**Language: C++**

**Philosophy:**

* C++ designed to allow programmer high degree of freedom to do what they want.
* Won’t stop you from doing things that don’t make sense.

**C++ GOOD AT:**

* Excels where high performance and precise control over memory and other resources needed.
* EX: video games, real time systems, embedded software

**INTRODUCTION TO COMPILER, LINKER, AND LIBRARIES**

* C++ compiler used to compile C++ program.
  + Compiler sequentially goes through each source code in prog and
    - Checks your code to make sure follows rules of C++ language, give error if does not.
    - Translates C++ source code to machine language file called object file.
      * Typically named name.o or name.obj where name is same name as the .cpp file
* After compiler creates object files, the linker kicks in.
  + First, takes all object files generated and combines them into single executable program.
  + It will also link library files.
    - A library file is a collection of precompiled code that has beek “packaged” up for reuse.
    - C++ has extensive library called C++ standard library containing functionality for printing on monitor, keyboard I/O , etc.
  + Linked makes sure all cross-file dependencies resolved properly.
    - Ex: If define something in one .cpp file, and use it in another, the linker connects the two together.
    - Will get link error if linker unable to reference something with its

Definition

**C++ Basics**

* *STATEMENTS* 
  + - a cpu prog is sequence of instructions that tell cpu what to do. Statements are a type of instruction that causes the prog to perform an action.
  + Examples include: Declaration, Jump, Expression, Compound, Selection, Iteration, and Try statements.
* *COMMENTS*
  + CAN USE // symbol to indicate a single-line comment
  + CAN USE /\* insert comments here \*/ to denote block comment
* *OBJECTS AND VARIABLES*
  + Object
    - Region of storage (usually memory) that has a value and other associated properties. When object defined, compiler auto determines where object will be placed in memory.
    - A named object is called a VARIABLE.
    - Name of the object called an IDENTIFIER
  + VARIABLE INSTANTIATION
    - To create variable, use special kind of declaration statement called a DEFINITION.
      * Ex: int x; //define a variable named x, of type int
    - Instantiation
      * When program runs, variable will be instantiated.
      * Fancy word meaning object will be created and assigned memory addr.
    - Variable Assignment and Initialization
      * After variable defined, can give it a balue using = operator.
        + This called a copy assignment in which it copies the value on the right-hand side of the = to the left and side of the operator.
      * To initialize a variable means to define and assign value at same time.
        + CAN INITIALIZE **3 DIF WAYS**:

Int width=5; //copy initialization

Int width(5); //direct initialization

Direct initialization same as copy for simply data tyipes but for more advanced types direct might be better.

Int width {5} ;

Called brace initialization.

If initialize with empty brace, then that’s zero initialization.

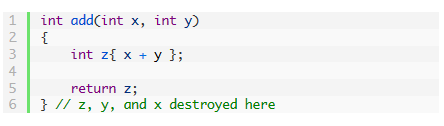
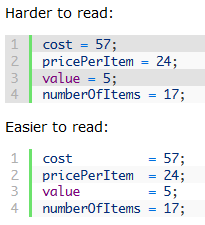
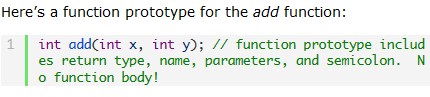
* *INTRO TO IOSTREAM: COUT, CIN, ENDL* 
  + Iostream lib part of C++ standard library dealing with basic input and output.
  + To use functionality defined within iostream, need to include iostream header at top of any code file that uses it.
    - **#include <iostream>**
  + Std::cout
    - Predefined var allows us to send data to console to be printed as text.
      * Ex: std::cout<<4;
      * Note the insertion operator <<
      * Can also print multiple things combining << operators
        + Ex: std::cout<<”hello”<<”world!”;
  + Std::endl
    - If want to print separate lines of ouput to console, need to tell console to move cursor to next line using std::endl
      * Ex: std::cout<<”hi”<<std::endl;
    - Std::endl vs \n
      * Using endl can be inefficient as it moves cursor to next line and flushes output.the \n character only moves cursor to next line.
      * TYPICALLY USE \n OVER endl
  + Std::cin
    - Reads input from kb using >> operator
      * Std::cout<< “Enter a number \n”;
      * Int x{ };
      * Std::cin>>x;
* *UNINITIALIZED VARS* 
  + **C++** doesn’t initialize most vars to a given value (such as zero) automatically.
    - When var assigned a mem location by compiler, default val of that var is whatever happens to already be there.
    - Like this for performance
      * Ex: if had to read in 100,000 values from file and had to initialize all of em first, slowww.
    - Using values of uninitialized vars can lead to crashes/bugs.
* *KEYWORDS AND NAMING IDENTIFIERS* 
  + KeyWords (reserved words) are words with special meaning for c++.
    - Ex: bool, int, class, const, enum, if, else…
  + IDENTIFIER NAMING RULES:
    - Name of a var.
      * Can’t have a keyword.
      * Only composed of letters, numbers, and underscores.
      * Must begin with a letter or underscore.
      * Case sensitive
    - BEST PRACTICES:
      * Var names should begin with lowercase letter
        + If one word then whole thing lower
      * Function names also start with lower
      * Names that start with capital letter typically used for user-defined types (like structs, classes, and enumerations)
      * If multi-word, can camelCase or use underscores
* *LITERALS* 
  + Literal
    - Fixed value that has been inserted directly into source code.
    - Literals have a fixed value and can’t be changed like a variable.

**C++ BASICS: FUNCTIONS AND FILES**

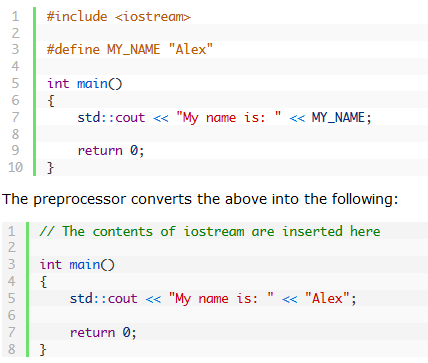
* *Functions -*Reusable sequence of statements designed to do a particular job.
  + Putting all code inside main make hard to manage.
  + Functions provide way to split prog into small chunks, easier to organize, test, and use.
    - C++ STL has lots of functions for you to use, however can define em as well.
  + When function call hit in execution, tells CPU to interrupt current function and execute another. Calls function named then CPU returns back to point. EXAMPLE:
    - Return-type identifier( params) //identifier replaced with name of func
    - {

//enter code here

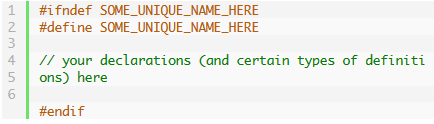
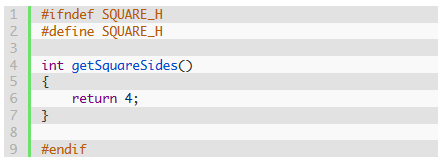
}

* + Functions can call other functions, **but NESTED functions in C++ not supported.**
  + **MAIN should return 0, and if you change it to void, compiler will implicitly return 0.**
  + When making functions don’t repeat yourself. If need to do something more than once, consider to remove redundancy.
* *Local Scope:* 
  + Function params, and vars defined in function called local variables.
    - LifeTime- time between creation and destruction of a var.
  + An identifiers SCOPE refers to where identifier can be accessed within code.
    - Names used in functions or vars declared in function body ONLY VISIBLE WITHIN function that declares them. So local vars within functions can be named without regard for names of vars in other functions.
* *WHY USE FUNCTIONS?* 
  + Organization – as programs grow, having all code in main too messy. Helps separate without having to think about rest of program.
  + Reusabiity – Once function written, can be called multiple times.
  + Testing – Functions reduce code redundency, so less code to test.
    - Self-Contained so can easily test function to see if works.
  + Extensibility- When need to extend program to handle unique cases, can just change function and it will carry throughout whole program.
  + Abstraction- In order to use function only need name, input, outputs, and where it lives. Don’t need to know how it works.
    - REFACTORING – when function become too long, too complicated, hard to understand can split into multiple subfunctions.
* *BEST PRACTICES FOR FUNCTIONS:* 
  + Lines no longer than 80 chars in length.
  + Use whitespace to make code easier to read
* *FORWARD DECLARATIONS AND DEFINITIONS* 
  + CANT call function if defined below main MUST BE ABOVE.
  + Can avoid this error by forward declarations or reordering.
  + FORWARD DECLARATION
    - Allows to tell compilre about existence of identifier before actually defining
    - TO WRITE A FORWARD DECLARATION :
      * We use a declaration statement calld a FUNCTION PROTOTYPE
      * Can define above main, and then add details  
        below it
        + DEFINITIONS ACTUALLY IMPLEMENT FUNCTION WhiLE   
          DECLARATIONS DO NOT.
* *NAMING COLLISIONS AND INTRO TO NAMESPACES* 
  + Most common naming collisions occur when
    - Two or more defs for a fn are introduced into sep files that are compiled into same prog. Results in linker error.
    - Two or more defs for a fn introduced into same file
  + A namespace is a region that allows you to declare names inside of it for purpose of disambiguation.
    - Provides scope to names declared insode of it.
    - Name declared in here won’t be mistaken for an identical name declared in another scope.
      * Global namespace- any name not defined in class, fn, or namespace.
      * STD namespace- std::cout 🡸 example.
        + Std is name of namespace that identifier cout is a part of.
        + **When use identifier defined inside a namespace, you have to tell compiler that the identifier lives inside the namespace.**
      * Explicit namespace- use explicit namespace prefixes to access identifiers defined in a namespace .
        + AVOID using namespace std;

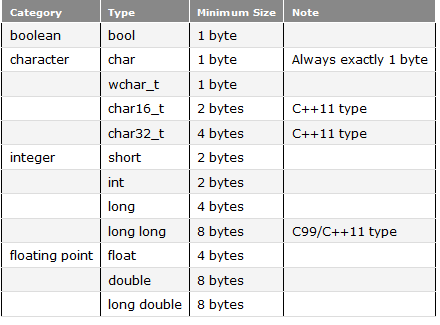
each The using directive tells compiler to check specified namespace when trying to resolve identifier that has no namespace prefix.

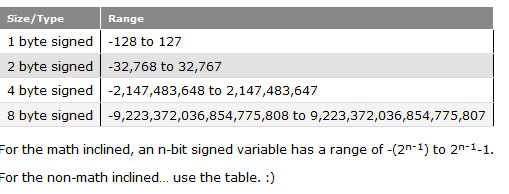
* *PreProcessor*
  + Essentially a separate program that manipulates text in each code file. When ran, looks for PREPROCESSOR DIRECTIVES
    - -instructions that start with # symbol and end with a newline (no semicolon)
      * The directives tell preprocessor to perform specific text manip.
        + EX: when #include, preprocessor replaces #include with contencts if the file.
        + #define Is directive can be used to create macro.

Macro – rule that defines how inpt text converted into replacement output.

* + Directives resolved before compilation from top to bottom.
* *HEADERFILES*
  + As programs grow larger, like to split into files.
  + Header files (.h extension) very popular.
    - Used to propagate declarations to code files.
    - Can save a lot of typing in multiple-file progs.
    - HeaderFiles should not contain function and var definitions generally.
      * Helps not to violate one def rule.
    - Consits of two things: **header guards**. And actual content of header file
      * USE A .H EXTENSION FOFR HEADER FILES
    - Code file should #include paired header file if exists.
    - USE ANGLED bRACKETS INSTEAD OF QUOTES FOR H FILES
* *HEADER GUARDS* 
  + Used to avoid name collisions (like protyping same function twice…)
    - Example:
* Preprocessor checks whether some unique name here has been defined, if not than contents included
* All header files should have header guards.
  + Ex:
* Have simpler alternate form of heade guards using #pragma directive.
* Serves same purpose, and has added benefit of being shorter
  + NOT official part of C++ lang doe, so not all compilers support it.

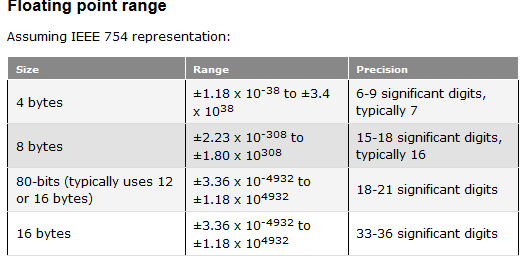
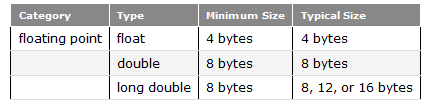
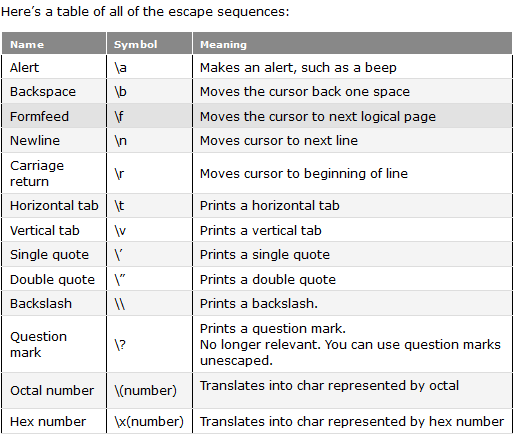
FUNDAMENTAL DATA TYPES

* Smallest unit of mem is binary digit.
* Memory organized into sequential units called memory addresses.
* All data is just sequence of bits, we use data type to tell compiler how to interpret contents of memory.
* **Void**
  + Easiest data type, NO TYPE. Used for functions not returning value. Can also put in parameter to indicate no params, but better to leave blank



* **Signed Integers**
  + An int is integral type represent pos and neg whole nums.
  + By default integers are signed

Overflow:

* + - Occurs when try to store val outside range of type. Results in data lost because obj not enough mem to store. Results in undefined behavior.
* **Unsigned Integers**
  + Can define using unsigned keyword (i.e. unsigned short us)…
  + 0-2^n –1 range.
    - What happens if store larger num in unsigned int cant hold?
      * Ex: put 280 in a 1 byte unsigned int.
        + Range of 1 byte unsined is 0-255. Cpu adds 1 so 256, and then the division of 280/256 remainder is kept, so 280/256= 1 remainder 24. So 24 kept.
    - Typically aavoid unsigned ints as if we subtract 2 of them and it is negative, what to do? What if mix signed unsigned ints?
* Floating Point Numbers
  + Can hold real numbers like 432.0 or -3.33
    - FLOAT
    - DOUBLE
    - LONG
  + Make sure type of literals match type of vars being assigned to.
  + Precision of floating point nums define how many sig digits can represent without info loss.
  + Can override precision default y using std::setprecision(16) //show 16 dig precision.
  + Favor double over float unless space at premium, as lack of precision in float often lead to inaccuracies.
* BOOLEAN VALUES
  + Vars that can only have two values true or false
  + Bool b; 🡸 example
  + If print out, will print 0 for false and 1 for true;
  + When asking for input cin will only accept 0 or 1 for true and false;
* IF STATEMENTS
  + Condition- expression that evaluates to Boolean val.
    - Ex: if(x==0) { do something }
    - Has if -else statements and if, else if, else
    - Can have non Boolean conditionals like if(4) for ex
      * If non zero value, convert to true otherwise false
* CHARS
  + Integral type so has underlying val stored as integer (1 byte) size
  + ASCII defines particular way to rep eng chars as number between 0 and 127
  + Can initialize as char ch2 {‘a’}
    - Or char ch1 {97}
  + ESCAPE SEQUENCES
    - Some chars in C++ have special meaning. Chars called escape sequences (start with a \ )
    - Like \n , or \t for tab
      * + Double quotes considered STRING

Collection of seq chars

* CONSTANTS
  + Useful to define vars with vals that cant be changed
    - Ex: const double gravity;
    - CANT be changed via assignment.
    - RUNTIME VS COMPILE
      * Runtime- initialization vals only resolved at runtime. Compiler can’t determine initial vals at compile time. Relies on user input for example.
      * Compile- initialization vals resolved at compile-time. So compiler can perform optimizations.
    - CONVENTIONS
      * Use upper case names for const vars
* OPERATOR PRECEDENCE AND ASSOCIATIVITY
  + Use parenthesis to make it clear how expression should evaluate even if not technically necessary.
  + To use exponents do:
    - Have to #include <cmath>
    - Std:: pow(3.0,4.0)}; //3 to the 4th power
  + INCREMENT DECREMENT
    - Prefix increment/ decrement
      * First operand incremented, THEN expression eval
    - Typically try to use prefix version of increment
  + CONDITIONAL OPERATOR (?)
    - Ternary operator (takes 3 operands)
    - Provide shorthand method for doing particular if/else statements
      * If take form
        + If(condition)

Statement 1

* + - * + Else

Statement2

* + - * Can write as (condition) ? expression 1: expression 2;
        + If condition true, expression 1 executed, else 2
        + EX: if(x>y)

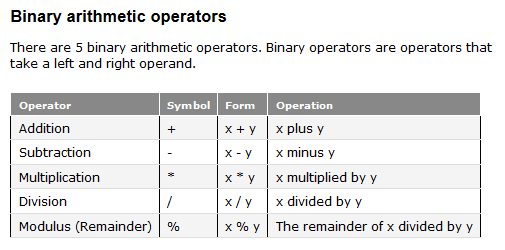
Larger=x;

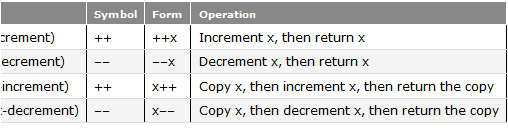
Else

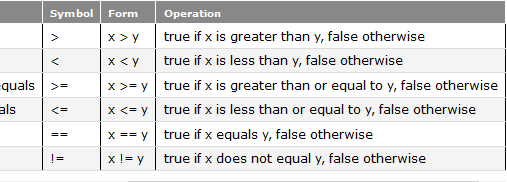
Larger=y;

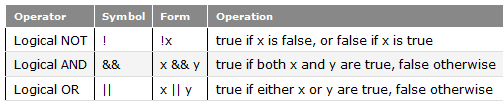
CAN REWRITE AS larger=(x>y) ? x : y ;

* ALWAYS parenthesize conditional part of operator





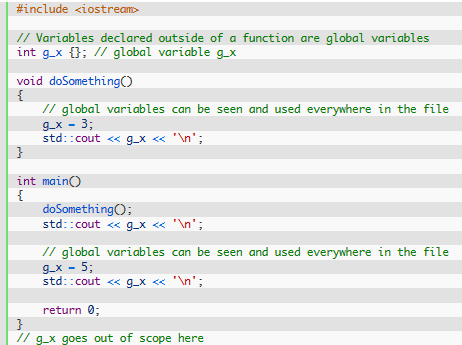
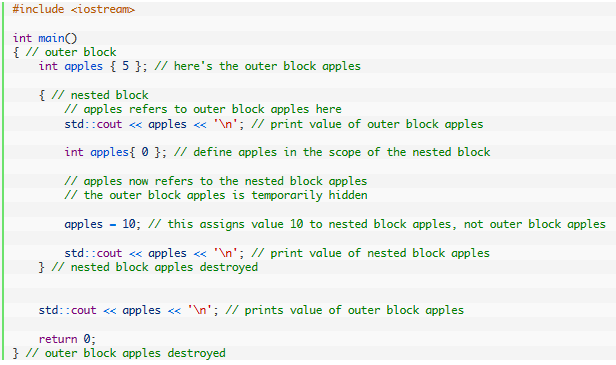




**User Defined NameSpaces**

* + - can define own namespace via namespace keword
    - ex: namespace foo{
      * + int something(int x, int y)
        + {

Return x+y;

* + - * + }
    - Int main()
    - {
      * Cout<<foo::something(a,b)<<endl;
* **LOCAL VARIABLES** 
  + Local vars have block scope so in scope from point of definition to end of block defined within.
  + Variables should be defined in most limited scopr.
    - If var only used within nested block, should be defined in that nested block.
* GLOBAL VARIABLES
  + Convention these vars declared at top of file below the includes but above any code.
    - Consider using a g or g\_ prefix to indicate global vars.
      * HAVE FILE SCOPE
    - Can also have const global vars.
    - GENERALLY SHOULD AVOID NON CONSTANT GLOBALS HOWEVER
* VARIABLE SHADOWING
  + What if var inside nested block has same name as var in outer? Called name hiding.

**WILL PRINT OUT**

**5**

**10**

**5**

* can also shadow global vars.
* GENERALLY ShOULD AVOID SHADOWIN, HOWEVER.

**STATIC VARIABLES:**

* Using static keyword on local var changes duration from automatic to static.
  + Means that var now created at start of prog, and destroyed at end like a global var. As a result, static var retain value even after it goes out of scope.

USING STATEMENTS

* Way to simplify things to utilize **using declaration**.
* Ex: using namespace std; using std::cout; …
  + Tells compiler we going to use object cout or std functions.
* Best to not use them in GLOBAL scope, but local scope is okay.
* Ex:
  + Int main()
  + {
    - Using namespace std;
    - ..
  + }

AUTO KEYWORD

* When initializing var, auto keyword can be used in place of type to tell compiler to infer var type from initialized type.
  + Called TYPE INFERENCE
  + Ex: auto d{5.0} 🡸 5.0 is double literal, so d will be type double
  + Also works with return types
    - Auto sum { add (5,6)} //sum will be int
  + In C++14 can now use auto as return type so
    - Auto add (int x, int y)
  + BEST TO AVOID using inference for function return types

IMPLICIT CONVERSION