Lecture 11: C syntax, types, semantics

David Hovemeyer

October 6, 2025

601.428/628 Compilers and Interpreters

Agenda

- Compiler project
- The C language, types
- C type representation
- C semantics

Compiler project

Compiler project

- Goal: implement a compiler for a substantial subset of C called "Nearly C"
  - https://github.com/daveho/nearly_c
- Target language: x86-64 assembly language
- The starter code gives you:
  - a working lexical analyzer (implemented using Flex)
  - a working AST-building parser (implemented using Bison)
  - Uses the `NodeBase` and `Node` classes from Assignments 1 and 2
  - You will work on the fun parts of the compiler

Assignments (likely)

- Assignment 3: Semantic analysis
- Assignment 4: Code generation
- Assignment 5: Better code generation

Why C?

- Still useful/relevant after $\approx$ 50 years
- Reasonably pleasant to write programs in
- *Relatively* straightforward to analyze and generate code for
  - Although: has some complexity

Why not C?

- Lack of type safety and memory safety
- Bugs and vulnerabilities due to undefined behavior
- Languages like Rust have fixed some of the fundamental issues with C for systems programming
- We'll still be using C, though

The C language

The C language

- You will need to be fairly knowledgeable about C syntax and semantics
- Today we will cover the essentials you will need to know for Assignment 3
  - Especially about C data types and the syntax of variable and function declaration/definition

Basic types

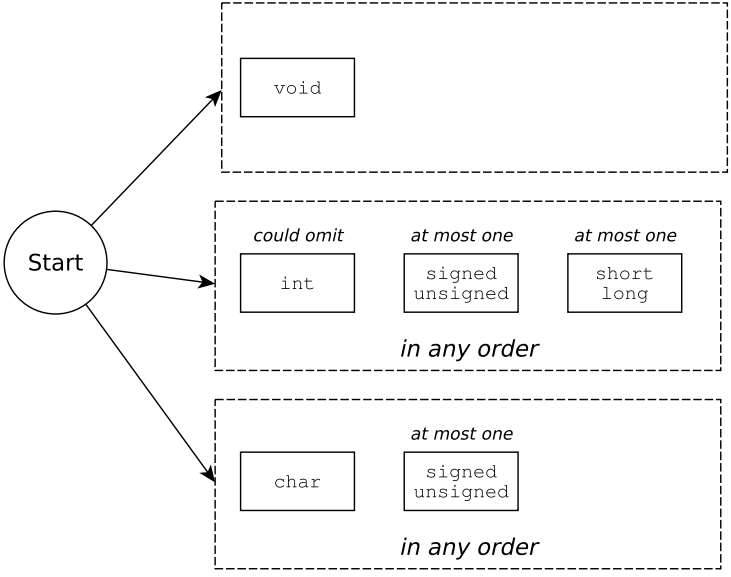
Basic types: `int`, `char`, `long`, etc. Also `void`.

- Any basic type (except `void`) can be `signed` or `unsigned`
- Technically, `short` is really `short int` and `long` is really `long int`
- If none of `char`/`int`/`void` are specified, `int` is assumed
  - So, `unsigned` by itself really means `unsigned int`
  - And this is also why `long` by itself really means `long int`
- Note that we won't be doing anything with floating point types or values, so forget about `float` and `double`

Type salad

- The basic type keywords can appear in any order!
- E.g., `int signed`, `unsigned int long`, etc.
- The compiler's semantic analyzer must make sense of whatever combination is specified
- Essential rules: in a basic type,
  - Only one of `char`, `int`, or `void` may be used
  - `void` cannot be combined with any other keywords
  - `long` and `short` can only be used with `int`
  - `long` and `short` are mutually exclusive
  - `signed` and `unsigned` are mutually exclusive

Basic type rules



Examples of basic types

- `int`
- `signed int`
- `unsigned int`
- `long unsigned`

Examples of invalid basic types

- `void int`
- `long short int`
- `long char`
- `signed unsigned int`
- `unsigned void`

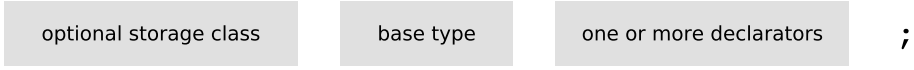
Declarators

C has a peculiar way of defining variables, known as “declarators”

A declarator specifies the name of a variable plus (potentially) part of its type

- Technically, functions are also defined using a syntax based on declarators, but we’ll ignore that for now

Variable declaration syntax



Base type is either a basic type or a struct type

- Full C would also allow a typedef name or a union type

Storage class is `auto`, `extern`, or `static`

- Full C would allow this to be mixed with the basic type keywords

Kinds of declarators

- *identifier*: a variable name
- `*` *declarator*: makes the type a pointer type
- *declarator* `[` *array size* `]`: makes the type an array type

Array modifiers have precedence over pointer modifiers. So, `int *p[10];` declares that `p` is an array of 10 pointers to `int` elements.

Examples

```
int a;          // a is a variable of type "int"

int *q;         // q is a variable of type "pointer to int"

char s[100];    // s is a variable of type "array of 100 char"

int *p[10];     // p is a variable of type "array of 10
                // pointers to int"

unsigned short **r[4][5];  // r is a variable of type "array
                           // of 4 arrays of 5 pointers
                           // to pointer to unsigned short"
```

List of declarators

A variable declaration can specify multiple declarators separated by commas. They share the base type.

Example:

```
int n, *p, a[10];

// n is a variable of type "int"
// p is a variable of type "pointer to int"
// a is a variable of type "array of 10 int"
```

Storage class

- `extern`: variable or function is in the global scope and is visible to other translation units; is the default for functions
- `static`: variable or function is visible only in this translation unit; for variables, lifetime of variable is the entire program execution (like a global variable)
- `auto`: storage is allocated in the current stack frame; is the default for local variables (and is seldom specified explicitly)

Function types

A function type includes

- the types of the parameter(s) (order is significant)
- the return type

Note that the parameter names are not really part of the type.

Function type examples

```
int add(int x, int y);  // add is a function of type
                        // "(int × int) → int"

void fill(char *s, char c, int n);
                      // fill is a function of type
                      // "(pointer to char × char × int) → void"
```

Struct types

A struct type consists of the types of its members (fields), in order.

The name of each member is also represented, since the compiler will need to know which field is being accessed when the `.` or `->` operators are used.

Struct type example

```
struct Player {
  int x, y;
  char symbol;
  short health;
};

// struct Player is
// (x, int) × (y, int) × (symbol, char) × (health, short)
```

Note the declarations of struct members are like normal variable declarations. The only difference is that they can't have a storage class.

Struct types in variable declarations

The name of a struct type can be specified as the base type in a variable declaration.

```
struct Player hero, *p, npcs[10];

// hero is a variable of type "struct Player"
// p is a variable of type "pointer to struct Player"
// npcs is a variable of type "array of 10 struct Player"
```

Type qualifiers

A type qualifier "qualifies" a type.

- `const`: if a variable's type is `const`, it can't be modified
- `volatile`: if a variable's type is `volatile`, the compiler must assume its value could change independently of the program's code

Note that `const` and `volatile` are not mutually exclusive.

Examples of qualified types

```
const int x;
  // x is a variable of type "const int"

volatile char *s;
  // s is a variable of type "pointer to volatile char"

const volatile int *done;
  // done is a variable of type "pointer to const volatile int"

const struct Player *p;
  // p is a variable of type "const struct Player"
```

C type representation

Type representation

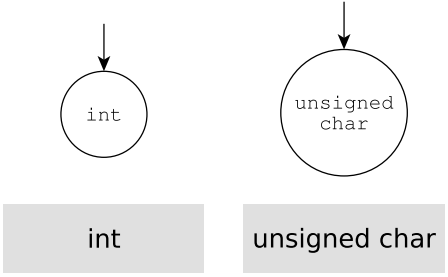
A C compiler will need to determine the type of each function, variable, and computed value.

A suitable *representation* mechanism is essential.

C types are recursive in nature, so trees are a natural representation.

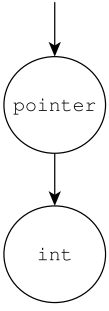
Basic types

Basic types are leaf nodes.

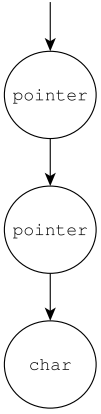


Pointer types

The child of a pointer type is the base type, i.e., the pointed-to type.



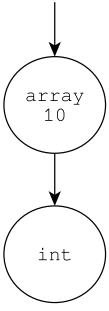
pointer to int



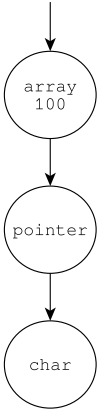
pointer to
pointer to char

Array types

Array type contains the size of the array, and the child is the element type.



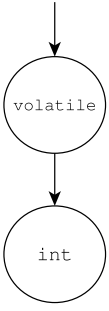
array of 10 int



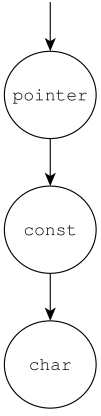
array of 100
pointer to char

Qualified types

Child of a type qualifier node is the type being qualified.



volatile int

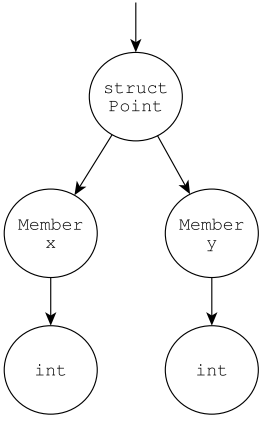


pointer to
const char

Struct type

A struct type indicates the name of the struct type, and the children are members (name and pointer to type)

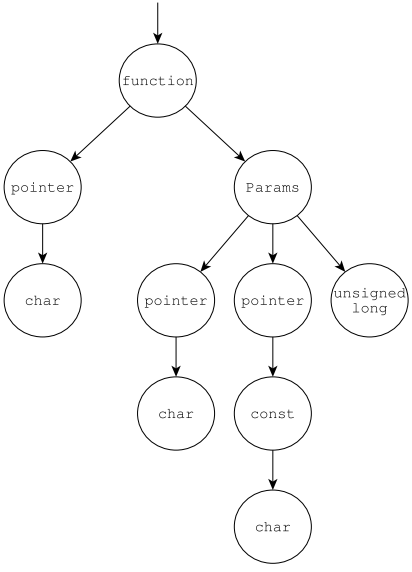
```
struct Point {
  int x, y;
};
```



Function type

A function type has children indicating return type and parameter types

```
char *strncpy(char *dest,
              const char *src,
              unsigned long n);
```



C semantics

Syntax vs. semantics

- The parser verifies that the input (sequence of tokens) can be derived from the language's syntax
- It doesn't verify that the input program is *semantically* valid
- The compiler's *semantic analyzer* checks the representation of the input source (usually, an AST):
  - Do uses of names refer to something?
  - Do the uses of variables, values, and functions conform to the language's type rules?

Semantic analysis $\rightarrow$ augmented AST

- The semantic analyzer will tyically *annotate* the input source representation: e.g.,
  - Annotate each variable reference with information about what variable it refers to
  - Annotate each expression with the type of value computed
- This information will be consumed by later phases of compilation (e.g., code generation)
- More on this next time

Overview of C semantics and type checking

What follows is a very brief overview of C semantic and type rules

lvalues and rvalues

- lvalue: a value that has a (potentially) assignable storage location
  - So-called because it is allowed on the left-hand side of an assignment
- rvalue: a computed value, cannot be assigned to
- A variable reference produces an lvalue
- Dereferencing a pointer produces an lvalue
- struct member references and array element references are lvalues
- Note that an lvalue's type could be qualified as `const`, in which case it can't be assigned to

Address-of and pointer dereference

- Address-of operator (`&`): when applied to an lvalue, yields a pointer which points to the lvalue's storage location in memory
  - Base type of pointer type is the lvalue's type
- Pointer dereference operator (`*`): when applied to a pointer value, yields an lvalue
  - Again, the lvalue's type is the pointer's base type

Pointer arithmetic

If `p` is a pointer pointing to an array element, and `i` is an integer, then `p + i` yields a pointer to the element `i` positions from the one that `p` points to

Note that `i` could be positive or negative: positive results in a pointer to a later element in the array, negative results in a pointer to a previous element in the array

Arrays and pointers

- Referring to an array is (generally) equivalent to referring to the address of the first element of the array
- The array subscript notation `a[i]` is simply an alternate syntax for `*(a + i)`

Struct member access

If `s` is an lvalue whose type is a struct type, then `s.x` is an lvalue referring to the member `x` of `s`

`p->x` is simply an alternate notation for `(*p).x`

Expressions

C has a large number of operators (we won't list them exhaustively here)

Generally, numeric types can be mixed freely in expressions

- E.g., you could compute the sum of a `char` value and an `int` value
- Generally, the value belonging to the less-precise type is "promoted" to the type of the value with the more-precise type
- When signed and unsigned values are mixed, the signed value becomes unsigned

What follows is an explanation of how expressions will handled in the compiler project (which might not exactly correspond to the rules for the actual C language)

Two-operand expressions

Steps for promotions and implicit conversions in binary (two-operand) expressions. (Note that this excludes left and right shifts.)

1. If either operand is has a type less precise than `int` or `unsigned int`, it is promoted to `int` or `unsigned int`
2. If one operand is less precise than the other, it is promoted to the more precise type
3. If the operands differ in signedness, the signed operand is implicitly converted to unsigned

Result type is `int` for relational and logical operators, otherwise is the type of the operands (which after steps 1–3 should be the same.)

Unary expressions

In a unary expression (`!`, `-`, `~`), the operand is promoted to `int` or `unsigned int` if necessary

The result of `!` (logical negation) is `int`, otherwise the result type is the same as the (possibly promoted) operand

Assignments

Assignment of a value belonging to any numeric type to an lvalue of any numeric type: allowed, may involve a promotion or truncation

Pointer assignments: left hand type must match right hand type exactly!

- Only exception: ok if left-hand side pointer base type is more qualified than right-hand pointer base type
- E.g., assignment of `char *` to `const char *` variable is allowed because `const char` is more qualified than `char`

Function calls

The arguments passed to a function call must have types which are legal to assign to the corresponding parameters

The result of a function call is the function's return type