Basic types and operators

Stat 580: Statistical Computing

- Theme: Black - White
- Printable version

References

- Part of this slide set is based on *Essential C* by Nick Parlante:

(for copyright reason, this notice is reproduced here.)

Basic types

Overview

- Variables are the basic objects in a program.
  - declared before use: type and name
  - the type determines
    - the set of values it can have
    - what operations can be performed
- There are two basic classes:
  - integer types
  - floating point types

Integer types

- There are four common integer types:
  - `char`, `short`, `int`, `long`
- They corresponds to different numbers of bits ("widths"):
  - the wider types can store a greater range of values
  - for instance,
    - size of `char` is 8 bits (-128 to 127, machine dependent)
    - size of `long` is at least 32 bits

Integer types

- The integer types can be preceded by the qualifier `unsigned`.
  - disallows representing negative numbers, but doubles the largest positive number representable
  - for instance, the range of values of `char` is -128 to 127 (machine dependent) while that of `unsigned char` is 0 to 255

Four integer types

`char` : ASCII character -- at least 8 bits

- pronounced as "car"
- `char` is basically always a byte (8 bits) which is enough to store a single ASCII character.
- 8 bits provides a signed range of -128 to 127 or an unsigned range of 0 to 255
- `char` is also required to be the "smallest addressable unit" for the machine -- each byte in memory has its own address

Four integer types

`short`: Small integer -- at least 16 bits

- 16 bits provide a signed range of -32768 to 32767
- typical size is 16 bits
- not used so much now

Four integer types

`int`: Default integer -- at least 16 bits, with 32 bits being typical

- defined to be the "most comfortable" size for the computer
- if you do not really care about the range for an integer variable, declare it `int` since that is likely to be an appropriate size (16 or 32 bit) for that machine

Four integer types

`long`: Large integer -- at least 32 bits

- typical size is 32 bits which gives a signed range of about -2 billion to +2 billion
- some compilers support `long long` for 64 bit integers

An example

```c
#include<stdio.h>
#include<limits.h>

int main() {

  printf("A byte is %u bits\n", CHAR_BIT);
  printf("A char has %lu bytes and %lu bits\n", sizeof(char),
        sizeof(char)*CHAR_BIT);
  printf("A short has %lu bytes and %lu bits\n", sizeof(short),
        sizeof(short)*CHAR_BIT);
  printf("A int has %lu bytes and %lu bits\n", sizeof(int),
        sizeof(int)*CHAR_BIT);
  printf("A long has %lu bytes and %lu bits\n", sizeof(long),
        sizeof(long)*CHAR_BIT);

  return 0;
}
```

Output on my machine:

```
A byte is 8 bits
A char has 1 bytes and 8 bits
A short has 2 bytes and 16 bits
A int has 4 bytes and 32 bits
A long has 8 bytes and 64 bits
```

Common integer constants

`char` constants

- a `char` constant is written with single quotes (') like 'A' or 'z'
- the `char` constant 'A' is really just a synonym for the ordinary integer value 65 which is the ASCII value for uppercase 'A'
- there are special `char` constants such as
  - '`\t`' for tab, for characters
  - '`\n`' newline character

Common integer constants

`int` constants

- numbers in the source code such as `234` default to type `int`
- they may be followed by an '`L`' (upper or lower case) to designate that the constant should be a `long` such as `42L`

Integer combination and promotion

- the integral types may be mixed together in arithmetic expressions
  - since they are all just integers with possibly different widths
  - e.g., `('b' + 5)`
- How does the compiler deal with different widths?
  - the compiler *promotes* the smaller type (e.g., `char`) to be the same size as the larger type (e.g., `int`) before combining the values
  - *promotions* are determined at compilation time based purely on the types of the values in the expressions
  - *promotions* do not lose information: they always convert from a type to compatible, larger type

Floating point types

Three floating point types:

- `float`: single precision floating point number (typical size: 32 bits)
- `double`: double precision floating point number (typical size: 64 bits)
- `long double`: possibly even bigger floating point number (somewhat obscure)

```
printf("A float has %lu bytes and %lu bits\n", sizeof(float),
       sizeof(float)*CHAR_BIT);

printf("A double has %lu bytes and %lu bits\n", sizeof(double),
       sizeof(double)*CHAR_BIT);

printf("A long double has %lu bytes and %lu bits\n", sizeof(long double),
       sizeof(long double)*CHAR_BIT);
```

An example

```
printf("A float has %lu bytes and %lu bits\n", sizeof(float),
       sizeof(float)*CHAR_BIT);

printf("A double has %lu bytes and %lu bits\n", sizeof(double),
       sizeof(double)*CHAR_BIT);

printf("A long double has %lu bytes and %lu bits\n", sizeof(long double),
       sizeof(long double)*CHAR_BIT);
```

On my machine:

```
A float has 4 bytes and 32 bits
A double has 8 bytes and 64 bits
A long double has 16 bytes and 128 bits
```

Floating point types

Floating point constants: (e.g. '`3.14`' in the source code)

- default type is `double`
- its type can be changed by suffixing with an '`f`' (`float`) or '`l`' (`long double`).

Promotion rule in arithmetic expressions:

- Simplified rules:
  - promotes smaller floating point types to larger floating types
  - promotes integer types to floating point types

Variables

Hinted from variable declaration:

```
int x;
```

A variable declaration *reserves* and *names* an area in memory at run time to hold a value of a particular variable

- **reserves**: need the size of the variable (determined by the type)
- **names**: need the name of the variable

Variables

Variable names in C

- case sensitive
- can contain digits and underscores (_), but may not begin with a digit
- C is a classical "compile time" language -- the names of the variables, their types, and their implementations are all flushed out by the compiler at compile time (as opposed to figuring such details out at run time like an interpreter)

Variable declaration

Different ways of declaration

- `int num=42;`
- `int x, y;`
- `int x=10, y=2;`
- `int x=10, y;`

Variables start with arbitrary values

```c
int num; /* declare int variable "num" */
num = 42; /* assign 42 to the variable "num" */
```

```c
int num; /* declare int variable "num" */
printf("Before initialization: %d\n", num);
num = 42; /* assign 42 to the variable "num" */
printf("after initialization: %d\n", num);
```

Truncation (not mathematical truncation)

```
char ch;
int i;
i = 321;
ch = i;     /* truncation of an int value to fit in a char */
```

Truncation moves a value from a type to a smaller type: *loss of information*!

- opposite to promotion
- may or may not lead to a compile time warning
- assigning from an *integer* to a smaller *integer* (e.g.. `long` to `int`, or `int` to `char`): drops the most significant bits
- assigning from a *floating point type* to an *integer*: drops the fractional part of the number (and drops the most significant bits if necessary)

An example

```
char ch;
int i;
i = 321;
ch = i;     /* truncation of an int value to fit in a char */
printf("(%d, %c)\n", ch, ch);
```

What is the output?

An example

```
char ch;
int i;
i = 321;
ch = i;     /* truncation of an int value to fit in a char */
printf("(%d, %c)\n", ch, ch); /* (65, A) */
```

- The assignment will drop the upper bits of the `int` 321.
- The lower 8 bits of the number 321 represents the number 65 (321 - 256).
- So the value of ch will be (`char`) 65 which happens to be 'A'.

Another example

```
char i, j;
double x, y;

x = 3.523434;
y = 321.523434;
i = x; /* truncation of an double value to fit in a char */
j = y; /* truncation of an double value to fit in a char */

printf("x = %f and i = %d\n", x, i);
printf("y = %f and j = %d\n", y, j);
```

What is the output?

Another example

```
char i, j;
double x, y;

x = 3.523434;
y = 321.523434;
i = x; /* truncation of an double value to fit in a char */
j = y; /* truncation of an double value to fit in a char */

printf("x = %f and i = %d\n", x, i);
printf("y = %f and j = %d\n", y, j);
```

The output is machine dependent (as the sizes of type are). On my machine:

```
x = 3.523434 and i = 3
y = 321.523434 and j = 65
```

- The assignment
  - throws away the decimals (mathematical truncation)
  - and then follows the previous rule of dropping upper bits

Integer division

```c
int x;
double y;

x = 1;
y = x / 2; /* integer division */
printf("%f\n", y);
```

Integer division

```c
int x;
double y;

x = 1;
y = x / 2; /* integer division */
printf("%f\n", y); /* 0.000000 */
```

Casting

Type casting is used to for type conversion for one single operation.

```
int x;
double y;

x = 1;

y = x / 2;
printf("%f\n", y);

y = x / 2.0;
printf("%f\n", y);

y = (double) x / 2;
printf("%f\n", y);

y = (int)((double) x / 2);
printf("%f\n", y);
```

Casting

Type casting is used to for type conversion for one single operation.

```c
int x;
double y;

x = 1;

y = x / 2; /* integer division */
printf("%f\n", y); /* 0.000000 */

y = x / 2.0; /* promotion */
printf("%f\n", y); /* 0.500000 */

y = (double) x / 2; /* casting */
printf("%f\n", y); /* 0.500000 */

y = (int)((double) x / 2); /* casting */
printf("%f\n", y); /* 0.000000 */
```

Boolean

C does not have a distinct boolean type

- `int` is used instead
- integer 0 as false
- all non-zero values as true

Basic operators

Assignment operators `=`

- assign a variable the value of an expression
  - e.g., `j = 1`
- two things actually happen:
  - `j` gets value `1`
  - the expression `j = 1` evaluates to `1`

Assignment operators `=`

What would happen if we run the following code?

```
x = y = z = 1;
```

- Precedence: check precedence table!
  - Associativity of `=`: Right-to-Left

1. Evaluate `z = 1`: `z` gets value `1` and `z = 1` evaluates to `1`
2. Evaluate `y = 1`: `y` gets value `1` and `y = 1` evaluates to `1`
3. Evaluate `x = 1`: `x` gets value `1` and `x = 1` evaluates to `1`

Mathematical operators:

- `+` (addition)
- `-` (negation/subtraction)
- `/` (division)
- `*` (multiplication)
- `%` (modulus)

Notes:

- Pay attention to the types of variables
  - be careful of *integer division*
- Precedence: check precedence table!

Increment operators:

- `++` (increment), `--` (decrement)
- `++i;` or `i++;` is equivalent to `i=i+1;`
- `--i;` or `i--;` is equivalent to `i=i-1;`
- `++i` (prefix): `i` is incremented before the expression is evaluated
- `i++` (postfix): `i` is incremented after the expression is evaluated
- Precedence: check precedence table!

Increment operators:

```c
int i, j, k;
i = 1;

i++;
printf("%d\n", i);

i = 1;
j = 2;
k = ++i + ++j;
printf("(i, j, k) = (%d, %d, %d)\n", i, j, k);

i = 1;
j = 2;
k = i++ + ++j;
printf("(i, j, k) = (%d, %d, %d)\n", i, j, k);
```

Increment operators:

```c
int i, j, k;
i = 1;

i++;
printf("%d\n", i); /* 2 */

i = 1;
j = 2;
k = ++i + ++j; /* 2 + 3 */
printf("(i, j, k) = (%d, %d, %d)\n", i, j, k); /* (2, 3, 5) */

i = 1;
j = 2;
k = i++ + ++j; /* 1 + 3 */
printf("(i, j, k) = (%d, %d, %d)\n", i, j, k); /* (2, 3, 4) */
```

Relational operators:

- `==` (equal)
- `!=` (not equal)
- `>` (greater than)
- `<` (less than)
- `>=` (greater or equal)
- `<=` (less or equal)

Relational operators:

Notes:

- operate on integer or floating point values and return a 0 or 1 boolean value
- usually used with control statements (e.g. `if`)
- `(x==3)` vs `(x=3)`
  - `(x==3)` returns `1` or `0` indicating `x` is `3` or not
  - `(x=3)` returns `3` (which is treated as true)!
- Precedence: check precedence table!

Logical operators:

- `!` (Boolean not, unary),
- `&&` (Boolean and)
- `||` (Boolean or)

Notes:

- Precedence: check precedence table!

Advanced assignment operators

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`

Notes:

- `i += j;` is equivalent to `i = i + j;`
- "variable operator= expression;" is equivalent to "variable = variable operator expression;"
- Precedence: check precedence table!