CS2100 AY24/24 LT11

C programming

Syntax

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159  // No ";"
double circle_area(double);
int main(void) {
    rim_area = circle_area(d2) – circle_area(d1);
    volume = rim_area * thickness;
    printf("Volume = %.2f\n", volume);
    return 0;
}
double circle_area(double diameter) {
    return PI * pow(diameter/2, 2);
}
```

- **No initialization**: Warning, uninitialized in this function.

Data Types

int: 4 bytes (in sunfire), -2,147,483,648 (-2^{31}) through +2,147,483,647 ($2^{31} – 1$)

float or **double**: 4 bytes for float and 8 bytes for double (in sunfire)

NO Boolean type in ANSI C. Instead, we use integers:

- 0 to represent false
- Any other value to represent true (1 is used as the representative value for true in output)

Name of a variable or function

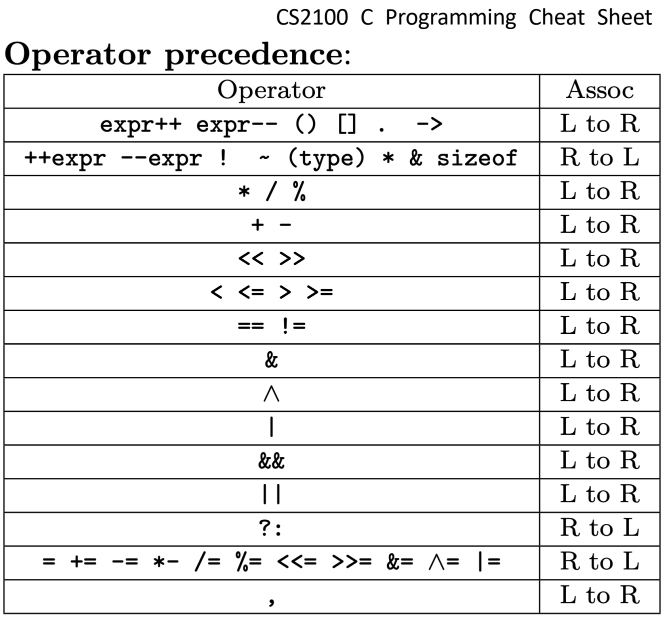
- May consist of letters (a-z, A-Z), digits (0-9) and underscores (_), but MUST NOT begin with a digit
- Must not be reserved words (i.e. int, void, return)
- Should avoid standard identifiers (i.e. Names of common functions, such as printf, scanf)

valid	**invalid**
maxEntries	1Letter
_X123	double
this_IS_a_long_name	joe's
	T*S
	ice cream

Side effect: An assignment statement does not just assigns, it also has the side effect of returning the value of its right-hand side expression

- Side effects have their use, but **avoid convoluted codes**, i.e. **a = 5 + (b = 10); // assign 10 to b, and 15 to a**
- rise **warning** as && is prior than ||, we need to add ().

```
int x, y, z, a = 4, b = -2, c = 0;
x = (a > b || b > c && a == b)
```



CS2100 C Programming Cheat Sheet

Operator precedence:

Operator	Assoc
expr++ expr-- () [] . ->	L to R
++expr --expr ! ~ (type) * & sizeof	R to L
* / %	L to R
+ -	L to R
<< >>	L to R
< <= > >=	L to R
== !=	L to R
&	L to R
∧	L to R
|	L to R
&&	L to R
||	L to R
?:	R to L
= += -= *- /= %= <<= >>= &= ∧= |=	R to L
,	L to R

Short-circuit evaluation

- **expr1 || expr2**: If expr1 is true, skip evaluating expr2 and return true immediately, as the result will always be true.
- **expr1 && expr2**: If expr1 is false, skip evaluating expr2 and return false immediately, as the result will always be false.

Compute: [caution] Round up of negative division and round down for positive division.

int r = 10 / 4.0 -> r = 2

int r = -10/ 4.0 -> r = -2

Loop [do while vs while]

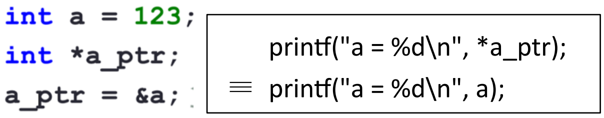
int num = 1;
while (num < 1) {
    num++;
}
printf("%d\n", num);
return 0;
	int num = 1;
	do {
	    num++;
	} while (num < 1);
	printf("%d\n", num);
	return 0;

Print **1**	Print **2**

- do while first perform operation and then check the condition. It will run at least once.
- while check condition before operation, it will run at least 0 time

Pointer

- refer to the address of a variable by using the address operator &
- %p is used as the format specifier for addresses
- Addresses are printed out in hexadecimal format
- The address of a variable **varies** from run to run, as the system allocates **any free memory** to the variable



```
int a = 123;
int *a_ptr;
a_ptr = &a;
```

```
    printf("a = %d\n", *a_ptr);
≡   printf("a = %d\n", a);
```

Once we make a_ptr points to a (as shown above), we can now access a directly as usual, or indirectly through a_ptr by using the indirection operator (also called dereferencing operator) *

Print pointer: printf("%p\n", a_ptr);

- Pass the addresses of two or more variables to a function so that the function can pass back to its caller new values for the variables
- Pass the address of the first element of an array to a function so that the function can access all elements in the array

Function

- A function **prototype** includes only the function's **return type**, the function's **name**, and the **data types** of the parameters (**names of parameters are optional**).
- put function prototypes at the top of the program, before the main() function, to inform the compiler of the functions that your program may use and their return types and parameter types.
- Function definitions to follow after the main() function.
- Without function prototypes, you will get **error/warning** messages from the compiler.

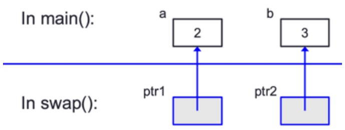
Scope Rule: Local parameters and variables are only accessible in the function they are declared

- Arguments from a caller are **passed by value** to a function's parameters.

Function invocation process

- When a function is called, an activation record is created in the call stack, and memory is allocated for the local parameters and variables of the function.
- Once the function is done, the activation record is removed, and memory allocated for the local parameters and variables is **released**.
- Hence, local parameters and variables of a function exist in memory *only during the execution of the function*. They are called automatic variables.
- In contrast, **static** variables exist in the memory even after the function is executed

Use **pointers** to modify value of a variable outside its scope.



Array

1. Initialize

an array can be initialized at the time of declaration

```
// a[0]=54, a[1]=9, a[2]=10
int a[3] = {54, 9, 10};
// b[0]=1, b[1]=2, b[2]=3
int b[] = {1, 2, 3};
// c[0]=17, c[1]=3, c[2]=10, c[3]=0, c[4]=0
int c[5] = {17, 3, 10};
```

Invalid initialize:

```
// warning issued: excess elements
int e[2] = {1, 2, 3};
// too late to do this;
// compilation error
int f[5];
f[5] = {8, 23, 12, -3, 6};
```

2. Assignment

```
#define N 10
int source[N] = { 10, 20, 30, 40, 50 };
int dest[N];
dest = source;  // illegal!
```

- An array name is a **fixed (constant) pointer**; it points to the first element of the array, and this cannot be altered.
- The code above attempts to alter **dest** to make it point elsewhere.

3. Function

Function prototype: both of the following are acceptable and equivalent

```
int sumArray(int [], int);
int sumArray(int arr[], int size);
// Use pointer
int sumArray(int *, int);
int sumArray(int *arr, int size) {
   ... // definition
}
```

- **No need** to put array size inside [ ]; even if array size is present, compiler just **ignores** it.
- Provide the array size through another parameter

Scope: function can modify the content of the array it received.

String

- A string is an array of characters, terminated by a null character ‘\0’ (which has an ASCII value of zero)
- Must have null character, otherwise, string functions will not work properly on it.
  (a) printf(“%s”, str) statement will print until it encounters a null character in str.
  (b) strlen(str) will count the number of characters up to (but not including) the null character.
  (c) result in illegal access of memory.
- Use string functions (include <string.h>) to manipulate strings

1. initialize

```
char str[6];
str[0] = 'e';
str[1] = 'g';
str[2] = 'g';
str[3] = '\0';
// Do not need ‘\0’
// it is automatically added
char name[] = "apple";
char name[] = {'a','p','p','l','e','\0'};
```

2. IO

Read string from stdin (keyboard)

```
// reads size – 1 char, or until newline
fgets(str, size, stdin)
// reads until white space
scanf("%s", str);
```

- fgets() also reads in the newline character. Hence, we may need to replace it with '\0' if necessary.

```
#include <stdio.h>
#define LENGTH 10
int main(void) { // V0.1
      char str[LENGTH];
      printf("Enter string:”);
      scanf("%s", str);
      printf("%s\n", str);
      return 0;
}
int main(void) { // V0.2
      char str[LENGTH];
      printf("Enter string:”);
      fgets(str, LENGTH, stdin);
      puts(str);
      return 0;
}
```

Input: My book

Output:

V0.1: My
V0.2: My book

Print string to stdout (monitor)

```
Print string to stdout (monitor)
puts(str); // terminates with newline
printf("%s\n", str);
```

String Function

- strlen(s): Return the number of characters in s
- strcmp(s1, s2): Compare the ASCII values of the corresponding characters in strings s1 and s2. Return
  - a negative integer if s1 is lexicographically less than s2, or
  - a positive integer if s1 is lexicographically greater than s2, or
  - 0 if s1 and s2 are equal.
- strncmp(s1, s2, n): Compare first n characters of s1 and s2.
- strcpy(s1, s2): Copy the string pointed to by s2 into array pointed to by s1. Returns s1.
- strncpy(s1, s2, n): Copy first n characters of string pointed to by s2 to s1.

Struct

The following is a definition of a type, NOT a declaration of a variable

- A type needs to be defined before we can declare variable of that type
- **No** memory is allocated to a type

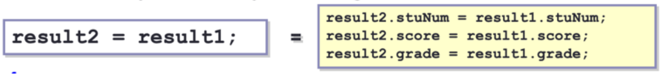
```
typedef struct {
      int day, month, year;
} date_t;
typedef struct {
      int cardNum;
      date_t expiryDate;
} card_t;
// This semi-colon ; is very important and
is often forgotten!
```

Initialize: card_t card1 = {888888, {31, 12, 2020}};

Reading:

```
result_t result1;
scanf("%d %f %c", &result1.stuNum,
                  &result1.score,
                  &result1.grade);
```

- If we use the structure variable's name, we are referring to the entire structure.
- Unlike arrays, we **may do assignments** with structures



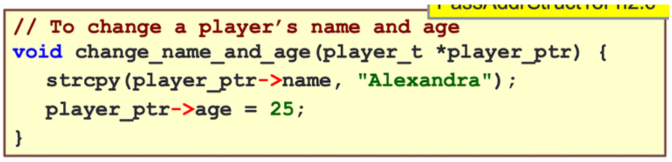
```
result2 = result1;
```

=

```
result2.stuNum = result1.stuNum;
result2.score = result1.score;
result2.grade = result1.grade;
```

Scope: Passing Structure to Function [Pass by value]

- The entire structure is **copied**
- members of the actual parameter are **copied** into the corresponding members of the formal parameter.
- The **original** structure variable **will not** be modified by the function.
- To modify original structure we need to pass **address**



```
// To change a player's name and age
void change_name_and_age(player_t *player_ptr) {
   strcpy(player_ptr->name, "Alexandra");
   player_ptr->age = 25;
}
```

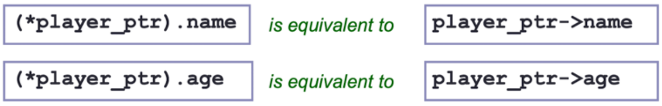
- If struct **has an array**, when **passing by value,** array will be **copied by value** instead by address. Hence, modify element in array will not cause array value change out of the function. This also apply to the situation when the struct has a struct!
- If struct has a **pointer**, when passing by value, stuct will be copied, then the pointer in stuct will be copied by value as well. However, as pointer store the address of a variable. The copy variable will point to the same variable. Hence, change in function will reflect on the pointed variable out of function!

Example:

```
typedef struct {
    int a;
    int arr[3];
    int *ptr;
} Inner;

typedef struct {
    Inner inner;
    int x;
} Outer;

void modify(Outer s) {
    s.x = 50;
    s.inner.a = 10;
    s.inner.arr[0] = 100;
    *(s.inner.ptr) = 200;
}

int main(void) {
    int val = 5;
    Inner inner = {1, {3, 4, 5}, &val};
    Outer outer = {inner, 10};
    modify(outer);
    return 0;
}
```

	x	a	arr	*(ptr)
Before	10	1	3,4,5	5
After	10	1	3,4,5	200

The Arrow Operator



`(*player_ptr).name` *is equivalent to* `player_ptr->name`

`(*player_ptr).age` *is equivalent to* `player_ptr->age`

[Q] Can we write *player_ptr.name instead of (*player_ptr).name?

No, because .(dot) has higher precedence than *, so *player_ptr.name means *(player_ptr.name)!