header file `cmath` (or in the old header style `math.h`).

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

int main()
{
        float number;

        cout << "Type in a real number.\n";
        cin >> number;
        cout.setf(ios::fixed);    //    LINE 10
        cout.precision(2);
        cout << "The square root of " << number << " is approximately ";
        cout << sqrt(number) << ".\n";

        return 0;
}
```

Program 2.3.1

This produces the output

```
Type in a real number.
200
The square root of 200.00 is approximately 14.14.
```

whereas replacing line 10 with "`cout.setf(ios::scientific)`" produces the output:

```
Type in a real number.
200
The square root of 2.00e+02 is approximately 1.41e+01.
```

We can also include tabbing in the output using a statement such as "`cout.width(20)`". This specifies that the next item output will have a width of at least 20 characters (with blank space appropriately added if necessary). This is useful in generating tables. However the C++ compiler has a default setting for this member function which makes it right justified. In order to produce output left-justified in a field we need to use some fancy input and output manipulation. The functions and operators which do the manipulation are to be found in the library file `iomanip` (old header style `iomanip.h`) and to do left justification we need to set a flag to a different value (i.e. left) using the `setiosflags` operator:

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

int main()
```

```
{
        int number;

        cout << setiosflags ( ios :: left );
        cout.width(20);
        cout << "Number" << "Square Root\n\n";

        cout.setf(ios::fixed);
        cout.precision(2);
        for (number = 1 ; number <= 10 ; number = number + 1) {
                cout.width(20);
                cout << number << sqrt( (double) number) << "\n";
        }

        return 0;
}
```

Program 2.3.2

This program produces the output

```
Number              Square Root

1                   1.00
2                   1.41
3                   1.73
4                   2.00
5                   2.24
6                   2.45
7                   2.65
8                   2.83
9                   3.00
10                  3.16
```

(In fact, the above programs work because "`cout`" is an identifier for an object belonging to the class "`stream`", and "`setf(...)`", "`precision(...)`" and "`width(...)`" are member functions of "`stream`". Don't worry too much about this for now - you will learn more about objects, classes and member functions later in the object-oriented part of the course.)

(BACK TO COURSE CONTENTS)

2.4 Declarations, Constants and Enumerations

As we have already seen, variables have to be declared before they can be used in a program, using program statements such as

```
float number;
```

Between this statement and the first statement which assigns "`number`" an explicit value, the value contained in the variable "`number`" is arbitrary. But in C++ it is possible (and desirable) to initialise variables with a particular value at the same time as declaring them. Hence we can write

```
double PI = 3.1415926535;
```

Furthermore, we can specify that a variable's value cannot be altered during the execution of a program with the reserved word "`const`":

Enumerations

Constants of type "int" may also be declared with an enumeration statement. For example, the declaration

```
enum { MON, TUES, WED, THURS, FRI, SAT, SUN };
```

is shorthand for

```
const int MON = 0;
const int TUES = 1;
const int WED = 2;
const int THURS = 3;
const int FRI = 4;
const int SAT = 5;
const int SUN = 6;
```

By default, members of an "enum" list are given the values 0, 1, 2, etc., but when "enum" members are explicitly initialised, uninitialised members of the list have values that are one more than the previous value on the list:

```
enum { MON = 1, TUES, WED, THURS, FRI, SAT = -1, SUN };
```

In this case, the value of "FRI" is 5, and the value of "SUN" is 0.

(BACK TO COURSE CONTENTS)

Where to put Constant and Variable Declarations

Generally speaking, it is considered good practice to put constant declarations before the "main" program heading, and variable declarations afterwards, in the body of "main". For example, the following is part of a program to draw a circle of a given radius on the screen and then print out its circumference:
(There is no need to type in this program)

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

const float PI = 3.1415926535;
const float SCREEN_WIDTH = 317.24;

int drawCircle(float diameter); /* this is a "function prototype" */

int main()
{
        float radius = 0;

        cout << "Type in the radius of the circle.\n";
        cin >> radius;

        drawCircle(radius * 2);

        cout.setf(ios::fixed);
        cout.precision(2);
        cout << "The circumference of a circle of radius " << radius;
        cout << " is approximately " << 2 * PI * radius << ".\n";
        return 0;
}

int drawCircle(float diameter)
{
        float radius = 0;

        if (diameter > SCREEN_WIDTH)
```

```
                radius = SCREEN_WIDTH / 2.0;
        else
                radius = diameter / 2.0;
        ...
        ...
}
```

After the definition of "`main()`", this program includes a definition of the function "`drawCircle(...)`", the details of which need not concern us here (we can simply think of "`drawCircle(...)`" as a function like "`sqrt(...)`"). But notice that although both "`main()`" and "`drawCircle(...)`" use the identifier "`radius`", this refers to a *different* variable in "`main()`" than in "`drawCircle(...)`". Had a variable "`radius`" been declared before the "`main`" program heading, it would have been a public or *global variable*. In this case, and assuming there was no other variable declaration inside the function "`drawCircle(...)`", if "`drawCircle(...)`" had assigned it the value "`SCREEN_WIDTH / 2.0`", "`main()`" would have subsequently printed out the wrong value for the circumference of the circle. We say that the (first) variable "`radius`" is *local to the main part of the program*, or *has the function* `main` *as its scope*. In contrast, it usually makes sense to make constants such as "`PI`" and "`SCREEN_WIDTH`" *global*, i.e. available to every function.

In any case, notice that the program above incorporates the safety measure of *echoing the input*. In other words, the given value of "`radius`" is printed on the screen again, just before the circumference of the circle is output.

(BACK TO COURSE CONTENTS)

2.5 Assignments and Expressions

Shorthand Arithmetic Assignment Statements

We have already seen how programs can include variable assignments such as

```
number = number + 1;
```

Since it is often the case that variables are assigned a new value in function of their old value, C++ provides a shorthand notation. Any of the operators "+" (addition), "-" (subtraction), "*" (multiplication), "/" (division) and "%" (modulus) can be prefixed to the assignment operator (=), as in the following examples (mostly copied from Savitch, Section 2.3):

Example:	**Equivalent to:**
`number += 1;`	`number = number + 1;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rush_factor;`	`time = time / rush_factor;`
`change %= 100;`	`change = change % 100;`
`amount *= count1 + count2;`	`amount = amount * (count1 + count2);`

The first of the above examples may written in even shorter form. Using the increment operator "++", we may simply write

```
number++;
```

The operator "++" may also be used as a prefix operator:

```
++number;
```

but care must be taken, since in some contexts the prefix and postfix modes of use have different effects. For example, the program fragment

```
x = 4;
y = x++;
```

results in "x" having the value 5 and "y" having the value 4, whereas

```
x = 4;
y = ++x;
```

results in both variables having value 5. This is because "++x" increments the value of "x" before its value is used, whereas "x++" increments the value afterwards. There is also an operator "--", which decrements variables by 1, and which can also be used in prefix or postfix form.

In general, assignment statements have a value equal to the value of the left hand side after the assignment. Hence the following is a legal expression which can be included in a program and which might be either evaluated as true or as false:

```
(y = ++x) == 5
```

It can be read as the assertion: "after x is incremented and its new value assigned to y, y's value is equal to 5".

(BACK TO COURSE CONTENTS)

Boolean Expressions and Operators

Intuitively, we think of expressions such as "`2 < 7`", "`1.2 != 3.7`" and "`6 >= 9`" as evaluating to "`true`" or "`false`" ("`!=`" means "not equal to"). Such expressions can be combined using the logical operators "`&&`" ("and"), "`||`" ("or") and "`!`" ("not"), as in the following examples:

Expression:	**True or False:**
`(6 <= 6) && (5 < 3)`	`false`
`(6 <= 6) || (5 < 3)`	`true`
`(5 != 6)`	`true`
`(5 < 3) && (6 <= 6) || (5 != 6)`	`true`
`(5 < 3) && ((6 <= 6) || (5 != 6))`	`false`
`!((5 < 3) && ((6 <= 6) || (5 != 6)))`	`true`

The fourth of these expressions is true because the operator "`&&`" has a higher precedence than the operator "`||`". You can check the relative precedence of the different C++ operators in a C++ programming manual or text book (see for example Savitch, Appendix 2). But if in doubt use `( )` parentheses, which in any case often make the program easier to read.

Compound Boolean expressions are typically used as the condition in "if statements" and "for loops". For example:

```
...
...
if (total_test_score >= 50 && total_test_score < 65)
        cout << "You have just scraped through the test.\n";
...
...
```

Once again, there is an important technical point concerning Boolean expressions. In C++, "`true`" is represented simply as any non-zero integer, and "`false`" is represented as the value 0. This can lead to errors. For example, it is quite easy to type "`=`" instead of "`==`". Unfortunately, the program fragment

```
...
...
if (number_of_people = 1)
        cout << "There is only one person.\n";
...
...
```

will always result in the message "`There is only one person`" being output to the screen, even if the previous value of the variable "`number_of_people`" was not 1.

(BACK TO COURSE CONTENTS)

2.6 Summary

In this lecture we have discussed variables in more detail. We have seen how variables are always of a particular data type, and have listed different ways in which variables may be temporarily or permanently assigned values. We have also seen how new values of various types can be generated by the use of operators. The material here is also covered in more detail in Savitch, Chapter 2, and Sections 4.2 and 4.5.

Exercises
