CS 201
Constants and Data Types

Debzani Deb

Exam 1(out of 58) Grade distribution

- Highest: 52
- Lowest: 18
- Average: 36.5
- See Lecture 11 for exam review.

Constant Declarations

- Up to now we have used #define to define a named constant.
- Another way is to define a constant inside the regular code.
- e.g. const double PI = 3.14159;
- PI = 3.0; is now illegal!
- The keyword const indicates a variable that does not change.
- Constants are useful for parameters which are used in the program but are do not need to be changed after the program is compiled.

const Restrictions

- Constants must be initialized when they are defined.
- Constant values can never be changed.
- They can never appear on the left of an assignment operator, =

const vs. #define

- Constants that are defined by using const are understood & checked by the C compiler itself immediately. So error messages are much more helpful.
- The **#define** directive is just substituted by the preprocessor and is not checked until the macro is used.
- **const** uses typical C syntax, while **#define** has its own syntax.
- **const** follows normal C scope rules, while constants defined by a **#define** directive continue on forever.

const vs. #define (cont.)

- The **#define** directive can only define simple constants.
- The **const** statement can define almost any type of C constant, including things like structures.
- The **#define** directive is essential for things like conditional compilation and other specialized uses.

Conditional Compilation

- One use is to "comment out" a bunch of code. Suppose you had:

```
...
i=12;
/* This is a one line comment */
J = 15;   /* This is a comment at the end of
a line */
K = -10;
X = sqrt (102.34);
/* Another one line comment */
...
```


Conditional Compilation

- You would like to eliminate all this code temporarily. Comment before and after? No, that doesn't work.

```
/*
i=12;
/* This is a one line comment */
J = 15;   /* This is a comment at the end of
a line */
K = -10;
X = sqrt (102.34);
/* Another one line comment */
*/
```

Terminates the comment here.

Conditional Compilation

- You can use a pre-processor statement to do this easily.

```
#ifdef _FLAG_01
i=12;
/* This is a one line comment */
J = 15;   /* This is a comment at the end of a
line */
K = -10;
X = sqrt (102.34);
/* Another one line comment */
#endif
```

None of this will be compiled into your program because you haven't ever defined a variable called _FLAG_01

Conditional Compilation

- Want to quickly turn on all the segments so marked? Simply define _FLAG_01.

```
#define _FLAG_01
#ifdef _FLAG_01
i=12;
/* This is a one line comment */
J = 15;   /* This is a comment at
the end of a line */
K = -10;
X = sqrt (102.34);
/* Another one line comment */
#endif
```

All this will be part of your program. And, any other section marked with _FLAG_01.

Other Base Constants

Base 10	Base 8	Base 16
6	06	0x6
9	011	0x9
15	017	0xF

const Pointers

- const int * num_ptr;
- Does NOT tell C that the variable num**_ptr** is a constant! Instead, it tells C that the data pointed to by num**_ptr** is a constant.
- Variable pointer to Constant data.
- The data cannot be changed, but the pointer can.
- int const * num_ptr does the same.

Pointer Is Constant

- If we put the **const** after the *, we tell C that the pointer is constant.
- int * const num_ptr;
- Constant pointer to variable data.
- The data can be changed, but the memory that contains the data is unmovable.

Or Both Unchangeable

- To make them both constants, put two const in.
- const int * const num_ptr;
- Constant pointer to constant data.
- The data cannot be changed, and the pointer cannot be changed.

WHY use Constants?

- Languages like Ada and Java provide automatic protection for the programmer. Similar to guards and safety switches on a table saw.
- C is like a spinning blade in space.
- Using const carefully can protect your code from unintended side effects.

Side Effects?

- One of the worst habits of so called "programmers" is being proud of their mastery of C's side effects! "Look how clever I am!"-type approach. You will see a lot of code written like that.
- However, no one can read the code!!!. So please don't be like them.
- Also codes with side effects may produce undefined results
  - Any combination of increment, decrement, and assignment operators (++, --, =, +=, -=, etc.) in a single expression which causes the same object either to be modified twice or modified and then inspected.
  - We have looked at one: i++* i++, another one is i = i++.
  - See http://c-faq.com/expr/ for some interesting discussion.
- It is good to know this tricks, but just for READING, not for CODING.

More Examples…

```
variable = other_variable ++;
```

- Main effect, assign the value of other_variable to variable.
- Side effect, increment other_variable.

More Examples…

```
if(variable = expression) x=0;
```

- Main effect: assign 0 to x if the value of the assignment is true (!= 0).
- Side effect: assign the value of expression to variable.
- Side effects are sort-of hidden actions taken by the language.
- Try to avoid them. If you use them, add comments!

Data Types

Primitive Data Types

- So far, we have used **3 data types**
  - int
  - char
  - double
- We have also seen some ways that these data types are related
  - int and double are both numbers, so we can perform math between them
  - We've used int values to store **true** or **false** (1 or 0)
- These data types are all consider **simple** or **scalar** data types because they only hold a single value.

Why we need different numeric data types?

- Operations involving integers are *much* faster to execute than those involving numbers of type double.
  - Depending on the computer, this difference may be very significant
- Integers take up less space than doubles.
  - int = 4 bytes
  - double = 8 bytes
- Operations with integers are always precise, whereas some loss of accuracy or round-off error may occur when dealing with type double numbers.
- These differences result from the way numbers are represented in the computer's memory.
  - The integer 13 is stored as 00000000 00000000 00000000 00001101
  - The double 13.0 is stored as 01000000 00101010 00000000 00000000 00000000 00000000 00000000 00000000

Representation of int

- Integers are stored in normal binary.
  - $13 = 1101 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$
  - When we perform math on integers, we just do simple binary arithmetic, which is very efficient and exact.
- int on linux
  - 4 bytes used.
  - (Where n = 32)

#include limits.h

INT_MIN **INT_MAX**

[-2147483648, 2147483647] $-2^{n-1} \xrightarrow{to} 2^{n-1}-1$

- $2^{4x8} = 2^{32} = 4294967296$
- 4294967296/2 = 2147483648

Important!!!

- Don't depend on your assumptions for size.
- Use the internal variables INT_MAX, INT_MIN to verify what you believe to be true.
- Make sure you use large enough data types to prevent overflow.

```
i = INT_MAX;
printf("%d %d\n", i, i+1);      // What is the output?
```

```
2147483647 -2147483648
```

We call is Arithmetic Overflow

Representation of double

- Double values are stored in **floating point** format, which is like scientific notation.
- The storage area is divided into two sections: **mantissa** and **exponent**.
  - The **mantissa** is between .5 and 1.0 for positive numbers and -.5 and -1.0 for negative numbers
  - The exponent is an integer
- These numbers are chosen so the following formula is true:
  - real number = mantissa * 2^{exponent}
- Because of the way they are stored, not all numbers can be represented exactly.

Representation (cont.)

- double
  - 8 bytes used.
  - #include float.h

On my machine, linux:

DBL_MIN=2.225074e-308

DBL_MAX=1.797693e+308

On my laptop, Windows Xp Pro:

DBL_MIN=2.225074e-308

DBL_MAX=1.797693e+308

Size of int/double

Type	Range in Typical Computer
short	-32,767 ... 32,767
unsigned short	0 ... 65,535
int	-2,147,483,647 ... 2,147,483,647
unsigned int	0 ... 4,294,967,295
long int	-2,147,483,647 ... 2,147,483,647
unsigned long int	0 ... 4,294,967,295

Type	Approximate Range
float	10^{-37} ... 10^{38}
double	10^{-307} ... 10^{308}
long double	10^{-4931} ... 10^{4932}

Numerical Inaccuracies

- Not all numbers can be exactly represented by the double format
  - Just as certain fractions cannot be represented exactly in the decimal number system (e.g., 1/3 is 0.3333...), some fractions cannot be represented exactly as binary numbers.
  - Think about representation of 1.55 in binary form

    $1 = 1*2^0$ (obvious)

    $.55 = .(1*2^{-1} + 0*2^{-2} + 0*2^{-3} + 0*2^{-4} + 1*2^{-5} + 1*2^{-6} + \ldots)$

    $= .(0.5 + 0.03125 + 0.015625 + \ldots)$
- Accuracy is totally dependent on the number of binary digits used to represent numbers: the more bits, the smaller the error.
- We call this the **representation error** (sometimes called round-off error)
- Because of this kind of error, an equality comparison of two type double values can lead to surprising results.

Numerical Inaccuracy: Example

```
int main(void)
{
   double y = .99, x = .33;
   while(y != 0)
   {
        y = y - x;
        printf("loop\n");
   }
   return 0;
}
```

What will be the Output?

Numerical Inaccuracy: Example

```
double sum = 0.0;

for(i=0; i<1000; i++)
        sum = sum + 1.55;

printf("sum 1.55 1000 times = %f\n", sum);
```

sum 1.55 1000 times = 1550.010864

Other Inaccuracies

- When you add a large number and a small number, the larger number may "cancel out" the smaller number, resulting in a **cancellation error**.
  - For example, 1000.0 + 0.0000001234 is equal to 1000.0 on some computers
- If two very small numbers are multiplied, the result may be too small to be represented accurately, so it will be represented as zero.
- This phenomenon is call **arithmetic underflow**.
- Similarly, if two large numbers are multiplied, the result may be too large to be represented.
- This phenomenon, called **arithmetic overflow**, is handled in different ways by different C compliers.

Automatic Conversion of Data Types

- In Chapter 2, we saw several cases in which data of one numeric type were automatically converted to another numeric type.
  Example: Given

```
int k = 5, m = 4, n;
double x = 1.5, y = 2.1, z;
```

k + x : x is of type double, so k is converted to double
z = k / m : since k and m are both of type int, so we get 1, and then convert that to 1.0 (double) to store in z.
n = x * y : we compute x * y to get 3.15 and then convert it to type int, so 3 is stored in n.

Explicit Conversion of Data Types

- In addition to automatic conversions, C also provides an explicit type conversion operation called a cast.
  `z = (double)k / (double)m;`
- Placing the name of the desired type in parentheses immediately before the value to be converted causes the value to be changed to the desired data format before it is used in the expression.
- Because this explicit conversion is a very high precedence operation, it is performed before the division.
- We could not achieve our goal by doing
  `z = (double)(k / m);` Why?

Representation and Conversion of char

- We have declared variables of type char and have used type char constants consisting of a single character enclosed in apostrophes.
- Each character has its own unique numeric code, the binary form of this code is stored in a memory cell that has a character value
  - One Byte per character
  - Collating sequence. ‘a’<‘b’, ‘A’ < ‘B’, ‘0’ < ‘1’.
  - How does C compute ’A’ < ’Z’? Ans: ’A’ equals 65 and ’Z’ equals 90, so ’A’ < ’Z’ is true.
  - How about ‘a’ < ‘A’ or ‘A’ < ‘a’??? Never be sure, always check on your machine.

Enumerated Types(1)

- Good solutions to many programming problems require new data types other than the primitive types.
- For example, in a calendar program you might need to distinguish between the different months: january, february, march, april, may, june, july, august, september, october, november, december.
- C allows you to associate a numeric code with each category by creating an enumerated type that has its own list of meaningful values.

```
typedef enum {january, february, march, april, may,
june, july, august, september, october, november,
december} month_t;
```

Defines a new type month_t

```
month_t month;
```

Defines a variable month of type month_t

Enumerated Types(2)

- Defining type month_t as shown causes the **enumeration constant** january to be represented as the integer 0, constant february to be represented as integer 1, and so on.
- Variable month and the twelve enumeration constants can be manipulated just as one would handle any other integers.
- Now you can write…..

```
month_t c_month, n_month;
c_month = january;
c_month++;
if (c_month == december)
    n_month = january;
else
    n_month = month_t(c_month + 1);
for(c_month = january; c_month <= july; ++c_month) { ... }
```


Enumerated Rules

- Enumerated constants must be identifiers. They can’t be
  - Numeric (1,3,-4)
  - Character (‘s’, ‘t’, ‘p’)
  - Or String (“This is a string”) Literals.
- An identifier cannot appear in more than one enumerated type definition.
- The identifiers you use can’t be used again for variable names
  - I couldn’t have a month_t type january and a variable january
- Relational, assignment, and even arithmetic operators can be used, just as with other integers.
- C provides no range checking to verify that the value stored in an enumerated type variable is valid.
  - `n_month = december + 3;` won’t cause any run-time error even though it is clearly invalid.