Odds & Ends

getchar, putchar, typedefs.

Reading and Writing Characters Using **`getchar`** and **`putchar`**

- For single-character input and output, `getchar` and `putchar` are an alternative to `scanf` and `printf`.
- `putchar` writes a character:

```
putchar(ch);
```

- Each time `getchar` is called, it reads one character, which it returns:

```
ch = getchar();
```

- `getchar` returns an `int` value rather than a `char` value, so `ch` will often have type `int`.
- `getchar` doesn't skip white-space characters as it reads.

Reading and Writing Characters Using **getchar** and **putchar**

- Using `getchar` and `putchar` (rather than `scanf` and `printf`) **saves execution time**.
  - `getchar` and `putchar` are much simpler than `scanf` and `printf`, which are designed to read and write many kinds of data in a variety of formats.
- **Disadvantage** of `getchar` is that the input keeps buffering till enter is pressed.
  - In this process you have to hit enter first to send anything to your program.

Reading and Writing Characters Using **getchar** and **putchar**

- Consider the `scanf` loop used to skip the rest of an input line:

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

- Rewriting this loop using `getchar` gives us the following:

```
do {
  ch = getchar();
} while (ch != '\n');
```

Reading and Writing Characters Using **getchar** and **putchar**

- Moving the call of `getchar` into the controlling expression allows us to condense the loop:

```
while ((ch = getchar()) != '\n')
  ;
```

- The `ch` variable isn't even needed; we can just compare the return value of `getchar` with the new-line character:

```
while (getchar() != '\n')
  ;
```


Reading and Writing Characters Using **getchar** and **putchar**

- `getchar` is useful in loops that skip characters as well as loops that search for characters.
- A statement that uses `getchar` to skip an indefinite number of blank characters:

```
while ((ch = getchar()) == ' ')
  ;
```

- When the loop terminates, `ch` will contain the first nonblank character that `getchar` encountered.

Reading and Writing Characters Using **getchar** and **putchar**

- Be careful when mixing `getchar` and `scanf`.
- `scanf` has a tendency to leave behind characters that it has "peeked" at but not read, including the new-line character:

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

What happens here?

Reading and Writing Characters Using **getchar** and **putchar**

- Be careful when mixing `getchar` and `scanf`.
- `scanf` has a tendency to leave behind characters that it has "peeked" at but not read, including the new-line character:

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

  `scanf` will leave behind any characters that weren't consumed during the reading of `i`, including (but not limited to) the new-line character.
- `getchar` will fetch the first leftover character.

Program: Determining the Length of a Message

- The `length.c` program displays the length of a message entered by the user:

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

- The length includes spaces and punctuation, but not the new-line character at the end of the message.
- We could use either `scanf` or `getchar` to read characters; most C programmers would choose `getchar`.
- `length2.c` is a shorter program that eliminates the variable used to store the character read by `getchar`.

length.c

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  char ch;
  int len = 0;

  printf("Enter a message: ");
  ch = getchar();
  while (ch != '\n') {
    len++;
    ch = getchar();
  }
  printf("Your message was %d character(s) long.\n", len);

  return 0;
}
```

length2.c

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  int len = 0;

  printf("Enter a message: ");
  while (getchar() != '\n'){
    len++;
  }
  printf("Your message was %d character(s) long.\n", len);

  return 0;
}
```


Type Definitions

- The `#define` directive can be used to create a “Boolean type” macro:

  `#define BOOL int`

- There’s a better way using a feature known as a ***type definition:***

  `typedef int Bool;`

- `Bool` can now be used in the same way as the built-in type names.
- Example:

  `Bool flag;    /* same as int flag; */`

Advantages of Type Definitions

- Type definitions can make a program more understandable.
- If the variables `cash_in` and `cash_out` will be used to store dollar amounts, declaring `Dollars` as

  ```
  typedef float Dollars;
  ```

  and then writing

  ```
  Dollars cash_in, cash_out;
  ```

  is more informative than just writing

  ```
  float cash_in, cash_out;
  ```


Advantages of Type Definitions

- Type definitions can also make a program easier to modify.
- To redefine `Dollars` as `double`, only the type definition need be changed:

  ```
  typedef double Dollars;
  ```

- Without the type definition, we would need to locate all `float` variables that store dollar amounts and change their declarations.

Type Definitions and Portability

- Type definitions are an important tool for writing portable programs.
- One of the problems with moving a program from one computer to another is that types may have different ranges on different machines.
- If `i` is an `int` variable, an assignment like

  ```
  i = 100000;
  ```

  is fine on a machine with 32-bit integers, but will fail on a machine with 16-bit integers.

Type Definitions and Portability

- For greater portability, consider using `typedef` to define new names for integer types.
- Suppose that we're writing a program that needs variables capable of storing product quantities in the range 0–50,000.
- We could use **`long`** variables for this purpose, but we'd rather use `int` variables, since arithmetic on **`int`** values may be **faster** than operations on `long` values. Also, `int` variables may take up **less space**.

Type Definitions and Portability

- Instead of using the `int` type to declare quantity variables, we can define our own "quantity" type:

  `typedef int Quantity;`

  and use this type to declare variables:

  `Quantity q;`
- When we transport the program to a machine with shorter integers, we'll change the type definition:

  `typedef long Quantity;`
- Note that changing the definition of `Quantity` may affect the way `Quantity` variables are used.

17

The `sizeof` Operator

- The value of the expression

  `sizeof (` ***type-name*** `)`

  is an unsigned integer representing the number of bytes required to store a value belonging to *type-name*.
  - `sizeof(char)` is always 1, but the sizes of the other types may vary.
  - On a 32-bit machine, `sizeof(int)` is normally 4.
- The `sizeof` operator can also be applied to **constants**, **variables**, and **expressions** in general.
  - If `i` and `j` are `int` variables, then `sizeof(i)` is 4 on a 32-bit machine, as is `sizeof(i + j)`.

18

Arrays

Array Initialization

- An array, like any other variable, can be given an initial value at the time it's declared.
- The most common form of ***array initializer*** is a list of constant expressions enclosed in braces and separated by commas:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Array Initialization

- If the initializer is shorter than the array, the remaining elements of the array are given the value 0:

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1,2,3,4,5,6,0,0,0,0} */
```

- Using this feature, we can easily initialize an array to all zeros:

```
int a[10] = {0};
/* initial value of a is {0,0,0,0,0,0,0,0,0,0} */
```

  There's a single 0 inside the braces because **it's illegal** for an initializer to be **completely empty**.

- It's also illegal for an initializer to be **longer than the array** it initializes.

Array Initialization

- If an initializer is present, the length of the array may be omitted:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- The compiler uses the length of the initializer to determine how long the array is.

Designated Initializers (C99)

- It's often the case that relatively few elements of an array need to be initialized explicitly; the other elements can be given default values.
- An example:

```
int a[15] =
  {0, 0, 29, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 48};
```

- For a large array, writing an initializer in this fashion is tedious and error-prone.

Designated Initializers (C99)

- C99's ***designated initializers*** can be used to solve this problem.
- Here's how we could redo the previous example using a designated initializer:

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

- Each number in brackets is said to be a ***designator.***

Designated Initializers (C99)

- Designated initializers are shorter and easier to read (at least for some arrays).
- Also, the order in which the elements are listed no longer matters.
- Another way to write the previous example:

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```


Designated Initializers (C99)

- If the array being initialized has length *n*, each designator must be between 0 and *n* – 1.
- If the length of the array is omitted, a designator can be any nonnegative integer.
  - The compiler will deduce the length of the array from the largest designator.
- The following array will have 24 elements:

```
int b[] = {[5] = 10, [23] = 13, [11] = 36, [15] = 29};
```

Designated Initializers (C99)

- An initializer may use both the older (element-by-element) technique and the newer (designated) technique:

```
int c[10] = {5, 1, 9, [4]=3, 7, 2, [8]=6};
```

- Output:

```
c[10] = {5, 1,9,0,3,7,2,0,6,0};
```

An online IDE https://ide.geeksforgeeks.org/index.php