Variables Outline

1. Variables Outline
2. C Character Set
3. Basic Data Types
4. Variables
5. C Variable Declaration
6. Variable Assignment
7. Variable Assignment Example
8. Changing a Variable’s Contents
9. Variable Initialization
10. C Variable Names
11. Literal Constants
12. Named Constants/Named Constant’s Value Can’t Be Changed
13. Why Literal Constants Are BAD BAD BAD
14. Why Named Constants Are Good
15. Output
16. Specifying the Value of a Variable Via
   Input from the Keyboard
17. Multiple Variables Per scanf Statement
18. Program Variables vs. Algebra Variables
19. Programming Exercise

C Character Set

These are the characters that C recognizes.

- **Letters**
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z
  a b c d e f g h i j k l m
  n o p q r s t u v w x y z

- **Digits**
  0 1 2 3 4 5 6 7 8 9

- **Special Characters**
  space (also known as blank)
  `" ( ) * + - / : = _`
  `! & $ ; < > % ? , . ^ # @ ~`
  `[ ] \ |`

C is case sensitive: it distinguishes between upper case and lower case letters.

**Keywords** in C — for example, the keyword int — **MUST** be in lower case. For example:

```c
#include <stdio.h>
int main ()
{ /* main */
  int height_in_cm;
  height_in_cm = 160;
  printf("My height is %d cm.n", height_in_cm);
} /* main */
```

Note: a **character string literal** is a sequence of characters delimited by a double quote at the beginning and a double quote at the end.
Basic Data Types

- Numeric
  - int
  - float
- Non-numeric
  - char

#include <stdio.h>

int main ()
{ /* main */
  float standard_deviation, relative_humidity;
  int  count, number_of_silly_people;
  char middle_initial, hometown[30];
} /* main */

Variables

A variable is an association between
- a name (chosen by the programmer), and
- a location in memory (chosen by the compiler).

Every variable has:
- a name, chosen by the programmer;
- an address (i.e., a location in memory), chosen by the compiler;
- a data type (e.g., int, float, char), chosen by the programmer;
- a value (which may be undefined), sometimes chosen by the programmer, and sometimes determined while the program is running (at runtime). The value is sometimes called the contents of the variable — that is, the value is the contents of the variable’s memory location.

The value of a variable can vary; that is, it can be changed at runtime. We’ll see how in a moment.

Jargon: compile time and runtime.

- Events that occur while a program is being compiled are said to happen at compile time.
- Events that occur while a program is running are said to happen at runtime.

For example, the address of a variable is chosen at compile time, while its value typically is determined at runtime.
C Variable Declaration

int x;

This declaration tells the compiler to choose a location in memory, name it \( x \), and think of it as an integer. Note that this declaration doesn’t specify a value.

The compiler might decide that \( x \) will live at, say, address 3980 or address 98234092 or address 56436. We don’t know and don’t care what address \( x \) lives at, because the compiler will take care of that for us; it’s enough to know that \( x \) has an address and that the address will stay the same throughout a given run of the program.

\[ x : \boxed{???????} \text{ (Address 56436)} \]

When \( x \) is first declared, we don’t know what its value is, because we haven’t put anything into its memory location yet, so we say that its value is undefined, or, informally, garbage.

Note: some compilers for some languages automatically initialize newly declared variables to default values (e.g., all integers get initialized to zero), but not every compiler does automatic initialization.

You should NEVER NEVER NEVER assume that the compiler will initialize your variables for you.

You should ALWAYS ALWAYS ALWAYS explicitly assign values to your variables in the body of the program.

Variable Assignment

An assignment statement sets the contents of a specified variable to a specified value:

\[ x = 5; \]

This statement tells the compiler to put the integer value 5 into the memory location named \( x \), like so:

\[ x : \boxed{5} \text{ (Address 56436)} \]

So, for example, we might have:

\[ \text{int } x; \]
\[ \downarrow \]
\[ x : \boxed{???????} \text{ (Address 56436)} \]
\[ x = 5; \]
\[ \downarrow \]
\[ x : \boxed{5} \text{ (Address 56436)} \]
\[ x = 12; \]
\[ \downarrow \]
\[ x : \boxed{12} \text{ (Address 56436)} \]

We say “\( x \) is assigned twelve” or “\( x \) gets twelve.”
Variable Assignment Example

```c
#include <stdio.h>

int main ()
{
    /* main */
    /*
     * *********************************************
     * Declaration section *
     */
    /*
     * Local variables *
     */
    /*
     * height_in_cm: my height in cm
     */
    int height_in_cm;
    /*
     * *********************************************
     * Execution section *
     */
    /*
     * Assign the integer value 160 to height_in_cm.
     */
    height_in_cm = 160;
    /*
     * Print height_in_cm to standard output.
     */
    printf("My height is %d cm.\n", height_in_cm);
    /* main */
    /* gcc -o assign assign.c */
    /* assign */
    My height is 160 cm.
}
```

Changing a Variable’s Contents

```c
#include <stdio.h>

int main ()
{
    /* main */
    /*
     * *********************************************
     * Declaration section *
     */
    /*
     * Local variables *
     */
    /*
     * height_in_cm: my height in cm
     */
    int height_in_cm;
    /*
     * *********************************************
     * Execution section *
     */
    /*
     * Assign the integer value 160 to height_in_cm.
     */
    height_in_cm = 160;
    /*
     * Print height_in_cm to standard output.
     */
    printf("My height is %d cm.\n", height_in_cm);
    /*
     * Assign the integer value 200 to height_in_cm.
     */
    height_in_cm = 200;
    /*
     * Print height_in_cm to standard output.
     */
    printf("My height is %d cm.\n", height_in_cm);
    /* main */
    /* gcc -o change change.c */
    /* change */
    My height is 160 cm.
    My height is 200 cm.
```
Variable Initialization

We can *initialize* a variable’s value in the variable declaration:

```c
int x = 5;
```

This statement is the same as declaring `x` and then assigning it 5.

```c
#include <stdio.h>
int main ()
{
    int height_in_cm = 160;
    printf("My height is %d cm.
", height_in_cm);
    return 0;
}
```

C Variable Names

*C identifiers* (including variable names) have the following properties:

- **Constructed using only these characters:**
  - Letters (case sensitive: it matters whether it’s upper or lower)
    ```c
    a b c d e f g h i j k l m
    n o p q r s t u v w x y z
    A B C D E F G H I J K L M
    N O P Q R S T U V W X Y Z
    ```
  - Digits
    ```c
    0 1 2 3 4 5 6 7 8 9
    ```
  - Underscore (*NOTE: NOT hyphen*)
    ```c
    _
    ```
  - The first character is a letter or an underscore:
    ```c
    a123_456 is good, and so is _a123_456 but not 1a23_456
    ```

Since variable names in C can be very long, the number of possible variable names is unbelievably enormous.

**Rule of Thumb for Choosing Variable Names**

A variable name should be so obvious that your favorite professor in your major, even if they know nothing about programming, could immediately tell what the variable name means.
Literal Constants

A constant is a value that cannot change.

So, the difference between a variable and a constant is that a variable’s value can vary, while a constant’s value is constant.

A literal constant is a constant whose value is specified literally:
- int literal constants (e.g., 5, 0, -127, 403298, -385092809)
- float literal constants (e.g., 5.2, 0.0, -127.5, 403298.2348, -3.85092809e+08)
- char literal constants (e.g., 'A', '7', '?')
- character string literal constants (e.g., "A", "Henry", "What's it to ya?")

Note: character string literal constants are sometimes called string literal constants, or just string literals for short.

Example:

```c
#include <stdio.h>
int main ()
/* main */
int income, tax;
printf("I'm going to calculate the federal income
n
tax on your 1997 income.
What was your 1997 income in dollars?

n");
scanf("%f", &income);tax = (income - (4150.0 + 2650.0)) * 0.15;
printf("The 1997 federal income tax on $%2.2fn",
income);
printf(" was $%2.2fn", tax);
/* main */
```

Named Constants

A named constant is a constant that has a name. It’s exactly like a variable, except that its value is set at compile time and CANNOT change at runtime. It’s exactly like a literal constant, except that it HAS A NAME.

In a named constant declaration, we indicate that it’s a constant via the const attribute, and we initialize it:

```c
const float pi = 3.1415926;
```

Example:

```c
#include <stdio.h>
int main ()
/* main */
const float pi = 3.1415926;
const float diameter_factor = 2.0;
const float area_power = 2.0;
circumference = pi * radius * diameter_factor;
area = pi * radius * radius;
printf("The circumference is %f, circumference and area.

n");
printf(" and the area is %f, area.

n", circumference, area);
/* main */
```

Named Constant’s Value Can’t Be Changed

```c
const float pi = 3.1415926;
pi = 3.0;
```

In a named constant declaration, we indicate that it’s a constant via the const attribute, and we initialize it:

```c
const float pi = 3.1415926;
```

Example:

```c
#include <stdio.h>
int main ()
/* main */
const float pi = 3.1415926;
pi = 3.0;
```

Named Constant’s Value Can’t Be Changed

```c
const float pi = 3.1415926;
pi = 3.0;
```

Example:

```c
#include <stdio.h>
int main ()
/* main */
```
Why Literal Constants Are BAD BAD BAD

When you embed literal constants in the body of your program, you make it much harder to maintain and upgrade your program.

```c
#include <stdio.h>

int main ()
{
    float income, tax;
    printf("I'm going to calculate the federal income tax on your 1997 income.\n");
    printf("What was your 1997 income in dollars?\n");
    scanf("%f", &income);
    tax = (income - (4150.0 + 2650.0)) * 0.15;
    printf("The 1997 federal income tax on $%2.2f\n", income);
    printf(" was $%2.2f.\n", tax);
}
```

```
gcc -o tax1997_literal tax1997_literal.c

tax1997_literal
I'm going to calculate the federal income tax on your 1997 income.
What was your 1997 income in dollars? 20000
The 1997 federal income tax on $20000.00 was $1980.00.
```

Why Named Constants Are Good

When you use named constants in the body of your program instead of literal constants, you isolate the constant values in the declaration section, making them trivial to find and to change.

```c
#include <stdio.h>

const float standard_deduction = 4150.0;
const float single_exemption = 2650.0;
const float tax_rate = 0.15;
const int tax_year = 1997;

int main ()
{
    float income, tax;
    printf("I'm going to calculate the federal income tax on your 1997 income.\n");
    printf("What was your 1997 income in dollars?\n");
    scanf("%f", &income);
    tax = (income - (standard_deduction + single_exemption)) * tax_rate;
    printf("The 1997 federal income tax on $%2.2f\n", income);
    printf(" was $%2.2f.\n", tax);
}
```

```
gcc -o tax1997_named tax1997_named.c

tax1997_named
I'm going to calculate the federal income tax on your 1997 income.
What was your 1997 income in dollars? 20000
The 1997 federal income tax on $20000.00 was $1980.00.
```
In C, we can print out multiple pieces of information on a single line of output text:

```c
#include <stdio.h>

int main ()
{
    const float pi = 3.1415926;
    const float diameter_factor = 2.0;
    const float area_power = 2.0;

    float radius, circumference, area;

    printf("I'm going to calculate a circle's
        circumference and area.
        What's the radius of the circle?
        
    ");
    scanf("%f", &radius);

    circumference = pi * radius * diameter_factor;
    area = pi * radius * radius;

    printf("The circumference is %f 
        and the area is %f.
        
    ");
}
```

Look at the last `printf` statement:

```c
printf(" and the area is %f.\n", area);
```

This `printf` statement outputs

- the text " and the area is ", followed by
- the value of the `float` variable named `area`, indicated by the `placeholder` `%f`, followed by
- the text "." followed by
- a carriage return, indicated by `\n`.  

**Output**

```
I'm going to calculate a circle's circumference and area.
What's the radius of the circle?

5
The circumference is 31.415924 and the area is 78.539810.
```

**Specifying the Value of a Variable Via Input from the Keyboard**

```
#include <stdio.h>

int main ()
{
    int height_in_cm;

    printf("What's my height in centimeters?
        
    ");
    scanf("%d", &height_in_cm);

    printf("My height is %d cm.
        
    ");
}
```

```
What's my height in centimeters?
160
My height is 160 cm.
```
Multiple Variables Per `scanf` Statement

C supports inputting multiple variables per `scanf` statement. At runtime, when the user types in the input values, they can separate the individual input values

- by blank spaces, and/or
- by tabs, and/or
- by carriage returns.

```c
#include <stdio.h>

int main ()
{ /* main */
    float average_height_in_m;
    int number_of_silly_people, number_of_toys;
    char middle_initial;

    printf("How many silly people are there in CS1313, and what's their average height in meters?\n");
    scanf("%d %f", &number_of_silly_people, &average_height_in_m);
    printf("There are %d silly people with an average height of %f m.\n", number_of_silly_people, average_height_in_m);

    printf("How many toys do I have, and what is my middle initial?\n");
    scanf("%d %c", &number_of_toys, &middle_initial);
    printf("I have %d toys. My middle initial is %c.\n", number_of_toys, middle_initial);
}
```

Program Variables vs. Algebra Variables

Variables in C (and many other programming languages) look and feel very similar to variables that you deal with in your math classes, from high school algebra on up.

This is on purpose.

The main difference between an algebra variable and a program variable is that a program variable can change its value during a run:

<table>
<thead>
<tr>
<th>Algebra</th>
<th>C</th>
<th>Output</th>
</tr>
</thead>
</table>
| Let \( x \) be 5. \[
\begin{align*}
    x &= 5; \\
    \cdot x &= 5
\end{align*}
\] | \[
\begin{align*}
    x &= 5; \\
    \cdot x &= 5
\end{align*}
\] | 5                 |
| Let \( y \) be 7. \[
\begin{align*}
    y &= 7; \\
    \cdot y &= 7
\end{align*}
\] | \[
\begin{align*}
    y &= 7; \\
    \cdot y &= 7
\end{align*}
\] | 7                 |
| \( z = x + y \) \[
\begin{align*}
    z &= x + y
\end{align*}
\] | \[
\begin{align*}
    z &= x + y
\end{align*}
\] | 12                |
| \( z = x \times y \) \[
\begin{align*}
    z &= x \times y
\end{align*}
\] | \[
\begin{align*}
    z &= x \times y
\end{align*}
\] | 35                |
Programming Exercise

Create a program that:

1. prompts the user for their age in years;
2. inputs the user’s age in years;
3. outputs the user’s age in years.

Begin by drawing a flowchart, and then write the program. The program does not have to have comments. The data type for the weight variable must be appropriate.