# Programming (general)

## Computer program basics

* A computer program consists of instructions executing one at a time. Basic instruction types are:
  + **Input**: A program gets data, perhaps from a file, keyboard, touchscreen, network, etc.
  + **Process**: A program performs computations on that data, such as adding two values like x + y.
  + **Output**: A program puts that data somewhere, such as to a file, screen, network, etc.
* Programs use **variables** to refer to data
  + The name is due to a variable's value varying as a program assigns a variable like x with new values.

## Computational thinking

* Mathematical thinking became increasingly important throughout the industrial age to enable people to successfully live and work.
  + In the information age, many people believe **computational thinking**, or creating a sequence of instructions to solve a problem, will become increasingly important for work and everyday life.
* A sequence of instructions that solves a problem is called an **algorithm**

# Programming basics

## A first program

* A program starts in **main()**, executing the statements within main's braces { }, one at a time.
* Each statement typically appears alone on a line and ends with a **semicolon**, as English sentences end with a period.
* The **int** **wage** statement creates an integer variable named **wage**. The wage = 20 statement assigns wage with 20.
* The **cout** statements output various values.
* The **return 0** statement ends the program (the 0 tells the operating system the program ended without error).
* **Code** is the textual representation of a program

### Example

| #include <iostream>  using namespace std;  int main() {  int wage;  wage = 20;  cout << "Salary is ";  cout << wage \* 40 \* 52;  cout << endl;  return 0;  } |
| --- |

## Basic input

* The following statement gets an input value and puts that value into variable x: **cin >>** **x;** cin is short for characters in.

### Example of cin

| #include <iostream>  using namespace std;    int main() {  int wage;    cin >> wage; //the users input to the variable wage    cout << "Salary is ";  cout << wage \* 40 \* 52;  cout << endl;    return 0;  } |
| --- |

## Basic output: Text

* The **cout** construct supports output; **cout** is short for characters out. Outputting text is achieved via: **cout << "desired text";**. Text in double quotes " " is known as a **string literal**.
  + Multiple cout statements continue printing on the same output line.
* The statement **cout << endl;** starts a new output line, called a **newline**.
  + Note endl is short for "end line". A common error is to type the number "1" or a capital I as in "in", instead of a lower case l as in "end line".

## Outputting a variable's value

* Outputting a variable's value is achieved via: **cout << x;**
* Programmers commonly use a single output statement for each line of output by combining the outputting of text, variable values, and a new line.

### Outputting multiple items with one statement

* + The programmer simply separates the items with **<<** symbols. Such combining can improve program readability because the program's code corresponds more closely to the program's output.

### Example

| #include <iostream>  using namespace std;  int main() {  int wage;  wage = 20;  cout << "Wage is: " << wage << endl;  cout << "Goodbye." << endl;  return 0;  } |
| --- |

* A new output line can also be produced by inserting **\n**, known as a newline character, within a string literal.

# Comments and whitespace

## Comments

* A **comment** is text a programmer adds to code, to be read by humans to better understand the code but ignored by the compiler. Two common kinds of comments exist:
  + A **single-line comment** starts with **//** and includes all the following text on that line. Single-line comments commonly appear after a statement on the same line.
  + A **multi-line comment** starts with **/\*** and ends with **\*/**, where all text between **/\* and \*/** is part of the comment.
    - A multi-line comment is also known as a **block comment**.

## Whitespace

* **Whitespace** refers to blank spaces (space and tab characters) between items within a statement and blank lines between statements (called newlines).
  + A compiler ignores most whitespace.

### Example of good Whitespace

| #include <iostream>  using namespace std;  int main() {  int myFirstVar; // Aligned comments yield less  int yetAnotherVar; // visual clutter  int thirdVar;    // Above blank line separates variable declarations from the rest  cout << "Enter a number: ";  cin >> myFirstVar;    // Above blank line separates user input statements from the rest  yetAnotherVar = myFirstVar; // Aligned = operators  thirdVar = yetAnotherVar + 1;  // Also notice the single-space on left and right of + and =  // (except when aligning the second = with the first =)  cout << "Final value is " << thirdVar << endl; // Single-space on each side of <<    return 0; // The above blank line separates the return from the rest  } |
| --- |

# Errors and Warnings

## Syntax Errors

* **Syntax error** - is to violate a programming language's rules on how symbols can be combined to create a program.

### Example

| main.cpp:6:27: error: expected ';' after expression  cout << "Traffic today"  ^  ; |
| --- |

## Unclear error messages

* Sometimes errors can be wrong, like it might say there is a missing “;” but really there's a missing “<<” just as an example. Look through the code to verify. Look at the code before it says where the error was.
  + Focus on the FIRST error message, ignoring the rest.
  + Look at the reported line of the first error message. If an error is found, fix it. Else, look at the previous few lines.
  + Compile, repeat.

## Logic errors

* A syntax error is known as a type of **compile-time error.**
* A **logic error**, also called a **bug**, is an error that occurs while a program runs.
  + The program would compile but would not run as intended.
    - Writing many lines of code without compiling and running is bad practice.
    - New programmers should compile and run programs after every few lines. Even experienced programmers compile and run frequently.

## Compiler warnings

* A compiler will sometimes report a **warning**, which doesn't stop the compiler from creating an executable program but indicates a possible logic error.

# Computers and programs (general)

## Switches

* A **switch** controls whether or not electricity flows through a wire.
  + In an electronically controlled switch, a positive voltage at the control input allows electricity to flow, while a zero voltage prevents the flow.
  + Engineers soon realized they could use electronically controlled switches to perform simple calculations.
  + The engineers treated a positive voltage as a "1" and a zero voltage as a "0".
    - 0s and 1s are known as **bits (binary digits)**.
  + They built connections of switches, known as **circuits**, to perform calculations such as multiplying two numbers.

## Processors and memory

* To support different calculations, circuits called **processors** were created to process (aka execute) a list of desired calculations, with each calculation called an **instruction**.
* A **memory** is a circuit that can store 0s and 1s in each of a series of thousands of addressed locations, like a series of addressed mailboxes that each can store an envelope (the 0s and 1s).

## Instructions

* Below are some sample types of instructions that a processor might be able to execute, where X, Y, Z, and num are each an integer.

### Example

| **Add X, #num, Y** | Adds data in memory location X to the number num, storing the result in location Y. |
| --- | --- |
| **Sub X, #num, Y** | Subtracts num from data in location X, storing the result in location Y. |
| **Mul X, #num, Y** | Multiplies data in location X by num, storing result in location Y. |
| **Div X, #num, Y** | Divides data in location X by num, storing the result in location Y. |
| **Jmp Z** | Tells the processor that the next instruction to execute is in memory location Z. |

* Memory stores instructions and data as 0s and 1s.
* The material will commonly draw the memory with the corresponding instructions and data to improve readability.
* The programmer-created sequence of instructions is called a **program**, **application**, or just **app**.

## Writing Computer Programs

* Instructions represented as 0s and 1s are known as machine instructions, and a sequence of **machine instructions,** together form an **executable program** (sometimes just called an **executable**).
* **Assemblers** to automatically translate human readable instructions, such as "Mul 97, #9, 98", known as **assembly** language instructions, into machine instructions.
  + The assembler program thus helped programmers write more complex programs.
* To support high-level languages, programmers **created compilers**, which are programs that automatically translate high-level language programs into executable programs.

# Variables and assignments (general)

## Variables and assignments

* In a program, a **variable** is a named item, such as x or numPeople, used to hold a value.
* An **assignment** assigns a variable with a value
  + In programming, = is an assignment of a left-side variable with a right-side value. = is NOT equality as in mathematics.
* Increasing a variable's value by 1, as in x = x + 1, is common, and known as **incrementing** the variable.

# Variables (int)

## Variable declarations

* A **variable declaration** is a statement that declares a new variable, specifying the variable's name and type.
  + int userAge;

## Assignment statement

* An **assignment statement** assigns the variable on the left-side of the = with the current value of the right-side expression.
* An **expression** may be a number like 80, a variable name like numApples, or a simple calculation like numApples + 1. Simple calculations can involve standard math operators like +, -, and \*, and parentheses as in 2 \* (numApples - 1).
  + An integer like 80 appearing in an expression is known as an **integer literal**.
  + Although not required, an integer variable is often assigned an initial value when declared.

# Identifiers

## Rules for identifiers

* A name created by a programmer for an item like a variable or function is called an **identifier**. An identifier must:
  + be a sequence of letters (a-z, A-Z), underscores (\_), and digits (0-9)
  + start with a letter or underscore
* Identifiers are **case sensitive**, meaning upper and lower case letters differ.
* A **reserved word** is a word that is part of the language, like int, short, or double. A reserved word is also known as a **keyword**.
  + A programmer cannot use a reserved word as an identifier. Many language editors will automatically color a program's reserved words.

| alignas (since C++11)  alignof (since C++11)  and  and\_eq  asm  auto  bitand  bitor  bool  break  case  catch  char  char16\_t (since C++11)  char32\_t (since C++11)  class  compl  const  constexpr (since C++11)  const\_cast  continue | decltype (since C++11)  default  delete  do  double  dynamic\_cast  else  enum  explicit  export  extern  false  float  for  friend  goto  if  inline  int  long  mutable | namespace  new  noexcept (since C++11)  not  not\_eq  nullptr (since C++11)  operator  or  or\_eq  private  protected  public  register  reinterpret\_cast  return  short  signed  sizeof  static  static\_assert (since C++11)  static\_cast | struct  switch  template  this  thread\_local (since C++11)  throw  true  try  typedef  typeid  typename  union  unsigned  using  virtual  void  volatile  wchar\_t  while  xor  xor\_eq |
| --- | --- | --- | --- |

## Style guidelines for identifiers

* While various (crazy-looking) identifiers may be valid, programmers may follow identifier naming conventions (style) defined by their company, team, teacher, etc. Two common conventions for naming variables are:
  + Camel case: **Lower camel case** abuts multiple words, capitalizing each word except the first, as in numApples or peopleOnBus.
  + Underscore separated: Words are lowercase and separated by an underscore, as in num\_apples or people\_on\_bus.

# Arithmetic Expressions (general)

## Basics

* An **expression** is any individual item or combination of items, like variables, literals, operators, and parentheses, that evaluates to a value
* A **literal** is a specific value in code
* An **operator** is a symbol that performs a built-in calculation

| Arithmetic operator | Description |
| --- | --- |
| + | The **addition** operator is **+**, as in x + y. |
| - | The **subtraction** operator is **-**, as in x - y. Also, the - operator is for **negation**, as in -x + y, or x + -y. |
| \* | The **multiplication** operator is **\***, as in x \* y. |
| / | The **division** operator is **/**, as in x / y. |

## Evaluation of expressions

* An expression **evaluates** to a value, which replaces the expression.
* An expression is evaluated using the order of standard mathematics, such order known in programming as **precedence rules**

| Operator/Convention | Description | Explanation |
| --- | --- | --- |
| **( )** | Items within parentheses are evaluated first | In 2 \* (x + 1), the x + 1 is evaluated first, with the result then multiplied by 2. |
| **unary -** | - used for negation (unary minus) is next | In 2 \* -x, the -x is computed first, with the result then multiplied by 2. |
| **\* / %** | Next to be evaluated are \*, /, and %, having equal precedence. | (% is discussed elsewhere) |
| **+ -** | Finally come + and - with equal precedence. | In y = 3 + 2 \* x, the 2 \* x is evaluated first, with the result then added to 3, because \* has higher precedence than +. Spacing doesn't matter: y = 3+2 \* x would still evaluate 2 \* x first. |
| **left-to-right** | If more than one operator of equal precedence could be evaluated, evaluation occurs left to right. | In y = x \* 2 / 3, the x \* 2 is first evaluated, with the result then divided by 3. |

# Arithmetic expressions (int)

## Compound operators

* Special operators called **compound operators** provide a shorthand way to update a variable, such as userAge **+=** 1 being shorthand for userAge = userAge + 1. Other compound operators include **-=**, **\*=**, **/=**, and **%=**

# Floating-point numbers (double)

## FLoating-point (double) variables

* A **floating-point number** is a real number containing a decimal point that can appear anywhere (or "float") in the number.
* A **double** variable stores a floating-point number. Ex: double milesTravel; declares a double variable.
* A **floating-point literal** is a number with a fractional part, even if the fraction is 0, as in 1.0, 0.0, or 99.573
  + Good practice is to always have a digit before the decimal point, as in 0.5, since .5 might mistakenly be viewed as 5.

## Choosing a variable type (double vs. int)

* A programmer should choose a variable's type based on the type of value held.
  + Integer variables are typically used for values that are counted, like 42 cars, 10 pizzas, or -95 days.
  + Floating-point variables are typically used for measurements, like 98.6 degrees, 0.00001 meters, or -55.667 degrees.
  + Floating-point variables are also used when dealing with fractions of countable items, such as the average number of cars per household.

## Floating-point division by zero

* If the dividend and divisor in floating-point division are both 0, the division results in a "not a number". **Not a number** (**NaN**) indicates an unrepresentable or undefined value. Printing a floating-point variable that is not a number outputs nan.

# Constant variables

* An initialized variable whose value cannot change is called a **constant variable**

### Example

| #include <iostream>  using namespace std;  /\*  \* Estimates distance of lightning based on seconds  \* between lightning and thunder  \*/  int main() {  const double SPEED\_OF\_SOUND = 761.207; // Miles/hour (sea level)  const double SECONDS\_PER\_HOUR = 3600.0; // Secs/hour  double secondsBetween;  double timeInHours;  double distInMiles;    cout << "Enter seconds between lightning and thunder: ";  cin >> secondsBetween;    timeInHours = secondsBetween / SECONDS\_PER\_HOUR;  distInMiles = SPEED\_OF\_SOUND \* timeInHours;    cout << "Lightning strike was approximately" << endl;  cout << distInMiles << " miles away." << endl;    return 0;  } |
| --- |

# Using math functions

## Basics

* A standard **math library** has about 20 math operations, known as functions. A programmer can include the library and then use those math functions.
  + #include <cmath>
* A **function** is a list of statements executed by invoking the function's name, such invoking is known as a **function call**. Any function input values, or **arguments**, appear within ( ), separated by commas if more than one.

| Function | Behavior | Example |
| --- | --- | --- |
| sqrt(x) | Square root of x | sqrt(9.0) evaluates to 3.0. |
| pow(x, y) | Power: xy | pow(6.0, 2.0) evaluates to 36.0. |
| fabs(x) | Absolute value of x | fabs(-99.5) evaluates to 99.5. |

* The "c" in cmath indicates that the library comes from a C language library.
* Some math functions for integers are in a library named cstdlib, requiring: #include <cstdlib>. Ex: abs() computes the absolute value of an integer.