Dictionary

*Note: I usually put a space between () to make it look nicer, e.g., ( ). However, there shouldn't be a space. The same can be said for {} and { }, or [] and [ ].*

**Key**

* Section Heading
* General comment about the heading
* Further explanation about the "General comment"

* Snippets of code (examples)
* General formatting of code
* Explanation of snippets of code
* Additional remark

* Miscellaneous

Saturday, October 3, 2015

4:08 PM

* **Terminology**
  + Sentinel value
    - A special value in the context of an algorithm which uses its presence as a condition of termination, typically in a loop or recursive algorithm.

* **Layout of C Programs**
  + pre-processor directives

global declarations

main( )

{

local variables to function main ;

statements associated with function main ;

}

f1( )

{

local variables to function 1 ;

statements associated with function 1 ;

}

f2 ( )

{

local variables to store function f2 ;

statements associated with function 2 ;

}

.

.

.

etc.

* **Brackets and Punctuation**
  + ( ) are used in conjunction with function names.
  + { } are used to delimit the C statements that are associated with that function.
  + ; is used to terminate C statements
    - C is a free format language and long statements can be continued, without truncation, onto the next line. The semicolon informs the C compiler that the end of the statement has been reached.
      * Free format also means that you can add as many spaces as you like to improve the look of your programs.

* **Basic Functions**
  + Nearly everything in C is a function.
  + int main( )

{

<insert code>

* Basically, code goes there. No angle brackets are needed.

}

* int refers to the function's return type.
  + e.g. return 0;
* main is the name of the function.

* **Commenting**
  + //single-line commenting
  + /\*Multi-line

commenting\*/

* **Preprocessor Directives**
  + Indicated by a # sign.
    - e.g. #include <name\_of\_library>
      * Tells the compiler to include the given file before compilation.
      * .h indicates a header file.
    - e.g. #include "filename"
      * Includes a file that isn't in C's standard library

* **Constants**
  + Generally used for values that never change.
    - e.g. the value of pi is always 3.14159
  + Examples of constants and their formatting are underneath.
  + #define <NAME\_IN\_CAPITAL> <value>
    - e.g. #define FEET\_IN\_YARD 3
      * No semi-colon at the end.
    - What is #define?
      * Searches and replaces every occurrence of the constant name with the constant value that follows its before sending the program to the compiler.
        + i.e. Replaces a given variable name with a value, even before the program is compiled.
  + const int YARDS\_IN\_MILE = 1760;
    - Alternative syntax for C 99.

* **Variables**
  + Variables are used to store and manipulate data.
  + Variables have to be specified as a certain data type:
    - i.e. see "Primitive Data Types" below
  + Global
    - A variable that is defined outside of all functions, including main.
      * i.e. along with the constants and libraries
    - All functions can use variables in the global scope.
  + Local
    - A variable that is defined inside a function.
    - Only that function can use that variable.

* **Primitive Data Types**
  + char is the data type for strings.
    - e.g. char name[14] = "Bucky Roberts";
    - Format code is %c for individual characters
    - Format code is %s for entire strings
      * Those strings have to be stored as array
      * s.
  + double is the data type to store decimal numbers to about 13 digits of precision.
    - Doubles take eight bytes to store.
    - Rounds last digit.
    - e.g. double costPerOunce;
    - Format code is %lf
      * %f also works for printf
      * lf stands for "large float"
  + float is the data type to store decimal numbers to about 6 digits of precision.
    - Floats take four bytes to store.
    - Rounds last digit.
    - e.g. float costPerOunce;
    - Format code is %f
  + int is the data type to store integers.
    - e.g. int numDogs;
    - Format code is %d
      * Most likely stands for "digit"

* **Operators**
  + =
    - Assignment operator.
      * e.g. int tuna = 14;
        + Sets the integer variable "tuna" equal to 14, i.e., assigns "tuna" to 14.
  + +, -, \*, /
    - In order from left to right: addition, subtraction, multiplication, and division.

* %
  + Modulo operator to find remainder from division.
    - e.g. (10 % 6) would yield a value of 4.
    - e.g. (4 % 20) would yield a value of 4
  + This operator is ONLY defined for integer operands.

* ==
  + "Equal to."
    - e.g. (3 \* 2 == 6);
* !=
  + "Not equal to."
* >, <
  + "Greater than" and "less than" operators.
* >=, <=
  + In order from left to right: Greater than or equal to, Less than or equal to.

* **Operands**
  + To increment:
    - x++;
    - ++x;
    - x = x + 1;
    - Raw numbers cannot be incremented.
      * 7++
        + Not a valid statement in C.
    - Only variables can be incremented.
    - However, only one variable can be used with the increment operand.
  + To decrement:
    - x--;
    - --x;
    - x = x - 1;

* **Shortcuts**
  + x += 10;
    - i.e. x = x + 10;
  + x \*= (3 + y);
    - i.e. x = x \* (3 + y);

* **Arithmetic Expressions**
  + Consist of numeric literals, arithmetic operators, and numeric variables

* **Order of Operations**
  1. Parentheses
     1. Innermost
  2. Function Call
  3. Cast
  4. Multiplication/Division/Modulo
  5. Addition/Subtraction
  6. Assignment to variable

* **Integer Division**
  + The quotient will always be **rounded down** to the nearest whole integer.
    - Technically, the quotient is truncated. It doesn't actually round down; the portion after the decimal is dropped.
    - e.g. 13/4 evaluates to 3.

* **Conflicting Data Types**
  + double + double = double
  + integer + integer = integer
  + double + integer = double
  + float + integer = float

* float variable1 = integer variable2 / integer variable3
  + The integer variables on the right-hand side will compute to an integer. And then, the float will store the integer with .000000 after the integer.

* int x = 8;

double y = x/3;

* First, the expression on the right-hand side of the equal sign will be evaluated to 2 because x is an integer.
* Then, the variable on the left-hand side will set equal to the value on the right side, and because y is a large float (i.e. double), y will store the value as 2.0.

* int x = 8;

double y = x/3.0;

* Although x is an integer, it is being divided by 3.0 which is a float. Therefore, the value 2.666667 would be stored in y.

* The key takeaway here is whenever a float/double and an integer mixes, the result will be a float/double, since the program would prefer to be more precise.
  + It is highly probable that if a float and a double mixes, the value would take on that of a double unless specified otherwise.
    - Such as if the left-hand side is specified as a float or integer.

* **Casting**
  + A cast is when you want an expression to temporarily be considered as a different type than it is.
  + The syntax for a cast is to put the type in parentheses right in front of the expression that is being casted.
  + int x = 8;

double y = (double) x/3;

* Note: The order of operations of a cast is highefar than division.
* The type of x does NOT change. But, for the purposes of the division, x ACTS as a double.

* int x = 8;

double y = (double) (x/3);

* y would have been set to 2.0 because the division would have occurred before the cast.
* Note: The order of operations of a cast is lower than parentheses.
  + Parentheses rule all.

* **Condition-Checking Statements and Boolean Expressions and Values**
  + If
  + Else

* **File I/O**
  + FILE \*fp;
    - fp = "file name"
      * Actually stands for "file pointer"
      * e.g. FILE \* fp\_input = fopen("input.txt", "r");
      * e.g. FILE \* fp\_output = fopen("output.txt", "w");
      * "a" means append
  + fprintf(<fileptr>, <format>, <vars>);
  + fscanf(<fileptr>, <format>, &<vars>);
    - e.g. fscanf(fp\_input, "%d", &read[i]);

* **Loops**
  + break;
    - Will break you out of the innermost loop in which the 'break' statement resides
  + continue;
    - Will skip the following code with the innermost loop in which the 'continue' statement resides, returning the flow of control to the top of that
  + for (int i = 0, n = strlen(s); i < n; i++)

{

//Code.

}

* This is an example of a more complex declaration of the 'for loop' as this includes a separate variable that we declare within the parentheses.
* We don't need curly braces if we only have one line of code.

* **One-Dimensional Arrays**
  + An array is a data structure that holds a number of related variables.
    - Thus, an array has a size which is the number of variables it can store.
      * All of these variables must be of the same type.
  + type <variable>[size];
    - int numbers[10];
      * This is an integer array called "numbers" of size 10.
      * This array has 10 "bins" to store integers.
        + For example, number[0] = 20;

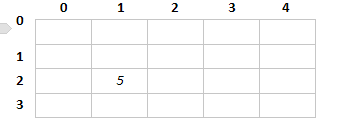
…

number[9] = number[0] + 2;

* Note that number[10] does not exist since arrays begin counting from zero.
* In other words, an array is a group of variables. The array "numbers" has 10 variables.
* All arrays are pointers, but not all pointers are arrays.
* fscanf(fp\_input, "%d", &numGuesses);

int read[numGuesses];

* Certain compilers will not allow the user to declare arrays with variables.
* The solution is to use special functions such as *malloc, calloc, realloc, or free* to dynamically allocate space for the array.
  + malloc is short for memory allocation.
* In short, this snippet of code is inconsistent, unreliable, and subsequently incorrect.

* **Two-Dimensional Arrays**
  + type <variable>[size1][size2];
    - int grid[4][5];
      * This code will create an array grid-table with 4 rows down and 5 columns across.
    - grid[2][1] = 5;
      * 

* Two-dimensional arrays are particularly useful for modeling Pascal's Triangle or Tic-Tac-Toe.

* **Strings**
  + A string is simply a sequence of characters strung together. In other words, a string is equivalent to char\*, which refers to the address of the beginning of a sequence of characters, which is also why strings cannot be compared with two equal signs because strings are set equal to the value of the address in memory rather than the data at that address.
  + All strings are *one-dimensional arrays* of single characters.
  + When taking a string input from a user, it's a good idea to check if the string is null in case the user has typed an atrocious number of characters that the system has no memory for. Alternatively, the system may not have the memory to store the string if the program is doing way too many things at once and eating up all of that memory.
    - Null is equal to byte/address 0 in RAM, so there cannot be legit data at address 0.
      * We can return 1 in these cases to signal that the program has gone wrong.
  + There are four ways to declare a string variable:
    - **1.** char <variable>[size];
      * char name[20];
        + Each bin will store a character, as specified by the user.

There are many different ways to do this, besides setting each individual bin equal to a single character.

* + - * char name[14] = "Bucky Roberts";
        + "Bucky Roberts" has 13 characters. However, the memory cost is 14 bytes due to the string terminator, which can be represented as:

\0

This is often dubbed as "null 0"

* + - * + Therefore, the array must contain at least 14 bins.
        + Note that it is still valid to use an array with more than 14 bins for the string "Bucky Roberts" since the remaining unfilled bins will not be used due to the string terminator.
        + However, it appears this method does not work for some compilers. Thus, for this example, the proper way to initialize a string variable:

char name[14] = {'B', 'u', 'c', 'k', 'y', ' ', 'R', 'o', 'b', 'e', 'r', 't', 's'};

* char\* <variable>;
  + See explanation above.

* **2.** scanf("%s%s%s", variable1, variable2, variable3)
  + scanf("%s", word);
  + Note that an ampersand is unneeded here because an array variable is really a memory address in C already.

* **3.** strcpy

* **Functions**

* **Prototyping a Function**
  + Do it above the first function along with #include <library>
  + Include its entirety
    - e.g. void convertDollars(float euro);
  + Alternatively, we don't need to prototype a function if we write the function before main. Sometimes, this is impossible to do however.

* int main( )

{

float euroPrice1 = 1.00;

float euroPrice2 = 5.00;

convertDollars(euroPrice1);

* Tells main that instead of stopping at return 0, move onto the convertDollars( ) function.
  + i.e. "calls" convertDollars( ) to the parameter euroPrice1, or 1.00.
    - Passes the argument euroPrice1 to a function.
  + Observation: Prototyping a function includes the data type. Calling a function does not.
* If there are two of these, then perform the tasks of that function twice.
  + e.g.:
    - convertDollars( );
    - convertDollars( );

convertDollars(euroPrice2);

return 0;

}

* int refers to the function's return type.
  + e.g. return 0;
  + e.g. If bonus = 5 \* int numYearsWorked, then:
    - return bonus;
* main is the name of the function.
* void convertDollars(float euro)

{

<insert code>

float usd = euro \* 1.37;

printf("%.2f Euros = %.2f USD\n", euro, usd);

return;

}

* void functions do not return anything.
  + However, it does need to return to the rest of the program.
* convertDollars is the name of the function (any name is okay).

* **Pointers**
  + Primitive data type.
  + Special type of variable that stores a memory address.
    - e.g. int\* p;

* **Pointer Arithmetic**
  + for (int i = 0, n = strlen(s); i <= n; i++)

\*(t+i) = \*(s+i);

* The nested assignment is equivalent to t[i] = s[i]. This code is used to copy over character values of an array or string.

* **Pass by Value vs. Pass by Reference**
  + Pass by Reference is also known as "Pass by Address."
  + Passing by Reference actually changes the variable.

* **Structures**
  + Structs differ from arrays in the sense that structs can be used to store multiple different data types.
    - Structs are particularly useful for storing employee information, in regards to ID (integer), name (string), age (integer), and the such.
    - Note that both arrays and structs are structures, but structs and structures are generally informally interexchangeable in conversations.
      * Typically, structures often refer to structs whereas arrays are simply called arrays.

* Typically, structs are declared near the top of the file, right after any #includes and #defines, but before any function prototypes.
  + The reason for this is that the function prototypes might reference the struct.
    - int read\_data(struct Shift shift\_data[ ]);
      * This function reads data from an input file into a structure called Shift.

* **To declare a struct:**
  + struct <Struct\_Name>

{

<type1> <var1>;

<type2> <var2>;

<type3> <var3>;

…

<typeN> <varN>;

};

* It should be noted that I added 3 spaces between <type> and <var> to clearly show spacing. In reality, there should only be one space.
* It should also be noted that there is a semicolon at the end of a structure.

* struct Employee

{

char first[30];

char last[30];

int ID;

double hourlyPay;

};

* **Alternatively:**
  + typedef struct

{

char first[30];

char last[30];

int ID;

double hourlyPay;

} employee\_t;

* Therefore, we can now use employee\_t to refer to struct rather than struct employee
* int equal(node\* listA, node\* listB)
  + as opposed to…
* int equal (struct node\* listA, struct node\* listB)

* **To access components of a struct variable:**
  + Once a struct is declared at the top of the file, a variable of the type of struct must be declared in a function in order for the struct to be used.
  + struct <Struct\_Name> <struct\_variable>;
    - This line is usually included as either a parameter before the other regular parameters, or at the top of a function after the regular variables declared for each function.
      * void print\_data(struct Struct shift[ ]; int numShifts);

* int main (void)

{

//Variables declared.

int numShifts;

struct Struct shift\_data[300];

…

return 0;

}

* No semicolon is needed at the end of a function, as opposed to the syntax for structs.
* The second type of variable of type struct declaration is *somewhat* similar to calling a function on a single line.

* struct Employee worker;
  + A different example that creates a variable of type struct employee.
  + After this line of code executes, here is a picture of what gets allocated:

Machine generated alternative text:
worker 
first 
hourlypay 

* The initial variables in the struct are called *components.*
* first, last, ID, and hourlyPay are all *components* of the struct *employee*

* **To access the individual components of a struct variable:**
  + The dot operator must be used.
    - That is, a period. Dot operator is a single dot/period, e.g., "." (without any quotation marks).
  + <struct\_variable>.<component>
  + worker.ID = 526;

* <var>.<component>

* **Linked Lists**
  + Linked lists are a way to store data with structures so that the programmer can automatically create a new place to store data whenever necessary.
    - i.e., a linked list allows the user to easily add a single piece of data to an existing piece of data while only allocating memory for the new piece of data.
  + Linked lists are the simplest form of a linked structure.
  + A linked list consists of a chain of data locations called *nodes*.
  + Each node holds a piece of information AND a link to the next node.

* **System Calls**
  + system("PAUSE");
    - Tells the program to halt. Note that "PAUSE" does not have to be capitalized. To be perfectly honest, this is a stupid line of code.

**Key**

* Section Heading
* General comment about the heading
* Further explanation about the "General comment"

* Snippets of code (examples)
* General formatting of code
* Explanation of snippets of code
* Additional remark

* Miscellaneous