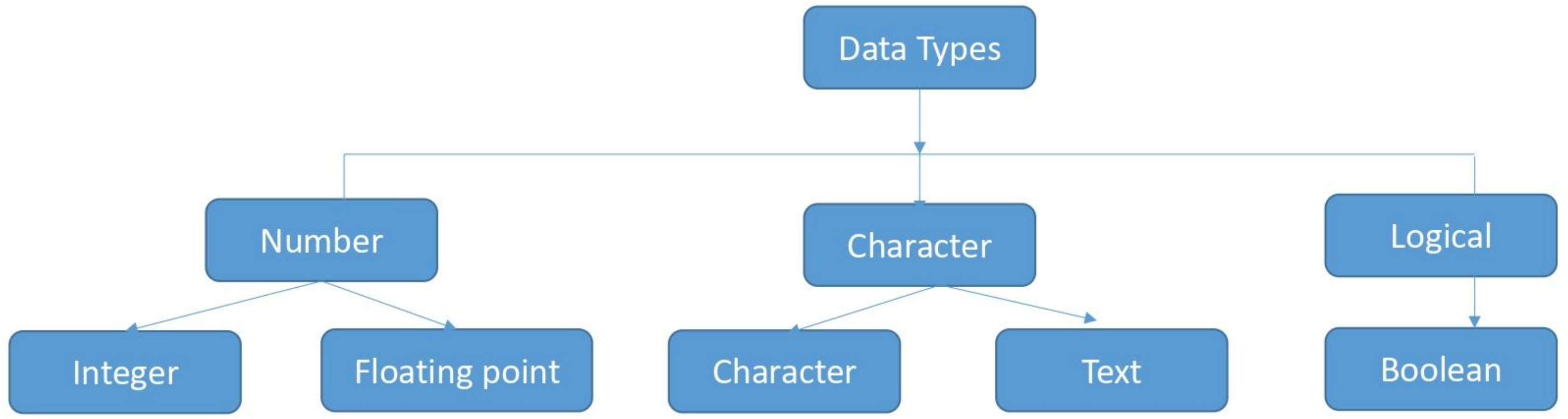
Module 2: Variables and Data Types

Instructor: Khouv Tannhuot
Phone: 087 42 47 36 (Telegram)

Data Types

- Data Type is type of data to be used by the program.
- The data type specifies the size and type of information the variable will store.



Keyword	int	float/double	char	string	bool
Value	10	10.50	'A'	"Hello"	true or false

Data Types

Several of the basic types can be modified using one or more of these type modifiers:

- signed
- unsigned
- short
- long

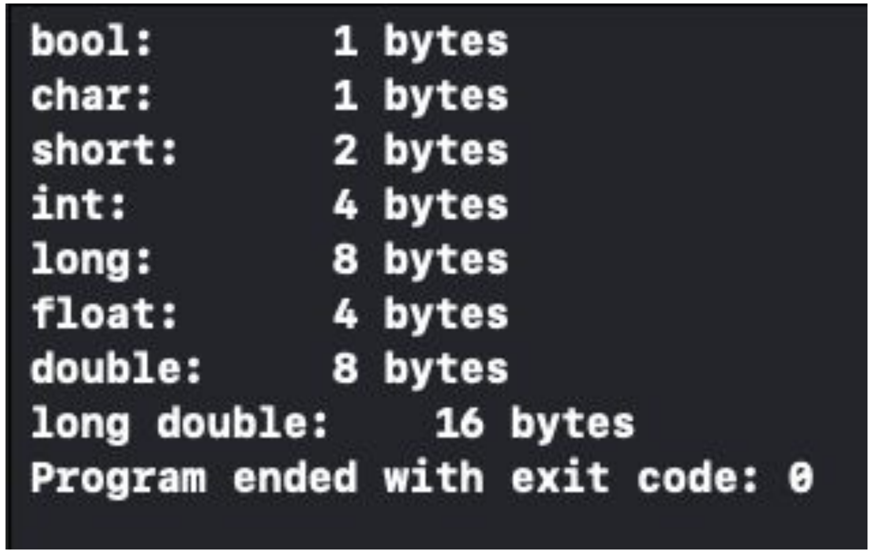
Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-9223372036854775808 to 9223372036854775807
signed long int	8bytes	same as long int
unsigned long int	8bytes	0 to 18446744073709551615
long long int	8bytes	-(2^63) to (2^63)-1
unsigned long long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	
double	8bytes	
long double	12bytes	

IT ACADEMY

Display the size of data types

```
int main(int argc, const char * argv[]) {
        std::cout << "bool:\t\t" << sizeof(bool) << " bytes\n";
        std::cout << "char:\t\t" << sizeof(char) << " bytes\n";
        std::cout << "short:\t\t" << sizeof(short) << " bytes\n";
        std::cout << "int:\t\t" << sizeof(int) << " bytes\n";
        std::cout << "long:\t\t" << sizeof(long) << " bytes\n";
        std::cout << "float:\t\t" << sizeof(float) << " bytes\n";
        std::cout << "double:\t\t" << sizeof(double) << " bytes\n";
        std::cout << "long double:\t" << sizeof(long double) << " bytes\n";
        return 0;
}
```

Output on MacOS:



```
bool:           1 bytes
char:           1 bytes
short:          2 bytes
int:            4 bytes
long:           8 bytes
float:          4 bytes
double:         8 bytes
long double:    16 bytes
Program ended with exit code: 0
```

Variable

- Variable is a data container, allow us to store temporary information in our program.

```
Syntax: datatype variable_name = value;
```

Example:

```
int age = 18;
string language = “C++”;
```

Variable

Every C++ variable has:

- **Name:** chosen by the programmer
- **Type:** specified in the declaration of the variable
- **Size:** determined by the type
- **Value:** the data stored in the variable’s memory location
- **Address:** the starting location in memory where the value is stored

Naming Conventions

Variable names can’t contain:

- whitespace characters
- mathematical symbols
- special characters
- cannot use reserved keyword
- Nor can they begin with a number, although numbers may be included elsewhere within the name

Example:

```
int maximum_number_of_login_attempts = 10;
int current_login_attempt = 0;
string server1 = “server 1”;
```

Variables vs Constants

Variable: is a name that can be used to store data that can be **changed** in computer memory while the program is running.

- Syntax:
  - data_type variable_name = value;
  - **int a = 10;**
  - **a = 20; //valid**

Constant variable: is a name that can be used to store data that **can not be** changed in computer memory while the program is running.

- Syntax:
  - const data_type variable_name = value;
  - **const int a = 10;**
  - **a = 20; //error because constant variable cannot change value**

Comments

Use comments to include non executable text in your code, as a note or reminder to yourself. Comments starts with a // and compiler will ignore them:

```
// This is a comment.
```

Multiline comments start with a forward-slash followed by an asterisk (/*) and end with an asterisk followed by a forward-slash (*/):

```
/*
This is also a comment
but is written over multiple lines.
*/
```

Literals

- Literals are also constants. They are literal values written in code.
- Integer literal: an actual integer number written in code (4, 10, 18)
  - If an integer literal is written with a leading 0, it’s interpreted as an octal value (base 8)
  - If an integer literal is written with a leading 0x, it’s interpreted as an octal value (base 16)

Example:

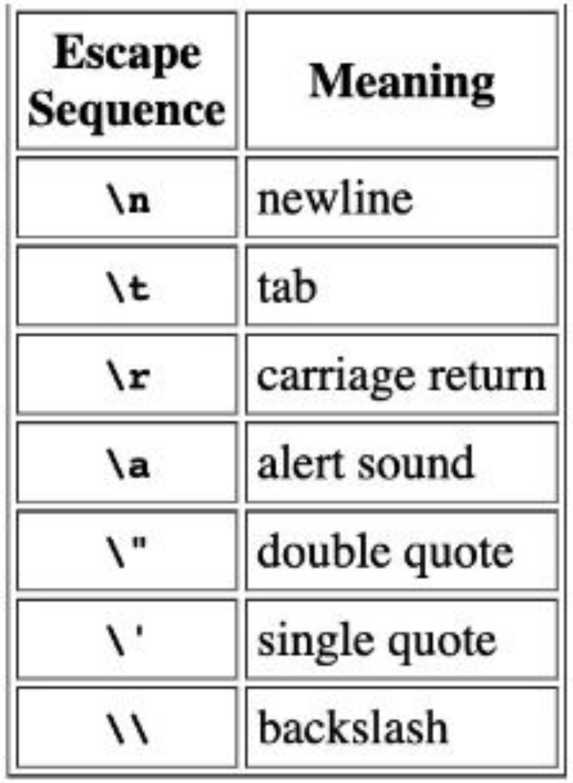
```
int x = 26;       // integer value 26
int y = 032;      // octal 32 = decimal value 26
int z = 0x1A;     // hex = decimal value 26
```

Literals

- Floating point literal: an actual decimal number written in code (4.5, -10.9, 2.0)
    - Note: these are interpreted as type double by standard C++ compilers
    - Can also be written in exponential (scientific) notation(3.12e5, 1.23e-10)
- Character literal: a character in single quotes: ('A', 'b', '\n')
- String Literal: a string in double quotes: ("Hello", "Bye", "Wow!\n")

Escape Sequences

- String and character literals can contain special escape sequences.
- They represent single characters that cannot be represented with a single character from the keyboard in your code.
- The backslash \ is the indicator of an escape sequence. The backslash and the next character are together considered ONE item (one char)



Escape Sequence	Meaning
`\n`	newline
`\t`	tab
`\r`	carriage return
`\a`	alert sound
`\"`	double quote
`\'`	single quote
`\\`	backslash

Practical

1. Write a program to display these texts to console

```
My name is - Dara
I'm loving  "C++ Programming"
C++ is fun
That's all!!!
```

2. Write a program display the output to console

```
Subject                  Marks
Fundamental Computer     90
C++ Programming 1        77
Graphic Design           69
```

Input and Output with stream in C++

- In C++ we use do I/O with "stream objects", which are tied to various input/output devices.

  Some pre-defined stream objects

- These stream objects are defined in the iostream library
  - **cout**: standard output stream
    - Of class type ostream (to be discussed later)
    - Usually defaults to the monitor
  - **cin**: standard input stream
    - Of class type istream (to be discussed later)
    - Usually defaults to the keyboard
  - **cerr**: standard error stream
    - Of class type ostream (to be discussed later)
    - Usually defaults to the monitor

Input and Output with stream in C++

- To use these streams, we need to include the **iostream** library into our programs.

  #include <iostream>

  using namespace std; //The using statement tells the compiler that all uses of these names (cout, cin, etc) will come from the "standard" namespace.

Using Output Streams

- output streams are frequently used with the **insertion operator <<**
  - The right side of the insertion operator can be a variable, a constant, a value, or the result of a computation or operation

Example:

```
cout << "HelloWorld";    //print a string literal
cout << 'a';             //print a character literal
cout << numStudents;     //print contents of a variable
cout << x+y-z;           //print result of a computation
cerr << "Error occurred";      //string literal printed to standard error
```

- When printing multiple items, the insertion operator can be "cascaded". Place another operator after an output item to insert a new output item:
  - `cout << "Average= " << avg << '\n';`
  - `cout << var1 << '\t' << var2 << '\t' << var3;`

Using Input Streams

- input streams are frequently used with the extraction **operator >>**
  - ➢ The right side of the extraction operator MUST be a memory location. For now, this means a single variable!
  - ➢ By default, all built-in versions of the extraction operator will ignore any leading "white-space" characters (spaces, tabs, newlines, etc)

  Examples:

```
int numStudents;
cin >> numStudents; // read an integer
double weight;
cin >> weight;      // read a double
cin >> '\n';        // ILLEGAL. Right side must be a variable
cin >> x + y;   // ILLEGAL. x + y is a computation, not a variable
```

- The extraction operator can be cascaded, as well:
  - ○ int x, y;
  - ○ double a;
  - ○ cin >> x >> y >> a; // read two integers and a double from input

Some special formatting for decimal numbers

- By default, decimal (floating-point) numbers will usually print out only as far as needed, up to a certain preset number of decimal places (before rounding the printed result)
  - ➢ double x = 4.5, y = 12.666666666666, z=5.0;
  - ➢ cout << x << endl; // will likely print 4.5
  - ➢ cout << y << endl; // will likely print 12.6667
  - ➢ cout << z << endl; // will likely print 5

Some special formatting for decimal numbers

- A special "magic formula" for controlling how many decimal places are printed:
  - ➢ cout.setf(ios::fixed); // specifies fixed point notation
  - ➢ cout.setf(ios::showpoint); // so that decimal point will always be shown
  - ➢ cout.precision(2); // sets floating point types to print to // 2 decimal places (or use your desired number)
- Any cout statements following these will output floating-point values in the usual notation, to 2 decimal places.
  - ➢ double x = 4.5, y = 12.666666666666, z=5.0;
  - ➢ cout << x << endl; //   print 4.50
  - ➢ cout << y << endl; //   print 12.67
  - ➢ cout << z << endl; //   print 5.00

Practice

Write a program to create employee variables and display them to console

Employee Information:

Employee ID is 100

Employee Name is Men Dara

Employee Salary is 500.50

Employee Sex is M

Employee Status is 1

Operators

- Operators are symbols that instruct the computer to perform a single, simple task, and most consist of just one or two characters (=, +, *, <=, or == - note that spaces between two-character operators are not allowed), but some operators consist of a complete word (new or return).
- Operands are **expressions** or values on which an operator acts or works.

Expression

- An **expression** is a combination of literals, variables, operators, and explicit function calls (not shown above) that produce a single output value

Example:

➢ 2	// 2 is a literal that evaluates to value 2
➢ "Hello world!"	// "Hello world!"is a literal that evaluates to text "Hello world!"
➢ x	// x is a variable that evaluates to the value of x
➢ 2 + 3	// 2 + 3 uses operator + to evaluate to value 5

Operators

1. Assignment Operator
2. Arithmetic Operators
3. Increment and Decrement Operators
4. Remainder Operator
5. Unary Operators
6. Comparison Operators
7. Logical Operators

Assignment Operator

Assignment operators are used to initializes or updates the value of a variable.

Example:

```
int b = 10;
int a = 5;

a = b;
// a is now equal to 10
```

Arithmetic Operators

1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)

Example:

```
int a = 5;
int b = 2;

cout << a + b;       // equals 7
cout << a - b;        // equals 3
cout << a * b;      // equals 10
cout << a / b; // equals 2
cout << "hello, " + "world";  // equals "hello, world"
```

PEMDAS => Parentheses → Exponent → Multiplication/Division → Addition/Subtraction

Automatic Type Conversions

- Typically, matching types are expected in expressions
- If types don't match, ambiguity must be resolved
- There are some legal automatic conversions between built-in types. Rules can be created for doing automatic type conversions between user-defined types, too
- For atomic data types, can go from "smaller" to "larger" types when loading a value into a storage location.
  - char -> short -> int -> long -> float -> double -> long double
- Should avoid mixing unsigned and signed types, if possible

Automatic Type Conversions

- int i1, i2;
- double d1, d2;
- char c1;
- unsigned int u1;

- d1 = i1; // legal.
- c1 = i1; // illegal. trying to stuff int into char (usually 1 byte)
- i1 = d1; // illegal. Might lose decimal point data.
- i1 = c1; // legal
- u1 = i1; // dangerous (possibly no warning)
- d2=d1+i2;//result of double + int is a double
- d2 = d1 / i2; // floating point division (at least one operand a float type)

Explicit type conversions (casting)

- Older C-style cast operations look like:
  - c1 = (char)i2; // cast a copy of the value of i2 as a char, and assign to c1
  - i1 = (int)d2; // cast a copy of the value of d2 as an int, and assign to i1
- Better to use newer C++ cast operators. For casting between regular variables, use static_cast
  - c1 = static_cast<char>(i2);
  - i1 = static_cast<int>(d2);
- Just for completeness, the newer C++ cast operators are:
  - static_cast
  - dynamic_cast
  - const_cast
  - reinterpret_cast

Increment and Decrement Operators

- ++x; // pre-increment (returns reference to new x)
- x++; // post-increment (returns value of old x) // shortcuts for x = x + 1
- --x; // pre-decrement
- x--; // post-decrement // shortcuts for x = x - 1
  - Pre-increment: incrementing is done before the value of x is used in the rest of the expression
  - Post-increment: incrementing is done after the value of x is used in the rest of the expression
  - Note - this only matters if the variable is actually used in another expression. These two statements by themselves have the same effect: x++; ++x;

Remainder Operator

The remainder (%) operator returns the remainder left over when one operand is divided by a second operand.

Example:

```
int a = 5;
int b = 2;

cout << a % b;     // equals 1
```

Unary Minus Operator

The sign of a numeric value can be toggled using a prefixed (-).

Example:

```
int three = 3;
int minus_three = -three;          // minusThree equals -3
int plus_three = -minus_three;  // plusThree equals 3, or "minus minus three"
```

The unary minus operator (-) is prepended directly before the value it operates on, without any white space.

Unary Plus Operator

The unary plus operator (+) simply returns the value it operates on, without any change:

Example:

```
int minus_six = -6;
int also_minusSix = +minus_six;  // also_minus_six equals -6
```

Although the unary plus operator doesn’t actually do anything, you can use it to provide symmetry in your code for positive numbers when also using the unary minus operator for negative numbers.

Compound Assignment Operators

Compound assignment operators that combine assignment (=) with another operation. One example is the addition assignment operator (+=):

Example:

```
int a = 1;
int a += 2;   // a is now equal to 3
```

The expression a += 2 is shorthand for a = a + 2. Effectively, the addition and the assignment are combined into one operator that performs both tasks at the same time.

Comparison Operators

1. Equal to (==)
2. Not equal to (!=)
3. Greater than (>)
4. Less than (<)
5. Greater than or equal to (>=)
6. Less than or equal to (<=)

Each of the comparison operators returns a Bool value to indicate whether or not the statement is true:

Example:

```
cout << 1 == 1    // true because 1 is equal to 1
cout << 2 != 1    // true because 2 isn't equal to 1
cout << 2 > 1     // true because 2 is greater than 1
cout << 1 < 2     // true because 1 is less than 2
cout << 1 >= 1    // true because 1 is greater than or equal to 1
cout << 2 <= 1    // false because 2 isn't less than or equal to 1
```

Logical Operators

Logical operators is used to modify or combine the Boolean logic values true and false. Python supports the three standard logical operators found in C-based languages:

1. Logical ! (not)
2. Logical && (and)
3. Logical || (or)

Logical Not(!) Operator

The logical NOT operator inverts a Boolean value so that true becomes false, and false becomes true.

Example:

```
    bool a = false;
    bool b = !a;      // equals True
    bool c = !b;      // equals False
```

Logical AND(&&) Operator

The logical AND operator creates logical expressions where both values must be true for the overall expression to also be true.

If either value is false, the overall expression will also be false. In fact, if the first value is false, the second value won’t even be evaluated, because it can’t possibly make the overall expression equate to true. This is known as short-circuit evaluation.

Example:

```
true && true         // equals true
true && false   // equals false
false && true   // equals false
false && false  // equals false
```

Logical OR(||) Operator

The logical OR operator is used to create logical expressions in which only one of the two values has to be true for the overall expression to be true.

Like the Logical AND operator above, the Logical OR operator uses short-circuit evaluation to consider its expressions. If the left side of a Logical OR expression is true, the right side isn't evaluated, because it can't change the outcome of the overall expression.

Example:

```
true || true  // equals true
true || false     // equals true
false || true     // equals true
false || false    // equals false
```

Combining Logical Operators

You can combine multiple logical operators to create longer compound expressions. The logical operators and and or are left-associative, meaning that compound expressions with multiple logical operators evaluate the leftmost subexpression first.

Example:

```
bool entered_door_code = true;
bool passed_retina_scan = false;
bool has_door_key = false;
bool knows_override_password = true;

entered_door_code && passed_retina_scan || has_door_key ||
knows_override_password
// equals true
```

Practice: age program

Create a program that asks the user for their age in years, and prints back the number of:

- months,
- days,
- hours,
- and seconds they've lived for.