Lecture Notes 1

Compiled vs. Interpreted Programming Languages

- Compiled
  - Code → Compiler → Assembler* → Linker → Execution
  - Compiler – Converts the code into either assembly or object code
    - Object Code – Package of "annotated" machine code
  - Assembler – Converts assembly into object code
    - * – Assembly step may be skipped in modern compiler (clang, gcc, etc.)
  - Linker – Combines object code files into larger files, libraries, or full executables
    - Symbol Resolution – Connects symbol (e.g. function name) to an address
  - C and C++ Traditional Pipeline
    - Code → Preprocessor → Compiler → Assembler → Linker → Execution
    - Preprocessor – Macro based text replacement system
- Interpreted
  - Code → Execution (on interpreter)
  - Interpreter – Program that understands programming language code and executes the commands
- Hybrid
  - Mixed Compiled & Interpreted
    - Compilation – Interpreted language is converted into byte code and executed on Virtual Machine
  - Just-In-Time (JIT) Compilation – Byte code is translated into machine code dynamically

Python to C++ Brief Overview

- Python Types to C++ (close equivalent)

Python	C++ (Equivalent)	Notes
`bool`	**`bool`**	
`True`	`true`	
`False`	`false`	
`int`	**`int`**	Or **`short`**, **`long`**, etc.
`42`	`42`	
`0x03`	`0x03`	
`0b10`	`0b10`	
`034`	`034`	
`float`	**`float`**	
`1.2`	`1.2`	
`2.3e4`	`2.3e4`	
`str`	**`std::string`**	Must include string header

`"Hello"`	`"Hello"`	
`'Hello'`	`"Hello"`	
`str[i]` `(or chr())`	**`char`**	Single character of string.
`"X"`	`'X'`	
`'X'`	`'X'`	
`list`	**`std::vector<type>`**	Or `std::list<>` must include header, must be container of specific type.
`tuple`	**`std::tuple<...>`**	Must include header and specify all types of the tuple.
`set`	**`std::set<type>`**	Must include set header and specify the type of the set.
`dict`	**`std::map<t1,t2>`**	Or `std::unordered_map`, must include header, and specify the mapping types.

- Comparisons, assignment and arithmetic

Python	C++ (Equivalent)	Notes
`==,!=, <, >, <=, >=`	**`==,!=, <, >, <=, >=`**	All same.
`and`	**`&&, and`**	C++ allows for `and` or `&&` for logical and
`or`	**`||, or`**	C++ allows for `or`, or `||` for logical or
`not`	**`!, not`**	C++ allows for `not`, or `!` for logical not
`&`	**`&, bitand`**	C++ allows for `bitand`, or `&` for bitwise and
`|`	**`|, bitor`**	C++ allows for `bitor`, or `|` for bitwise or
`~`	**`~, compl`**	C++ allows for `compl`, or `~` for bitwise complement
`^`	**`^, xor`**	C++ allows for `xor`, or `^` for bitwise exclusive or
`=` **`+=`** **`-=`** **`*=`**	**`=`** **`+=,++`** **`-=, --`** **`*=`**	C++ has a special case for adding/decrementing 1.
`a *= 5`	`a *= 5;`	
`a += 1`	`a++;`	
`&=` **`|=`** **`^=`**	**`&=, and_eq`** **`|=, or_eq`** **`^=, xor_eq`**	C++ has two options.
`+, -, *`	**`+, -, *`**	All same.

`a = b + 4`	`a = b + 4;`	Don't forget semicolon!
`/, //` **`/=, //=`**	**`/`** **`/=`**	Division will be done on the type, so integer division will only be done if both operands are integers.
`a = b / 3.0`	`a = b / 3.0;`	b can be an integer of float.
`a = b // 3`	`a = b / 3;`	Assumes b is an integer.
`%` **`%=`**	**`%, fmod()`** **`%=, = fmod()`**	Modulus can only be done on integral types in C++, use fmod for floating point modulus. Must include math header for fmod.
`a = b % 3`	`a = b % 3;`	Assumes b is an integer.
`a = b % 3.0`	`a = fmod(b,3.0)`	b can be integer of float
`**`	**`pow()`**	Must include math header for pow.
`a = b ** c`	`a = pow(b,c)`	

- Control Flow

Python	C++ (Equivalent)	Notes
`if cond:` **`    block`**	**`if(cond){`** **`    block`** **`}`**	Don't forget the curly braces {}!

`if a == b:`
`    c = 5`
	`if(a == b){`
	`    c = 5;`
	`}`	

`if cond:` **`    block1`** **`else:`** **`    block2`**	**`if(cond){`** **`    block1`** **`}`** **`else{`** **`    block2`** **`}`**	Don't forget the curly braces {}!
`if cond1:` **`    block1`** **`elif cond2:`** **`    block2`** **`else:`** **`    block3`**	**`if(cond1){`** **`    block1`** **`}`** **`else if(cond2){`** **`    block2`** **`}`** **`else{`** **`    block3`** **`}`**	Don't forget the curly braces {}!
`while cond:` **`    block`**	**`while(cond){`** **`    block`** **`}`**	Don't forget the curly braces {}!

`while a < 5:`
`    a = a + 1`
	`while(a < 5){`
	`    a++;`
	`}`	

for *x* in *cont*: ** *block***	**for(auto x:*cont*){** ** *block*** **}**	C++11 added range based for loops.

for x in y:
    x = x + 1
	for(auto &a:y){
	    a++;
	}	

for *x* in range(y,z): ** *block***	**for(int x=y;x<z;x++)** **{** ** *block*** **}**	C++11 added range based for loops.

for x in range(0,3):
    y[x] = 3
	for(int x=0;x<3;x++)
	{
	    y[x]=3;
	}	

- Functions

Python	C++ (Equivalent)	Notes
def foo(*parms*): ** *block*** ** return *val***	**rtype foo(ptype p){** ** block** ** return val;** **}**	C++ functions need to specify the return type and the type of each parameter.

def foo(x):
    return x + 4
	int foo(int x){
	    return x + 4;
	}	

- Misc (Entry Point, and Imports)

Python	C++ (Equivalent)	Notes
if __name__=='__main': ** *block***	**int main(int argc,** **char *argv[]){** ** *block*** **}**	The entry point for C++ is main.

- Parameter Types

Python	C++ (Equivalent)	Notes

`def foo(x):`
`  x = 4`
`  return x + 3`
 	
`y = 5`
`z = foo(y)`
`# y == 5, z == 7`
	`int foo(int x){`
	`  x = 4;`
	`  return x + 3;`
	`}`
	`...`
	`y = 5;`
	`z = foo(y);`
	`// y == 5, z == 7`
		When basic numeric types are passed, they act like passing a copy of the value in C++.

`def foo(x):`
`  x.append(1)`
 	
`y = [2, 3]`
`foo(y)`
`# y == [2, 3, 1]`
	`void foo(std::vector< int`
	`  > &x){`
	`  x.push_back(1);`
	`}`
	`...`
	`std::vector<int> y =`
	`  {2, 3};`
	`foo(y);`
	`// y == {2, 3, 1}`
		When a list, or dict, is passed to a function they act like a C++ reference, so modification actually changes the variable argument.