600.120 Intermediate Programming, Spring 2017

Misha Kazhdan

Outline

- Why C/C++?
- Variables, types, and operators
- Control Structures
- Loops

Why C/C++?

- Ubiquitous
  - A huge fraction of the world’s most crucial code is written in C
    - Java’s JVM
    - Python’s interpreter
    - Web servers
    - …
- Mature
- Efficient

Why C/C++?

- Ubiquitous
- Mature
  - C has been around since the 70s
  - C++ has been around since the 80s
    - Many systems are built on it, so many jobs require it
- Efficient

Why C/C++?

- Ubiquitous
- Mature
- Efficient
  - Gives you lower-level access to the way your program runs
  - Enables more opportunity for optimization
    - Memory layout
    - When/how memory is allocated and deallocated
    - Allows you to leverage specialty hardware

What happens when you compile?

1. Preprocessor
   - Prepares the code for compilation
     - Code inclusion
     - Macro expansion
     - Conditional compilation
     - Comment removal
2. Compiler
3. Linker

What happens when you compile?

1. Preprocessor
2. Compiler
   - Turns human-readable *source code* into *object code*
   - May yield compiler warnings/errors
3. Linker

What happens when you compile?

1. Preprocessor
2. Compiler
3. Linker
   - Bring together all the relevant *object code* into a single executable file
   - May yield linker warnings/errors

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

- **#include** is a preprocessor directive, similar to import

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

- **#include** is a preprocessor directive, similar to import
- Explanatory comment before function is good practice

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

- `#include` is a preprocessor directive, similar to import
- Explanatory comment before function is good practice
- `main` is a function, every program has exactly one
  - `int` is its return value
  - `main( void )` says that `main` takes no parameters

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

- `#include` is a preprocessor directive, similar to import
- Explanatory comment before function is good practice
- `main` is a function, every program has exactly one
  - `int` is its return value
  - `main( void )` says that `main` takes no parameters
- Prints a string to the console followed by a newline

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

- #include is a preprocessor directive, similar to import
- Explanatory comment before function is good practice
- main is a function, every program has exactly one
  - int is its return value
  - main(void) says that main takes no parameters
- Prints a string to the console followed by a newline
- Returns the state of the program when it terminated
  - A value of zero indicates no error

Hello World

```
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
        printf( "Hello, world!\n" );
        return 0;
}
```

Q: What if we omit the line **#include <stdio.h>**?

```
>> gcc helloWorldErr.c -std=c99 -pedantic -Wall -Wextra
```

A: The compiler doesn't know what **printf** should mean.

```
helloWorldErr.c: In function main:
helloWorldErr.c:4:3: warning: implicit declaration of function printf [-Wimplicit-function-declaration]
  printf( "hello world\n" );
  ^~~~~~
helloWorldErr.c:4:3: warning: incompatible implicit declaration of built-in function printf
helloWorldErr.c:4:3: note: include <stdio.h> or provide a declaration of printf
```

Variables

```
int num_students;
```

- When declared, a variable gets a *type* (`int`) and a *name* (`num_students`)
  - C/C++ are *typed languages*: every variable must have a type
- A variable also has a *value* that may change throughout the program's life

Assignment

```
int num_students;
num_students = 32;
```

- When declared, a variable gets a *type* (`int`) and a *name* (`num_students`)
  - C/C++ are *typed languages*: every variable must have a type
- A variable also has a *value* that may change throughout the program's life
- `=` is the *assignment operator*, which modifies a variable's value

Assignment

```
int num_students = 32;
```

- When declared, a variable gets a *type* (`int`) and a *name* (`num_students`)
  - C/C++ are *typed languages*: every variable must have a type
- A variable also has a *value* that may change throughout the program's life
- `=` is the *assignment operator*, which modifies a variable's value
- It is good practice to declare and assign *at the same time*
  - Otherwise you have variables with *undefined* (random) values

  ⇒ The way the code misbehaves from run to run will not be consistent

  ⇒ It may be very hard to debug the code

Types

```
int num_students = 32;
```

- Integer types:
  - `[unsigned] char`: [un]signed character (typically 1 byte)
  - `[unsigned] int`: [un]signed integer (typically 4 bytes)

- Floating-point types:
  - `float`: single-precision floating point number (typically 4 bytes)
  - `double`: double-precision floating point number (typically 8 bytes)

https://en.wikipedia.org/wiki/C_data_types

Operators

Take one or two values (*operands*) and combine to get a new value

Unary:

-	negation	-num_students

Binary

+	addition	3 + 4
-	subtraction	num_students - 4
*	multiplication	3 * num_students
/	division	num_students / num_students
%	modulus	num_students % 4

What happens if you add an integer and a float?
What happens if you divide an odd number by two?

Mysterious program

```
#include <stdio.h>
int main(void)
{
        int x = 75;
        float y = 5.0 / 9.0 * (x - 32);
        printf( "%0.2f\n" ,  y );
        return 0;
}
```

This program compiles and runs, but the naming convention makes it hard to “read”.

Less mysterious program

```
#include <stdio.h>
int main(void)
{
        int x = 75;
        float y = 5.0
        printf( "%0.2
        return 0;
}
```

This program does the same thing, but is more “readable”.

```
#include <stdio.h>
// Convert 75 degrees Fahrenheit to Celsius, print result
int main( void )
{
        int fahrenheit = 75;
        float celsius = 5.0 / 9.0 * (fahrenheit - 32);
        // print up to 2 decimal places
        printf( "%0.2f\n" , celsius );
        return 0;
}
```

Precedence

- Will this code compile?
  - Yes
- Is it correct?
  - No

```
#include <stdio.h>
// Convert 75 degrees Fahrenheit to Celsius, print result
int main( void )
{
        int fahrenheit = 75;
        float celsius = 5.0 / 9.0 * fahrenheit - 32;
        // print up to 2 decimal places
        printf( "%0.2f\n" , celsius );
        return 0;
}
```

Precedence

- C/C++ have rules about what order operations should be performed
- Know where to look up the rules and use parentheses when in doubt

Precedence	Operator	Associativity
1	++ -- () [] . -> (*type*){*list*}	Left-to-right
2	++ -- + - ! ~ (*type*) * & sizeof _Alignof	Right-to-left
3	* / %	Left-to-right
4	+ -	
5	<< >>	
6	< <= > >=	
7	== !=	
8	&	
9	^	
10	|	
11	&&	
12	||	

http://en.cppreference.com/w/c/language/operator_precedence

Precedence

- C/C++ have rules about what order operations should be performed
- Know where to look up the rules and use parentheses when in doubt

```
...
float celsius = 5.0 / 9.0 * fahrenheit - 32;
...
```

```
...
float celsius = ( ( 5.0 / 9.0 ) * fahrenheit ) - 32;
...
```

Precedence	Operator	Associativity
1	++ -- () [] . -> (*type*){*list*}	Left-to-right
2	++ -- + - ! ~ (*type*) * & sizeof _Alignof	Right-to-left
	* / %	Left-to-right
	+ -	
	<< >>	
6	< <= > >=	
7	== !=	
	&	
	^	
	|	
	&&	
	||	

Even less mysterious program

```
#include <stdio.h>
// Convert 75 degrees Fahrenheit to Celsius, print result
int main( void )
{
        int base = 32;
        float factor = 5.0 / 9.0;
        int fahrenheit = 75;
        float celsius = factor * (fahrenheit - base);
        // print up to 2 decimal places
        printf( "%0.2f\n" , celsius );
        return 0;
}
```

And still less mysterious program

```
#include <stdio.h>
// Convert 75 degrees Fahrenheit to Celsius, print result
int main( void )
{
        const int base = 32;
        const float factor = 5.0 / 9.0;
        int fahrenheit = 75;
        float celsius = factor * (fahrenheit - base);
        // print up to 2 decimal places
        printf( "%0.2f\n" , celsius );
        return 0;
}
```

const keyword

```
const int base = 32;
```

- The **const** keyword indicates that the variable cannot be modified after it’s been declared

const keyword

```
#include <stdio.h>
// Convert 75 degrees fahrenheit to celsius, print result
int main( void )
{
        const int base = 32;
        const float factor = 5.0 / 9.0;
        const int fahrenheit = 75;
        const float celsius = factor * ( fahrenheit – base );
        printf( "%0.2f\n" , celsius ); // print up to 2 decimal places
        fahrenheit = 70;
        celsius = factor * ( fahrenheit – base );
        printf( "%0.2f\n" , celsius ); // print up to 2 decimal places
        return 0;
}
```

const keyword

```
#include <stdio.h>
// Convert 75 degrees fahrenheit to celsius, print result
int main( void )
{
        const int base = 32;
        const float factor = 5.0 / 9.0;
        const int fahrenheit = 75;
        const float celsius = factor * ( fahrenheit – base );
        printf( "%0.2f\n" , celsius ); // print up to 2 decimal places
}
```

```
>> gcc convert_fc_var3.c -std=c99 -pedantic -Wall -Wextra
helloWorldErr.c: In function main:
helloWorldErr.c:8:14: error: assignment of read-only variable fahrenheit
   fahrenheit = 70;
              ^
helloWorldErr.c:9:11: error: assignment of read-only variable celsius
   celsius = 5.0 / 9.0 * fahrenheit - 32;
```

Types (more)

- Boolean type
  - `#include <stdbool.h>`
  - type is `bool`, value is either `true` or `false`
  - Integer types can also function as bools, where 0=false, non-0=true
    - This is quite common, since bool was only introduced in C99
    - Generally, C mindset is “Booleans are just integers”

Operators (more)

Takes Boolean value(s) and returns a Boolean value

- Unary:

`!`	logical “not”	`!A` is true iff. `A` is false

- Binary:

`&&`	logical “and”	`(A && B)` is true iff. both `A` and `B` are true
`||`	logical “or”	`(A || B)` is true iff. either or both `A` and `B` are true

Operators (more)

Takes integer/floating-point value and returns a Boolean value

- Equality operators:

`==`	`(A == B)` is true iff `A` equals `B`*
`!=`	`(A != B)` is true iff `A` does not equal `B`*

- Relational operators

`>`	`(A > B)` is true iff `A` is greater than `B`
`<`	`(A < B)` is true iff `A` is less than `B`
`>=`	`(A >= B)` is true iff `A` is greater than or equal to `B`*
`<=`	`(A <= B)` is true iff `A` is less than or equal to `B`*

*You should avoid using these to compare floating point values!

Operators (more)

Short-circuiting:

- When C evaluates the composition of logical expression. . .

```
if( (statement _1) || (statement_2) )
if( (statement _1) && (statement_2) )
```

. . . it *short circuits* as soon as answer is definitely true or definitely false.

- `if( a == 7 || b == 7 ):`

  When (`a==7`) is true, the entire expression is true so we don't need to test if (`b == 7`) is true.

- `if( a == 7 && b == 7 ):`

  When (`a==7`) is false, the entire expression is false so we don't need to test if (`b == 7`) is true.

Types (more)

- Character type
  - a `char` variable holds a single character:
    - `char digit = '4';`
    - `char bang = '!';`
  - These *must* be single quotes; double quotes are for strings, not `char`s
  - Behind the scenes, `char` is just like `int`:

    `char digit = '4'-1;`

    `digit` now contains the character '3'

ASCII

- The ASCII standard governs the mapping between characters and integers.

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SoH	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	SoTxt	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	EoTxt	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EoT	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	Enq	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	Ack	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	Bell	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	Bsp	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	HTab	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LFeed	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VTab	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FFeed	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SOut	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SIn	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAck	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	Syn	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	EoTB	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	Can	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EoM	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	Sub	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	Esc	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FSep	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GSep	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RSep	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	USep	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	Delete

charstable.com

ASCII

Q: What does this print?

```
#include <stdio.h>
int main( void )
{
        char char_0 = '0';
        int int_0 = char_0 - '0';
        printf( "Character printed as character: %c\n" , char_0 );
        printf( "Character printed as integer: %d\n" , char_0 );
        printf( "Integer printed as integer: %d\n" ,  int_0 );
}
```

```
>> ./a.out
Character printed as character: 0
Character printed as integer: 48
Integer printed as integer: 0
>>
```

Control structures

- The `if` statement evaluates a Boolean predicate and executes the code in braces if the predicate is true.

```
#include <stdio.h>
int main( void )
{
        int n = 12;
        if( n % 2 == 0 )
        {
                printf( "E\n" );
        }
        return 0;
}
```

```
>> ./a.out
E
>>
```

Control structures

- The if statement evaluates a Boolean predicate and executes the code in braces if the predicate is true.
- It no braces are provided, the if only affects the next command (i.e. up to the next “;”).

```
#include <stdio.h>
int main( void )
{
        int n = 12;
        if( n % 2 == 0 )
                printf( “E\n” );
        return 0;
}
```

Note: White-space / indentation has no effect on what the if applies to.

Control structures

```
#include <stdio.h>
int main( void )
{
        int n = 12;
        if( n % 2 == 0 ) printf( "E\n" );
        return 0;
}
```

- The `if` statement evaluates a Boolean predicate and executes the code in braces if the predicate is true.
- It no braces are provided, the `if` only affects the next command (i.e. up to the next ";").
- Can even put on one line (if it's readable).

Note: White-space / indentation has no effect on what the `if` applies to.

Control structures

- The **if / else** statement evaluates a Boolean predicate and follows the **if** branch if the predicate is true and the **else** branch otherwise.

```
#include <stdio.h>
int main( void )
{
        int n = 13;
        if( n % 2 == 0 )
        {
                printf( "E\n" );
        }
        else
        {
                printf( "O\n" );
        }
        return 0;
}
```

```
>> ./a.out
O
>>
```

Control structures

- The **if / else** statement evaluates a Boolean predicate and follows the **if** branch if the predicate is true and the **else** branch otherwise.
- It no braces are provided, the **if / else** only effect the next command (i.e. up to the next “;”).

```
#include <stdio.h>
int main( void )
{
        int n = 13;
        if( n % 2 == 0 ) printf( “E\n” );
        else             printf( “O\n” );
        return 0;
}
```

Control structures

- The **if / else** statement evaluates a Boolean predicate and follows the **if** branch if the predicate is true and the **else** branch otherwise.
- It no braces are provided, the **if / else** only effect the next command (i.e. up to the next “;”).
- The **else** is always associated to the last (unmatched) **if**.

```
#include <stdio.h>
int main( void )
{
        int n = 13;
        if( n % 2 == 0 )
                if( n==11 ) printf( “11\n” );
                else        printf( “E\n” );
        return 0;
}
```

```
>> ./a.out
>>
```

Control structures

- The **if / else** statement evaluates a Boolean predicate and follows the **if** branch if the predicate is true and the **else** branch otherwise.
- It no braces are provided, the **if / else** only effect the next command (i.e. up to the next ";").
- The **else** is always associated to the last (unmatched) **if**.

```
#include <stdio.h>
int main( void )
{
        int n = 13;
        if( n % 2 == 0 )
        {
                if( n==11 ) printf( "11\n" );
        }
        else printf( "O\n" );
        return 0;
}
```

```
>> ./a.out
O
>>
```

Control structures

- The **if / else if / else** statement evaluates a sequence of Boolean predicates, and executes the code when the predicate is true.

```
#include <stdio.h>
int main( void )
{
        int x = 79;
        if      ( x >= 90 ) printf( "A\n" );
        else if( x >= 80 ) printf( "B\n" );
        else if(x >= 70)   printf( "C\n" );
        else if(x >= 60)   printf( "D\n" );
        else               printf( "F\n" );
        return 0;
}
```

```
>> ./a.out
C
>>
```

Control structures

- The **switch** statement tests if a value matches one of a set of prescribed cases and executes *all* the code after if it does.
  - **switch**:
    Specifies the value to be tested
  - **case**:
    specifies the case to execute
  - **break**:
    do not continue to the next case
  - **default**:
    if nothing else matched...

```
#include <stdio.h>
int main( void )
{
        char grade = 'C';
        int points = 0;
        switch( grade )
        {
                case 'A':
                        points = 4;
                        break;
                case 'B':
                        points = 3;
                        break;
                case 'C':
                        points = 2;
                        break;
                case 'D':
                        points = 1;
                        break;
                default:
                        points = 0;
                        break;
        }
        printf( "Grade %c -> %d GPA points\n", grade, points);
}
```

```
>> ./a.out
Grade C -> 2 points
>>
```

Compound assignment

Combine binary operators with assignment operators:

```
A += B;   ⇒   A = A+B;
A -= B;   ⇒   A = A-B;
A * =B;   ⇒   A = A*B;
A /= B;   ⇒   A = A/B;
A %= B;   ⇒   A = A%B;
```

The right hand side can be either a variable or a constant

Increment and decrement

Increase / decrease the value by one:

A++;	⇒	*A* = *A*+1;
A--;	⇒	*A* = *A*-1;
++*A*;	⇒	*A* = *A*+1;
--*A*;	⇒	*A* = *A*-1;

The difference between *A*++ and ++*A* (or *A*-- and --*A*) is precedence.

Increment and decrement

Increase / decrease the value by one:

```
B = A++;  ⇒  { B = A;   A = A+1;        }
B = A--;  ⇒  { B = A;   A = A-1;        }
B = ++A;  ⇒  {          A = A+1;   B = A; }
B = --A;  ⇒  {          A = A-1;   B = A; }
```

Increment and decrement

Increase / decrease the value by one:

```
#include <stdio.h>
int main( void )
{
        int i = 0;
        if( ++i ) printf( "++i was non-zero\n" );
        printf( "i=%d\n" ,  i );

        i = 0;
        if( i++ ) printf( "i++ was non-zero\n" );
        printf( "i=%d\n" , i );
}
```

```
>> ./a.out
++i was non-zero
i=1
i=1
>>
```

Loops

The **for** loop

```
#include <stdio.h>
int main( void )
{
        for( int i=0 ; i<10 ; i++ )
        {
                d\n" , i );
        }
}
```

```
>> ./a.out
0
1
2
3
4
5
6
7
8
9
>>
```

Loops

The **for** loop:

- Initializes a loop variable

```
#include <stdio.h>
int main( void )
{
        for( int i=0 ; i<10 ; i++ )
        {
                printf( "%d\n" , i );
        }
}
```

Loops

The `for` loop:

- Initializes a loop variable
- Iterates while the looping condition is met

```
#include <stdio.h>
int main( void )
{
        for( int i=0 ; i<10 ; i++ )
        {
                printf( "%d\n" , i );
        }
}
```

Loops

The `for` loop:

- Initializes a loop variable
- Iterates while the looping condition is met
- Adjusts the loop value <u>after</u> each iteration

```
#include <stdio.h>
int main( void )
{
        for( int i=0 ; i<10 ; i++ )
        {
                printf( "%d\n" , i );
        }
}
```

Loops

The `for` loop:

- Initializes a loop variable
- Iterates while the looping condition is met
- Adjusts the loop value <u>after</u> each iteration
- Performs the calculation in braces at each iteration

```
#include <stdio.h>
int main( void )
{
        for( int i=0 ; i<10 ; i++ )
        {
                printf( "%d\n" , i );
        }
}
```

Loops

The `for` loop:

- Initializes a loop value
- Iterates while the looping condition is met
- Adjusts the loop value <u>after</u> each iteration
- Performs the calculation in braces at each iteration
  - If no braces are provided, it performs the next command

```
#include <stdio.h>
int main( void )
{
        for( int i=0 ; i<10 ; i++ )
                printf( "%d\n" , i );
}
```

Loops

The **while** loop:

- Iterates until the **while** condition fails.
- Performs the calculation in braces at each iteration

```
#include <stdio.h>
int main( void )
{
        int i = 1;
        while( (i%7) != 0 )
        {
                printf( "%d\n" , i );
                i++;
        }
}
```

```
>> ./a.out
1
2
3
4
5
6
>>
```

Loops

The **while** loop:

- Iterates until the **while** condition fails.
- Performs the calculation in braces at each iteration

How about this?

```
#include <stdio.h>
int main( void )
{
        int i = 0;
        while( (i%7) != 0 )
        {
                printf( "%d\n" , i );
                i++;
        }
}
```

```
>> ./a.out
>>
```

Loops

The **while** loop:

- Iterates until the **while** condition fails.
- Performs the calculation in braces at each iteration
  - If no braces are provided, it performs the next command

```
#include <stdio.h>
int main( void )
{
        int i = 1;
        while( (i%7) != 0 )
                printf( "%d\n" , i++ );
}
```

Loops

The **while** loop:

- Iterates until the **while** condition fails.

```
#include <stdio.h>
int main( void )
{
        int i = 1;
        while( (i%7) != 0 )
                printf( "%d\n" , i++ );
}
```

Note that a **for** loop can always be implemented as a **while** loop.

```
#include <stdio.h>
int main( void )
{
        for( int i=1 ; (i%7) != 0 ; i++ )
                printf( "%d\n" , i );
}
```

Loops

The do / while loop:

- Like a while loop, but is always guaranteed to perform at least one iteration (i.e. tests the condition after the loop, not before)
- Performs the calculation in braces at each iteration

```
#include <stdio.h>
int main( void )
{
        int i = 0;
        do
        {
                printf( "%d\n" , i );
                i++;
        }
        while( (i%7) != 0 );
}
```

Loops

The do / while loop:

- Like a while loop, but is always guaranteed to perform at least one iteration (i.e. tests the condition after the loop, not before)
- Performs the calculation in braces at each iteration
  - If no braces are provided, it performs the next command

```
#include <stdio.h>
int main( void )
{
        int i = 0;
        do printf( "%d\n" , i++ );
        while( (i%7) != 0 );
}
```

Loops (summary)

- `while( boolean expression ) { statements }`
  - Iterates 0 times, as long as `boolean expression` is true
  - Execute `statements` at each iteration
- `do { statements } while ( boolean expression )`
  - Iterates 1 times, as long as `boolean expression` is true
  - Execute `statements` at each iteration
- `for( init ; boolean expression ; update ) { statements }`
  - `init` happens first; usually declares & assigns "index variable"
  - Iterates 0 times, as long as `boolean expression` is true
  - Execute `statements` at each iteration
  - `update` is run after `statements`; often it increments the loop variable (`i++`)

Loops (summary)

- `while( boolean expression ) { stat`
  - Iterates 0 times, as long as `boolean exp`
  - Execute `statements` at each iteration
- `do { statements } while ( boolean e`
  - Iterates 1 times, as long as `boolean exp`
  - Execute `statements` at each iteration
- `for( init ; boolean expression`
  - `init` happens first; usually declares & ass
  - Iterates 0 times, as long as `boolean exp`
  - Execute `statements` at each iteration

```
#include <stdio.h>
int main( void )
{
        int i = 0;
        do
        {
                printf( "%d\n" , i++ );
                if( (i%7) != 0 )
                        break;
        }
        while( true );
}
```

If `statements` has the command break, the code terminates the loop regardless of whether or not `boolean expression` is true.