ADVANCED COMPUTING

C++ Review – Part I

Sebastien Donadio
sdonadio@uchicago.edu

FINM-32700: Computing for Finance in Python

Syllabus

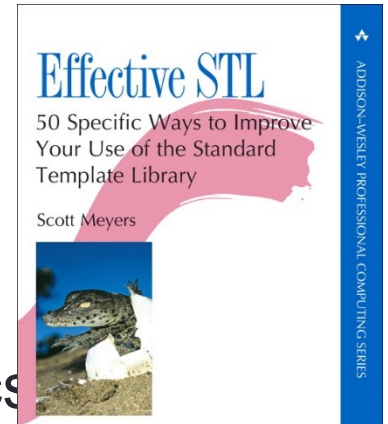
- Weekly Programming assignments: 20%
- Course Project: 20%
- Midterm: 25%
- Final Exam: 30%
- Participation: 5%

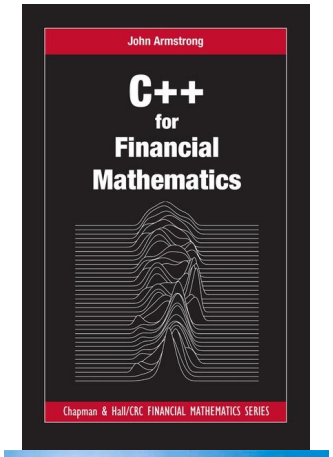
Why are you learning C++?

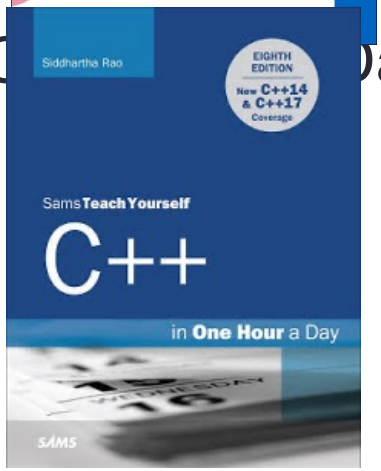
- Because it is fun?
- Because you love C therefore you will love C++?
- Is it because it is difficult to learn?
- Since it is difficult to learn, you can get a good-paying job?
- If you know C++, you also know Java?

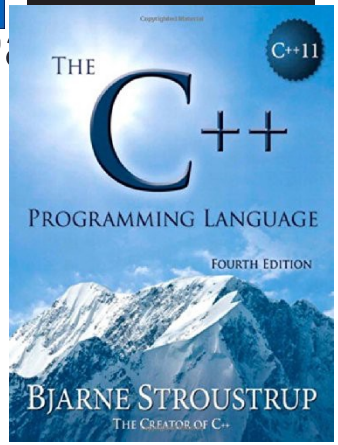
Books

- Reading the following books is
  - Effective STL, Scott Meyers
  - Quantlib bibliography
  - C++ for Financial Mathematics
  - Sams Teach Yourself C++ in C
    Edition)

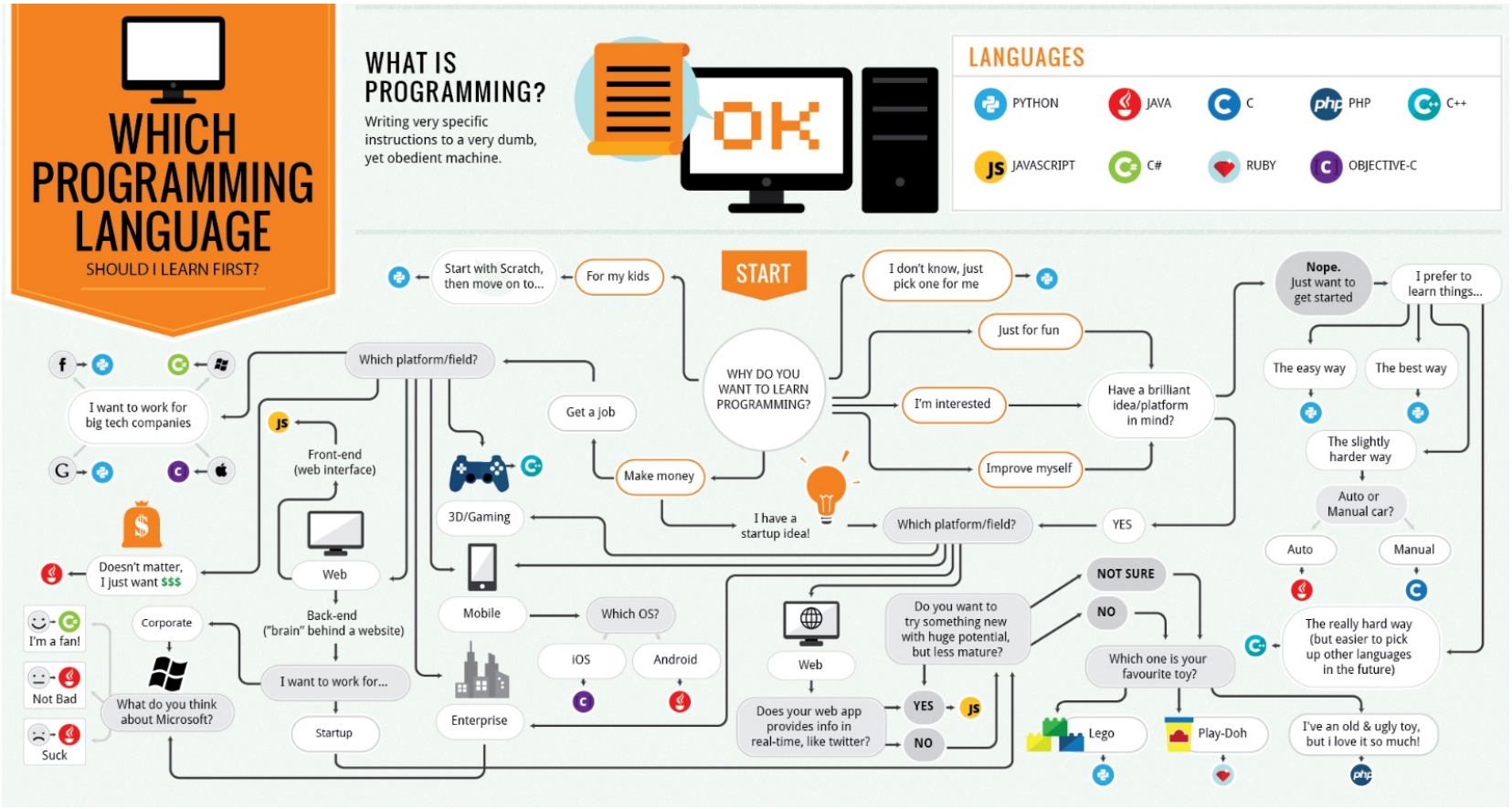




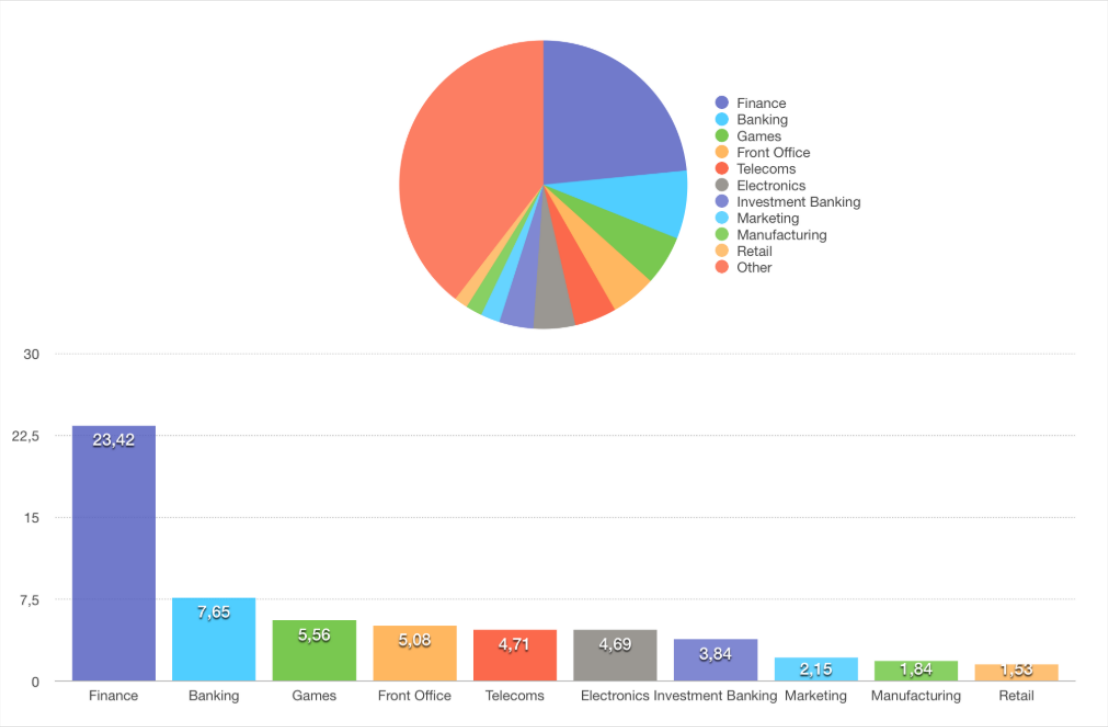




Programming Language

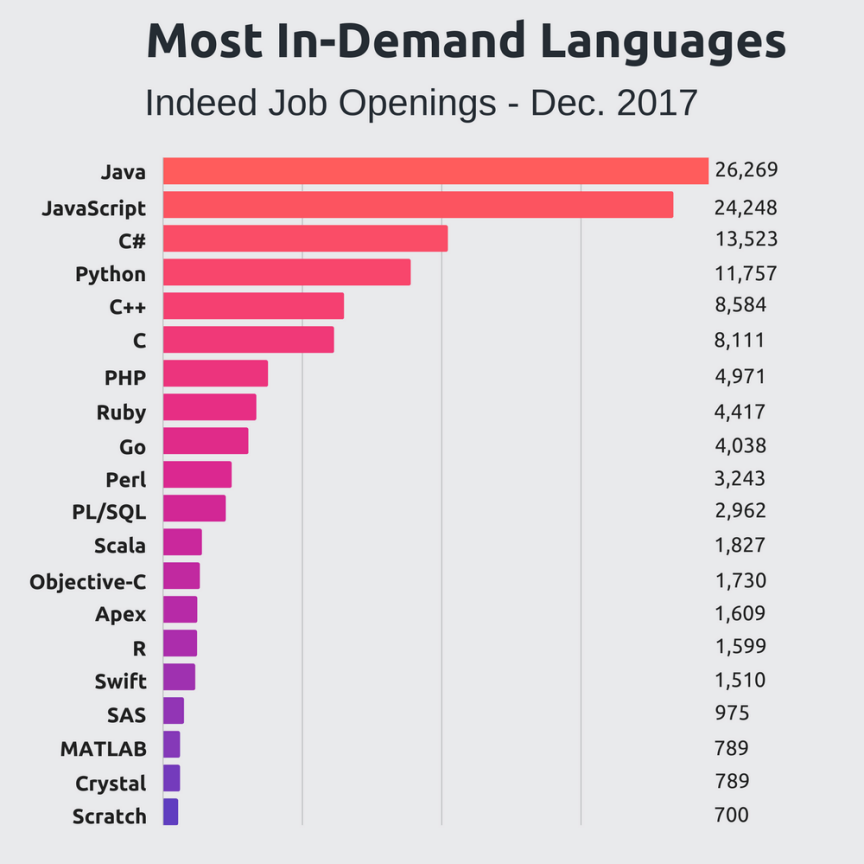


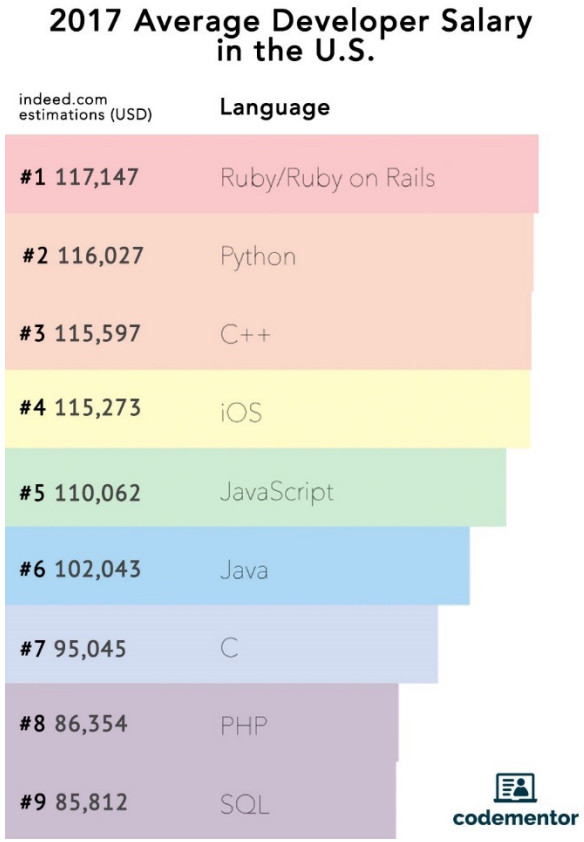
C++ in Finance



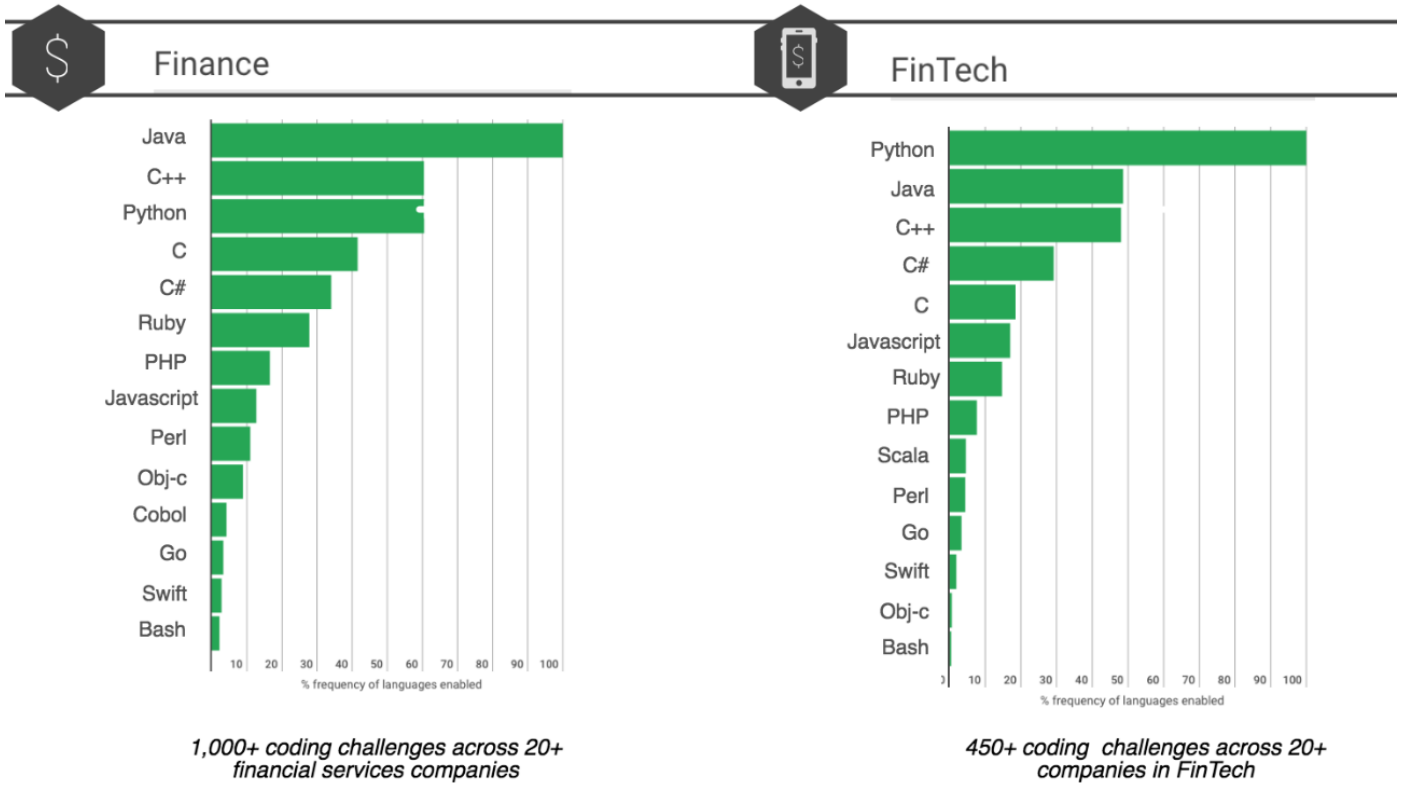
The top compiler is GCC (65% share), trailed by Clang with 20%. Taking only Windows platforms, 36% use Visual C++ and 34% use GCC.

Job market


Most In-Demand Languages
Indeed Job Openings - Dec. 2017
Java 26,269
JavaScript 24,248
C# 13,523
Python 11,757
C++ 8,584
C 8,111
PHP 4,971
Ruby 4,417
Go 4,038
Perl 3,243
PL/SQL 2,962
Scala 1,827
Objective-C 1,730
Apex 1,609
R 1,599
Swift 1,510
SAS 975
MATLAB 789
Crystal 789
Scratch 700


2017 Average Developer Salary in the U.S.
indeed.com estimations (USD)
Language
#1 117,147 Ruby/Ruby on Rails
#2 116,027 Python
#3 115,597 C++
#4 115,273 iOS
#5 110,062 JavaScript
#6 102,043 Java
#7 95,045 C
#8 86,354 PHP
#9 85,812 SQL
codementor

C++ in Finance

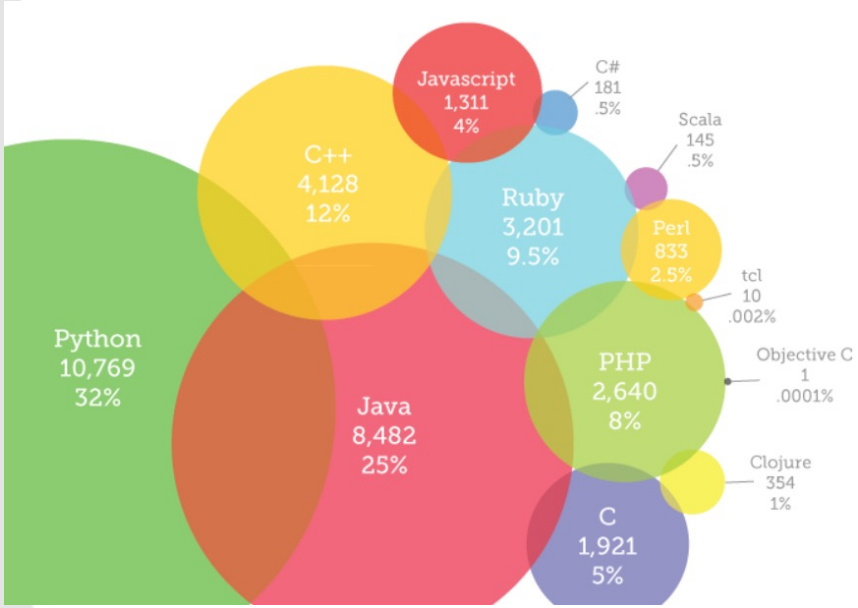


1,000+ coding challenges across 20+ financial services companies

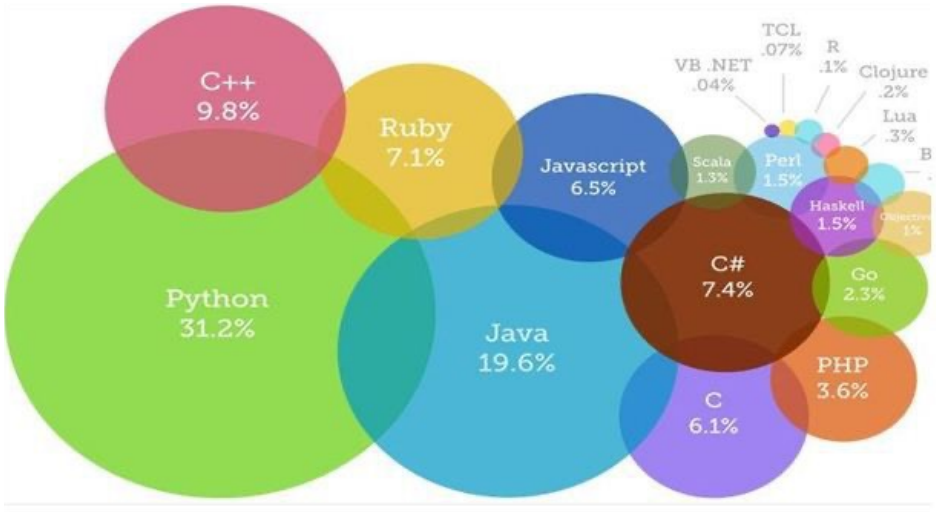
450+ coding challenges across 20+ companies in FinTech

Trends

2012



2018



Programming Language Community

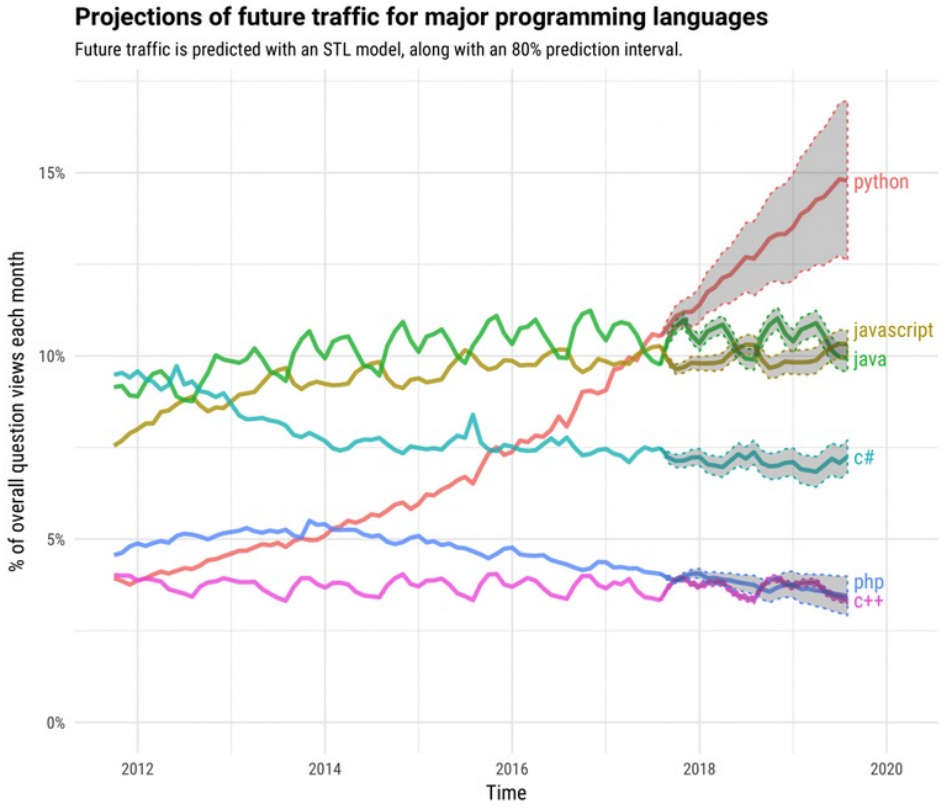


Image: Stack Overflow

Why are you learning C++?

- Not the most difficult one to learn ☺
  - Even if Basic and Pascal are easier
  - Ada, Prolog, Caml and of course Assembly Languages are very difficul

- General-purpose language
- Strong bias toward systems programming
- One layer above assembly language
- Paradigms: Imperative, Object-oriented, Functional, Meta-programming
- International standards
- High performance, type-safe, statically verifiable native software

Mother Tongues

Tracing the roots of computer languages through the ages

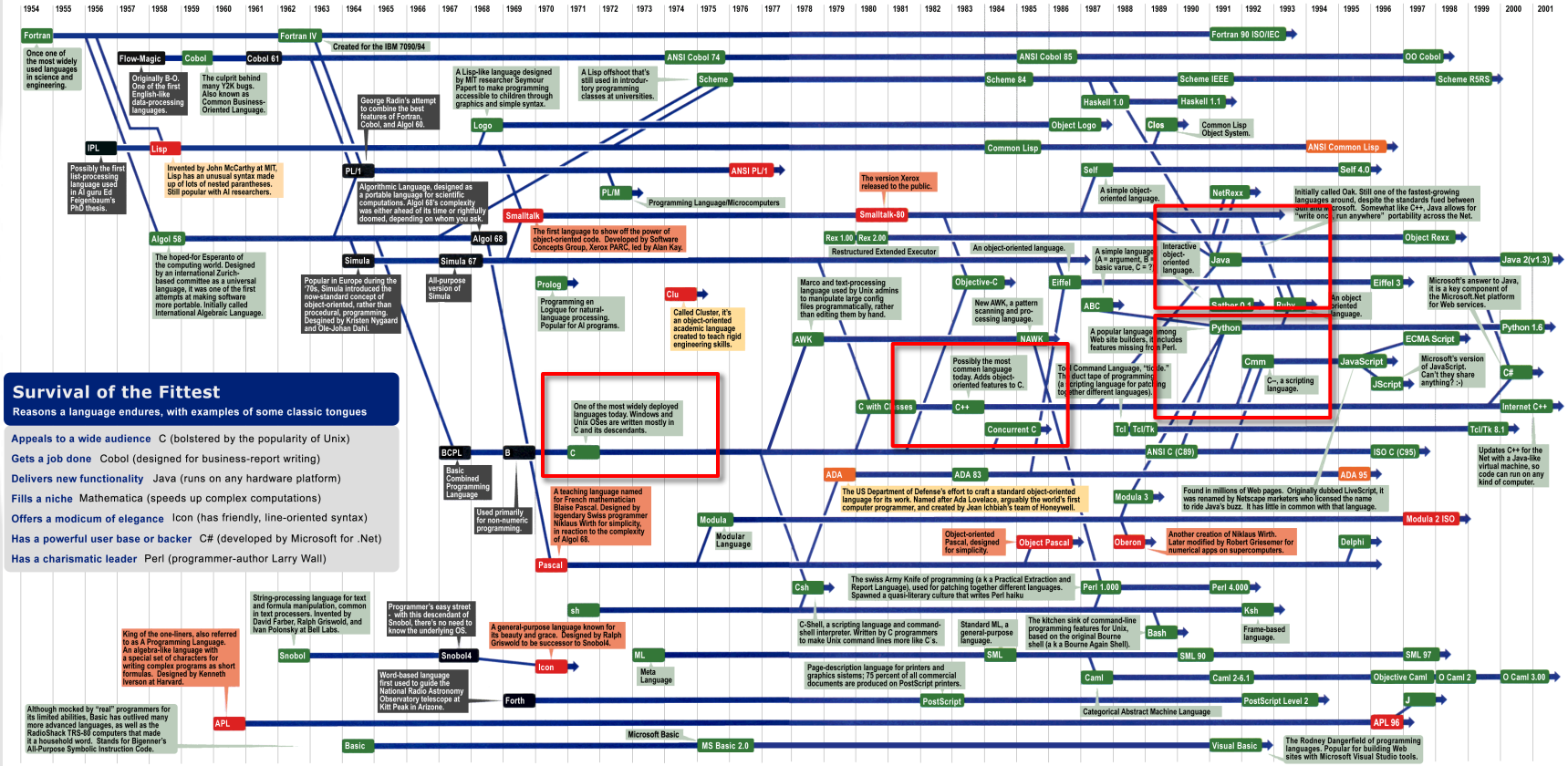
Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers-electronic lexicographers, if you will-aim to save, or at least document the lingo of classic software. They're combining the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua frangas. Among the most endangered are Ada, APL, B (the predecessor of C), Lsp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Musuem in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at *HTTP://www.informatik.uni-freiburg.de/Java/misc/lang_list.html*. - Michael Mendeno

Key

- 1954 Year Introduced
- **Active:** thousands of users
- **Protected:** taught at universities; compilers available
- **Endangered:** usage dropping off
- **Extinct:** no known active users or up-to-date compilers
- **Lineage continues**



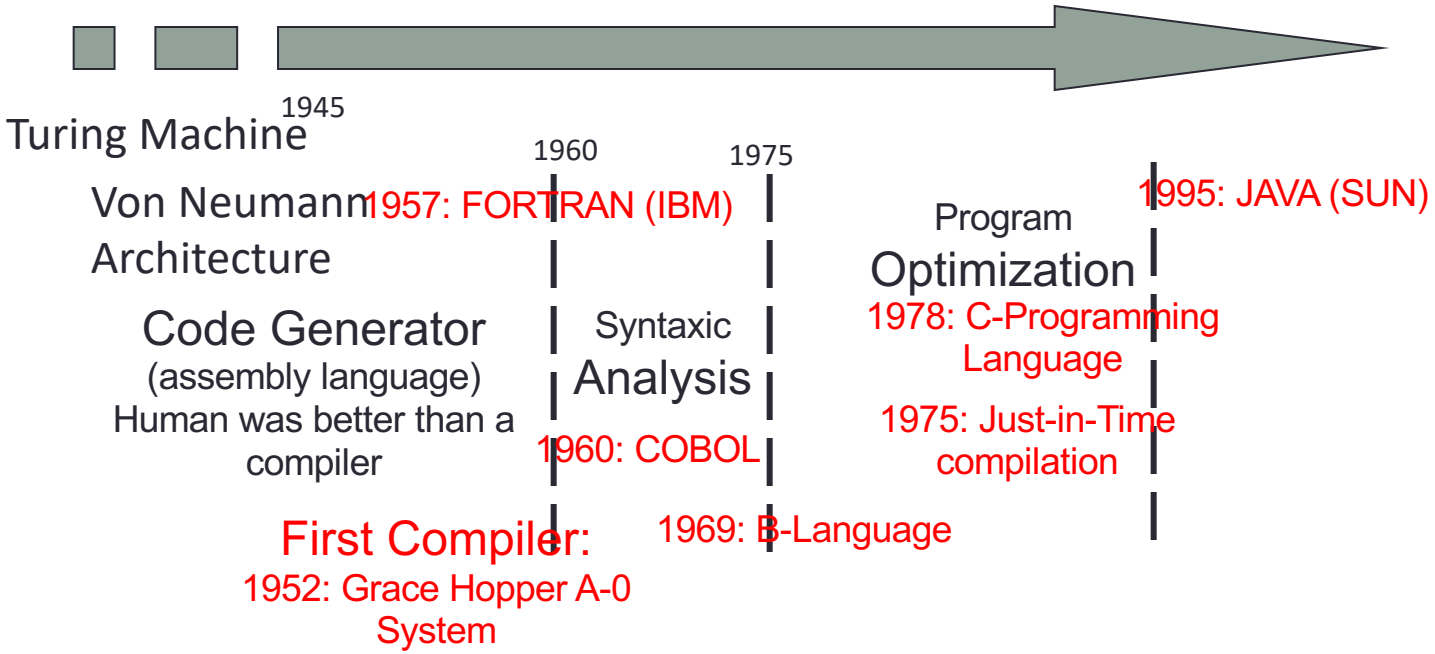
Survival of the Fittest

Reasons a language endures, with examples of some classic tongues

- **Appeals to a wide audience** C (bolstered by the popularity of Unix)
- **Gets a job done** Cobol (designed for business-report writing)
- **Delivers new functionality** Java (runs on any hardware platform)
- **Fills a niche** Mathematica (speeds up complex computations)
- **Offers a modicum of elegance** Icon (has friendly, line-oriented syntax)
- **Has a powerful user base or backer** C# (developed by Microsoft for .Net)
- **Has a charismatic leader** Perl (programmer-author Larry Wall)

Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University

Modern Compiler History

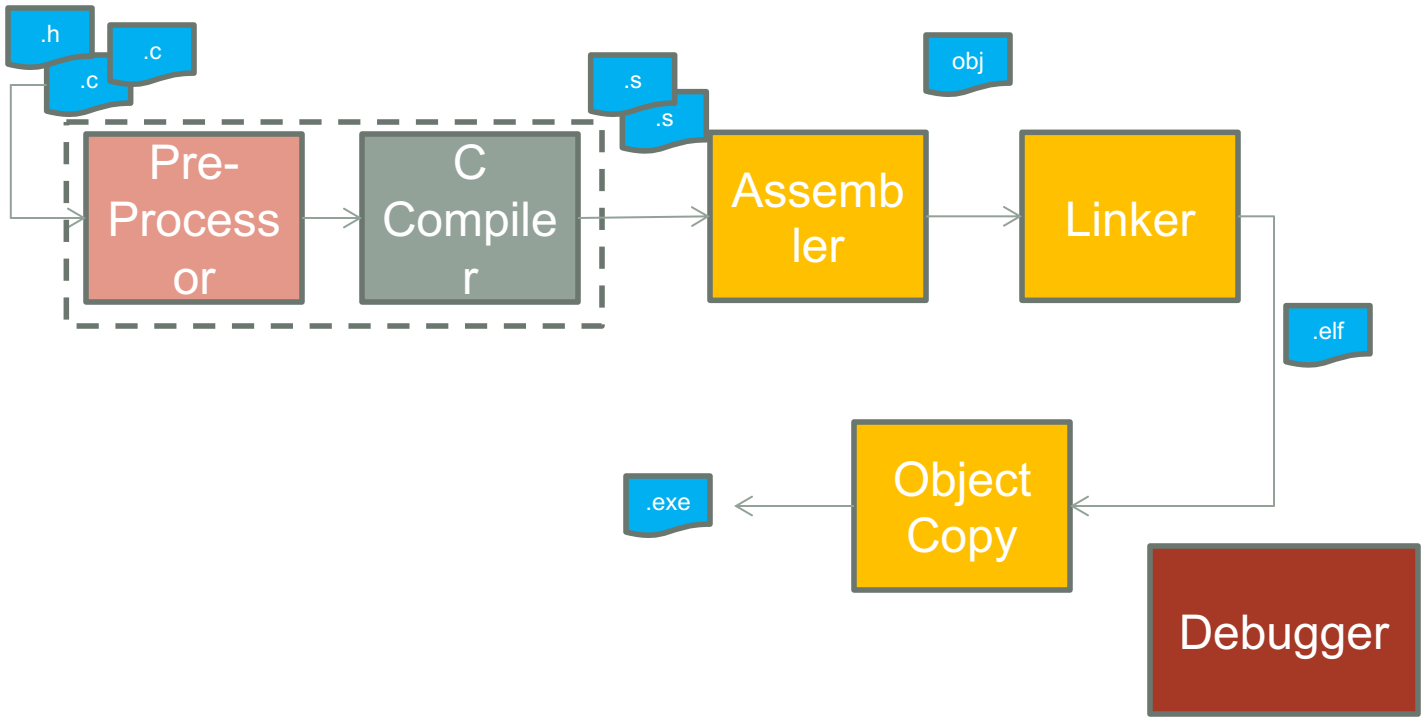


Compiler / Interpreter

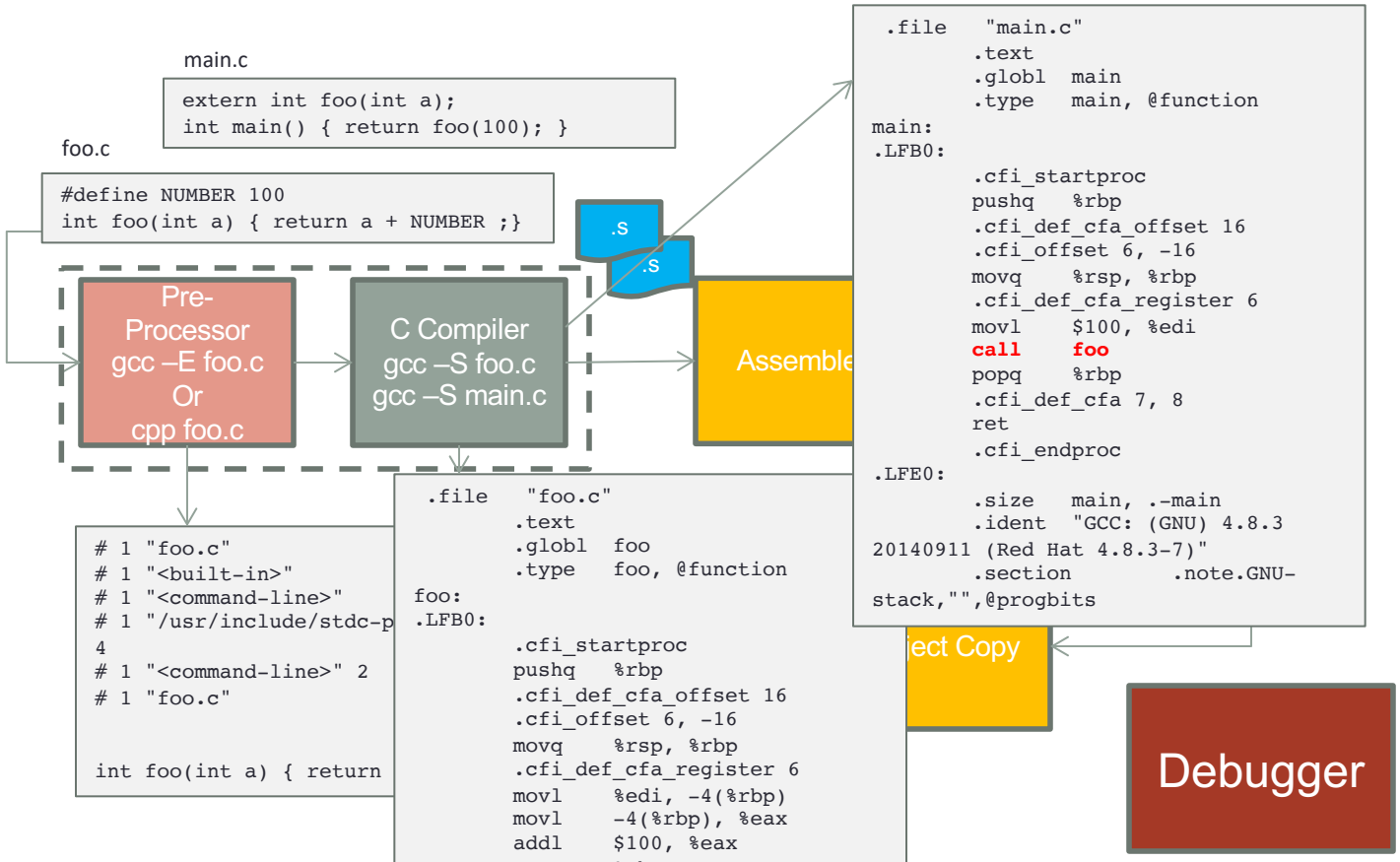
Compiler	Interpreter
• Intermediate Object Code Generated • Faster • More Memory Needed • Needs to be compiled before execution Example: C/C++	• **No** Intermediate Object Code Generated • Slower • Less Memory Needed • **No** Need to be compiled before execution Example: Python/JAVA*

* Java is more complex: compiled to bytecode and interpreted

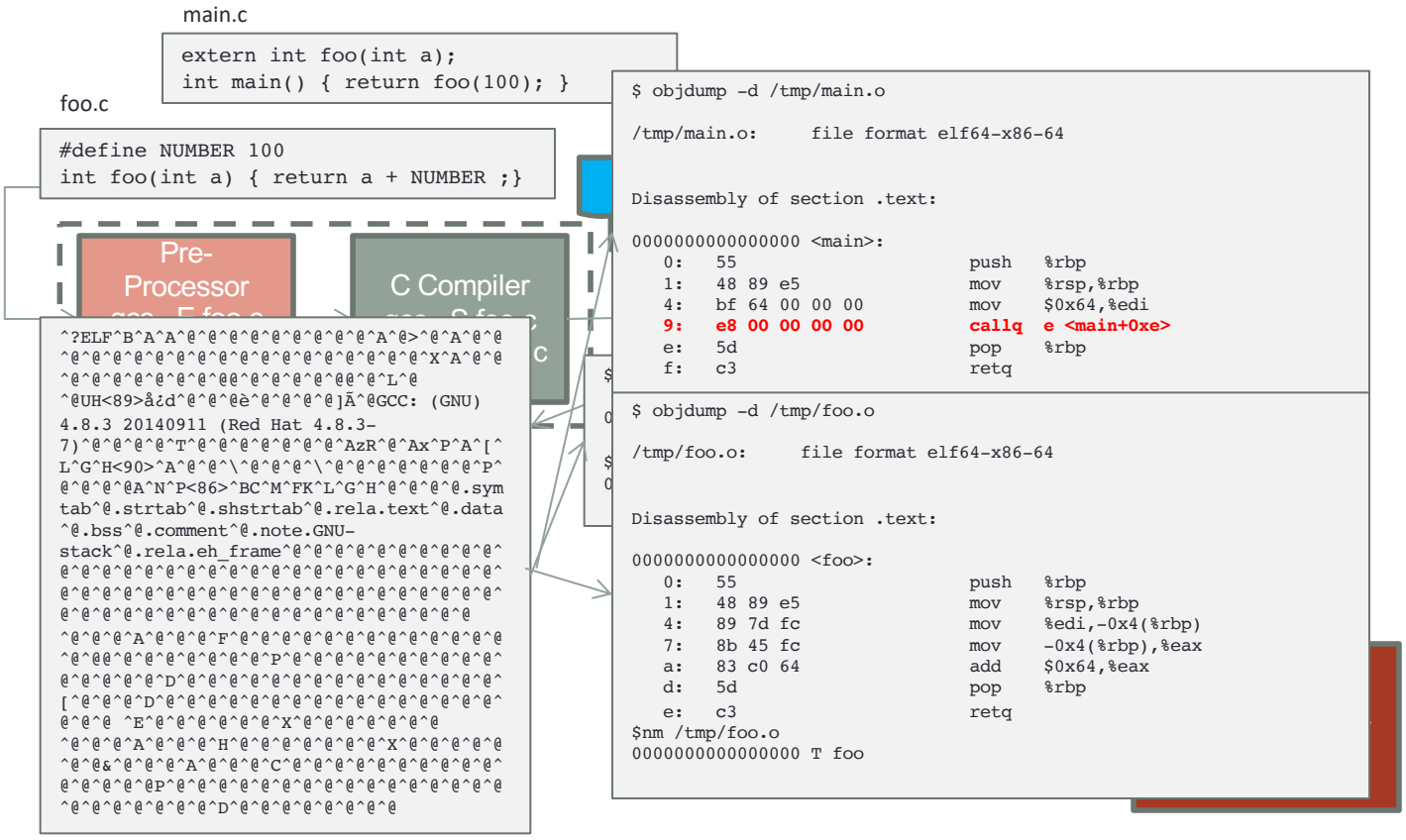
Compiler / Tool-chain (example: GNU Toolchain)



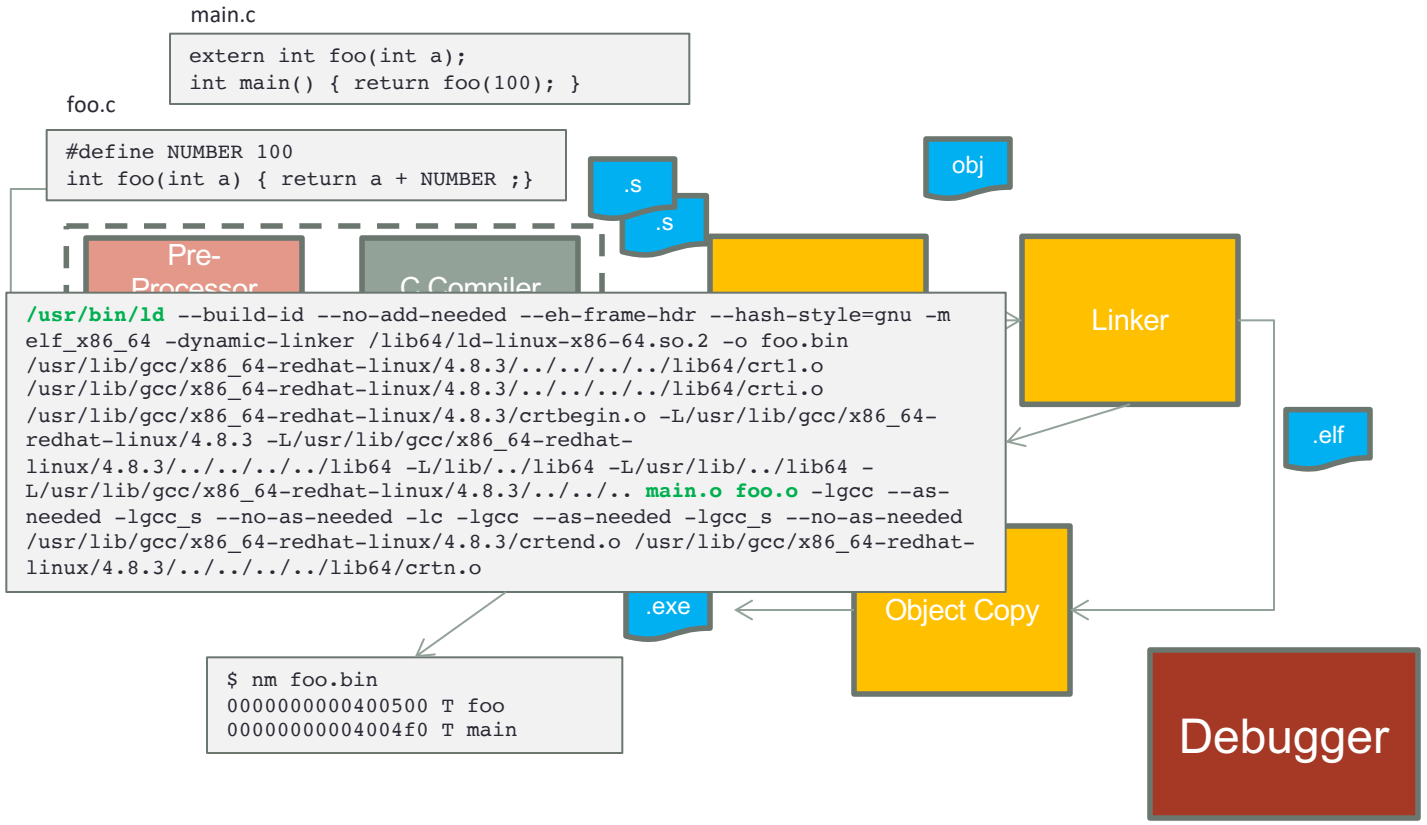
Concrete Example



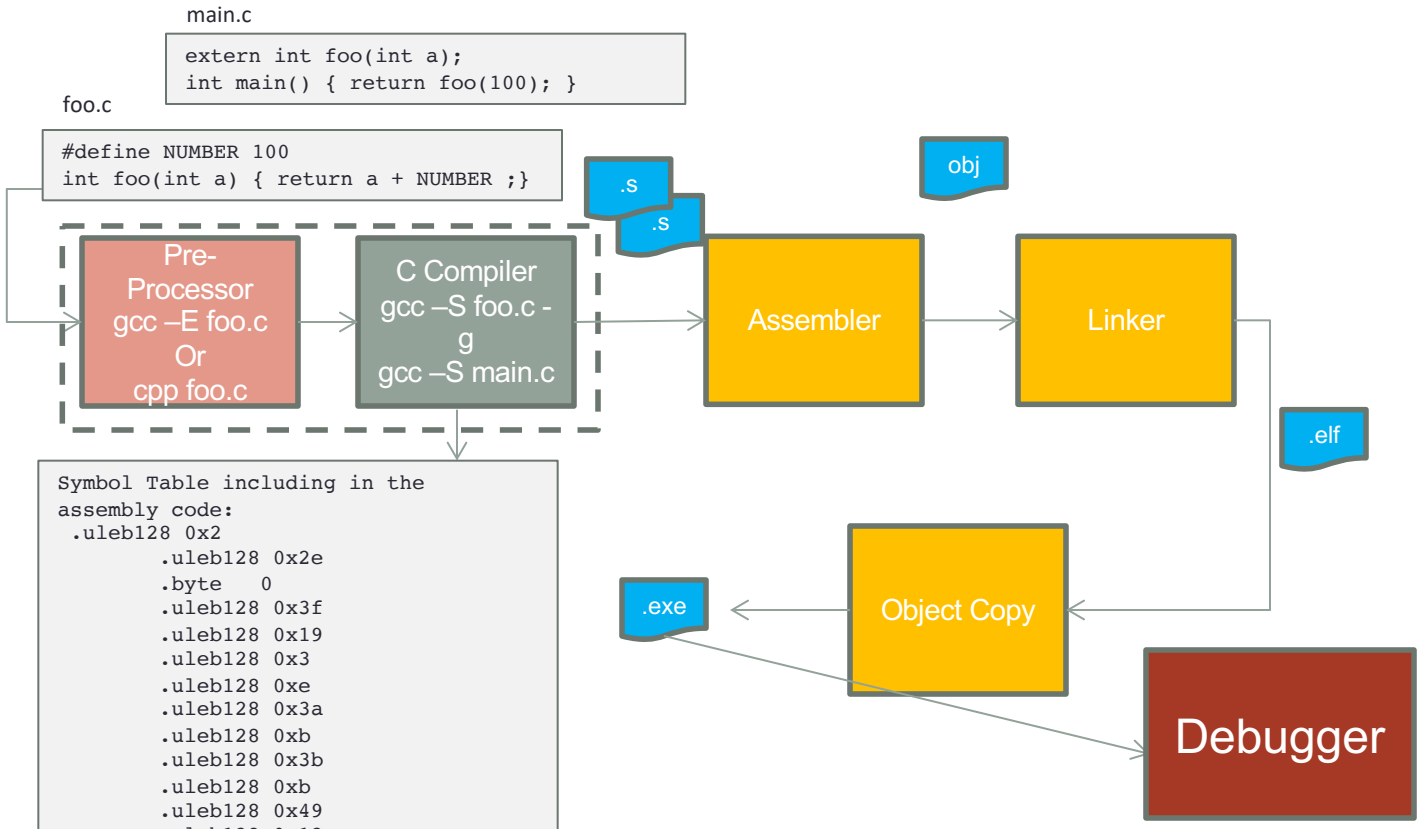
Concrete Example



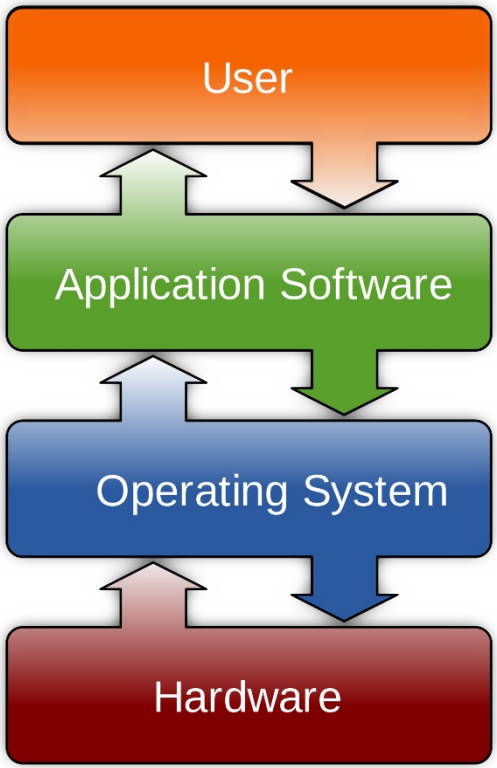
Concrete Example



Concrete Example



Trading System



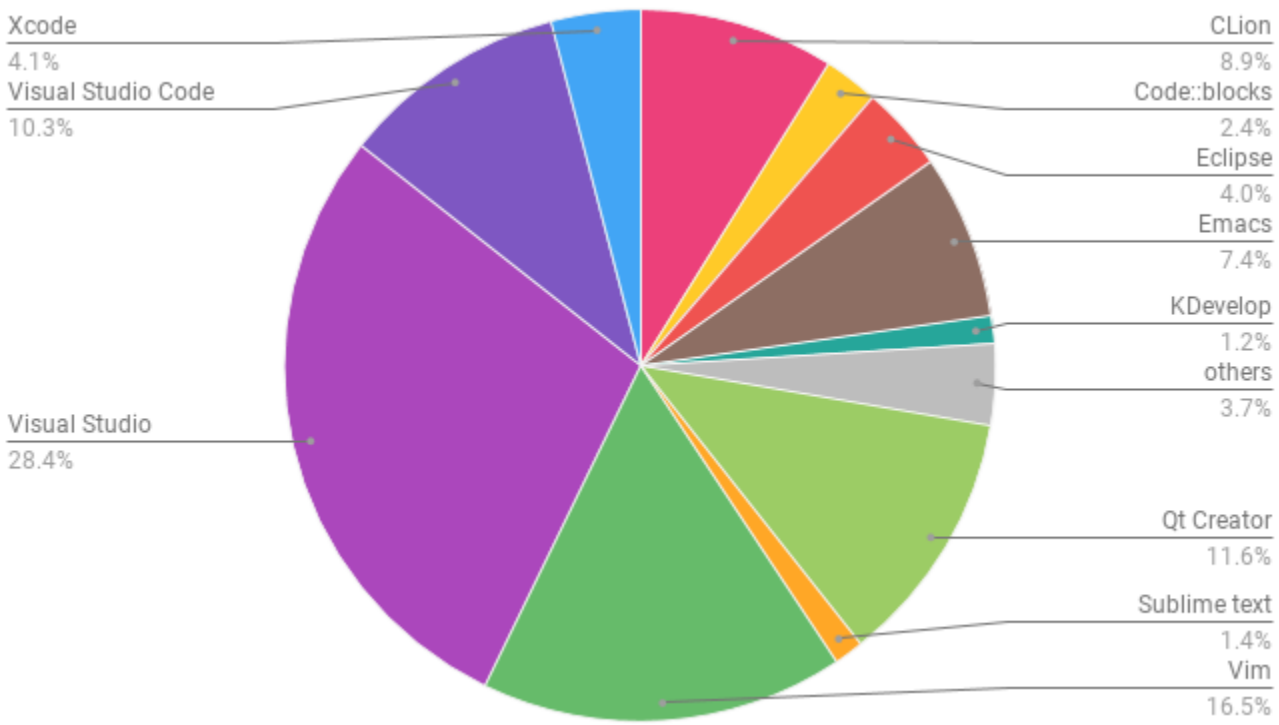
Traders

Trading system

Unix – Windows – MacOS

IDE (Integrated Development Environment)

- Microsoft Visual Studio
- JetBrains Clion
- Code::Blocks
- Eclipse CDT
- Atom
- Vim ☺



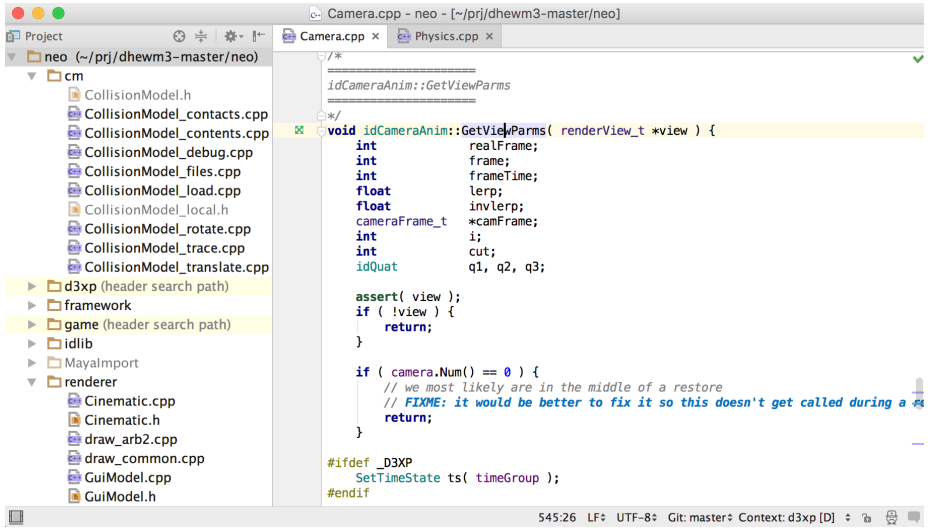
Clion with Cygwin

Download Clion

- https://www.jetbrains.com/clion/download/#section=windows

How to install it on Windows?

- https://www.jetbrains.com/help/clion/quick-tutorial-on-configuring-clion-on-windows.html#Cygwin



Create your first code “Hello class”

- If you didn’t have time to install Clion, just use:
- http://www.tutorialspoint.com/compile_cpp_online.php
- And t

```
1: // Preprocessor directive that includes header iostream
2: #include <iostream>
3:
4: // Start of your program: function block main()
5: int main()
6: {
7: /* Write to the screen */
8: std::cout << "Hello class" << std::endl;
9:
10: // Return a value to the OS
11: return 0;
12: }
```

Dissection of the C++ program "Hello class!"

- Lines 1, 4, 7, and 10, which start with a // or with a /*, are called *comments* and are ignored by the compiler. These comments are for humans to read.

- In Line 2, #include <filename> tells the preprocessor to take the contents of the file (iostream, in this case) and include them at the line where the directive is made
  - ***iostream*** is a standard header file that enables the usage of std::cout

- In Line 5, the body of the program characterized by the function main().
  - In many (`int main (int argc, char* argv[])` like this:

- In Line 11, Conventionally programmers return 0 in the event of success or –1 in the event of error

Dissection of the C++ program “Hello class!”

- Lines 8, we are using std:: in the expression
- `std::cout << "Hello class" << std::endl`
- We use the std (pronounced “standard”) namespace to invoke functions, streams, and utilities that have been ratified by

```
1: // Preprocessor directive
2: #include <iostream>
3:
4: // Start of your program
5: int main()
6: {
7: // Tell the compiler what namespace to search in
8: using namespace std;
9:
10: /* Write to the screen using std::cout */
11: cout << "Hello World" << endl;
12:
13: // Return a value to the OS
14: return 0;
15: }
```

Stdin, stdcerr, stdout

- std::cout (pronounced “standard see-out”) to write simple text data to the console
- std::cin (“standard see-in”) to read text and numbers (entered using the keyboard) from the console

```
#include <iostream>

int main(int args, char *argv[])
{
    int number;
    std::cin >> number;
    if (number==0)
    {
        std::cerr << "0 cannot be used!" << std::endl;
        return -1;
    }
    std::cout << number;
    return number;
}
```

Small quizz

- 1- What is the definition between an interpreter and a compiler?
- 2- Can you explain the function of the linker?
- 3- What does #include do?

Using variables and constants

- Variables: help the programmer temporarily store data for
- a finite amount of time

- Declaration:
- `VariableType VariableName;`
- OR `VariableType VariableName = InitVariable;`

- Constants: help the programmer define artifacts that are not allowed to change

Scope of a variable

```
#include <iostream>

int number;

int main(int args, char *argv[])
{
    number=4;
    {
        number = 6;
        int number;
        number=2;
    };

    std::cout << number << std::endl;
    return number;
}
```

- Ordinary variables like the ones we have declared this far have a well-defined scope within which they're valid and can be used.
- When used outside their scope, the variable names will not be recognized by the compiler and your program won't compile. Beyond its scope, a variable is an unidentified entity that the compiler knows nothing of.

Data types

Bits	Name	Range (assuming two's complement for signed)	Decimal Digits (approx.)	Uses	Implementations				
					C/C++	C#	Pascal and Delphi	Java	SQL[a]
4	nibble, semioctet	*Signed:* From −8 to 7, from $-(2^3)$ to $2^3 - 1$	1	Binary-coded decimal, single decimal digit representation.	n/a	n/a	n/a	n/a	n/a
		Unsigned: From 0 to 15 which equals $2^4 - 1$	2						
8	byte, octet	*Signed:* From −128 to 127, from $-(2^7)$ to $2^7 - 1$	3	ASCII characters	`int8_t`, `char`[b]	`sbyte`	`Shortint`	`byte`	`tinyint`
		Unsigned: From 0 to 255 which equals $2^8 - 1$	3		`uint8_t`, `char`[b]	`byte`	`Byte`	n/a	`unsigned tinyint`
16	halfword, word, short	*Signed:* From −32,768 to 32,767, from $-(2^{15})$ to $2^{15} - 1$	5	UCS-2 characters	`int16_t`, `short`[b], `int`[b]	`short`	`Smallint`	`short`	`smallint`
		Unsigned: From 0 to 65,535 which equals $2^{16} - 1$	5		`uint16_t`	`ushort`	`Word`	`char`[d]	`unsigned smallint`
32	word, long, doubleword, longword, int	*Signed:* From −2,147,483,648 to 2,147,483,647, from $-(2^{31})$ to $2^{31} - 1$	10	UTF-32 characters, True color with alpha, FourCC, pointers in 32-bit computing	`int32_t`, `int`[b], `long`[b]	`int`	`LongInt`; `Integer`[c]	`int`	`int`
		Unsigned: From 0 to 4,294,967,295 which equals $2^{32} - 1$	10		`uint32_t`	`uint`	`LongWord`; `DWord`; `Cardinal`[c]	n/a	`unsigned int`
64	word, doubleword, longword, long long, quad, quadword, qword, int64	*Signed:* From −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, from $-(2^{63})$ to $2^{63} - 1$	19	Time (milliseconds since the Unix epoch), pointers in 64-bit computing	`int64_t`, `long`[b], `long long`[b]	`long`	`Int64`	`long`	`bigint`
		Unsigned: From 0 to 18,446,744,073,709,551,615 which equals $2^{64} - 1$	20		`uint64_t`	`ulong`	`UInt64`; `QWord`	n/a	`unsigned bigint`
128	octaword, double quadword	*Signed:* From −170,141,183,460,469,231,731,687,303,715,884,105,728 to 170,141,183,460,469,231,731,687,303,715,884,105,727, from $-(2^{127})$ to $2^{127} - 1$	39	Complicated scientific calculations	C: only available as non-standard compiler-specific extension	n/a	n/a	n/a	n/a
		Unsigned: From 0 to 340,282,366,920,938,463,463,374,607,431,768,211,455 which equals $2^{128} - 1$	39	IPv6 addresses GUIDs					
n	*n*-bit integer (general case)	*Signed:* $(-(2^{n-1}))$ to $(2^{n-1} - 1)$	$\lceil (n-1) \log_{10} 2 \rceil$		Ada: `range -2**(n-1)..2**(n-1)-1`				
		Unsigned: 0 to $(2^n - 1)$	$\lceil n \log_{10} 2 \rceil$		Ada: `range 0..2**n-1`, `mod 2**n`; standard libraries' or third-party arbitrary arithmetic libraries' BigDecimal or Decimal classes in many languages such as Python, C++, etc.				

You would use an unsigned variable type when you expect only positive values. So, if you're counting the number of apples, don't use `int`; use `unsigned int`. The latter can hold twice as many values in the positive range as the former can.

Variable size

- Operator *sizeof* will return the size of a variable in bytes

```
#include <iostream>
2:
3: int main()
4: {
5: using namespace std;
6: cout << "Computing the size of some C++ inbuilt variable types" << endl;
7:
8: cout << "Size of bool: " << sizeof(bool) << endl;
9: cout << "Size of char: " << sizeof(char) << endl;
10: cout << "Size of unsigned short int: " << sizeof(unsigned short) << endl;
11: cout << "Size of short int: " << sizeof(short) << endl;
12: cout << "Size of unsigned long int: " << sizeof(unsigned long) << endl;
13: cout << "Size of long: " << sizeof(long) << endl;
14: cout << "Size of int: " << sizeof(int) << endl;
15: cout << "Size of unsigned long long: "<< sizeof(unsigned long long)<<
endl;
16: cout << "Size of long long: " << sizeof(long long) << endl;
17: cout << "Size of unsigned int: " << sizeof(unsigned int) << endl;
18: cout << "Size of float: " << sizeof(float) << endl;
19: cout << "Size of double: " << sizeof(double) << endl;
20:
21: cout << "The output changes with compiler, hardware and OS" << endl;
22:
23: return 0;
24: }
```

Auto / Typedef

- Since the C++11 norm, you can now use *auto* as a type
- Compile will infer the data type (Compiler type-inference)

- You can use *typedef* to substitute variable type to a name that you think more convenient

Contants

- Constants in C++ can be
- Literal constants
  - String, numbers (int, float), binary literals,…
- Declared constants using the const keyword
  - Const It is good programming practice to define variables that are not supposed to change their values as *const*.
- Constant expressions using the constexpr keyword (new since C++11)
  - constexpr double GetPi() {return 22.0 / 7;}
- Enumerated constants using the enum keyword
  - enum
- Defined constants that are not recommended and deprecated
  - *#define*
  - Defining constants using the preprocessor via #define is depre-
CAUTION cated and should not be used.

Enumerations

```cpp
#include <iostream>

enum order_type{
    GTC,
    GTD,
    FOK=10,
    ICEBERG
};

int main()
{
    std::cout << order_type::GTC << std::endl;
    std::cout << order_type::GTD << std::endl;
    std::cout << order_type::FOK << std::endl;
    std::cout << order_type::ICEBERG << std::endl;
}
```

```
C:\Users\sebastiend\CLionProjects\ieor-cplusplus\cmake-build-debug\ieor_cplusplus.exe
0
1
10
11
```

Quiz

- 1- Why should I initialize the value of variables?
- 2- Why shouldn't I use global variables all the time?
- 3- Why does C++ give me the option of using short int and int and long
- int? Why not just always use the integer that always allows for the highest
- number to be stored within?
- 4- What is the difference between a signed and an unsigned integer?
- 5- Why should you not use #define to declare a constant?

Previous session Quiz

- 1- What are the different major steps of compilation?
- 2- What is a namespace? Give an example of a namespace.
- 3- Why should I initialize the value of variables?
- 4- What is the purpose of a linker?
- 5- Can you give an example of code using the pre-processor?

Arrays and Strings

- What is an array?
  - It is a collection of element
  - All these elements have the same type
  - This collection is a complete set
- How do we initialize an array?
  - Type VariableType[CONST_NUMBER]
  - Type VariableType[CONST_NUMBER or empty]={} //={0} or ={1,2,3…}
- Be aware of boundaries and initialization

- Why having strings and arrays on the same slide?

Strings

- `std::cout << "Hello World";`
- This is equivalent to using the array declaration:
- ```
  char sayHello[] = {'H', 'e', 'l', 'l', 'o',
  ' ', 'W', 'o', 'r', 'l', 'd','\0'};
  ```
- `std::cout << sayHello << std::endl;`
- What is '\0'?
  ```

String manipulation

- Let's test the functions:

```
#include <string.h>
size_t strlen(const char *s);
```

```
#include <string.h>
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
```

```
#include <string.h>
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

Quiz

- Create the function concatenation taking 2 strings as arguments

std::string

- C++ standard created a type string:
- efficient and safer way to deal with text input
- perform string manipulations like concatenations. std::string is not static in size like a char array implementation of a C-style string is
- can scale up when more data needs to be stored in it.

Dynamic arrays

- The class vector from the STL library
- `#include <vector>`
- We will come back a bit later

Quiz

- 1- Why initializing arrays?
- 2- Would you like to keep using C-style strings?
- 3- Does the length of a string include the character ‘\0’ at the end?
- 4- You forget to end your C-style string with a null-terminator. What happens when you use it?

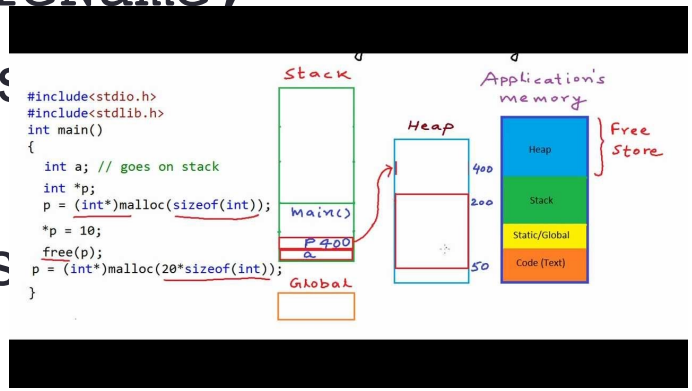
Pointers

- What is a pointer?
- A variable
- It contains an address in memory
- How to create them?
- `PointedType * PointerVariableName;`
- Like for all the other variable, you mus
- `PointerVariableName = NULL;`
- Size of a pointer: use sizeof (depends compiler/OS)
- 32-bit / 64-bit difference

Example:

```
#include <iostream>
int main()
{
    int *int1 = new int();
    int *int2 = new int[20]();
    *int1=3;
    int2[0]=1;int2[1]=1;int2[2]=1;int2[3]=1;
    std::cout << *int1 << std::endl;
    std::cout << int2[0] << int2[1] << int2[2] <<
int2[3]  << std::endl;
    delete int1;
    delete[] int2;

}
```



Pointers

- Play with & and *
- New and Delete
- Increment and decrement a pointer -- and ++
- Can we use `const`?
- Reference to arrays

- Main mistakes by using pointers:
  - Memory leaks
  - Point to invalid memory locations
  - Dangling pointers (after delete)

Must DO

- Always initialize pointer
- Release Memory

DO NOT

- Access a block of memory or use a pointer after it has been released by `delete`
- Invoke `delete` on a memory address more than once

Reference

- A reference is **an alias for** a variable
- When you declare a reference you need to associate it to a variable
- Syntax:
- `VarType original = Value;`

`VarType &ReferenceVariable = original;`

```
#include <iostream>
int main()
{
    int original = 30;
    const int& constRef = original;
    constRef = 40; // Not allowed: constRef can't
change value in original
    int& ref2 = constRef; // Not allowed: ref2 is not
const
    const int& constRef2 = constRef; // OK
    return 0;
}
```

Quiz

- 1- I have two pointers:
  - `int* pointToAnInt = new int;`
  - `int* pCopy = pointToAnInt;`
  - Am I not better off calling delete using both to ensure that the memory is gone?
- 2- What would you use?
  - `void CalculateArea (const double* const ptrRadius, double* const ptrArea);`
  - `void CalculateArea (const double& radius, double& area);`
- 3- Difference between:
  - `int myNumbers[100];`
  - `int* myArrays[100];`

Statements and Operators

- An *operator* is something that transforms data. Common operators in C++:
- = Assignment operator, takes the value on the right and puts it into the variable on the left.
- + Addition operator
- - Subtraction operator
- * Multiplication operator
- / Division operator
- % Modulus operator (remainder after a division)
- ++, --, +=, -=, *=, /=, ==, !=, <, >, <=, >=, !, &, &&, |, ||, ^, <<, >>, sizeof

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	++ -- () [] . ->	Suffix/postfix increment and decrement Function call Array subscripting Element selection by reference Element selection through pointer	
3	++ -- + - ! ~ (*type*) * & sizeof new, new[] delete, delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Dynamic memory allocation Dynamic memory deallocation	Right-to-left
4	.* ->*	Pointer to member	Left-to-right
5	* / %	Multiplication, division, and remainder	
6	+ -	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
9	== !=	For relational = and ≠ respectively	
10	&	Bitwise AND	
11	^	Bitwise XOR (exclusive or)	
12	|	Bitwise OR (inclusive or)	
13	&&	Logical AND	
14	||	Logical OR	
15	?: = += -= *= /= %= <<= >>= &= ^= |=	Ternary conditional Direct assignment (provided by default for C++ classes) Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	Right-to-left
16	throw	Throw operator (for exceptions)	
17	,	Comma	Left-to-right

```
1   #include <iostream>
2   #include <list>
3
4   #define LOG(X,Y) {std::cout << "case " << X << ": "  << Y <<
std::endl;}
5
6   int function1(int a) {
7       a++;
8       return a;
9   }
10
11  int function2(int &a) {
12      a++;
13      return a;
14  }
15
16  int function3(int *a) {
17      (*a)++;
18      return *a;
19  }
20
21
22  int main() {
23      int a=2;
24      LOG(1,function1(a));
25      LOG(2,function2(a));
26      LOG(3,function2(a));
27      LOG(4,function3(&a));
28      return 0;
29  ;}
```

Controlling Program Flow

- if…else

```
#include <iostream>
int main() {
    int a = 1;
    if (a<2)
        if(a<1)
            std::cout << "Ending 1" << std::endl;
    else
        std::cout << "Ending 2" << std::endl;

    return 0;
}
```

- switch

```
switch(expression)
{
    case LabelA:
    DoSomething;
    break;
    case LabelB:
    DoSomethingElse;
    break;
    // And so on...
    default:
    DoStuffWhenExpressionIsNotHandledAbove;
    break;
}
```

Controlling Program Flow

- Conditional Execution Using Operator (?:)

```
#include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5: cout << "Enter two numbers" << endl;
6: int num1 = 0, num2 = 0;
7: cin >> num1;
8: cin >> num2;
9:
10: int max = (num1 > num2)? num1 : num2;
11: cout << "The greater of " << num1 << " and " \
12: << num2 << " is: " << max << endl;
13:
14: return 0;
15: }
```

- goto
- while
- do..while
- for loop
- continue/break

Range based for-loop

Since C++11

```
for (VarType varName : sequence)
{
   // Use varName that contains an element from sequence
}
```

```
#include <iostream>

int main()
{
    std::string s("hello class !!");
    for (char varName : s)
    {
        std::cout << varName << std::endl;
    }
}
```

```
#include <iostream>

int main()
{
    int someNums[] = { 1, 101, -1, 40, 2040 };
    for (int varName : someNums)
    {
        std::cout << varName << std::endl;
    }
}
```

Try auto

Quiz

- 1- Create a Multi-dimensional array and iterate through all the values.
- 2- What happens if I forget a break in a switch-case statement?
- 3- Give the code of an infinite loop.
- 4- How can I exit an infinite loop?
- 5- while (-1) will result in doing what?
- 6- code a matrix-matrix multiply

Functions

- Function definition:
- ReturnType FunctionName([parameters][, parameters])

- A function can have no return value: void.
- A function can have no arguments
- A function can have arguments with default value
- A function can be recursive (functions which invoke themselves)
- A function can have many returns
- A function can return auto type

Function overloading

- Functions with the same name and return type but with different parameters or set of parameters are said to be overloaded functions.

```
double Area(double radius); // for circle
double Area(double radius, double height); // for cylinder
```

Different types of argument

- Passing an argument by values
- Passing an array of values to a function
- Passing an argument by reference

Inline function

Why do we have inline function?

```
0: #include <iostream>
1: using namespace std;
2:
3: // define an inline function that doubles
4: inline long DoubleNum (int inputNum)
5: {
6: return inputNum * 2;
7: }
8:
9: int main()
10: {
11: cout << "Enter an integer: ";
12: int inputNum = 0;
13: cin >> inputNum;
14:
15: // Call inline function
16: cout << "Double is: " <<
DoubleNum(inputNum) << endl;
17:
18: return
```

Lambda functions

- There are anonymous functions
- They are simplified functions

```
0: #include <iostream>
1: #include <algorithm>
2: #include <vector>
3: using namespace std;
4:
5: void DisplayNums(vector<int>& dynArray)
6: {
7: for_each (dynArray.begin(), dynArray.end(), \
8: [](int Element) {cout << Element << " ";} );
9:
10: cout << endl;
11: }
12:
13: int main()
14: {
15: vector<int> myNums;
16: myNums.push_back(501);
17: myNums.push_back(-1);
18: myNums.push_back(25);
19: myNums.push_back(-35);
20:
21: DisplayNums(myNums);
22:
23: cout << "Sorting them in descending order" << endl;
24:
25: sort (myNums.begin(), myNums.end(), \
26: [](int Num1, int Num2) {return (Num2 < Num1); } );
27:
28: DisplayNums(myNums);
29:
30: return 0;
31: }
```

Quiz

- 1- What happens if my recursive function never ends?
- 2- Why not inlining all the functions?
- 3- I have declared two functions, both with the same name and return type but different parameter lists. What are these called?

Classes

- You declare a class using the keyword `class` followed by the name of the class, followed by a statement block {…}

```
#include <stdint.h>
#include <iostream>

class Order
{
// Member attributes:
    std::string order_id;
    bool        is_bid;
    uint64_t    quantity;
    double      price;
// Member functions:
    void cancel();
    void fill(uint64_t qty);
    ...
};
```

Instance of an object

- An Object as an Instance of a Class
- Creating an object of type class Human is similar to creating an instance of another type:

```
double pi= 3.1415; // a variable of type double
Order order1; // first Order: an object of class Order
```

- We can also create a pointer:

```
int* pointsToNum = new int; // an integer allocated dynamically
delete pointsToNum; // de-allocating memory when done using
Order* order2 = new Order(); // dynamically allocated Orde
delete order2; // de-allocating memory
```

Public/Private Member Access

- You can access member with “.” and “->” outside the object.
- As long as the attributes is Public

Constructors/Destructors

- A **constructor** is the first function being called after allocation
- A **constructor** is a special function that takes the name of the class and returns no value.
- So, class Order will have this constructor:

```
class Order
{
    public:
    Order(); // declaration of a constructor
};
```

- A **destructor** is a automatically invoked when an object is destroyed.

```
class Order
{
public:
~Order();
};
```

Features of constructor

- Default constructor is a constructor without parameters
- You can choose of having:
- No default constructor
- Constructor Parameters with Default Values
- Constructor with Initialization List

Copy Constructor

- How to declare a copy constructor:

```
class Order
{
Order(const Order & Order); // copy constructor
};
Order :: Order(const Order & orderSource)
{
// Copy constructor implementation code
}
```

- When do we need a copy constructor implementation?
- Shallow copy/Deep copy

```
Order::operator= (const Order& orderSource)
{
//... copy assignment operator code
}
```

- Keep in mind that the copy assignment operator should follow the same rule

this pointer

- An important concept in C++, this is a reserved keyword applicable within the scope of a class and contains the address of the object. In other words, the value of this is &object.

```
void setOrderType(int orderype)
{
this->ordertype = ordertype; // same as
ordertype = ordertype
}
```

Struct <> Class

- Interview question:
- `struct` is a keyword from the days of C, and for all practical purposes it is treated by a C++ compiler similarly to a class.

- Unless specified, **members in a struct are public by default** (private for a class), and unless specified, a struct features **public inheritance from a base** struct (private for a class).

Friend of a Class

- A class **does not permit** external access to **its data members and methods** that are declared **private**.

- This rule is waived for a friend.
- You can friend a function or a class.
- Example:

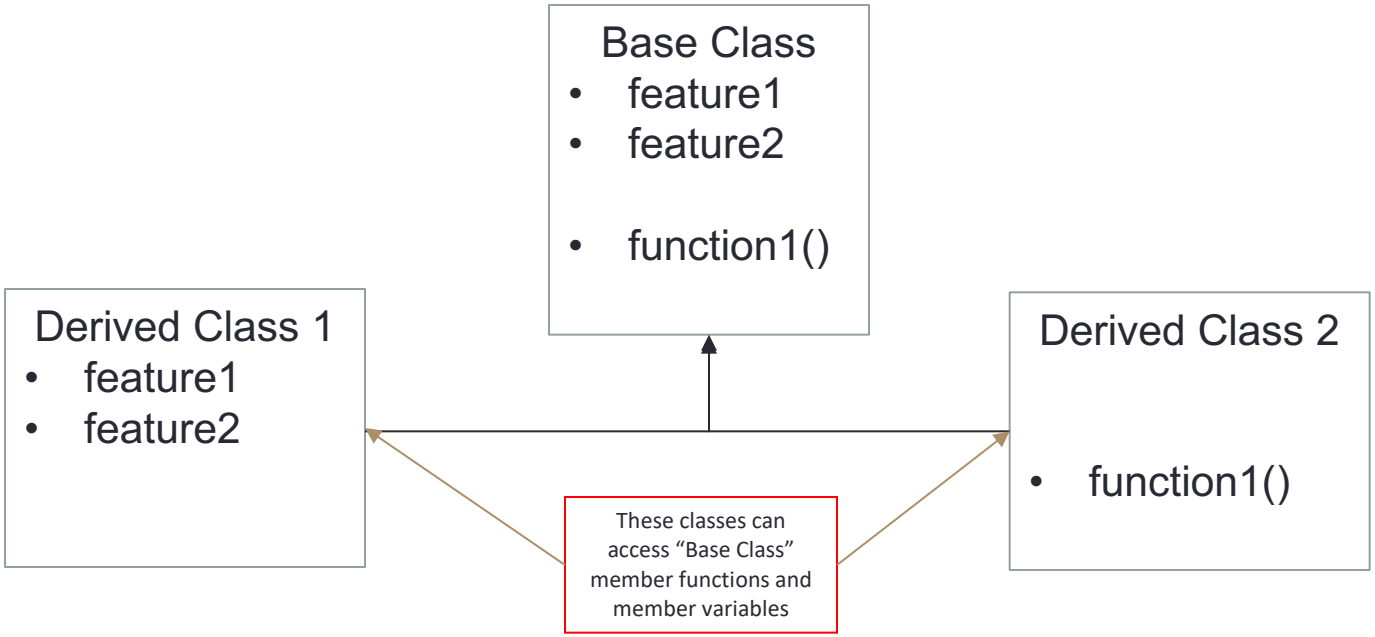
```
class Order
{
private:
    void UpdateOutstandPosition(const Order& order)
public:
    Order(); // declaration of a constructor
};
```

Quiz

- 1- What is the difference between the instance of a class and an object of that
- Class and the class itself?
- 2- Should I always program a copy constructor?
- 3- When I create an instance of a class using new, where is the class created?

Concept of Inheritance

Inheritance is one of the key features of Object-oriented programming in C++. It allows user to create a new class (derived class) from an existing class(base class).



Inheritance

```
class Base
{
// ... base class members
};
class Derived: access-specifier Base
{
// ... derived class members
};
```

The `access-specifier` can be one of `public` (most frequently used) where a "derived class is a base class" relationship; `private` or `protected` for a "derived class has a base class" relationship.

Implement an example of C++ inheritance

- You have “Financial Products” which has the following attributes:
- symbol
- price
- You have “Option” having the following attributes:
- expiration date
- You have “Fixed income” having the following attributes
- interest rate
- term date

Inheritance Features

- Derived class Overriding Base class's methods
- Invoking Methods of a Base Class in a Derived Class
  - Use scope resolution operator (::) with the base class name
- Constructors/Destructors orders?
- Overriding issue with base class methods (using resolution operator, using `using`)
- Public/Protected/Private inheritance
- Multiple inheritance

```
class Derived: access-specifier Base1, access-specifier Base2
{
    // class members
};
```

- Slicing

Quiz

- 1- In what order are the constructors created for inheritance?
- 2- What is the nature of inheritance with this code snippet?
  - `class Derived: Base`
  - `{`
  - `// ... Derived members`
  - `};`

Polymorphism

- In Greek:
- Poly: many
- Morph: form

```
class Base
{
virtual ReturnType FunctionName (Parameter List);
};
class Derived
{
ReturnType FunctionName (Parameter List);
};
```

Abstract Base Class

- A virtual method is said to be *pure virtual* when it has a declaration as shown in the following:

```
class AbstractBase
{
public:
virtual void DoSomething() = 0; // pure virtual method
};
```

Operator

- `return_type operator operator_symbol (...parameter list...);`

- Option 1 (using the increment operator):
- `++ holiday;`
- Option 2 (using a member function Increment()):
- `holiday.Increment();`

Unary operators

Operator	Name
++	Increment
--	Decrement
*	Pointer dereference
->	Member selection
!	Logical NOT
&	Address-of

Casting operators

- Need
- What examples of casting have we already encountered?
- Type-safe and type-strong

- C-style casts not popular, why?
- `char* staticStr = "Hello World!";`
- `int* intArray = staticStr;`
- or
- `int* intArray = (int*)staticStr;`

C++ Casting Operators

- static_cast
- dynamic_cast
- reinterpret_cast
- const_cast

- Usage:
- `destination_type result =`
- `cast_operator<destination_type>(object_to_cast);`

Static cast

- Similar as C-type casting
- Main difference: compile-time check to ensure that the pointer is being cast to a related type
- We will use static cast most of the time with inheritance:

```
Base* objBase = new Derived ();
Derived* objDer = static_cast<Derived*>(objBase); // ok!
// class Unrelated is not related to Base

Unrelated* notRelated = static_cast<Unrelated*>(objBase); // Error
// The cast is not permitted as types are unrelated
```

Static cast: upcast/downcast

- Casting a Derived* to a Base* is called *upcasting* and can be
- done without any explicit casting operator:
- `Derived objDerived;`
- `Base* objBase = &objDerived; // ok!`

- Casting a Base* to a Derived* is called *downcasting* and cannot
- be done without usage of explicit casting operators:
- `Derived objDerived;`
- `Base* objBase = &objDerived; // Upcast -> ok!`
- `Derived* objDer = objBase; // Error: Downcast needs explicit cast`

Dynamic cast

- opposite of static casting
- executes the cast at runtime
- dynamic_cast operation can be checked to see whether the attempt at casting succeeded
- The typical usage syntax of the dynamic_cast operator is

```
destination_type* Dest = dynamic_cast<class_type*>(Source);
if(Dest) // Check for success of the casting operation
        Dest->CallFunc ();
```

- For example:

```
Base* objBase = new Derived();
// Perform a downcast
Derived* objDer = dynamic_cast<Derived*>(objBase);
if(objDer) // Check for success of the cast
        objDer->CallDerivedFunction ();
```

- Real example: Cast a base class into a derived class

Reinterpret cast

- Same as C-type casting
- allow the programmer to cast one object type to another, regardless of whether or not the types are related
- forces the compiler to accept situations that static_cast would normally not permit

Const cast

- const_cast enables you to turn off the const access modifier to an object

```
class SomeClass{
public:
// ...
void DisplayMembers(); //problem - display function isn't const
};
```

- const_cast enables you to turn off the const access modifier to an object

```
void DisplayAllData (const SomeClass& object)
{
object.DisplayMembers (); // Compile failure
// reason: call to a non-const member using a const reference
}
```

Preprocessors

- The preprocessor, as the name indicates, is what runs before the compiler starts

- Preprocessor directives are characterized by the fact that they all start with a # sign.
- Example:

```
// instruct preprocessor to insert contents of iostream here
#include <iostream>
// define a macro constant
#define ARRAY_LENGTH 25
int numbers[ARRAY_LENGTH]; // array of 25 integers
// define a macro function
#define SQUARE(x) ((x) * (x))
int TwentyFive = SQUARE(5);
```

Using Macros for Protection against Multiple Inclusion

- C++ programmers typically declare their classes and functions in .H files called header files.
- The respective functions are defined in .CPP files that include the header files using the #include<header> preprocessor directive.
- For the preprocessor however, two header files that include each other is a problem of recursive nature.
- Solution use pre-processor:

```
#ifndef HEADER1_H _//multiple inclusion guard:
#define HEADER1_H_ // preprocessor will read this and following lines once
#include <header2.h>
class Class1
{
// class members
};
#endif // end of header1.h
```

Using **#define** to Write Macro Functions

- `#define SQUARE(x) ((x) * (x))`

- This helps determine the square of a number. Similarly, a macro that calculates the area of a circle looks like this:

- `#define PI 3.1416`
- `#define AREA_CIRCLE(r) (PI*(r)*(r))`

Macro assert

- `assert (expression that evaluates to true or false);`
- A sample use of assert() that validates the contents of a pointer is

```
#include <assert.h>
int main()
{
char* sayHello = new char [25];
assert(sayHello != NULL); // throws a message if pointer is NULL
// other code
delete [] sayHello;
return 0;
}
```

Macro Pros and Cons

- Pros:
- Reuse code, instead of creating 2 functions for different types,
- You can use this macro function MIN on integers:
- `cout << MIN(25, 101) << endl;`
- But you can reuse the same on double, too:
- `cout << MIN(0.1, 0.2) << endl;`

- Cons:
- Macros do not support any form of type safety
- USE TEMPLATE !!

Template Function Introduction

- Template Declaration:
- You begin the declaration of a template using the template keyword followed by a type parameter list. The format of this declaration is
- `template <parameter list>`
- `template function / class declaration..`
- The keyword template marks the start of a template declaration and is followed by the template parameter list. This parameter list contains the keyword typename that defines the template parameter objType, making it a placeholder for the type of the object that the template is being instantiated for.
- `template <typename T1, typename T2 = T1>`
- `bool TemplateFunction(const T1& param1, const T2& param2);`

Template Class Introduction

- A simple template class that uses a single parameter T to hold a member variable can be written as the following:

```
template <typename T>
class HoldVarTypeT
{
private:
T value;
public:
void SetValue (const T& newValue) { value = newValue; }
T& GetValue() {return value;}
};
```

Quiz

- 1- Why should I use inclusion guards in my header files?
- 2- When should I favor macro functions over templates if the functionality in
- question can be implemented in both?
- 3- Do I need to specify template arguments when invoking a template function?
- 4- Consider the following macro:
- `#define SPLIT(x) x / 5`
- What is the result if this is called with 20?
- 5- What is the result if the SPLIT macro in Question 2 is called with 10+10?
- 6- Implement a template function for swap that exchanges two variables.