MORE ABOUT VARIABLES AND OPERATORS

CS 23200

Outline

- Variables
  - Constants and their types
  - Type casting
  - Double/float calculation
  - Automatic and external variables
  - Scope
  - Initialization
  - #define directive
  - enums
- Operators miscellany

Constants and Their Types

- 1234 :
  - int
- 1234L :
  - long
- 1234.0 :
  - double
- 1234.0F :
  - float
- 1234.0L :
  - long double

- 'x' :
  - an int (not a character!)

ASCII Codes

Dec	Glyph
...	...
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
...	...

Dec	Glyph
...	...
65	A
66	B
67	C
...	...
90	Z
...	...
97	a
98	b
99	c
...	...

Digits are contiguous

Capital letters are contiguous

Lower-case letters are contiguous

Character Constants as ints

```
char c;
if(c >= '0' && c <= '9')
```

- Tests if c is a digit character

```
char c;
if((c >= 'A' && c <= 'Z') ||
   (c >= 'a' && c <= 'z'))
```

- Tests if c is a letter (upper or lower case)
- Should typically use standard library functions for these sorts of things, since they don't rely upon a particular character encoding

Character Constants as ints

```
char c;
if(c >= 'A' && c <= 'Z')
{
      c = c - 'A' + 'a';
}
/* changes upper-case letters to lower-case */
```

```
char c;
int x;
if(c >= '0' && c <= '9'){
      /* set x to the value given by digit c */
      x = _____;
}
x = c - '0';
```

'3' != 3

```
printf("Enter a number: ");

int theNum = 0;
int c;
while(!feof(stdin)){
   int digitVal;
   c = getchar();
   if(! isdigit(c) ){
      break;
   }
   digitVal = c - '0'; /* this translation is CRUCIAL */
   theNum = theNum * 10 + digitVal;
}
```

Same principles apply to characters read from files

Constants and Their Types

- 0x or 0X prefix : hexadecimal
  - Base 16
  - Digits are 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f
  - 2 hex digits == 1 byte
  - 0xff == 255
  - 0xa3 == 163
- 0 prefix : octal
  - Base 8
  - 03 == 3
  - 013 == 11 (not 13!)

Outline

- Variables
  - Constants and their types
  - Type casting
  - Double/float calculation
  - Automatic and external variables
  - Scope
  - Initialization
  - #define directive
  - enums
- Operators miscellany

Automatic Type Conversions

- Examples:
  - double x = 3;
  - 4.0 / 9
- Rules:
  - If either operand is a floating point, the other will be converted to floating point
  - If both operands are integral, they will remain integral

Casting Operator

- Used to manually "squeeze" something into a smaller or lower type

```
int x = getchar();
if (x != EOF){
    char c = (char) x;
    ...
```

- Be careful that you do not lose information!
  - The value of x must fall in the range representable by a char (e.g., 0 through 255)
    - Otherwise, only one byte of x is kept!
  - Casting floating point to integral truncates any fractional part
    - int z = (int) 3.9; // x == 3

An Exercise: Expression Types

Find the people with expressions that are the same type as yours

Expression Types

- long
  - 734L
  - (long)(58365738104.486)
  - 7L + (short)18
  - -1845L
  - 30000000000L
- int
  - 'v'
  - 7 / 3
  - 793 - 128
  - 'b' + 18
  - 79374 % 372
  - -18439 * (int)(3.141592)
  - -4982
  - 1 / 2
- float
  - 3.587F
  - (float) 7 / 3
  - -9.8e4F + 7
  - 1.9F
  - 1.0F / 2.0F
- double
  - 7.0 / 3
  - 1.7e8 + 1000
  - 1.000000037
  - 2.0 * 3 / 7
  - 1.0 / 2.0
  - 1.0 / 2.0F

What are Outputs?

```
#include <stdio.h>
int main(){
      int a = 32767;
      short b;
      printf ("size of int = %ld, size of short = %ld\n",
sizeof(int), sizeof(short));

      b = (short)a;
      printf ("a = %d, b = %d\n", a, b);

      a ++;
      b = (short)a;
      printf ("a = %d, b = %d\n", a, b);

      return 0;
}
```

A Production Issue

```
void do_something(short argu)
{
      ……
}
int main()
{
      int db_table_key;
      ……..
      do_something(db_table_key);
      …….
}
```

Outline

- Variables
  - Constants and their types
  - Type casting
  - Double/float calculation
  - Automatic and external variables
  - Scope
  - Initialization
  - #define directive
  - enums
- Operators miscellany

What are outputs?

```
#include <stdio.h>
int main()
{
	double a = 1.03;
	double b = 0.42;
	double c;
	c = a - b;

	printf ("The result is %.20f\n", c);

	printf ("Calculate 1.00 - 9 * 0.1 = %.20f\n",
1.00 - 9 * 0.1);
}
```

Notes About Double/Float

- Avoid float and double if exact answers are required!!!
- Moreover, to test a floating point value for equality to some other number, it is best to declare a "nearness to x" epsilon value. For example, instead of checking to see if double/float x is equal to 2 as follows:

  if (x == 2) …

  it is better to use:

  if (abs(x - 2) < epsilon) …

  (assuming we have epsilon defined correctly!)

Software Disaster

Software Disaster
Disasters Channel
2 years ago • 56,156 views
During the Gulf War in the early 1990's, Operation Desert Storm use…

- https://www.youtube.com/watch?v=aYFVfbvFEjs

Outline

- Variables
  - Constants and their types
  - Type casting
  - Double/float calculation
  - Automatic and external variables
  - Scope
  - Initialization
  - #define directive
  - enums
- Operators miscellany

Global (External) Variables

- Our examples have declared variables inside functions
  - Called **automatic variables**
- Can declare variables outside functions
  - Called **external variables**
  - Also called **global** variables
- Poor Style
- Global variables violate modularity
  - Difficult to keep track of what functions are modifying which data

Outline

- Variables
  - Constants and their types
  - Type casting
  - Double/float calculation
  - Automatic and external variables
  - Scope
  - Initialization
  - #define directive
  - enums
- Operators miscellany

Variable Scope

- Scope: the part of the program where a variable exists
- External variables: from declaration to end of file
- Automatic variables
  - Declared at the beginning of a block
  - Scope is from the declaration through the end of the block
  - Can create blocks at any point inside a function

```
{
   int x = 7;
   ...
   /* x is in scope */
}
/* x is out of scope */
```

The static Keyword

- A static variable inside a function keeps its value across function calls
  - Initialization is done one time only

```
int someFunction(const int* array){
   static int numFnCalls = 0;
   numFnCalls++;

   printf("someFunction has been called %d times.\n",
          numFnCalls);

   ...
}
```

Outline

- Variables
  - Constants and their types
  - Type casting
  - Double/float calculation
  - Automatic and external variables
  - Scope
  - Initialization
  - #define directive
  - enums
- Operators miscellany

Initialization

- Automatic variables
  - By default, no initialization: variables have undefined contents
  - Initialization values can be any valid expression

```
int main(){
   int numBeans = countBeans(); /* allowed */
   int i;                       /* uninitialized */
   ...
}
```

Outline

- Variables
  - Constants and their types
  - Type casting
  - Double/float calculation
  - Automatic and external variables
  - Scope
  - Initialization
  - #define directive
  - enums
- Operators miscellany

#define

```
#include <stdio.h>
#define BUFFER_SIZE 1000
int main(){
   char buffer[BUFFER_SIZE];
   ...
```

- Before code is compiled, `BUFFER_SIZE` is replaced by whatever follows `BUFFER_SIZE` in the `#define` statement
- General form: `#define` *`expression value`*

Use const instead of #define

- Problems with `#define`:
  - The #define names have no type
  - Not even "existence" checking!

```
#define BUFFER_SIZE
...
char buffer[BUFFER_SIZE] = "to fill in";
buffer[BUFFER_SIZE -1] = '\0';
```

  - This will compile without even a warning
  - The `#define` could be in a different file, making it very difficult to pinpoint the invalid array access (buffer[-1])
- Using `const` variables avoids these problems

Outline

- Variables
  - Constants and their types
  - Type casting
  - Double/float calculation
  - Automatic and external variables
  - Scope
  - Initialization
  - #define directive
  - enums
- Operators miscellany

An Example

```
int main(){
   const int ID232 = 0;
   const int ID271 = 1;
   const int ID161 = 2;
   const int ID160 = 3;
   ...
```

- There is a shorter way to define groups of related constants...

enums

```
enum classIDs {ID232, ID271, ID161, ID160,
                        NUM_CLASSES};
```

- The `enum` constants have type `int`
- The first constant is 0 (unless overridden)
- Each subsequent constant is one more than the last (unless overridden)

```
enum classIDs {ID232=2, ID271, ID161=1, ID160};
```

- ID271 has value _____
- ID160 has value _____
- ID232 and ID160 are allowed to have the same value

Outline

- Variables
  - Constants and their types
  - Type casting
  - Double/float calculation
  - Automatic and external variables
  - Scope
  - Initialization
  - #define directive
  - enums
- Operators miscellany

Prefix and Postfix

- Both `++i` and `i++` increment the value of `i`
  - `++i` == value of `i` **after** incrementing
  - `i++` == value of `i` **before** incrementing
- Similarly for `--i` and `i--`

Prefix and Postfix

- If the value of `i++` or `++i` is not used, their effects are the same
- Same or different?
- `for(i=0; i<n; ++i)` and `for(i=0; i<n; i++)`
- `int x = i++;` and `int x = ++i;`
  - What are the values if `i == 3` at the start?

Assignment Operators

- `lhs += rhs;`
  - Equivalent to `lhs = (lhs) + (rhs);`
    - Except that `lhs` is evaluated only once

```
int x = 7, y = 4;
x += y * 2;
```

- What is x?
- Most binary operators (e.g., `+,-,*,/,%`) have similar corresponding assignment operators

Conditional Expressions

- `(a > b) ? a : b`
  - Evaluates to the max of `a` and `b`
- General form:
  - `condition ? ifTrue : otherwise`
- Another example of compactness versus clarity
  - Can always just write out the if ... else ... statement, though more variables may be needed
- You should not use these unless there is a very compelling reason

Operator Precedence

higher precedence ↑

Operators	Associativity
`()  []`	L to R
`!  ~  ++  --  (unary)+  (unary)-  (type)  sizeof`	R to L
`*  /  %`	L to R
`+  -`	L to R
`<<  >>`	L to R
`<  <=  >  >=`	L to R
`==  !=`	L to R
`&`	L to R
`^`	L to R
`|`	L to R
`&&`	L to R
`||`	L to R
`?:`	R to L
`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=`	R to L
`,`	L to R

Operator Precedence and Associativity

```
int x=7, y=4;
int j = x * y / x;
```

- What is `j`?
- Left-to-right associativity: `j = (x * y) / x`

- `int j = x * (y / x);` gives a different result

```
if(x == 3 && y == x || y == 4)
```

- Is this true?
  - Yes, because `||` has lower precedence than `&&`

Precedence

- Sometimes adding parentheses to an expression can help with clarity, even if the order of precedence does not require them

- Some orders are left up to the compiler
  - Function calls on the same line
    - int x = f(buffer) + g(buffer);
    - Either f or g could be called first, which may affect the input to the other function
  - a[i] = i++;
  - Put things in separate statements if order is important!

Summary

- Variables
  - Constants and their types
  - Type casting
  - Double/float calculation
  - Automatic and external variables
  - Scope
  - Initialization
  - #define directive
  - enums
- Operators miscellany

Big Picture

- ☑ Writing programs on *nix computers
  - ☑ File system commands
  - ☑ Editing, compiling, running, and debugging programs
- C Language
  - ☑ Variables, operators, basic I/O, control flow
  - ☑ Functions, more variables and operators
  - What is next?
    - Pointers: an important part of C that is not explicit in Java
    - Organizing data in structures
    - Multi-file programs
    - …

Upcoming

- gdb lab
  - Will go though some problems similar in style to the upcoming project
  - Can bring gdb references
    - Slides
    - "Programming with GNU Tools"
    - Other internet resources you find useful
  - Can bring problem-solving references

Optional Material

~~Additional Type Casting Information~~

~~Bit Operators~~

Automatic Type Conversions

- `4.5L * 'x'`
  - The `int 'x'` is converted to a `long double`
  - The result is a `long double`
- `4.0 / 5`
  - The `int` 5 is converted to a `double`
  - The result is a `double` (`0.8`)
- `char c = 5; short x = 4; x / c;`
  - Both the `char` and `short` are converted to `int`
  - The result is an `int` (`0`)

Automatic Type Conversions

- **The basics:**
  - If either operand is a floating point, the other will be converted to floating point
  - If both operands are integral, they will remain integral
- The details:
  - A binary operator with operands of two different types will promote the "lower" type to the "higher" type
    - `long double` is highest, followed by `double`, then `float`
    - `char` and `short` are always promoted to `int`
  - If, after all this, there is a `long` operand, convert the other operand to `long`

Automatic Type Conversions

- `long x = 7;`
  - The `int 7` is converted to a `long` by the binary operator `=`
- `x / 8.9F;`
  - The `long x` is converted to a `float`
- `x / 8;`
  - The `int 8` is converted to a `long`
- Function arguments are also automatically converted to higher types

```
double timesPi(const double x);
...
timesPi( 7 );
/* the int 7 is converted to a double */
```

Type Casting

- The casting operator `(type)` provides a way to explicitly convert types

```
int x, y;
...
(long) x / y;
```

  - Explicit cast of `x` invokes an automatic cast of `y`
- Even if an automatic cast would apply, writing the cast operator makes it obvious that a cast is taking place

Type Casting

- The value of the variable is not changed
- As if a temporary variable of the specified type was created and assigned
  - Using a cast

```
long result;
int x = 9;
result = (long)x + 7;
```

  - Using a temporary variable

```
long result;
int x = 9;
long temp = x;
result = temp + 7;
```

Issues with Casting

```
int x,y;
char c;
...
y = x;
c = (char)x;
x = (int) c;
```

- Is x == y?
- BEWARE of coercing to lower types using the casting operator
  - You MUST be sure that the value of the higher type can "fit" in the lower type, or you risk losing information

Casting Floating Point to Integral

```
double x = 4.7;
int y = x;
```

- What is the value of `y`?
  - Fractional part is truncated (not rounded)
- Standard library provides ceiling (`ceil`) and floor (`floor`) functions

Bitwise Operators

- Apply only to integral types
- Not commonly used, but good to know their existence
- & : bitwise AND
- | : bitwise OR
- ~ : bitwise NOT (i.e., complement)

```
if(x & 1){
   /* x is odd */
   ...
```

- What is the value of x after the assignment?

```
int x, y;
x = ~(y & ~y) & (y | ~y) & y;
```

Bit Masks

- Used to extract some bits
- The 1's in the mask determine which bits are extracted
- Other bits are set to 0

```
value:          01110110      value:          11000111
mask:           11110000      mask:           11110000
value & mask:   01110000      value & mask:   11000000
                ^^^^                          ^^^^
                bits from value               bits from value
```

Other Bitwise Operators

- ^ : bitwise exclusive OR (not exponentiation!)
  - 0xF0 ^ 0x1F == 0xEF
  - 0xFF ^ 0xFF == 0x0
- `x << y` shifts `x` left by `y` bits
  - Equivalent to multiplying by 2^y
- `x >> y` shifts `x` right by `y` bits
  - If `x` is unsigned, equivalent to integer division by 2^y
  - If `x` is signed, result depends upon machine implementation

A Bit Exercise

- Print out the bits corresponding to the value

```
void printBits(const int x){
   const int numBits = sizeof(x) * 8;
   int i;

   printf("The bit representation of the int %d is:", x);
   for(i=numBits-1; i >= 0; i--){
      const int mask = (1 << i);
      int bit = 1;
      if((mask & x) == 0){
         bit = 0;
      }
      printf("%d", bit);
   }
   printf("\n");
}
```