CS 240 Programming in C

Storage classes and Operators

Feb 2, 2023

Storage Classes

- A storage class defines the scope (visibility) and lifetime of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program
  - **auto** - default storage class for all local variables.
  - **extern** - used to define a global variable or function, which will also be used in other files.
  - **static** - The static storage class instructs the compiler to keep a local variable in existence during the lifetime of the program instead of creating and destroying it each time it comes into and goes out of scope.
  - **register** - used to define local variables that should be stored in a CPU register instead of memory/cache.
- Book Chapter: 4, page: 73
- More: https://www.geeksforgeeks.org/storage-classes-in-c/

External Variables

- In C, the function name has to be unique, even if they are compiled separately from different files since they will be used to reference their binary code body.
- Variables usually exist inside a function body, and for each function there can not exist two variables that assume the same name; inside different functions, there can be variables that assume the same name.
- Well, there can also exist variables outside any function, and these variables are called external variables.

External Variables

- Like function names, external variable names must also be unique from each other.
- External variables and internal variables can share the same name, and they reference different memory addresses. And they have different access scopes.

Block and Scope

- Block: A section of code that is grouped together
  - In C, blocks are delimited by curly braces

    ```
    { [block statements] }
    ```
  - or the parenthesis of main function

    ```
    int main () { int i = 0, j = 1 };
    ```
- Scope: the area of a program where a variable can be referenced
  - For each different entity that an identifier designates, the the identifier is visible (i.e., can be used) only within a region of program text called its *scope*

Block and Internal Variable

- Variables defined within a block are local to the block where they are defined which means that they are not accessible the outside of the block; they come and go with the block of codes executing and finishing.
- Internal variables are also often called "auto" variables. Inside a function block, these two definitions are equal (it is just the "auto" keyword is often ignored):

```
int j = 0;
auto int j;
```

Block

- If a variable was not defined within this block, then it will resort to the outer block for the definition of this variable, until outside the function within the same file.
- Let's see demos.

External Variables

- For accessing an external variable that is not defined within this source code, we have to use the "extern" keyword.
- Let's see a demo.

A Summary

```
        Question:

Are external variables the variables defined
by the "extern" keyword?
```

External Variables and the "extern" key word

- No.
- An external variable is just a variable being defined outside any function.
- The "extern" keyword is used for searching the external/global variable reference somewhere else. It means there is no variable definition here within this function.

Declaration vs Definition

- A variable declared by "extern" is a declaration, which does not cause memory allocation.
- A variable definition means at this line of code, this variable will be allocated and reside in memory.

```
extern int i;  Declaration
extern int i=0; This is Definition, not Declaration

int i; int i=0; These are all variable definitions
```

Declaration vs Definition

- A variable definition is also a declaration, but a declaration is not necessary to be a definition.
- A variable is to be used has to at least have a declaration first.

External Variables

Advantages:

- If a large number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists.
- External variables also retain their values after the exit of a function call, since no function owns it solely.
- External (global) variables are favored in high-performance computing. They allow additional optimization by compilers.

External Variables

Disadvantages:

- It is problematic for decoupling a program structure, which makes a big software into less dependent parts such that it is easy for maintaining and testing etc.
- If their value gets corrupted, hard to trace the reason. They make functions dependent on their external environment
- In fact, software architecture/design standards often prohibit the use of external variables

External Variables (External Static)

- External variables can be accessed by any function in the program.
- what if we want to limit its scope?
- The static declaration applied to an external variable or function limits the scope of that object to the source file being compiled.

Static Local Variable

- Static local variable is a local variable that retains and stores its value between function calls or blocks and remains visible only to the function or block in which it is defined.

Example: Static External

```
#include <stdlib.h>
double drand48(void);
void srand48(long int seedval);
```

- The pseudorandom number generator `drand48()` is a family of functions
- They keep an external static variable X as the seed of the generators
- We must call `srand48()` to initialize the seed to generate a different sequence of numbers.

Example: Static Internal

```
#include <stdio.h>
int counter(){
    static int num;
    return num++;
}

int main(void){
    for (int i=0;i<5;i++) counter();
    printf("%d\n", counter());
}
```

The `register` Variables

- A register declaration advises the compiler that this variable will be heavily used
- We want it placed in a machine register, but the compiler is free to ignore this suggestion if it needs registers
- Can only be applied to automatic variables

The `register` Variables

- Register variables can be defined to local variables within functions or blocks, they are stored in CPU registers instead of RAM to have quick access to these variables.

  `Example: register int age;`

- A register variable may actually not be placed into registers in many situations.
- And it is not possible to parse the address of a register variable regardless of whether the variable is actually placed in a register.
- The specific restrictions on the number and types of register variables vary from machine to machine.

Example: Register Variables

The variables declared using the register have no default value. These variables are often declared at the beginning of a program.

```
#include <stdio.h>

int main(void) {
{
    register int i;
    int *p=&i ;
    /*it produces an error when the compilation occurs,
    we cannot get a memory location when dealing
    with CPU register*/

    return 0;
}
```

Summary

Storage Class	Declaration	Storage	Default Initial Value	Scope	Lifetime
auto	Inside a function/block	Memory	Unpredictable	Within the function/block	Within the function/block
register	Inside a function/block	CPU Registers	Garbage	Within the function/block	Within the function/block
extern	Outside all functions	Memory	Zero	Entire the file and other files where the variable is declared as extern	program runtime
Static (local)	Inside a function/block	Memory	Zero	Within the function/block	program runtime
Static (global)	Outside all functions	Memory	Zero	Global	program runtime

Operators in C

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators [1]

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

[1] Link: https://www.tutorialspoint.com/cprogramming/c_operators.htm

Arithmetic Operators

Assume variable A holds 10 and variable B holds 20 then

Operator	Description	Example
+	Adds two operands.	A + B = 30
–	Subtracts second operand from the first.	A – B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Relational Operators

Assume variable A holds 10 and variable B holds 20 then

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

Assume variable A holds 1 and variable B holds 0, then

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>&&</td><td>Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.</td><td>(A && B) is false.</td></tr>
<tr><td>||</td><td>Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.</td><td>(A || B) is true.</td></tr>
<tr><td>!</td><td>Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.</td><td>!(A && B) is true.</td></tr>
</table>

Bitwise Operators

The bitwise operator works on bits and performs the bit-by-bit operation.

p	q	p & q	p I q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Bitwise Operators

The bitwise operator works on bits and performs the bit-by-bit operation.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Assignment Operators

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Misc Operators

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

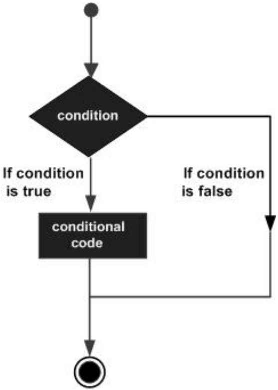
Operators Precedence Table

Reference on Chapter 2.12.

Decision Making

Decision-making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Decision Making



Condition

C programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed a false value.

The ? : Operator

```
Exp1 ? Exp2 : Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

- The value of a ? expression is determined like this
- Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire? expression.
- If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

Header Files

Library Functions in Different Header Files

- assert.h - Program assertion functions
- ctype.h - Character type functions
- locale.h - Localization functions
- math.h - Mathematics functions
- setjmp.h - Jump functions
- signal.h - Signal handling functions
- stdarg.h - Variable arguments handling functions
- stdio.h - Standard Input/Output functions
- stdlib.h - Standard Utility functions
- string.h - String handling functions
- time.h - Date time functions

Ctype.h Functions

Function	Return Type	Use
isalnum(c)	int	Determine if the argument is alphanumeric or not
isalpha(c)	int	Determine if the argument is alphabetic or not
isascii(c)	int	Determine if the argument is ASCII character or not
isdigit(c)	int	Determine if the argument is a decimal digit or not.
toascii(c)	int	Convert value of argument to ASCII
tolower(c)	int	Convert character to lower case
toupper(c)	int	Convert letter to uppercase

[2]

[2] https://www.startertutorials.com/blog/functions-in-c.html

Math.h Functions

Function	Return Type	Use
ceil(d)	double	Returns a value rounded up to next higher integer
floor(d)	double	Returns a value rounded up to next lower integer
cos(d)	double	Returns the cosine of d
sin(d)	double	Returns the sine of d
tan(d)	double	Returns the tangent of d
exp(d)	double	Raise e to the power of d
fabs(d)	double	Returns the absolute value of d
pow(d1, d2)	double	Returns d1 raised to the power of d2
sqrt(d)	double	Returns the square root of d

Stdlib.h Functions

Function	Return Type	Use
abs(i)	int	Return the absolute value of i
exit(u)	void	Close all file and buffers, and terminate the program
rand(void)	int	Return a random positive integer
calloc(u1, u2)	void*	Allocate memory for an array having u1 elements, each of length u2 bytes
malloc(u)	void*	Allocate u bytes of memory
realloc(p,u)	void*	Allocate u bytes of new memory to the pointer variable p
free(p)	void	Free a block of memory whose beginning is indicated by p

String.h Functions

Function	Return Type	Use
strcmp(s1,s2)	int	Compare two strings
strcpy(s1,s2)	char*	Copy string s2 to s1
strlen(s)	int	Return the number of characters in string s
strrev(s)	char*	Return the reverse of the string s

time.h Functions

Function	Return type	Use
difftime(11,12)	double	Return the difference between 11 ~ 12.
time(p)	long int	Return the number of seconds elapsed beyond a designated base time