Outline

Why learn C?

Programming Language Taxonomy

Programming Language Constructs

Syntax v. Semantics

Formal Syntax

Syntax Trees

Control Flow Diagrams
Programming Languages…
there’s so many

Top 12 most popular programming languages on sourceforge.net (05.09.2013)

1. Java
2. C++
3. PHP
4. C
5. C#
6. Python
7. JavaScript
8. Perl
9. Visual Basic
10. Delphi/Kylix
11. Assembly
12. Ruby
OK, so Java is the best.

Why are you teaching us the fourth best??
OK, so Java is the best.

Why are you teaching us the fourth best??

Actually, that is not how it works. Programming languages are tools. It is important to choose the best tool for the job at hand!
Why teach C?

- C is a relatively low-level programming language and has benefits in many engineering disciplines.
- A low-level programming language gives the programmer a lot of flexibility to tune for performance.
- A low-level programming language helps (forces?) the programmer to understand what the machine is doing in the background.
- Transitioning from a low-level programming language to a high-level programming language is (generally) easier than vice versa.

However, these benefits come at a cost. C generally does a bad job at protecting the programmer from him/her-self.
Outline

Why learn C?

Programming Language Taxonomy

Programming Language Constructs

Syntax v. Semantics

Formal Syntax

Syntax Trees

Control Flow Diagrams
Considering Programming Languages

There are many ways to break down programming languages, but we will consider them from two perspectives.

**Programming Paradigm**
- Procedural Programming
- Object-oriented Programming
- Functional Programming
- Logic Programming

**Execution Level**
- Compiled Languages
- Interpreted Languages
Procedural Programming
Focus on the method (i.e., *procedure*) of your program. Solve a problem by thinking about the plan of action. *To solve this problem, do this step first, then do this step, followed by this step...*

Object-oriented Programming
Focus on the data (i.e., *objects*) of your programs. Actions are completed with respect to data. Instead of considering a function that acts on an object, we consider the object and what it can do. *door->open v. open(door)*
Functional Programming
Focus on evaluating expressions (e.g., functions). Functions can be used as input and output with no global state that is manipulated.

\[
f(x) = 2(x + 1)
g(x) = 2x + 2
\]

Are \( f \) and \( g \) the same function? ...same procedure?

Logic Programming
Focus on formal first order logic to represent programs. Programs are written as a set of “rules” and queries can be run over these to output a solution.

Barret is a Bulldog
Bulldog is a Dog

Is Barret a Dog?
Compiled Languages
A program that is compiled is converted to a language that the machine
directly understands and can execute. This conversion is called
*compilation*.

Interpreted Languages
A program that cannot be executed directly on the machine, but is rather
interpreted and executed by another piece of software, is *interpreted*.
Some languages may be converted to another form, these are still
interpreted languages if the converted form cannot be directly executed by
the machine.
Outline

Why learn C?

Programming Language Taxonomy

Programming Language Constructs

Syntax v. Semantics

Formal Syntax

Syntax Trees

Control Flow Diagrams
While programming languages can widely differ, they (generally) share some constructs that are language independent (though syntax is not). This is due to the fact that most programming languages must, on some level, provide a mechanism for:

- representing data
- operating on data
- controlling the flow of a program
A *variable* is a place holder for some data. Hopefully, you are familiar with this construct from algebra.

\[ x = 2y \]

\( x \) and \( y \) are variables that are a place holder for some number.

In programming the concept is the same, let’s take a look at the above equation as a C assignment:

\[ x = 2 \times y; \]

\( x \) and \( y \) are variables that are a place holder for some number.

Easy, right!? 
A variable is a place holder for some data. Hopefully, you are familiar with this construct from algebra.

\[ x = 2y \]

\( x \) and \( y \) are variables that are a place holder for some number.

In programming the concept is the same, let’s take a look at the above equation as a C assignment:

\[ x = 2 \times y; \]

\( x \) and \( y \) are variables that are a place holder for some number.

Easy, right!? Well, it’s a tad more complicated, but not much…
In most programming languages, variables are typed. That is, the compiler or interpreter keeps track of what *kind* of variable we are talking about as to avoid confusion. In some cases, the programmer must explicitly tell the compiler or interpreter what type a variable is. Continuing with our example:

```c
int x;
int y;
x = 2 * y;
```

In the above example, we explicitly tell the compiler that `x` and `y` are integers.
Types of Types

For now we will consider 3 distinct types:

- **int** variables of this type represent integers
- **float** variables of this type represent real numbers
- **bool** variables of this type represent a boolean (true/false) value

The act of explicitly telling the compiler what type a variable is is referred to as a *declaration.*
Now that we have covered variables and their types, we may want to perform some operation on those variables. The simplest of these is an assignment. We use the equals sign (’=’) to denote assignment as in the example below.

```
//declarations
int x;
float y;
bool z;

//assignments
x = 5;
y = 3.14;
z = true;
```
Another simple operation is the arithmetic operation. This includes addition (’+’), subtraction (’-’), multiplication (’*’), division (’/’), and modulo (’%’).

```c
//declarations
int x;
int y;
int z;

//assignments with arithmetic operations
x = 8;
y = 2 + 2;
z = (x - y) / 4;
```
The modulo operation (’%’) is an operation you all know, though perhaps not as modulo.

Remember learning division in grade school? What is \( \frac{3}{2} \)?
The modulo operation (‘%’) is an operation you all know, though perhaps not as modulo.

Remember learning division in grade school? What is \( \frac{3}{2} \)?
1.5??
The modulo operation (’%’) is an operation you all know, though perhaps not as modulo.

Remember learning division in grade school? What is $\frac{3}{2}$? 1.5??

WRONG!
The modulo operation (’%’) is an operation you all know, though perhaps not as modulo.

Remember learning division in grade school? What is $\frac{3}{2}$? 1.5??

WRONG!

OK, technically its not wrong, but remember the days when the answer was “1 remainder 1”? That’s all the modulo operation is. It is the remainder after the division of two numbers. So, after the statement

```
x = 3 % 2;
```

is executed, x has been assigned the value of 1.
We haven’t yet considered the boolean type. Well, we can perform logical operations on boolean types as well. These include not ('!'), and ('&&'), and or ('||').

```
//declarations
bool x;
bool y;
bool z;

//assignments with logical operations
x = true;
y = x || false;
z = !y;
```
Operations

Relational

A relational operation is used to compare values. These include 'equal to' ('=='), 'not equal to' ('!='), 'greater than' ('>'), 'less than' ('<'), 'greater than or equal to' ('>='), and 'less than or equal to' ('<=').

These operations tend not to be used with assignments, but rather with control statements. We will consider control statements in detail in just a bit, but as an example...

```cpp
//declaration
int x;

//assignment
x=5;

//control statement with relational operator
if(x < 10){
    ...do something...
}
```
Control Statements

Branches

A branch statement allows the programmer to control the flow of the program by determining what code block should be executed next based on some condition. The most common of these is the if–else construct.

```plaintext
//control statement
if(x < 10){
    ...do something...
} else{
    ...do something else...
}
```
A loop statement allows the programmer to control the flow of the program by repeating a code block some number of times based on some condition. The most simple of these is the while loop.

```java
//while loop
while(x < 10){
    ...do something...
}
```
Outline

Why learn C?

Programming Language Taxonomy

Programming Language Constructs

Syntax v. Semantics

Formal Syntax

Syntax Trees

Control Flow Diagrams
Semantics is defined as *the meanings of words and phrases in a particular context.*

An English sentence has a *meaning*, this conveys *information* from the producer to the consumer.

Analogous to this, a program has a meaning. In computer science, semantics is the study of mathematically defining the meaning of a programming language.
Syntax

Syntax is defined as *the way in which linguistic elements (words) are put together to form constituents (phrases or clauses).*

An English sentence has a correct syntax. Generally a sentence consists of a *subject*, then a *verb*, followed by a *predicate*.

Analogous to this, a program must have correct syntax.
In computer science, the syntax of a programming language is strictly defined (more-so than in natural language).

This includes:

- The definition of keywords.
- Rules as to how tokens (e.g., variables) are to look. For example, is _##! a legal variable name?
- The grammar rules as to how the overall structure should look.
- The context constraints. (e.g., Must I declare a variable before I use it?)
Outline

Why learn C?

Programming Language Taxonomy

Programming Language Constructs

Syntax v. Semantics

Formal Syntax

Syntax Trees

Control Flow Diagrams
The first phase of checking syntax is performed by the *scanner*. Its responsibilities include:
identifying:
- reserved words (e.g., if, while, int, etc.)
- names (e.g., names of variables, functions, etc.)
- constants (e.g., numbers, strings, etc.)

and removing:
- white space
- comments

The second phase is performed by the *parser*. This phase analyzes the overall structure of the program.
We’ll learn by example...

Let us start by defining what letters and digits are...

\[
\text{letter} ::= \$|a|\ldots|z|A|\ldots|Z \\
\text{digit} ::= 0|\ldots|9
\]

**letter** and **digit** represent character sets. The ’|’ symbol separates possible characters (akin to “or”).
We can use these to define further constructs...

\[
\begin{align*}
\text{digit} & \ ::= \ 0|...|9 \\
\text{number} & \ ::= \ \text{digit} \ *
\end{align*}
\]

The ’*’ symbol means “repeated 0 or more times” and is with respect to the argument directly before it.

Does \text{number} define a real number? ...an integer? ...a positive integer?
Scanner

cont.

\[
\text{digit} ::= 0|...|9
\]

\[
\text{number} ::= \text{digit digit} \ast
\]

How about this? ...a real number? ...an integer? ...a positive integer?
Scanner
cont.

digit ::= 0|...|9
number ::= −?digit digit *

The '?' symbol means the preceding symbol is optional.

How about now? ...a real number? ...an integer? ...a positive integer?
Scanner

cont.

digit ::= 0|...|9

number ::= −?digit digit * (.digit digit *)?

Parenthesis (’(‘ and ’)’) can be used to group symbols.

And now? ...a real number? ...an integer? ...a positive integer?
Now let's define names (i.e., variable and function names). I can tell you that names must start with a letter and may contain any number of letters or numbers following the initial letter.
Scanner

cont.

\[
\text{letter} ::= \$ \mid a \ldots z \mid A \ldots Z
\]

\[
\text{digit} ::= 0 \ldots 9
\]

\[
\text{name} ::= \text{letter} (\text{letter} \mid \text{digit})^*\]

\text{name} is constructed from \text{letters} and \text{digits}.

So,

\_178 \ x \ this\_is\_a\_name

are legal names, while

2cool \ !! \ -42

are illegal names.
We have now effectively told the scanner what a **number** looks like and what a **name** looks like.

Expressions that are constructed with character (sets) and the following operations

- Alternation (`'|'`)
- Iteration (`'*'`)
- Option (`'?`)
- Concatenation

are called **regular expressions**
With the scanner, we are able to take the following:
\[2 \times (\text{var}_x + 25)\]

and identify it as:
\[\text{number} \times (\text{name} + \text{number})\]

However, is “\text{number} \times (\text{name} + \text{number})” proper program structure??

This is where the parser comes in.
The notation used to express the program syntax is referred to as *Extended Backus–Naur Form (EBNF)*.

EBNF can be used to express *context-free grammars (CFG)*.
Programs are hierarchical in structure

\[
\begin{align*}
  \text{unop} & ::= - \\
  \text{binop} & ::= -|+|*|/|% \\
  \text{expr} & ::= \text{number}\|\text{name}\|((\text{expr})| \\
  & ::= \text{unop}\ \text{expr}\|\text{expr}\ \text{binop}\ \text{expr}
\end{align*}
\]

Note: \(-, +, *, /, \) and \(\%\) are terminals (i.e., there are used in the programming language and will have no rules expanding them)
EBNF Example (cont.)

unop ::= -
binop ::= -|+|*|/|% 
expr ::= number | name | (expr) |
      ::= unop expr | expr binop expr

2 * (var x + 25)

scanner

number * (name + number)

parser

expr
Outline

Why learn C?

Programming Language Taxonomy

Programming Language Constructs

Syntax v. Semantics

Formal Syntax

Syntax Trees

Control Flow Diagrams
Such an expression can also be expressed as a syntax tree. This is a tree representation based on the EBNF definitions we defined.

\[
\begin{align*}
\text{unop} & ::= - \\
\text{binop} & ::= -|+|\ast|/|\% \\
\text{expr} & ::= \text{number}\ |	ext{name}\ |	ext{(expr)} \\
& ::= \text{unop }\text{expr}\ |\text{expr }\text{binop }\text{expr}
\end{align*}
\]

\[2 \ast (\text{var}_x + 25)\]
int x;
x = read();
if (x > 0)
  write(1);
else
  write(0);
Program EBNF

program ::= decl * stmt*
decl ::= type name(name)*;
type ::= int|float|bool
stmt ::= ;|{stmt*}|
      ::= name = expr; | name = read(); | write(expr);
      ::= if(cond)stmt
      ::= if(cond)stmt else stmt
      ::= while(cond)stmt
Program EBNF (cont.)

expr ::= number | name | (expr) |
::= unop expr | expr binop expr
unop ::= -
binop ::= - | + | * | / | %
cond ::= true | false | (cond) |
::= expr comp expr |
::= cunop cond | cond cbinop cond
comp ::= == | != | <= | < | >= | >
cunop ::= !
cbinop ::= && | ||
Outline

Why learn C?

Programming Language Taxonomy

Programming Language Constructs

Syntax v. Semantics

Formal Syntax

Syntax Trees

Control Flow Diagrams
To understand a program and what it does, it is important to understand the *control flow* of the program. The control flow simply describes the “path” the processor takes through the program during execution. This can be simplified through the use of a *control flow diagram*.

**Components of a Control Flow Diagram**

- **Start Node**
  - Start

- **End Node**
  - Stop
Control Flow Diagram Components

- **Input**: $x = \text{read}();$
- **Output**: $\text{write}(y);$
- **Expression**: $x = x - y;$

Prof. C. Eckert, Dr. J. Pföh

PL Intro - Control Flow Diagrams

Technische Universität München

Control Flow Diagram Components

Branch:

Join:
int x, y;
x = read();
y = read();
while (x != y)
   if (x < y)
      y = y - x;
   else
      x = x - y;
write(x);
Control Flow Diagram

Walk-through

Start

\[ x = \text{read}(); \]
\[ y = \text{read}(); \]

\[ x \neq y \]

no

write(x);

Stop

yes

\[ x < y \]

no

\[ x = x - y; \]

yes

\[ y = y - x; \]
Control Flow Diagram

Walk-through

Start

\[ x = \text{read}(); \]
\[ y = \text{read}(); \]

\[ x \neq y \]

- no
  - write(x);
  - Stop

- yes
  - x < y
    - no
      - x = x - y;
    - yes
      - y = y - x;

x == 18, y == 24
Control Flow Diagram

Walk-through

Start

\[x = \text{read}();\]
\[y = \text{read}();\]

x \neq y

\[x = x - y;\]
\[y = y - x;\]

Stop

x = 18, y = 24
Control Flow Diagram

Walk-through

Start

x = read();
y = read();

x != y

write(x);

x < y

x = x - y;
y = y - x;

Stop

x == 18, y == 24
Control Flow Diagram
Walk-through

Start

\[ x = \text{read}(); \]
\[ y = \text{read}(); \]

\[ x \neq y \]

- no: write(x);
- yes:

\[ x = x - y; \]
\[ y = y - x; \]

x == 18, y == 6

Stop
Control Flow Diagram

Walk-through

Start

\( x = \text{read}(); \)
\( y = \text{read}(); \)

\( x \neq y \)

x \neq y

Write(x);

Stop

\( x = x - y; \)

\( y = y - x; \)

x \leq y

x \leq 18, y \leq 6
Control Flow Diagram

Walk-through

Start

\[ x = \text{read}(); \]
\[ y = \text{read}(); \]

\[ x \neq y \]

\[ \text{write}(x); \]

\[ x = x - y; \]

\[ y = y - x; \]

\[ x == 12, y == 6 \]
Control Flow Diagram

Walk-through

Start

\[ x = \text{read}(); \]
\[ y = \text{read}(); \]

x \neq y

yes

x < y

no

write(x);

Stop

x = x - y;

y = y - x;

x = 12, y = 6

no

yes
Control Flow Diagram
Walk-through

Start

\[ x = \text{read}(); \]
\[ y = \text{read}(); \]

\[ x \neq y \]

- no: write(x);
- yes: x = x - y;

Stop

x == 6, y == 6

- no: x < y
- yes: y = y - x;
Control Flow Diagram
Walk-through

Start

\[ x = \text{read}(); \]
\[ y = \text{read}(); \]

\(x \neq y\)

\text{no}

write(x);

\text{Stop}

\text{yes}

\[ x = x - y; \]

\text{no}

\[ x < y \]

\text{yes}

\[ y = y - x; \]

\(x = 6, \ y = 6\)
Control Flow Diagram

Walk-through

Start

\[ x = \text{read}(); \]
\[ y = \text{read}(); \]

\[ x \neq y \]

no

\[ \text{write}(x); \]

Stop

yes

\[ x = x - y; \]

\[ x = x \]

\[ y = y - x; \]

\[ x \leq 6, y \leq 6 \]
Control Flow Diagram

Walk-through

Start

\[ x = \text{read}(); \]
\[ y = \text{read}(); \]

\[ x \neq y \]

no

write(x);

Stop

yes

\[ x < y \]

no

\[ x = x - y; \]

yes

\[ y = y - x; \]