Principles of Programming Languages

CS 245 — Spring 2019
kmicinski.com/cs245

Kristopher Micinski

Jocelyn Dunkley       Myriam (Mimi) Benkoussa
This class is about the **principles** of programming languages:

what kinds of languages are there? how do they work “under the hood”? how are they implemented?

We’re also going to learn 4-8, or more (depending on how we count), different programming languages!
First, a bit of history…
Humans create tools to help them solve problems

Programming Languages allow us to precisely express solutions to certain classes of problems
How do we (partially) formally specify what we want?
How do we (partially) formally specify what we want?

How do we write a program that does what we want?
How do we (partially) formally specify what we want?

How do we write a program that does what we want?

How do we run that program?
How do we (partially) formally specify what we want?

How do we write a program that does what we want?

How do we run that program?

How can we be convinced that program is correct?
~500 BC
Allows us to solve arithmetic problems (if you know how to use it)

≈500 BC

Not really a program, but machine that allows us to perform computation
Jacquard Loom (1804)

Reads punched cards to build, e.g., tapestries

You write a **program** to build the material

https://www.youtube.com/watch?v=OlJns3fPltE
Analog targeting computer (USS North Carolina)
 Helps aim guns given target distance / speed
 Works using gears..
Not general purpose!!

Analog targeting computer (USS North Carolina)
Helps aim guns given target distance / speed
Works using gears..
Ada Lovelace
Translated memoir describing general-purpose computer (1842)
Wrote notes of how to use this to compute Bernoulli numbers
Alonzo Church
Created lambda calculus (1936)
Alonzo Church

Created lambda calculus (1936)

Lambda calculus: mathematically specified language

Notably: a **general purpose** language

But **ridiculously simple**

\[
e ::= \ x \\
| \ \lambda x. \ e \\
| \ e \ e
\]

Alonzo Church
At this time, there were no “computer scientists”
Most people studying this were mathematicians, engineers, etc…

Also, nobody had actually **built** a general-purpose computer

So we were free to think about what languages would look like without thinking about hardware
Alan Turing (Church’s Student)

Invented Turing machines (1936)
Model of computation that includes:
- Read / Write Tape (memory)
- Head (current position on tape)
- Current state
- Instructions

Alan Turing
Alan Turing

Model of computation that includes:
- Read / Write Tape (memory)
- Head (current position on tape)
- Current state
- Instructions

Church-Turing Thesis:
Any **computable** function can be computed by **some** Turing machine
Turing’s Bombe
Cracks enigma by semi-brute-force exploiting a flaw in German code scheme
Even after his work cracking enigma, Turing was prosecuted for his homosexuality
He committed suicide at the age of 41
Several general-purpose languages came about, mainly targeted at mechanical computers in the early 50s.

These languages mostly resembled Turing machines and grew into the assembly languages we see today!
Corrado Böhm

Wrote first meta-circular compiler (1951)
In only 114 lines of code
John Backus

1954 — FORTRAN invented at IBM
First general purpose language w/ compiler that had widespread use
Also invented BNF
Grace Hopper

1955—Writes FLOW-MATIC (inspires COBOL)
1958—Invents LISP (inspiration for Scheme/Racket)

Gets variable scoping **wrong** because he failed to read **all** of Church’s 1936 paper...
“To use functions as arguments, one needs a notation for functions, and it seemed natural to use the λ-notation of Church (1941). I didn’t understand the rest of his book, so I wasn’t tempted to try to implement his more general mechanism for defining functions.”

“I must confess that I regarded this difficulty as just a bug and expressed confidence that Steve Russell would soon fix it. He did fix it but by inventing the so-called FUNARG device that took the lexical environment along with the functional argument. Similar difficulties later showed up in Algol 60, and Russell’s turned out to be one of the more comprehensive solutions to the problem.”

—John McCarthy, History of Lisp, 1979

http://jmc.stanford.edu/articles/lisp/lisp.pdf

Remember this when we talk about closures :-}
Margret Hamilton

1960s: leads team that writes assembly code for Apollo rockets / lunar module / command module
Think of how much testing this required!
Amazing what people can do even w/ weak languages!
Mid 60s: Ken Thompson and Dennis Ritchie get fed up hacking on the crummy code in MULTICS.

Start writing UNIX for fun to get away from their bad code—First versions written in assembly in 1969.
Writing in assembly is error-prone, so they created the C programming language—a derivative of BCPL (language around Bell labs at the time)

Early 70s: rewrite UNIX in C, create most famous operating system of all time

(My Mac’s kernel is based on UNIX!)
Barbara Liskov

Early 70s: CLU—classes, abstract types, iterators

Liskov Substitution Principle: subtyping!
Bjarne Stroustrup

1979—Extended C to add classes, creates C++ (or C with classes)
And many others…!
Now, back to the lowest level...
Binary: The native language of the processor

- Modern processors are very fast
- (m/b)illions of instructions per sec

Processors execute a small number of very basic instructions

MOV r1, r2  ADD r1, r2, r3
IFZERO r1, +20

These instructions written in a binary encoding (Why?)
Binary: The native language of the processor

- Modern processors are very fast
- (m/b)illions of instructions per sec

Processors execute a small number of very basic instructions

MOV r1, r2  ADD r1, r2, r3
IFZERO r1, +20

These instructions written in a binary encoding (Why?)

Compact representation  Quick to decode and execute
Thousands of different processors

Each speaks a different language

Called its *architecture*

Different versions of architecture add features, etc..
So I need to turn this into something my i7 speaks...
To do that, I use a compiler
“Compile a file named sumnums.cpp, and output an executable file named sumnums”

clang++ sumnums.cpp -o sumnums
“Compile a file named sumnums.cpp, and output an executable file named sumnums”

```
clang++ sumnums.cpp -o sumnums
```

(Ton of options here, especially for large projects with complex configs / multifiles)
```cpp
#include <iostream>
#include <string>
using namespace std;

int sum(const unsigned int number) {
    int i = number;
    int accumulator = number;
    while (i > 0) {
        accumulator += i;
        i--;
    }
    return accumulator;
}

// This program accepts 1 argument
int main(int argc, char *argv[]) {
    int number;

    if (argc < 2) {
        cerr << "This program needs at least one argument.\n";
        exit(1);
    }
    try {
        number = stoi(argv[1]);
    } catch (const invalid_argument &ia) {
        cerr << "Invalid argument: " << ia.what() << "\n";
        exit(1);
    }
    if (number < 0) {
        cerr << "This program expects a non-negative argument.\n";
        exit(1);
    }
    cout << "I am going to sum the numbers from 0 to " << argv[0] << "\n";
    cout << "Sum: " << sum(number) << "\n";
    return 0;
}
```
So, the compiler turns C++ into a giant list of these instructions...
So, the compiler turns C++ into a giant list of these instructions...

These are written in assembly
(Human-readable binary)
Let’s see what assembly the compiler generates...
clang++ -S sumnums.cpp

(Note I really used:
clang++ -S sumnums -fno-asynchronous-unwind-tables
This is because otherwise extra debugging overhead is inserted.)
Divided up by function

```
.globl __TEXT,.text,regular,pure_instructions
.macoser_version_min 10, 12
.p2align 4, 0x90

.Z3sumj:
    pushq  %rbp
    movq  %rsp, %rbp
    movl  %edi, -4(%rbp)
    movl  -4(%rbp), %edi
    movl  %edi, -8(%rbp)
    movl  -4(%rbp), %edi
    movl  %edi, -12(%rbp)
    cmp   $0, -8(%rbp)
    jle   LBB0_3

.LBB0_1:     # =>This Inner Loop Header: Depth=1
    movl  -8(%rbp), %eax
    addl  -12(%rbp), %eax
    movl  %eax, -12(%rbp)
    movl  -8(%rbp), %eax
    addl  $1, %eax
    movl  %eax, -8(%rbp)
    jmp   LBB0_1

.LBB0_3:
    movl  -12(%rbp), %eax
    popq  %rbp
    retq

.globl _main
.p2align 4, 0x90

.main:
```

Divided up by function

Implementation of sum
Don’t worry that this code is hard to understand for now. (It also confuses me..)
I can manually transform the assembly to the binary...

as sumnums.s
Crud...
Insight: my program needs a lot of other stuff to run...

This is kept in a *library*

(But keep in mind, that’s also **just code**. Nothing particularly magical)
Your code

\[ \text{Your code} \]

+ 

Executable file

\[ \text{Executable file} \]

= 

EXE

\[ \text{EXE} \]
Syllabus

http://kmicinski.com/cs245/syllabus/
A draft is freely available at:  isocpp.org/tour
Grade breakdown

50% : ~8 coding projects

10% : weekly labs

35% : Two midterms (~6 weeks in and ~12 weeks in)

5% : Participation (graded in various ways)
Autograder
Academic honesty
All submissions are graded using Clang 5, Racket 7, Python 3.7 on an Ubuntu 18.04 LTR server.

If you have any trouble configuring this (or a compatible environment) on your home machine, I highly recommend you develop with:

![lubuntu](image1)

![VirtualBox](image2)
What programming paradigms have you heard of?

See if you know (or know of) any that your neighbors don’t—or vice versa.
Programming languages: paradigms

- **Imperative languages** emphasize issuing commands that tell the machine what to do next at each step of evaluation.

- **Structured languages** emphasize structured control-flow (i.e., not goto) that can be properly nested, especially sequencing, conditionals, and looping constructs (while, for, do).

- **Procedural programming** is imperative programming with subroutines —emphasizes abstracting behaviors over data.

- **Object-oriented programming** emphasizes encapsulation of behaviors (methods) and data (fields) within classes, abstract modular schema for program values, that are instantiated as objects at run-time. Inheritance hierarchies used to promote code-reuse.

- **Reactive programming** emphasizes responding to events.
Programming languages: paradigms

- **Dynamic languages** emphasize permitting arbitrary manipulation of program values, control, and the environment at runtime. Primarily these use duck typing / structural typing. A related paradigm is that of **reflective programming**—dynamically modifying types at runtime.

- **Static languages** emphasize bounding program behavior ahead-of-time. Primarily these use nominal typing and are type-checked.

- **Array languages** emphasize concisely manipulating arrays, matrices.

- **Functional programming** emphasizes immutability, like math. Programs are constructed from pipelines of composed functions that transform inputs to outputs without affecting their environment.

- **Logic programming** emphasizes declarations, propositions, logical constraints. The programmer states what must be true of a solution.
Programming languages: imperative paradigm

Place first board and rails
While fence incomplete:
  move half-a-foot to the left
  position a new board
  position a nail
  hammer nail into top rail
...

function build_fence(len):
    if len == 1:
        return rails_and_first_picket()
    else:
        return add_one_picket(build_fence(len-1))
def fence.
fence is 5 ft tall.
fence has two rails.
fence has 50 pickets,
    each picket is 4" wide
    every picket is 2" from at least one other.
C++ is a superset of C with object-oriented features and generics/templates.

Focusing on classic/vanilla C++ written from scratch...

C++ is an example of the imperative, structured, procedural, static, and object-oriented language paradigms.
Introduction to C++ syntax and semantics

C++

The syntax of a language is the rules one must follow for a program to be parsed correctly. E.g., braces must match {}, identifiers begin with a character in [\_A-Za-z], semi-colons, etc.

The semantics of a language is the rules by which programs are run or evaluated to a result or behavior. E.g., operator precedence, order of operations, dynamic dispatch (which method is it), etc.
C++ syntax: comments

/* Multi-line or “C style” comments begin with a slash-star
 **********************************************
   ...and end with star-slash */

// Single-line or “C++ style” comments start with two slashes
// and end with a newline

/* Multi-line comments cannot be nested

   ...like this: /* */ // <-- this closes the whole comment

*/ // <-- this dangles
C++ syntax: identifiers, strings, numbers

(The basics are very similar to Java, as Java was designed to have C-like syntax.)

IDs match \[\_a-zA-Z][\_a-zA-Z0-9]*)\], and are not reserved keywords

Numbers can take a number of forms in C/C++... e.g.

- x
- _0123
- A_0
- a12

2.0, 2f

0xffffff00ff

30500ULL

Characters are between single-quotes: e.g., ’a’, ‘\n’

Strings are between double-quotes: “Hello World\n”

Strings in C/C++ are just arrays of chars: e.g., char[16]
**C++ syntax: reserved keywords**

<table>
<thead>
<tr>
<th>alignas</th>
<th>constexpr</th>
<th>inline</th>
<th>short</th>
<th>bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignof</td>
<td>const_cast</td>
<td>int</td>
<td>signed</td>
<td>char</td>
</tr>
<tr>
<td>and</td>
<td>continue</td>
<td>long</td>
<td>sizeof</td>
<td>char8_t</td>
</tr>
<tr>
<td>and_eq</td>
<td>decltype</td>
<td>mutable</td>
<td>static</td>
<td>char16_t</td>
</tr>
<tr>
<td>asm</td>
<td>default</td>
<td>namespace</td>
<td>static_assert</td>
<td>char32_t</td>
</tr>
<tr>
<td>auto(1)</td>
<td>delete</td>
<td>new</td>
<td>static_cast</td>
<td>class</td>
</tr>
<tr>
<td>bitand</td>
<td>do</td>
<td>noexcept</td>
<td>struct</td>
<td>compl</td>
</tr>
<tr>
<td>bitor</td>
<td>double</td>
<td>not</td>
<td>switch</td>
<td>catch</td>
</tr>
<tr>
<td>bool</td>
<td>dynamic_cast</td>
<td>not_eq</td>
<td>synchronized</td>
<td>char</td>
</tr>
<tr>
<td>break</td>
<td>else</td>
<td>nullptr</td>
<td>template</td>
<td>char</td>
</tr>
<tr>
<td>case</td>
<td>enum</td>
<td>operator</td>
<td>this</td>
<td>char</td>
</tr>
<tr>
<td>catch</td>
<td>explicit</td>
<td>or</td>
<td>thread_local</td>
<td>thread_local</td>
</tr>
<tr>
<td>char</td>
<td>export</td>
<td>or_eq</td>
<td>throw</td>
<td>char</td>
</tr>
<tr>
<td>char8_t</td>
<td>extern</td>
<td>private</td>
<td>true</td>
<td>char</td>
</tr>
<tr>
<td>char16_t</td>
<td>false</td>
<td>protected</td>
<td>try</td>
<td>virtual</td>
</tr>
<tr>
<td>char32_t</td>
<td>float</td>
<td>public</td>
<td>typedef</td>
<td>void</td>
</tr>
<tr>
<td>class(1)</td>
<td>for</td>
<td>reflexpr</td>
<td>typeid</td>
<td>volatile</td>
</tr>
<tr>
<td>compl</td>
<td>friend</td>
<td>register</td>
<td>typename</td>
<td>wchar_t</td>
</tr>
<tr>
<td>concept</td>
<td>goto</td>
<td>requires</td>
<td>union</td>
<td>while</td>
</tr>
<tr>
<td>const</td>
<td>if</td>
<td>return</td>
<td>unsigned</td>
<td>xor</td>
</tr>
<tr>
<td>consteval</td>
<td>import</td>
<td></td>
<td>using</td>
<td>xor_eq</td>
</tr>
</tbody>
</table>
C++ semantics: memory model

Each variable in C++ exists somewhere in memory

C++ thinks of this as a giant array of bytes
C++ semantics: memory model

Note: some lengths differ depending on architecture

- char: 1 byte
- u16: 2 bytes
- int: 4 bytes
- long long: 8 bytes

(for a 32-bit architecture...)

Note: some lengths differ depending on architecture.
C++ syntax: includes and macros

By convention, .cpp files are used for source, .h for libraries/declarations.

#include “path/to/file.h”

#include will textually replace this line with the entire contents of a file.

#include <library>

#define defines a macro: in this case, textually replace occurrences of “MAX” with “255”.

#define MAX 255
C++ syntax: anatomy of a function

The smallest valid C program.

```cpp
int main()
{
    return 0;
}
```
C++ syntax: anatomy of a function

The smallest valid C program.

main(...) is the entry-point of the program

```c
int main()
{
    return 0;
}
```

Returns status code 0, success.

All statements end with a semi-colon, as in Java.

In C/C++ the preferred style is for curly braces to line up on the same row or column.
As in Java though, whitespace only separates tokens and is not otherwise meaningful.
C++ syntax: anatomy of a function

“Hello World”

#include <iostream>

int main()
{
    std::cout << “Hello World”
               << std::endl;
    return 0;
}
Clang++: compiling and running

“Hello World”

```
$ clang++ -o hello hello.cpp
$ ls
hello hello.cpp
$ ./hello
Hello World
$
```

- `-g` compiles for debugging,
- `-std=c++14` compiles with C++14 features
- `-O2` compiles with optimization level 2
C++ syntax: arrays, dereferencing a pointer

An array (len=5) can be allocated on the stack using syntax `T a[5];`
or on the heap using syntax `T* a = new T[5];`

Using the prefix, unary operator * will explicitly dereference a pointer.
if `a` is of type `int*`, then `*a` is of type `int`.

```cpp
int main()
{
    int* iarr = new int[5];
    *iarr = 99;
    // is the same as
    iarr[0] = 99;
    // ... 
}
```
C++ syntax: structs

A custom type containing two publicly visible fields: x, and y.

```cpp
struct Point
{
    int x;
    int y;
};

int main()
{
    Point p;
    p.x = 5; // field access
    // ...
}
```
C++ syntax: new and delete

keyword “new” allocates an object on the heap, “delete” frees it

```cpp
struct Point
{
    int x;
    int y;
};

int main()
{
    Point* p = new Point();
    p->x = 5; // Same as (*p).x = 5
    delete p;
    //...
}
```
C++ syntax: pass by reference

Using `T&` in place of `T*` means the pointer itself cannot be manipulated and dereference is implicit! These are called *references*.

```cpp
bool x_gt_y(const P& p)
{
    return p.x > p.y;
}
```
C++ semantics: reading command-line arguments

Give main arguments argc and argv as below.

```c++
#include <iostream>

int main(int argc, const char** argv)
{
    if (argc <= 1) return 1; // failure
    std::cout << argv[1] << std::endl;
    return 0; // success
}
```
C++ semantics: pointers

A type T* means a pointer to something of type T.

For example, an int* is a word of memory containing the location, in memory, of an integer. An int** is an address pointing to a location containing an address to an integer.
C++ semantics: pointers

Pointers in C do not have lengths. You can read as many words or bytes at the location as you wish. Thus pointers are all really arrays of length 1 or greater.

A string in C is just an array of chars, or a char*

```
char* 0xefc11c40
```

```
<table>
<thead>
<tr>
<th>char</th>
<th>char</th>
<th>char</th>
<th>char</th>
<th>char</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘H’</td>
<td>‘e’</td>
<td>‘l’</td>
<td>‘l’</td>
<td>‘o’</td>
</tr>
</tbody>
</table>
```
C++ semantics: **const pointers**

A type may be preceded by keyword const, this tells the compiler to check that the value cannot be modified!

A const string in C is a `const char*`

If the pointer itself is also const, then it is a `const char* const`
Let’s try out some examples