Values, Operators, Variables

CSE 333 Winter 2021

Instructor: John Zahorjan

Teaching Assistants:

Matthew Arnold	Nonthakit Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

Preliminaries

- C is beautiful
- What does that mean?
  - A very few, pretty simple concepts are all it has/needs
  - It uses those simple concepts to enable “natural ways of writing things” – ways programmers were already accustomed to
- **Aside**: if simple is beautiful than complicated must be un-beautiful
  - **Special cases are complicated – avoid them**
  - If…then…else is complicated – avoid them
    - Especially nested ones
  - A method with multiple return statements is complicated
  - Scattering declarations throughout your code is complicated – put them at the top

Preliminaries

- Complex vs. Non-Complex
  - The more you have to know about the rest of the code and the dynamic state of the execution to understand that some statement is correct, the worse the code

- C provides mechanisms
- It relies on conventions
  - A trivial example: "every" main() checks the number of arguments and calls a function whose name is usage() if something looks wrong
  - It's not part of the language; not built into compilers; not enforced in any way
  - When you read someone else's code and you see a call to usage(), you know what it's doing

- C mechanisms don't make it impossible to write correct code
- Most of the time, if you follow the conventions, the code looks pretty "normal"

Expressions

- An expression is something that can be evaluated to produce a value
  - 6 + 10 / 3
    - 10 / 3 evaluates to 3 => 6 + 3 => 9
- A literal is a simple value, known at coding time
  - 6
  - ‘c’
- How did I decide that 10 / 3 evaluated to 3?
  - I did integer division. Why?
  - It isn’t the symbol ‘/’ telling me to do integer division, it’s the type of the operands to the ‘/’ operation

Operators / Functions

- An expression can be just a literal
- Expressions often involve operators/functions:
  - 6 + 10 / 3
  - 6 + gcd(98, 32)
- Functions and operators are very similar, even if the syntax looks different
- Both take inputs of certain type(s) and "return" a value of a certain type
- In C, <u>the type returned can be determined at compile time</u>
- This is all that type checking means in C…

Beautiful or Bizarre? Assignment Statements

- C doesn't have assignment statements
  - It has an assignment operator
  - It's very low precedence
- x = y + 2;
  - Normal; looks like "an assignment statement"
- x = y = 2 + z;
  - Totally legal
- 2+z;
  - Totally legal (but the compiler might helpfully issue a warning)

What Does ‘=‘ Mean in C?

- Assignment is about copying the contents of memory from someplace to someplace else
  - (Technical detail: sometimes “memory” is a register)
- Let’s consider the general form: lhs = expression;
- The lhs has to be “an lvalue” – something that can be assigned to
  - X // a simple variable
  - Array[4] // this means what you think, but is a pointer dereference (later…)
- The lhs has a type, and the type has a size
  - That’s how many bytes are going to be copied

Type Conversion

- The expression on the right-hand-side (rhs) is evaluated
  - The result is a value of some type
  - That type has a size
  - That type may not be the same as the type of the lhs
- IF the lhs and the rhs have the same type, then they're of the same size and C generates code to copy bytes
- If the lhs and rhs are of different types, there are two possibilities
  1. C converts the type of the rhs quantity to the type of the lhs
  2. You get a compile time error
- C **really** doesn't like to issue compile time errors…

Examples

- int x = 20;
  int y = x * 2;
  - 4 bytes are allocated on stack for x, and 4 for y
  - Code is generated to move the small integer 20 to the 4 bytes named x
  - Code is generated to fetch the four bytes named x into a register, shift the register left one bit (multiply by 2), and then copy the 4 bytes in the register to the four bytes named y

Examples

- int x = 20;
  float y = x * 2;
  - What happens with x is as before
  - C understands that simply copying the bits that result from x*2 into y wouldn't be a good idea, even though y is 4 bytes
  - So, C does an implicit cast, generating code that converts the integer result of x*2 to its float representation
    - And then copies the four bytes of that float to y
- int x = '0';
  - C generates code that extends the 8 bits representing '0' to 32 bits, then moves them into the 4 bytes named x

Assignment and Implicit Conversions

- The rules about implicit conversion are complicated
- There are lots of them
- Mostly, they "just work"…
  - You get what you expect, mostly
- I'm not going to try to go over them in class
- It is hard to generate an example of an assignment to an int variable that causes a compile time error
- It's easy to generate examples that don't do what you probably think, though

Summary So Far

- lhs = rhs
  - The lhs names some memory where values will be written
  - The rhs identifies a value
- x = x
  - The ‘x’ on the lhs means the address that x represents
  - The ‘x’ on the rhs means the last value assigned to x
- This is ALWAYS what assignment means in C, even when it may not seem like it
  - And even when you want it to mean something different
- (Aside: passing an argument to a function is assignment)

Array in C

- Originally, C didn't have arrays
  - But it has always had syntax that looked just like arrays
    - E.g., x[4] = x[5] + 1;
- These days the compiler knows something about arrays
  - E.g., you probably used sizeof() on a thing we'd call an array in ex01

Arrays in General

- An array is a data structure whose keys are consecutive non-negative integers and that can perform lookup in constant time
- The implementation requires as many consecutive bytes in memory as the total size of the array
  - int example[100]; // requires 400 consecutive bytes
- The consecutive bytes allow constant time lookup
  - array[n] is located at
    - The starting address of the elements of array plus
    - n * the number of bytes required for each element
  - Example: array[10] is at starting address of array + 40 if array holds ints

C and Arrays

- C has something that works just like an array if you use it without error
- If your code has errors, though, what it does is undefined
- If you write array[n] = 0, C will generate code that assigns 0 to the four bytes in memory at address “starting address of array + n*4” (assuming array hold ints)
- What if n == -3?
  - “undefined”
  - In practice, you’ll be operating on the bytes at offset -12 from the start of the array
  - Those bytes are likely some other variable in your program

C and Arrays

- Sure, C *could* check that n was in bounds
  - It can't check at compile time, though
  - It would have to generate code to check during execution
  - But that would slow down every C program, including those that didn't contain any array bound errors
- So… you write code yourself to check array bounds if you're worried your code isn't right
  - Don't make every program pay the penalty
- There are, of course, libraries that will provide an array bounds checked array
  - And you can just write code yourself, maybe more simply

Okay, So Arrays in C Are Hunks of Memory

- int a[10]; // 40 bytes
  char c[10]; // 10 bytes
- How do I access elements?
  - a[2] => generate code that takes the starting address of a and adds 2*4 to it, and the four bytes at that location are what you want
- C: Generalize and simplify => Pointers

Pointers

- C needs the following things
  - Address computations that support arrays / array indexing
  - A data type that can store the address returned by malloc(), so that programs can dynamically allocate space
  - A way for a variable to serve as a reference to another variable, like we need when building linked lists and other linked data structures
- The unifying concept for these things in C is the pointer
- Pointers hold memory addresses
- You can put any address you want into a pointer variable, but you'd be crazy to do so
  - Unless you're an operating system, maybe

Pointers Explained

- int *pInt; // "pInt" is an 8-byte variable that can
  // store a memory address
- pInt = 2; // Set the address in pInt to 2 (never do this)
- *pInt = 2; // Write the four bytes at the address given
  // by the 8 bytes of pInt with the value 2

- **The pointer is 8 bytes**
- **The thing at the address it contains is presumed to be as long as the type it was declared to point at**

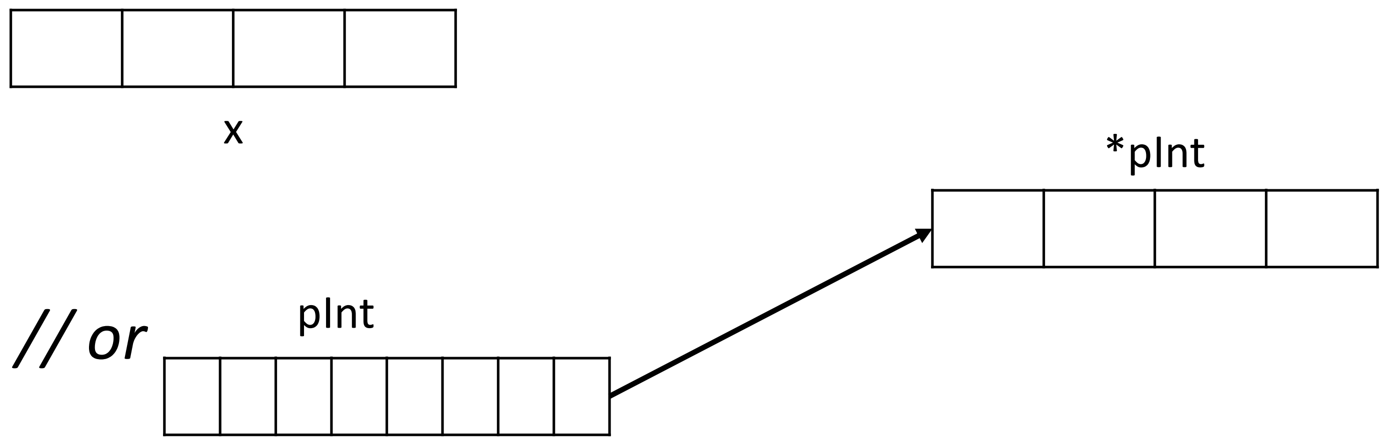
The Dereference Operator, *

- On the lhs:
  - *pInt = 2;
  - The lhs is going to evaluate to an address to write
    - 'pInt' evaluates to the address of the 8 bytes named pInt
    - '*pInt' takes the 8 bytes stored in bytes named pInt and that value becomes the effective address

- On the rhs:
  - pOther = pInt;
    - The value of pInt on the rhs is the contents of the 8 bytes named pInt
  - y = *pInt;
    - *pInt means get an address from the 8 bytes pInt, then go to that address and get four bytes

Visually

- int x;
- int *pInt; *// or int* pInt;*



Setting Pointers Sanely

- When you dynamically allocate something
  - struct node *pNode = (struct node*)malloc(sizeof(node));

- When you want to create an alias for some existing variable
  - int x;
    int *pInt = &x; // "address of" operator

Array Names

- int array[10];
  - The symbol “array” behaves like a pointer literal
    - Its value is the starting address of the 40 bytes of the array
    - Its value is stored by the compiler during compilation
    - There is no memory allocated to store its value at run time
  - You can say array[2] = 0, meaning something like *(0x7ffe0354ce5c + 2*4) = 0
- In contrast:
  - ```
    int *p;        // this allocates 8 bytes named p, not an array
    p[2] = 0;      // a terrible mistake, as you're writing over memory
                   // that holds some other variable (probably) because
                   // p is not initialized
    ```
    ```

When Things Aren't What A Java Programmer Thinks

- C's pointer syntax makes it easy to forget it doesn't really have arrays

- ```
  int x[] = {1,2,3};
  int y[] = {100, 101, 102};
  x = y;
  printf("%d\n", x[0]);
  ```

- What happens?
  - Compile time error
  - Prints 100
  - Prints some crazy number
  ```

Second Try

- ```
  int x[] = {1,2,3};
  int y[] = {100, 101, 102};
  x[0] = y;
  printf("%d\n", x[0]);
  ```

- What happens?
  - Compile time error
  - Prints 100
  - Prints some crazy number
  ```

Second Try

- ```
  int x[] = {1,2,3};
  int y[] = {100, 101, 102};
  x[0] = y;
  printf("%d\n", x[0]);
  ```

- What happens?
  - Compile time error
  - Prints 100
  - Prints some crazy number

```
test.c: In function 'main':
test.c:6:8: warning: assignment to 'int' from 'int *' makes integer from pointer without a cast [-Wint-conversion]
   6 |   x[0] = y;
     |        ^
[attu7] ~/tmp> ./a.out
867687208
```
```

Multi-dimensional Arrays

- Generic 2D format:

  `type name[rows][cols] = {{values},…,{values}};`

  - Still allocates a single, contiguous chunk of memory
  - C is *row-major*

```
// a 2-row, 3-column array of doubles
double grid[2][3];

// a 3-row, 5-column array of ints
int matrix[3][5] = {
  {0, 1, 2, 3, 4},
  {0, 2, 4, 6, 8},
  {1, 3, 5, 7, 9}
};
```

  - What is the address computation corresponding to matrix[2][3]?

C Parameter Passing

- **All assignment copies bytes**
- **Parameter passing is assignment**
  - Assign the value of the arguments to the local variables that are the parameters (names used in the function)
- **Parameter passing is "by value"**
  - The argument is some expression, .e.g., x or x+y or ptr or *ptr or intArray
- **ALWAYS**

Parameter Passing Examples

- int x = 12;
  int *pInt = &x;
  int intArray[] = {0, 1, -2, 3, -4, 5, -6, 7, -8, 9};

Function prototype	Call	Value of y in Function

int sub(int y);	sub(x);
	sub(pInt);
	sub(intArray);
		12
		<the address of caller's x> as an int
		<the address of caller's intArray[0]>

int sub(int *y)	sub(x);
	sub(pInt);
	
	sub(intArray);
		*y is the four byte int at address 12
		*y is the caller's x
		y[3] is a runtime bug!
		*y is the caller's intArray[0]
		y[0]…y[n] are the caller's intArray

int sub(int y[])	sub(x)	y[0] is the four byte int at address 12 y[0] is the caller's x; y[1] is an error y[0]…y[n] are the caller's intArray

Arrays as Parameters

- You **cannot** pass an array as a parameter
- You can pass the starting address of the array
- The function's parameter type determines the size of the element(s) the parameter points at

- If you want to create a function "that operates on an array" you have to supply
  - The array's starting address
  - The array's length
- void zeroArray(int *array, int size); // or
  void zeroArray(int array[], int size);

Warning

- In Java an array name basically names the elements of the array
- In C, the array name is an address, not an array
  - It's the [] operator (as in intArray[3] or pInt[n]) that "makes it an array"
    - Even if it isn't…

- Except that int exampleArray[100]; allocates space for 100 elements and no space for the symbol "exampleArray"
- And int *pIntArray allocates 8 bytes to hold a pointer, but no space for the array the programmer presumably intends pIntArray will point at

Another Use of Pointers

- A method can return only one value
- What if you want to return more than one value?
  - For example, you want to return a success/failure indicator AND some result computed when successful
- You could return a struct that contained fields for both
  - That's not typically done
- What is typical is to
  - Return success/failure as the return value
  - Return the other value(s) through an output parameter(s)
- Why does it matter that the success/indicator is the actual return value?

Output Parameters

```
int max(int val_array[], int size, int *result) {
  if ( val_array == NULL ) return 1;
  if ( result != NULL ) {
    *result = val_array[0];
    for ( int i=1; i<size; i++ )
      if ( val_array[i] > *result ) *result = val_array[i];
  }
  return 0;
}

int main() {
  int vals[] = {1, -2, 3, 17, 10, 29, -4};
  int result;
  if ( !max(vals, sizeof(vals)/sizeof(int), &result) )
       printf("%d\n", result);
  else printf("Call to max() failed\n");
  return 0;
}
```