# Landmarks and Landmines of C

*Some of this handout written by Robert Plummer.*

At the beginning here, we're going to move impossibly fast (whereas later we will only move ridiculously fast). For Friday's class, you should carefully read all of Chapter 1 on your own and be prepared to start dissecting and manipulating a variety of simple C programs. We won't teach the basic C syntax and constructs, just highlight some of the more unusual features together. This handout contains a smattering of facts about the C language, mentions some of the basics, and points out a few pitfalls to avoid.

## Most C programs have the same component sections

—A comment explaining what the program does

This is a good programming practice that makes your code more accessible to the reader (who might be you!)

—Library inclusions

These `#include` statements let the compiler know what libraries you want to use functions from. Many standard libraries come with all C compilers. You will also use libraries you write yourself, or in this course, libraries that we provide, such as **`genlib`** or **`simpio`**.

—Constant definitions

By using the `#define` mechanism to define constants, you make your program more readable and avoid "magic numbers". You also make it easy to change the value of the constant.

—Function prototypes

In C, the normal style is to write the code for functions after the main program. This means that functions are often called before the code appears. Function prototypes let the compiler compile the calls correctly and make sure they are consistent with the function code.

—Main program

The main "program" is actually a function called **`main`**. A prototype for this function is not required. Execution of your program begins with the first line of the main function. The text book begins the function **`main`** with

```
main()
```

That is what CodeWarrior expects, but Visual C++ wants you to write

```
void main(void)
```

—Function definitions

All programs worth writing involve breaking the code into functions, so that no part of the program becomes large and intractable. Good decomposition leads to code that is clear, logical, and easy to understand.

## Variables must be declared before use

Variables are local to the function in which they are declared. We will use global variables (declared outside any function) only rarely in this course.

## All values have a "type", and every variable has a declared type

C has standard data types for storing integers, floating point numbers, and characters (see text for details). The CS106-specific `genlib` library adds data types called **`string`** and **`bool`**.

### printf is used for output and has lots of formatting options

Each variable type has its format code, along with options for controlling the alignment, number of decimal places, etc. See table on p. 19 of the text. Only bother memorizing the basics of what you need, learn the more obscure options on a "need to know" basis.

### The simpio functions read input

The **`GetLine`**, **`GetInteger`**, and **`GetReal`** functions are simple-to-use routines that read one piece of data from the user. These are CS106-specific functions, not standard C, but save you from having to deal with the messy features of input until later in the course.

### Expressions evaluate using rules of precedence and associativity

**`a + b * c`** does the multiplication first because of precedence, and **`a - b - c`** subtracts **`b`** from **`a`** first because of associativity. You can write **`x = y = z = 0`** in C because (1) an assignment is an expression that has as its result the value assigned, and (2) the assignment operator associates to the right.

### In expressions of mixed types, values are promoted to the richer type

For example, if an int **`score`** is multiplied by a double **`curve`**, the value of the int is converted to type double before the multiplication. This does not affect the value of **`score`**, just the outcome of the multiplication.

### Assignment does a type conversion if necessary

The following code

```
int i;
i = 2.9;
```

assigns **`2`** to **`i`** since the type is converted and floating point numbers are converted to ints by truncating the fractional component.

### Integer division truncates

**`29/10`** evaluates to **`2.`** If you really want a fractional result from the division, change the expression so at least one of the operands is of floating-point type (either by appending a .0 to an integer constant expression or using a typecast to float or double).

### C has arithmetic shorthand operators

Be sure you understand the meaning of the following representative expressions that exhibit some of the available shorthands:

```
y = a + x++;
z = ++x + a;
salary *= payIncrement;
```

### Logical operators use "short-circuit" evaluation

The logical connectives and/or evaluate left-to-right and stop as soon as the overall truth result can be determined. This can be taken advantage of to compute quick tests before the more lengthy ones, guard against division by zero, etc.

```
if (y != 0 && x % y == 0)
```

### Don't confuse boolean logic with bitwise

If you mistakenly use **&** where you meant **&&**, the result might be a valid expression and compile (surprise!) but not do what you want. The single **&** and **|** are bitwise operators, the double **&&** and **||** are logical operators. We will not use the bitwise operators much in this class.

### Statements end with semicolons and are grouped with curly braces.

Putting a set of statements in curly braces creates a compound statement or a block. braces There is more than one convention for the placement of brackets. Some people prefer

```
if (x < 0)
{
    y = x;
    z = 0;
}
```

Others would write

```
if (x < 0) {
    y = x;
    z = 0;
}
```

Take your pick, but be consistent within any given program! The amount of indentation is also something that is up to you, but again, should be consistently applied.

### Any non-zero expression is true

C does not have an official Boolean type (**bool** is an addition from the CS106 **genlib**) and the language considers any non-zero expression to be true, and any zero expression to be false. This means it is possible to just write:

```
if (x) {
```

Which mean to the same thing as:

```
if (x != 0) {
```

Pay close attention to this until it becomes second nature. Something that is "wrong" in C often compiles but produces unexpected results. Writing **=** when you meant **==** is a common mistake to watch for.

### if and switch are conditionals

The test must always be enclosed in parenthesis. A sample **if**:

```
if (time != 0) rate = distance / time;
```

The **if** statement has an optional **else**. You can string if/elses together in something known as a "cascading if":

```
if (rank == 1) {
    printf("Gold medal.\n");
    points = 10;
} else if (rank == 2) {
    printf("Silver medal.\n");
    points = 5;
} else if (rank == 3) {
    printf("Bronze medal.\n");
    points = 2;
} else {
    printf("Consolation prize.\n");
```

```
        points = 1;
}
```

A **switch** statement can be used to route control to different cases based on matching an integer to a specific set of values. Re-writing the above using switch:

```
switch(rank)
  {
     case 1:
           printf("Gold medal.\n");
           points = 10;
           break;
     case 2:
           printf("Silver medal.\n");
           points = 5;
           break;
     case 3:
           printf("Bronze medal.\n");
           points = 3;
           break;
     default:
           printf("Consolation prize.\n");
           points = 1;
           break;
  }
```

Be sure you understand why the **break** is there and what happens if you leave it out!

The **switch** does not allow for ranges or more complicated tests. The following cannot be converted to a **switch**, since we are testing for ranges of values and the variable involved is of type **double**:

```
double score;
string letterGrade;

score = GetReal();

if (score > 90)
    letterGrade = "A";
else if (score > 80)
    letterGrade = "B";
else if (score > 70)
    letterGrade = "C";
else
    letterGrade = "You don't want to know.";
```

### Loops are for and while and do-while

The **for** loop is mostly common use to iterate forward over a sequence of numbers. Changing the initialization or increment portion of the loop allows you to create down-to loops, loop by 2s, etc.

Common loop pattern :

```
sum = 0;
for (i = 0; i < 10; i++)
   sum += i;
```

We can also rewrite this using **while** instead of **for**:

```
sum = 0;
i = 0;
while (i < 10) {
```

```
      sum += i;
      i++;
   }
```

In general, any **for** loop can be re-written into a **while** and vice versa. It is usually preferable to use a **for** loop for straightforward iterative tasks and **while** for those of indefinite iteration.

A **do-while** loop is similar to an ordinary **while** except that the test is performed at the bottom of the loop. It is used rarely, in those situations where you are sure the loop body needs to execute at least once.

```
do {
   printf("Your answer? ");
   response = GetInteger();
} while (response != correctAnswer)
```

### while(TRUE) with break handles the loop-and-a-half problem

Rather than

```
get a value from user
while (value != sentinelValue) {
   process the value
   get a value from user
}
```

we can write

```
while (TRUE) {
   get a value from user
   if (value == sentinel) break;
   process the value
 }
```

The **break** statement can be used to exit any of the three loop types. If there are nested loops, a **break** only exits the inner loop.