Variables, Types, Operators

Alexandra Stefan

Summary

- Variable declaration, initialization, and use.
  - Possible syntax errors
    - Syntax – rules for code to compile.

- Types:
  - Define what are legal values for a variable.
  - Both variables and values have types.
  - A variable can only have one type throughout the program.
  - Common errors:
    - Initialize a variable with a value of another type
    - Use data of a ‘wrong’ type for an operator
    - Use data of a ‘wrong’ type as function argument
      - Function takes arguments of a different type.

- Type casting:
  - Casting: (float)

- ++/--, +=/-= (not recommended in exams)

- Named constants: #define DAYS_PER_WEEK 7
  - #define – directive can be used in one other way, but we will not cover that.

Declaring a Variable

- A program uses *variables* to store and refer to data.
- You can think of a variable as a 'box' that holds data and has a label. You can only get to the box using the label.
  - Unlike an actual box, a variable is NEVER empty. **It always has a value**.

- You create a variable, by doing a **variable declaration**.

- There are two ways to declare a variable:

  type variable_name; *// declare only: name and type*

  type variable_name = initial_value; *// declare and initialize*

- For example:

```
int x;      //declaration only. Will x be empty?
int num_of_fingers = 5; //declare and initialize
float radius = 20.231;
```

  - Several variables can be declared on the same line:

```
int age, count = 0, total = 0, temp;
```

  - Each variable has a *type*, a *name*, a *value*, and a *memory address*.
    - We will rarely use the memory address of a variable in this class
    - Initialize every variable, otherwise it will have a random (junk) value.

Variable names

- Rules for variable names:
  - The name can contain only letters, digits (0-9), and underscore(_). Ok: `temp3, temp_3, in2ft, minGrade`
  - It cannot start with a digit. Invalid names: `3temp, 001`
  - It is NOT recommended to start with _.

- The name of a variable should indicate what that variable is used for (what data it holds). E.g. `count, age, total`

- Spelling errors - Be sure to check the spelling! Every time you use a variable name the spelling must be identical to the one from declaration.
  - **`Age`** is not the same as **`age`** . These are 2 different variable names for C.

Declaration/Initialization before Use

- C executes the code line-by-line from top to bottom of the given block.
- **A variable must be declared before we try to use it.**
- This code does not compile. Gives error: "age undeclared …"

// incorrect code: Variable **age** is NOT declared before use.

```
age = 5;   // type is missing, so this is not a declaration for 'a'
printf("age = %d\n",age);
```

A variable cannot be redeclared (in the same scope)

- **A variable can only be declared once within its scope. You cannot redeclare/redefine it (neither with the same type nor with a different type)**
- This code does not compile. Gives error: "redefinition of 'count' …"

```
int count = 5;
printf("count = %d\n",count);
int count = 10;      // error: count is redeclared/redefined
//float count = 10.6; // error: count is redeclared/redefined
```

Variable SCOPE – difficult now, easier later

- The variable scope is the part of the program where the variable is visible (where it exists). The scope:
  - starts - at declaration and
  - ends - where the first enclosing block ends (with }). Here a “block” is group of instructions enclosed by { } .

```
float price1 = 3.99 ;
{
   float price2 = 19.99;  //price2 is visible only in this block of {}
   printf("price1 = %f, price2 = %f\n", price1,  price2);
   // price1 is visible in here.
 }
// price1 is visible here.
// price2 is NOT visible here
printf("price is: %f\n", price1);
```

- There cannot be 2 variables with the same name in the same block.
- If you put all your variables at the top of the function you are safe

value vs address (pointer) of a variable

```
/* Just like a mailbox, a variable has BOTH a content (or value) and an address.
   A variable is stored at a memory location. That is its address.
   The content of the variable is what is written (as 0/1 bits) at that address.
 E.g. mailbox 230 (i.e. at address 230) in the picture may contain value 5.   */

int age = 0;
printf("Enter your age: ");
scanf("%d", &age); // assume user enters 10
printf("\n content of age is %d, address of age is %p", age , &age );
```

- Address/memory address/pointer – all mean the same thing
- The address (of the variable) is still a value. It is NOT the name of the variable.
- When we use the variable name the CONTENT/VALUE is used.
- When we use `&name` the memory address of that variable is used.
- Example where the address of a variable is needed: **scanf**
  - Use & for int, char, float, double (and other primitive types):
    - &age – gives the [memory] address of age in scanf("%d", &age);
  - No & for strings, because the "value" of a string variable is its address: e.g. scanf("%s", lastName);

types: int, float, double, char

- int
  - Stores integer numbers.
  - E.g.: 6, 9230, -15, 0
  - Beware that division between 2 int values gives just the integer part of the division. E.g. 7/10 gives 0.
  - `%d` (format specifier)

- float
  - Stores real numbers (that can have decimals)
  - E.g.: 13.89, -0.0037, 5892, 17023.34
  - Division works as expected.
  - , is not allowed. E.g. wrong: 13,909 Correct: 13909
  - `%f` (format specifier)
- double:
  - like float, but use `%lf`
  - Can represent larger numbers and with more precision than float
- char
  - Stores a single character/symbol. It can be a letter, a digit or other symbols: `.,;)*`, `tab`, `space`, `enter`
  - Hardcoded data of type char must be enclosed in single quotes:
    - E.g. `'A'`, `'k'`, `'+'`, `'3'`, `'!'`, `'?'`, `' '`(space)
  - `%c` (format specifier)
  - Use `" %c"` to skip whitespaces (space, tab, enter) when reading char with scanf
    - E.g. `scanf(" %c", &ch);`
    - Exception: when want to detect whitespaces. E.g. in a game, Enter= shoot, tab= duck
    - NOTE: when printing, use `"%c"` as normal.
  - ASCII codes - Each symbol (char) has an integer value. - See **ASCII table**
    - Try printf("value 65 printed as char is: %c\n", 65); // try other values: 66,67, 64, 48,49,50
    - Try printf("value 200 printed as char is: %c\n", 200);

```
int n = 9;
printf("%d", n);
printf("\nEnter n: ");
scanf("%d", &n);
```

```
float avg = 95.7;
printf("%f", avg);
printf("\nEnter avg: ");
scanf("%f", &avg);
```

```
double tempF = 82.5;
printf("%lf", tempF);
printf("\nEnter tempF: ");
scanf("%lf", &tempF);
```

```
char ch = 'A';
printf("%c", ch);
printf("\nEnter ch: ");

// skips Enter, tab, space
scanf(" %c", &ch);
```

See examples of more types and keywords

- More keywords:
  - You do NOT need to memorize any new ones from the list. Just be aware of this list and memorize the ones we cover
  - https://www.tutorialspoint.com/cprogramming/c_basic_syntax.htm
- More types
  - https://www.tutorialspoint.com/cprogramming/c_data_types.htm
  - Each type has a min and max value that it can store
    - See value ranges (e.g. unsigned char: [0,255])
    - Each type has a specific size (number of bytes) and that will affects how many values of that type we can have – this will be revisited later
  - Integer types:
    - char, int, short, long
    - unsigned char, unsigned int, unsigned short, unsigned long
  - Floating-point types:
    - float
    - double
    - long double

Types, type casting

- The variable type dictates:
  - What values a variable can take
  - How that variable can be used
  - What operations are allowed
  - How operations will be performed (e.g. see integer division)

- Both variables and values have a type. E.g.: 152, 23.98, 'A'

- Once declared, a variable will only have that type. It cannot be changed.

- Explicit type casting – we indicate in the code that the DATA from a variable (or expression) should be used as data of a different type ***in this evaluation***.
  - This does NOT change the type of the variable. When you later use it, it still has its original type.
  - Syntax: *(new_type) variable* E.g. `(float)A_count`
  - E.g.: `A_percent = ((float)A_count)/total;`
  - Or: `A_percent = A_count/(float)total;`
- Implicit type casting:
  - 2+10.1 - calculated using ALU (arithmetic logic unit) or FPU (floating point unit)?
    - These processing units work with data (bits), not variable names
    - The data must be written according to the type format (and it will have a specific size)
    - ALU – only integer types (int, long), FPU – only floating point types (e.g. float, double)
    - The result will be data of the same type
  - in 2+10.1, the 2 will automatically be converted to a floating point type (fractional number) and the FPU will be used.

types: char[]

- String(more specifically “array of char”)
  - Stores text.
  - E.g.: "Monday ", "Today is rainy", "Janet",
  - Must specify max length+1 when created
  - Declaration: `char name[101] = {};`
    - `101` indicates that it can store strings with at most 100 chars
    - `={}` (or `=""`) initializes it to the empty string (no letter in it)
  - `%s`
  - NO & in `scanf: scanf("%s",name); // not &name`
  - `"%[^n]s"` for spaces in string
  - `" %[^n]s"` for spaces in string and skip whitespaces at the beginning of string
  - Special function for strings:
    - Must include the library string.h: `#include <string.h>`
    - To copy use: strcpy(toStr, fromStr);
    - To get length use strlen(var). E.g. `strlen(name);`
      - Gives length of the string stored (e.g. 5), not original capacity (e.g. 101).
    - To read about more functions, read about the string.h library

```
#include <stdio.h>
#include <string.h>

int main(){
   int num;
   char name[101]= {};  // initializes it to the empty string

   printf("len(s) = %d", strlen(name));

   printf("Enter your name: ");
   scanf("%s", name);  // enter first and last

   printf("Your name is |%s|", name);

   printf("\nEnter your name again: ");
   scanf(" %[^\n]s", name);
   printf("Your name is |%s|", name);

   return 0;
}
```

in: scanf(" %[^\n]s", s);
remove the space between “ and %
and see what happens

Named Constants: const var and macros

- Sometimes we need to use constants in our programs. Examples:
  - 7 - days in a week.
  - 3.14159 - π(pi); other constants from science.
  - data specific to your application (e.g. minimum number of staff at a desk during a work day)
- Name them – this is better than just using the value
  - Use all uppercase letters
    - This is the naming convention for constant values
    - Using lowercase letters or a mix, will still compile and run, but is bad style
  - Change in one place – applies in all places - If you need to change the value, you will change it in one place, and it applies wherever that name is used.
  - Code is more readable
    - the name indicates what the data is about: DAYS_OF_WEEK, MAX_SIZE, SPEED_OF_LIGHT
    - It also indicates that it is a constant (thus should not be modified). Use all uppercase letters.
  - The compiler will do some checks for us (e.g. not allow us to modify it)
- 2 ways to achieve this:
  - **Macro: #define (we will use this)**
  - Constant variable

Named Constants: macros and const variable

Syntax	***#define NAME value*** **(define is a keyword)**	***const type name = value;*** **(const is a keyword)**
Is this in the test?	**Yes. This will be in tests. You must know it.**	This will not be.
Where in the program	**At the top, (below #include), outside any function**	Inside a function
Scope/Visibility (consistent with general scope rules for variables)	**Anywhere in that file** **(If in a .h file, in files that include it. See this in later classes.)**	Inside the function where they are declared
	The preprocessor will replace *NAME* with *value* before compilation. E.g. replace DAYS_PER_WEEK with 7 in example below. (Thus the compiled code is the SAME as if 7 was hardcoded directly.)	
Usage	You can use their value in anyway (e.g. for printing or in calculations) . Compiler gives syntax error if you try to modify their value.	

macro example

```
#define DAYS_PER_WEEK 7      // note no   "="   or   ";"
#define MIN_PAY 100
int main(){
  int days = 3 * DAYS_PER_WEEK;  // this usage is fine
  printf("%d weeks have %d days (%d).\n", 3, days, DAYS_PER_WEEK);
  DAYS_PER_WEEK = 10;   //syntax error; does not compile
```

Constant variable example

```
int main(){
  const int DAYS_PER_WEEK_2 = 7;  // const variable declare + init
  int days = 3 * DAYS_PER_WEEK_2;
  printf("%d weeks have %d days (%d).\n", 3, days, DAYS_PER_WEEK_2);
  DAYS_PER_WEEK_2 = 10;  //syntax error, does not compile
```

Arithmetical Operators

- * , /, + , -
- = (Assignment) E.g. `age = 10;`
  = vs == (assignment vs compare for equal)
- * (multiplication), / (division), +, -
- % (remainder from integer division) – returns ONLY the remainder. See next page.
- You can use the same variable twice in an expression (both to the left and right of =)

```
total = total + B_count;
```

- +=, -=, *=, /=, %=, ++, --

12+9/3 = _____
Operator precedence and typically left-to-right associativity:
https://www.tutorialspoint.com/operator-precedence-and-associativity-in-c
Or
https://en.cppreference.com/w/c/language/operator_precedence

Do NOT use +=, -= etc in an exam
(miss a symbol => wrong answer)

```
//Assume total is 14 and B_count is 9

total = total + B_count;
        ⇩
total = 14 + 9;
        ⇩
total = 23;
```

B_count: 9

total: ~~14~~ 23

Step 1: evaluate expression on right side.
a) replace `total` and `B_count` with their values.
b) calculate result value, say 23.
Step 2: update variable from left side:
write result value, say 23, in "box" for `total`.

No history of past values: it will NEVER remember that `total` had value 14 in the past.

Short form	Normal form.
`count += 2;`	`count = count + 2;`
`total *= 5;`	`total = total * 5;`
`avg /= count;`	`avg = avg / count;`
`budget -= pay;`	`budget=budget-pay;`
`count++;`	`count = count + 1;`
`count--;`	`count = count - 1;`

Using the short form has the same result as using the normal form.

% operator – the remainder from division

The % (modulo) operator is [frequently] used in programming.

Gives the remainder (from integer division).

Sample usage:

- Get a random number in a certain range.
- Number conversion from base 10 to another base (e.g. base 2)
- Crop out digits from a number:
  - 237%10 = 7 (the units digit) ,
  - 237/10 = 23 , 23%10 = 3 (the tens digit)
    all in one expression: (237/10)%10
- Produce a pattern:
  - Alternate between 2 options - Identify even/odd
    - Print black/white squares on a chess board ; Take turns in playing a game
  - *––*––*––*…. – when value%3 ==0 print *, else print –
  - Every 5 days start the sprinklers
  - Display progress at a specific rate: 10%, 20%, 30%
  - Distribute action over k queues (request N goes to queue N%k)
- Restart a count (from 0)
  - E.g. keep counting up, but use count%7 will always give a value in the range 0,1,2,3,4,5,6
    - map day of month to day of week;
    - array wrap-around – more details later

Math review:

45 % 6 = 3
Because:
45÷6 = 7 remainder 3

307 % 5 = 2
Because:
307÷5 = 61 remainder 2

6 % 45 = 6
Calculated as:
6÷45 = 0 remainder 6

If a<b then
a%b = a
E.g.
6%45 = 6
19%100 = 19

Note that = in code means assignment.

Good practice

1. Declare all variables at the beginning of the program (or function).
2. Initialize all the variables (at declaration)
    1. A variable is never ‘empty’. Even if you do not initialize it, it will have a value depending on the 0/1 values of bits from where it is stored. (like a board that was not erased)
3. Give meaningful names to variables:
    - E.g. `total_price, tax, count,`
4. Use all uppercase letters for named constants.
    - E.g. `DAYS_PER_WEEK, IN2CM`
5. Fully parenthesize larger expressions to ensure they are evaluated in the order you want. It also makes the code more readable.

Dictionary

- Hardcode /hardcoded data = using a specific number or value in the program instead of allowing it to be entered by the user. With hardcoded data, at every run of the program you use that same data. With user input, the user can give different data at each new run, so it is more flexible.
- Constant - named value created with `#define` or `const`
- Type
- Type cast
- String
- Modulo (%)
- +=, …
- Variable vs value

Practice

- For each line say if it will it give a syntax error.

```
#define DAYS_PER_WEEK 7
int main() {
  int age = 14;  // line 1
  int PI;        // line 2
  age--;          // line 3
  DAYS_PER_WEEK++;    // line 4
  DAYS_PER_WEEK = 7;  // line 5
  PI = 3.1415;        // line 6
  16++;               // line 7
  printf()++;         // line 8
}
```