

CS265

Advanced Programming

Techniques

C Fundamentals

Variables and Data Types

- Every variable must have a **type**
- Variables must be declared before they are used
- C data types

integer data types

floating point numbers data types

pointer data types

boolean (after C99)

- with unique handling of some

character data types (C uses small integers instead)

string data types (C uses array of small integers instead)

logical / boolean data type (uses 0 for false, non-zero for true)

Integer Data Types

Default is **signed**

`char`

`short`

`int`

`long`

`long long int` (after C99)

Unsigned can be used to hold non-negative values (use U to indicate unsigned values)

`unsigned char`

`unsigned short`

`unsigned int`

`unsigned long`

`unsigned long long int` (after C99)

Signed and Unsigned Integers

- The leftmost bit of a **signed** integer (known as the **sign bit**) is 0 if the number is positive or zero, 1 if it's negative.

0	1	1	0	0	0	1	0
---	---	---	---	---	---	---	---



sign bit = 0 for positive number
1 for negative number

- An integer with no sign bit (the leftmost bit is considered part of the number's magnitude) is said to be **unsigned**.
- By default, C uses signed integers

Integer Literals

123 classic decimal

0173

Integer Literals

123 classic decimal

0173 **starts with 0, so it is an octal number (0173 = 123)**

0x7B

Integer Literals

123 classic decimal

0173 starts with 0, so it is an octal number ($0173 = 123$)

0x7B **starts with 0x, so it is hexadecimal number ($0x7B = 123$)**

0x7BL

Integer Literals

123 classic decimal

0173 starts with 0, so it is an octal number ($0173 = 123$)

0x7B starts with 0x, so it is hexadecimal number ($0x7B = 123$)

0x7BL **has an L suffix, meaning it is a long**

(short) 0x7B

Integer Literals

123 classic decimal

0173 starts with 0, so it is an octal number ($0173 = 123$)

0x7B starts with 0x, so it is hexadecimal number ($0x7B = 123$)

0x7BL has an L suffix, meaning it is a long

(short) 0x7B **is a short**

123U, 0173U, 0x7BU

Integer Literals

123 classic decimal

0173 starts with 0, so it is an octal number ($0173 = 123$)

0x7B starts with 0x, so it is hexadecimal number ($0x7B = 123$)

0x7BL has an L suffix, meaning it is a long

(short) 0x7B is a short

123U, 0173U, 0x7BU **end with U so they are unsigned**

123UL

Integer Literals

123 classic decimal

0173 starts with 0, so it is an octal number ($0173 = 123$)

0x7B starts with 0x, so it is hexadecimal number ($0x7B = 123$)

0x7BL has an L suffix, meaning it is a long

(short) 0x7B is a short

123U, 0173U, 0x7BU end with U so they are unsigned

123UL **is unsigned long**

(unsigned short) 0x7B

Integer Literals

123 classic decimal

0173 starts with 0, so it is an octal number ($0173 = 123$)

0x7B starts with 0x, so it is hexadecimal number ($0x7B = 123$)

0x7BL has an L suffix, meaning it is a long

(short) 0x7B is a short

123U, 0173U, 0x7BU end with U so they are unsigned

123UL is unsigned long

(unsigned short) 0x7B **is unsigned short**

The order of U and L does not matter and neither does the case

Defining integers

Examples of valid definitions of an integer variable v

```
short int v;  
int v;  
long int v;  
long v;                (can drop the int)  
unsigned short int v;  
unsigned int v;  
unsigned long int v;
```

Defining integers

Which is correct?

```
unsigned long int v;
```

```
int unsigned long v;
```

```
long unsigned int v;
```

Defining integers

Which is correct?

```
unsigned long int v;
```

```
int unsigned long v;
```

```
long unsigned int v;
```

All are correct! The order does not matter

Integer Types

- Typical ranges on a 32-bit machine:

Type	<i>Smallest Value</i>	<i>Largest Value</i>
short int	-32,768	32,767
unsigned short int	0	65,535
int	-2,147,483,648	2,147,483,647
unsigned int	0	4,294,967,295
long int	-2,147,483,648	2,147,483,647
unsigned long int	0	4,294,967,295

- Typical ranges on a 64-bit machine:

Type	<i>Smallest Value</i>	<i>Largest Value</i>
short int	-32,768	32,767
unsigned short int	0	65,535
int	-2,147,483,648	2,147,483,647
unsigned int	0	4,294,967,295
long int	-2^{63}	$2^{63}-1$
unsigned long int	0	$2^{64}-1$

Floating Point Numbers

There are three

<code>float</code>	single precision floating point
<code>double</code>	double precision floating point
<code>long double</code>	extended precision floating point

The size is unspecified (depends on the machine)

Floating Point Literals

- any number that contains a . or a E is a float
- default is double
- append F to indicate float
- append L to make it a long float

Examples

double: 123.456, 1E-2, -1.23456E4

float: 123.456F, 1E-2F, -1.23456E4F

long double: 123.456L, 1E-2L, -1.23456E4L

Character data type

- The values of type `char` can vary from one computer to another, because different machines may have different underlying character sets.
- When a character appears in a computation, C uses its integer value.
- Consider the following examples, which assume the ASCII character set:

```
char ch;  
int i;  
  
i = 'a';          /* i is now 97    */  
ch = 65;          /* ch is now 'A' */  
ch = ch + 1;      /* ch is now 'B' */  
ch++;            /* ch is now 'C' */
```

- The `char` type—like the integer types—exists in both signed and unsigned versions. Most of the time, the difference doesn't matter.

```
signed char sch;  
unsigned char uch;
```

Logical Data Types

- C did not have a separate logical data type until C99
- Convention
 - use `int` or `char`
 - 0 means FALSE
 - Non-0 means TRUE

Workaround #1

- One way to work around this limitation was to declare an `int` variable and then assign it either 0 or 1:

```
int flag;  
  
flag = 0;  
...  
flag = 1;
```

- And check the value of the flag

```
if (flag) {...}  
if (!flag) {...}
```

- It works, but it does not result in programs that are easy to understand!

Workaround #2

- Define macros with names such as `TRUE` and `FALSE`:

```
#define TRUE 1  
#define FALSE 0
```

- Assignments to `flag` now have a more natural appearance:

```
flag = FALSE;  
...  
flag = TRUE;
```

- To test the value of `flag` we can use

```
if (flag == TRUE) {...}  
if (flag) {...}  
if (!flag) {...}  
if (flag == FALSE) {...}
```

Boolean Type and <stdbool.h> in C99

- C99 introduces the native type `_Bool`
- `_Bool` is an integer data type that can only have the values 0 and 1
- Attempting to store a nonzero value into a `_Bool` variable will cause the variable to be assigned 1
- The associated standard header `<stdbool.h>` introduces the macros `bool`, `true` and `false` for boolean tests.

```
#include <stdio.h>
#include <stdbool.h>

int main() {
    _Bool i = true;
    bool j = false;

    if (j == true) {
        printf("value is true\n");
    }
    else
        printf("value is false\n");
}
```

On tux, the header file `<stdbool.h>` is in `/usr/lib/gcc/*/*/include`

Assignment

variable v = expression e

- Evaluates the expression e and copies its value into variable v .

```
float f;
```

```
f = 72.99;    /* ok */
```

```
f = 72.99f;   /* better! */
```

- If v and e don't have the same type, then the value of e is converted to the type of v as the assignment takes place:

```
int i;
```

```
float f;
```

```
i = 72.99f;   /* i is now 72 */
```

```
f = 136;      /* f is now 136.0 */
```


Assignment

- Watch out for chained assignments

```
int i;  
float f;  
  
f = i = 33.3f;
```

- What is the value of `i` and `f`?

Assignment

- Watch out for chained assignments?

```
int i;  
float f;  
  
f = i = 33.3f;
```

- What is the value of `i` and `f`?
- `i` is assigned the value 33, then `f` is assigned 33.0 (not 33.3).

Variable Initialization

- Some variables are automatically set to zero when a program begins to execute
- **most are not**
- A variable that doesn't have a default value and hasn't yet been assigned a value by the program is said to be ***uninitialized***.
- Attempting to access the value of an uninitialized variable may yield an unpredictable result.
- With some compilers, worse behavior—even a program crash—may occur.

I/O functions

- `printf` – prints output
- `scanf` - reads input

printf function to print output

- C language decision to use the same function for all data types
- Instead of

`print_int()`, `print_float()`, `print_character()`, etc.

- We use `printf()` function with `%` format with a specifier to display the value of specific data types

```
#include <stdio.h>
```

```
int printf( const char *format, ... ) ;
```

printf format

format

`%[flags][width][.precision] specifier`

`flags` Justification, padding, leading +/-

`width` The minimum total field width

`precision` The number of decimals (might be truncated)

`specifier` Specifies the data type, and presentation

printf output - specifier

format

%[flags][width][.precision] **specifier**

% and specifier are the only two required fields

Format Specifier	Type
%c	character
%d	signed integer
%e or %E	scientific notation of floats
%f	float values
%g or %G	similar as %e or %E
%hi	signed integer (short)
%hu	unsigned integer (short)
%i	unsigned integer
%l or %ld or %li	long
%lf	double

Format Specifier	Type
%Lf	long double
%lu	unsigned int or unsigned long
%lli or %lld	long long
%llu	unsigned long long
%o	octal representation
%p	pointer
%s	string
%u	unsigned int
%x or %X	hexadecimal representation
%n	prints nothing

printf output - flags

format

%[**flags**][width][.precision] specifier

- Left-justify
- + Forces leading sign, even for positive numbers
- # With o, x, or X specifier, output is preceded by 0, 0x, or 0X
- 0 Pads with 0, rather than space

printf output - width

format

`%[flags][width][.precision] specifier`

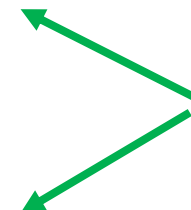
- `width` is an optional integer that specifies the minimum characters to print

- If `width` is positive the output is right justified

`%10d` displays the number 123 as123

- If `width` is negative the output is left justified

`%-10d` displays the number 123 as 123.....



• represents the space character

- If the value to be printed requires more than `width` characters, the field `width` automatically expands to the necessary size.

printf output - precision

format

`%[flags][width][.precision]` specifier

- `precision` is an optional precision specifier for the number of decimal points after the .
- defaults to 6 if it is missing
- to print the line

```
Profit: $2150.48
```

- use

```
printf("Profit: $%.2f\n", profit);
```

Compilers and `printf`

- Compilers aren't required to check that the number of conversion specifications in a format string matches the number of output items.

```
printf("%d %d\n", i);    /*** WRONG ***/
```

```
printf("%d\n", i, j);    /*** WRONG ***/
```

Example

```
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void)
{
    int i;
    float x;

    i = 40;
    x = 839.21f;

    printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
    printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

    return 0;
}
```

Output:

```
|40|    40|40    |   040|
|  839.210| 8.392e+02|839.21    |
```

Escape sequences for `printf`

- The `\n` code that used in format strings is called an ***escape sequence***.
- Escape sequences enable strings to contain nonprinting (control) characters and characters that have a special meaning (such as ").
- A partial list of escape sequences:

Alert (bell)	<code>\a</code>
Backspace	<code>\b</code>
New line	<code>\n</code>
Horizontal tab	<code>\t</code>

Reading Input – the `scanf` function

- `scanf` is the C library's counterpart to `printf`.
- `scanf` requires a **format string** to specify the appearance of the input data.
- Example of using `scanf` to read an `int` value:

```
scanf("%d", &i);  
/* reads an integer; stores into i */
```

- The `&` symbol is usually (but not always) required when using `scanf`.

Reading Input

- Reading a `float` value requires a slightly different call

```
scanf("%f", &x);
```

- `"%f"` tells `scanf` to look for an input value in `float` format (the number may contain a decimal point, but it doesn't have to).

The conversions allowed with `scanf` are essentially the same as those used with `printf`.

The `scanf` function

- In many cases, a `scanf` format string will contain only conversion specifications:

```
int i, j;
```

```
float x, y;
```

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- Sample input:

```
1 -20 .3 -4.0e3
```

`scanf` will assign 1, -20, 0.3, and -4000.0 to `i`, `j`, `x`, and `y`, respectively.

The `scanf` function

- When using `scanf`, the programmer must check that the number of conversion specifications matches the number of input variables and that each conversion is appropriate for the corresponding variable.
- Another trap involves the `&` symbol, which normally precedes each variable in a `scanf` call.
- The `&` is usually (but not always) required, and it's the programmer's responsibility to remember to use it.

How `scanf` works

- `scanf` tries to match groups of input characters with conversion specifications in the format string.
- For each conversion specification, `scanf` tries to locate an item of the appropriate type in the input data, skipping blank space if necessary.
- `scanf` then reads the item, stopping when it reaches a character that can't belong to the item.
 - If the item was read successfully, `scanf` continues processing the rest of the format string.
 - If not, `scanf` returns immediately.

How scanf works

- As it searches for a number, `scanf` ignores **white-space characters** (space, horizontal and vertical tab, form-feed, and new-line).

- A call of `scanf` that reads four numbers:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- The numbers can be on one line or spread over several lines:

```
1
-20  .3
    -4.0e3
```

- `scanf` sees a stream of characters (␣ represents new-line, • is space):

```
••1␣-20•••.3␣•••-4.0e3␣
ssrsrrrrsssrsssrsssrsssr (s = skipped; r = read)
```

- `scanf` “peeks” at the final new-line without reading it.

How scanf works

- When asked to read an integer, `scanf` first searches for a digit, a plus sign, or a minus sign; it then reads digits until it reaches a nondigit.
- When asked to read a floating-point number, `scanf` looks for
 - a plus or minus sign (optional), followed by
 - digits (possibly containing a decimal point), followed by
 - an exponent (optional). An exponent consists of the letter `e` (or `E`), an optional sign, and one or more digits.
- `%e`, `%f`, and `%g` are interchangeable when used with `scanf`.

How `scanf` works

- When `scanf` encounters a character that can't be part of the current item, the character is “put back” to be read again during the scanning of the next input item or during the next call of `scanf`.

Examples

- Sample input:

1-20.3-4.0e3x

- The call of `scanf` is the same as before:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- Here's how `scanf` would process the new input:

- `%d` Stores 1 into `i` and puts the - character back.
- `%d` Stores -20 into `j` and puts the . character back.
- `%f` Stores 0.3 into `x` and puts the - character back.
- `%f` Stores -4.0×10^3 into `y` and puts the new-line character back.

Ordinary Characters in `scanf`

- `scanf` reads the input until it reaches a non-white character, and then it compares this character with the format string
- Examples:
 - If the format string is `"%d/%d"` and the input is `•5/•96`, `scanf` succeeds.
 - If the input is `•5•/•96`, `scanf` fails, because the `/` in the format string doesn't match the space in the input.
- To allow spaces after the first number, use the format string `"%d /%d"` instead.



Be aware of spaces and special characters
in the `scanf` format string

Confusing `printf` and `scanf` – The `&` character

- Although calls of `scanf` and `printf` may appear similar, there are significant differences between the two.
- One common mistake is to put `&` in front of variables in a call of `printf`:

```
printf("%d %d\n", &i, &j);  /** WRONG **/
```



Do not put `&` in front of the variable in `printf`
`scanf` usually needs the `&` in front of the variable (but not always)

Confusing `printf` and `scanf` – The `\n` character

- Putting a new-line character at the end of a `scanf` format string is usually a bad idea.
- To `scanf`, a new-line character in a format string is equivalent to a space; both cause `scanf` to advance to the next non-white-space character.
- If the format string is `"%d\n"`, `scanf` will skip white space, read an integer, then skip to the next non-white-space character.
- A format string like this can cause an interactive program to “hang.”

Bad idea



putting a new line at the end of a `scanf` format string is usually a bad idea

Constants

- Using a feature known as [macro definition](#), we can name a constant:

```
#define INCHES_PER_POUND 166
```

- When a program is compiled, the preprocessor replaces each macro by the value that it represents.

- During preprocessing, the statement

```
weight = (volume + INCHES_PER_POUND - 1) / INCHES_PER_POUND;
```

will become

```
weight = (volume + 166 - 1) / 166;
```

Defining names for constants

- The value of a macro can be an expression:
`#define RECIPROCAL_OF_PI (1.0f / 3.14159f)`
- If it contains operators, the expression and the operators should be enclosed in parentheses.
- Using only upper-case letters in macro names is a common convention.

e.g.

```
#define quad(x) x*x  
int a = quad(2+3);
```

```
#define TEMPERATURE_WITH (-1)  
#define TEMPERATURE_WITHOUT -1  
int w = 10-TEMPERATURE_WITH;  
int wo=10-TEMPERATURE_WITHOUT;
```

Identifiers

- Names for variables, functions, macros, and other entities are called **identifiers**.
- An identifier may contain letters, digits, and underscores, but must begin with a letter or underscore:

```
times10  get_next_char  _done
```

- It's usually best to avoid identifiers that begin with an underscore.
- Examples of illegal identifiers:

```
10times  get-next-char
```

- C is **case-sensitive**: it distinguishes between upper-case and lower-case letters in identifiers.
- For example, the following identifiers are all different:

```
job  joB  jOb  jOB  Job  JoB  JOB  JOB
```

Identifiers

- Many programmers use only lower-case letters in identifiers (other than macros), with underscores inserted for legibility:

`symbol_table current_page name_and_address`

- Other programmers use an upper-case letter to begin each word within an identifier:

`symbolTable currentPage nameAndAddress`

- C places no limit on the maximum length of an identifier

Keywords

- The following **keywords** can't be used as identifiers:

auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

*C99 reserved 5 more words

_Bool	_Complex	_Imaginary	inline	restrict
-------	----------	------------	--------	----------

**C11 reserved 7 more words

_Alignas	_Alignof	_Atomic
_Generic	_Noreturn	_Static_assert
_Thread_local		

Operators

- C has a rich collection of operators, including
 - arithmetic operators
 - relational operators
 - logical operators
 - assignment operators
 - increment and decrement operatorsand many others

C Language Overview

Data types	char, int, float, double, bool
Integer types	(unsigned, signed) char (unsigned, signed) short (unsigned, signed) int (unsigned, signed) long
Floating point	float double long double
Constants	#define MAX 1000 const int MAX = 1000;
Type conversions / cast	(type-name) expression
Arithmetic Operators	+, -, *, /, %.
Logical Operations	&&, , !
Comparisons	==, !=, <, >, <=, >=
Bitwise logic	&, ^, , ~
Bitwise shifts	<<, >>
Assignment	=
Assignment (extended)	=, +=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=

C Language Overview

Data types	char, int, float, double, bool
Conditional evaluation	? :
Equality Testing	== !=
Calling functions	()
Increment and decrement	++ --
Object size	sizeof
Member selection	-> .
Reference and dereference (pointers, arrays)	& * []
Type conversion	(new type)

Precedence and Associativity of Operators

This table lists the precedence and associativity of C operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){list}	Compound literal(c99)	
2	++ --	Prefix increment and decrement ^[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of ^[note 2]	
	_Alignof	Alignment requirement(c11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10		Bitwise OR (inclusive or)	
11	&&	Logical AND	
12		Logical OR	
13	?:	Ternary conditional ^[note 3]	Right-to-left
14 ^[note 4]	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= =	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Compound Assignment

- $v += e$ isn't "equivalent" to $v = v + e$.
- One problem is operator precedence: $i *= j + k$ isn't the same as $i = i * j + k$.
- There are also rare cases in which $v += e$ differs from $v = v + e$ because v itself has a side effect.
- Similar remarks apply to the other compound assignment operators.

Compound Assignment

- When using the compound assignment operators, be careful not to switch the two characters that make up the operator.
- Although `i =+ j` will compile, it is equivalent to `i = (+j)`, which merely copies the value of `j` into `i`.

Be careful when using `+=` not to confuse it with `=+`

Increment and Decrement Operators

- Two of the most common operations on a variable are “incrementing” (adding 1) and “decrementing” (subtracting 1):

```
i = i + 1;  
j = j - 1;
```

- Incrementing and decrementing can be done using the compound assignment operators:

```
i += 1;  
j -= 1;
```

- C provides special ++ ([increment](#)) and -- ([decrement](#)) operators.
- The ++ operator adds 1 to its operand. The -- operator subtracts 1.
- The increment and decrement operators are tricky to use:
 - They can be used as [prefix](#) operators (++i and --i) or [postfix](#) operators (i++ and i--).
 - They have side effects: they modify the values of their operands.

Increment and Decrement Operators

- Evaluating the expression `++i` (a “pre-increment”) yields `i + 1` and—as a side effect—increments `i`:

```
i = 1;
printf("i is %d\n", ++i);    /* prints "i is 2" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

- Evaluating the expression `i++` (a “post-increment”) produces the result `i`, but causes `i` to be incremented afterwards:

```
i = 1;
printf("i is %d\n", i++);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

Increment and Decrement Operators

- `++i` means “increment `i` immediately,” while `i++` means “use the old value of `i` for now, but increment `i` later.”
- How much later? The C standard doesn’t specify a precise time, but it’s safe to assume that `i` will be incremented before the next statement is executed.

Increment and Decrement Operators

- When ++ or -- is used more than once in the same expression, the result can often be hard to understand.

- Example:

```
i = 1;  
j = 2;  
k = ++i + j++;
```

The last statement is equivalent to

```
i = i + 1;  
k = i + j;  
j = j + 1;
```

The final values of `i`, `j`, and `k` are 2, 3, and 4, respectively.

Increment and Decrement Operators

- In contrast, executing the statements

```
i = 1;
```

```
j = 2;
```

```
k = i++ + j++;
```

will give *i*, *j*, and *k* the values 2, 3, and 3, respectively.

If statement

- Typical form

```
if (expression) {  
    statements1  
}  
  
else {  
    statements2  
}
```

- *Ternary form*

$expr_1 \quad ? \quad expr_2 \quad : \quad expr_3$

Instead of

```
if (a < b)  
    { c = a; }  
else  
    { c = b; }
```

Use

```
c = (a < b) ? a : b;
```

Switch statement

```
switch (expression) {  
    case const-expr :  
        statements;  
        break; /* optional */  
    case const-expr :  
        statements;  
        break; /* optional */  
    default:  
        statements;  
}
```

For loop

- `for (expr1; expr2; expr3)`
 statements

- This is valid C

```
int i;  
for (i=0; i< 10; i++ { ...}
```

- and so is this

```
for( ; ; ) {  
    printf("This loop will run forever.\n");  
}
```

- and this

```
for (int i=0; i< 10; i++ { ...}
```

- But not this

```
for i in 0..10 {...}
```

While Loops

- Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

```
while (expression) {  
    statements;  
}
```

- Like a while statement, except that it tests the condition at the end of the loop body, so the statement or group of statements execute at least once

```
do  
    statements;  
while (expression);
```

Loop Control Statements

`break`

- Terminates the `loop` or `switch` statement and transfers execution to the statement immediately following the loop or switch.

`continue:`

- Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

`goto`

- Transfers control to the labeled statement.

```
if (expression) { goto error; }
```

```
...
```

```
error: printf("This is an error");
```

Lessons

- Lesson 1: Learn C to become a power programmer
- Lesson 2: C / C++ are the defacto systems programming languages



Resources

- These notes
- *C Programming: A modern Approach* by K.N. King, 2008
 - Chapters 2-7
- K&R Book <http://tinyurl.com/yaemm9vh> (it covers an older version of C but it is a great way to learn the basics)