Operating Systems Exercise 1

Georgios Georgiadis Negin F.Nejad

Prelude

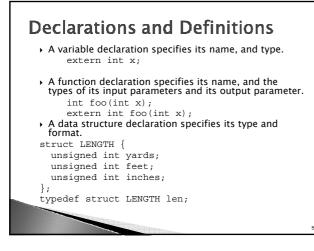
- We do **not** teach programming
- Take a course
- We do <u>not</u> teach C
- Read a book
- The C Programming Language, Kernighan, Richie
- This is a brief tutorial on C's traps and pitfalls
- $^\circ$ For those that already know C programming
- "C Traps and Pitfalls", Andrew Koenig, Addison– Wesley 1989 (link at course's webpage)

Overview

- Declarations and Definitions
- Memory Allocation
- Pointers and Arrays
- Lexical Pitfalls
- Syntactic Pitfalls
- Semantic Pitfalls

Declarations and Definitions

- We can declare something without defining it • But we cannot define it without declaring it.
- The confusing part is that the definition will repeat the declaration specifications.



Declarations and Definitions

• A function definition specifies the exact sequence of operations to execute when it is called.

int foo(int x) {return 1};

• A data structure definition will reserve space in memory for it.

len length;

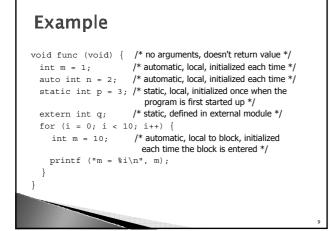
Memory Allocation

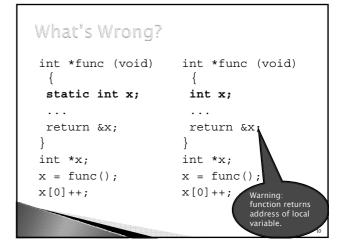
- Static/global allocation

 Each static or global variable defines one block of space, of a fixed size.
 - The space is allocated once, when your program is started (part of the $\exp c$ operation), and is never freed.
- Automatic allocation
 - Such as a function argument or a local variable. The space for an automatic variable is allocated when the compound statement containing the declaration is entered, and is freed when that compound statement is exited.
 - $\,\circ\,$ The size of the automatic storage should be a constant.
- Dynamic Memory Allocation not covered.

Example #include ...

- int i; /* i is static, and visible to the entire program */
- extern j; $/\star$ j is static, and visible to the entire program */
- static int k; /* k is static, and visible to the routines in this source file */





Overview

- Declarations and Definitions
- Memory Allocation
- Pointer and Arrays
- Lexical Pitfalls
- Syntactic Pitfalls
- Semantic Pitfalls

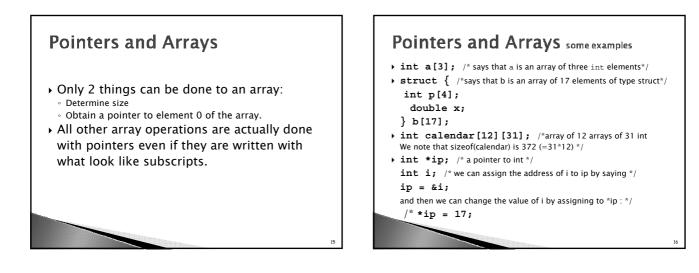
Pointers and Arrays

- The C notions of pointers and arrays are inseparably joined
- C has only one dimensional arrays, and the size of an array must be fixed as a constant in compilation time.
- However, an element of an array may be an object of any type.

Pointers and Arrays

```
> Arrays
int manyNumbers[3];
int manyNumbers[3]={1,2,3};
int mult[2][2] = { {1,2}, {3,4} };
> Strings
char name[20];
char address[] = "a long"
strcpy(address, "Chalmers");
if (strcmp(address, "Chalmers") == 0)
{ ... }
```

Pointers Example void swap(int *t1, int *t2) { int tmp; tmp = *t1; *t1 = *t2; *t2 = tmp; }



Pointers Arithmetic

- If a pointer happens to point to an element of an array, we can add/subtract an integer to that pointer to obtain a pointer to the next element of that array.
 But very different from integer arithmetic!
- ip+1 does <u>NOT</u> point to the next memory location.If we have written
- int *q = p + i;

then we should be able to obtain i from writing q-p.

There is no way to guarantee even that the distance between p and q is an integral multiple of an array element!

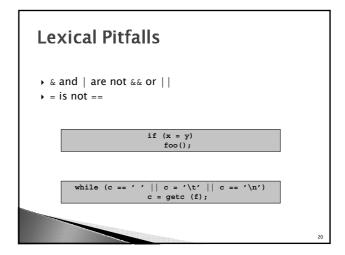
Pointers Arithmetic

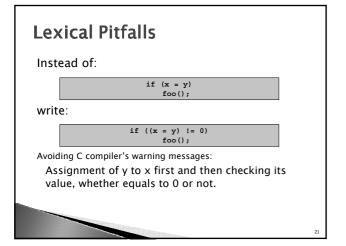
int a[3];

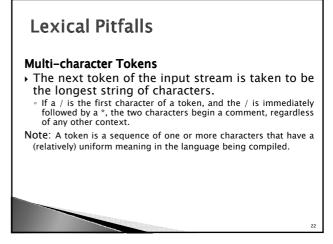
- p=a;
 // a pointer to the first element of the array
 p=&a;
- // wrong! A pointer to an array assign to a pointer to int
- > Does sizeof(p) equal to the sizeof(a)?
- *a = 84; sets the element 0 to 84
- * (a+i) is no different a[i]
- > Since a+i equals i+a then a[i] and [i] a is the same. Also, calendar[4][7] <=> *(calendar[4]+7) <=> *(*(calendar+4) +7)

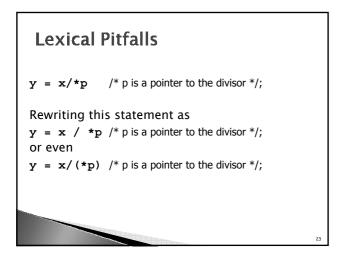
Overview

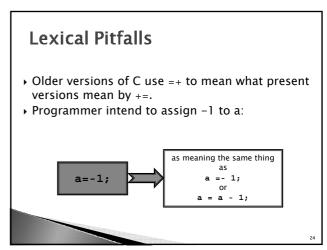
- Declarations and Definitions
- Memory Allocation
- Pointer and Arrays
- Lexical Pitfalls
- Syntactic Pitfalls
- Semantic Pitfalls











Strings and Characters are Different!

- Single and double quotes mean very things in C language.
 - A character enclosed in single quotes is just another way of writing an integer.
 - The integer that corresponds to the given character in the implementation's collating sequence.
 - Thus, in an ASCII implementation, 'a' means exactly the same thing as 0141 or 97.

Strings and Characters are Different!

- A string enclosed in double quotes, is a short-hand way of writing a pointer to a nameless array.
 This array will be initialized with the characters
- between the quotes and an extra character whose binary value is zero.

printf ("Hello world\n"); Same as char hello[] = {'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\n', 0}; printf (hello);

Strings and Characters are Different!

Saying
printf('\n');
instead of
printf ("\n");

Is not the same

• Using a pointer instead of an integer (or vice versa) will often cause a warning message.

Strings and Characters are Different!

- Writing 'yes' instead of "yes" is not the same!
- "yes" means "the address of the first of four consecutive memory locations containing y, e, s, and a null character, respectively."
- 'yes' means "an integer that is composed of the values of the characters y, e, and s."

Strings and Characters are Different!

What are the following in C?

- `0'
- ▶ "0"
- ▶ 0
- ▶ NULL
- `\\0'

Strings and Characters are Different!

What are the following in C?

- `0' an integer value of a character
- "0" a string the encodes zero
- 0 the integer 0
- NULL (#define NULL ((void *)0))
- \blacktriangleright '\0' the first character of ASCII table, NULL

Overview

- Declarations and Definitions
- Memory Allocation
- Pointer and Arrays
- Lexical Pitfalls
- Syntactic Pitfalls
- Semantic Pitfalls

Syntactic Pitfalls - Type Cast

Is the following piece of code correct?

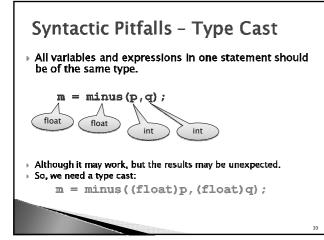
```
float minus(float a, float b){return a-b;}
```

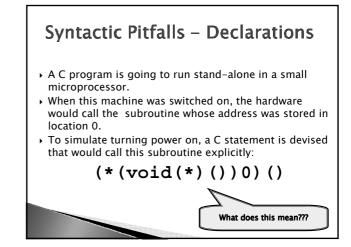
… int p,q; float m;

p = 1;

q = 2;

m = minus(p,q);





Syntactic Pitfalls - Declarations

- > Using a typedef declaration, we could have solved the problem more clearly: typedef void (*funcptr)(); (* (funcptr) 0)();
- But imagine we couldn't use typedef

Syntactic Pitfalls - Declarations

- > float f, g; The expressions f and g, when evaluated, will be of type float.
- > Parentheses may be used freely: float ((f)); means that ((f)) evaluates to a float and therefore, by inference, that f is also a float.
- Similar logic applies to function and pointer types. float ff(); means that the expression ff() is a float, and therefore that ff is a function that returns a float.

Syntactic Pitfalls - Declarations

float *pf;

*pf is a float and therefore pf is a pointer to a float.
> float *g() (*h) ();

Says *g() and (*h) () are float expressions.

note:

- $^\circ~{\boldsymbol{g}}$ is a function that returns a pointer to a float.
- $^\circ\,\,{\rm \textbf{h}}$ is a pointer to a function that returns a <code>float</code>.

Syntactic Pitfalls – Declarations

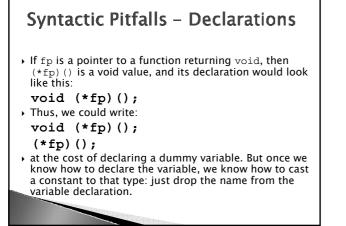
- Knowing variable declaration allows us to write a cast for that type
- Remove the *variable name* and the *semicolon* from the declaration and enclose the whole thing in parentheses.
- > float *g(); declares g to be a function
 returning a pointer to a float
- (float *()) is a cast to this type.

Syntactic Pitfalls - Declarations

- Suppose that we have a variable fp that contains a function pointer and we want to call the function to which fp points.
 - (*fp)();
- > If ${\tt fp}$ is a pointer to a function, ${\tt *fp}$ is the function itself, so $({\tt *fp})$ () is the way to invoke it.
- The parentheses in (*fp) are essential
- The expression would otherwise be interpreted as * (fp()).
 We have now reduced the problem to that of finding an appropriate expression to replace fp.

Syntactic Pitfalls - Declarations

- If C could read our mind about types, we could write:
 - (*0)();
- This doesn't work because the * operator insists on having a pointer as its operand.
- Furthermore, the operand must be a pointer to a function so that the result of * can be called.
- Thus, we need to cast 0 into a type described as "pointer to function returning void."



Syntactic Pitfalls – Declarations

- Thus, we cast 0 to a "pointer to function returning void" by saying: (void(*)())0
- > and we can now replace fp by (void(*)())0: (*(void(*)())0)();
- The semicolon on the end turns the expression into a statement.

Syntactic Pitfalls - Declarations

We are now ready to think what does the following expression means

Overview

- Declarations and Definitions
- Memory Allocation
- Pointer and Arrays
- Lexical Pitfalls
- Syntactic Pitfalls
- Semantic Pitfalls

Operators Precedence

- Constant FLAG is an integer with exactly one bit turned on in its binary representation (in other words, a power of two),
 We want to test whether the integer variable flags has that bit
- turned on.
- if (flags & FLAG) ...
- /* if statement tests whether the expression in the parentheses evaluates to 0 or not. */
- More explicit if statement: if (flags & FLAG != 0) ...
- The statement is now easier to understand, however it is wrong!! because != binds more tightly than &, so the interpretation is now: if (flags & (FLAG != 0)) ...

Operators Precedence

- We have two integer variables, h and 1, whose values are between 0 and 15,
- We want to set r to an 8-bit value whose low-order bits are those of 1 and whose high-order bits are those of h.
 r = h<<4 + 1;
- Unfortunately, this is wrong.
- > Addition binds more tightly than shifting
 r = h << (4 + 1);</pre>
- Here are two ways to get it right: r = (h << 4) + 1; r = h << 4 | 1;</pre>

Operators Precedence

- To avoid these problems
 - Parenthesize everything
 - Problem! expressions with too many parentheses are hard to understand.
 - Try to remember the precedence levels in C!
 - $\circ\,$ Unfortunately, there are fifteen of them, so this is not always easy to do.
 - Classify operators into groups; subscripting, function calls, unary operators, etc.
 - The C Programming Language, Kernighan, Richie

Watch Those Semicolons!

- if (x[i] > big);
- big = x[i]; The semicolon on the first line will not upset the compiler, but the code fragment means something quite different from: if (x[i] > big) big = x[i];
- The first one is equivalent to: if (x[i] > big) { }
- big = x[i];
- which is, of course, equivalent to:
- **big** = x[i]; (unless x, i, or big is a macro with side effects).

Watch Those Semicolons!

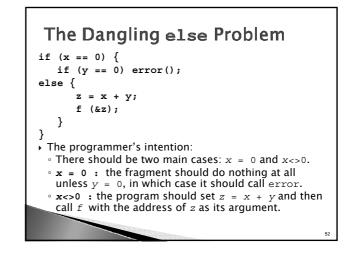
>Forgotten semicolons!
struct foo {
 int x;
}
f()
{
 ...
}
>Semicolon missing between the first } and f
>The effect of this is to declare that the function f
returns a struct foo, which is defined as part of this
declaration.
>If the semicolon were present, f would be defined by
default as returning an integer.

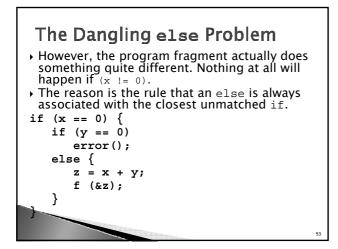
The Switch Statement

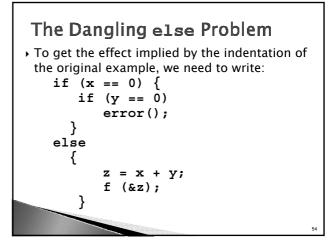
switch (color) {
case 1: printf ("red");
case 2: printf ("yellow");
case 3: printf ("blue");
}
Labels in C behave as true labels. Control can flow
through a case label.
redyellowblue, yellowblue, blue

The Switch Statement

```
switch (color) {
case 1: printf ("red");
break;
case 2: printf ("yellow");
break;
case 3: printf ("blue");
break;
}
red, yellow, blue
```







Conclusion • We discussed: • Declarations and Definitions webpage) Memory Allocation • Pointer and Arrays • Lexical Pitfalls Syntactic Pitfalls • Semantic Pitfalls • Only some of C's pitfalls were discussed here due to time constraints. Now it's your turn!

Conclusion

- Study "*C Traps and Pitfalls*", Andrew Koenig, Addison–Wesley 1989 (link at course's
- Contains many more pitfalls!
- If still uncertain, study "The C Programming Language", Dennis M. Richie, Brian W. Kernighan
- In conjunction with some C code.
- Google Code Search is a nice tool.
- Look for more tutorial at course's webpage.