
Week 2

XMUT-NWEN 241 - 2026 T2

Systems Programming

Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

C Fundamentals

Identifiers

- Identifier is used to name **macros**, variables, **functions**, **structs**, **unions**, and other entities in a computer program
- Java and C have similar rules for identifiers, except:
 - In C, \$ is not allowed in identifiers (though some compilers allow \$)

Rules on Identifiers

- **An identifier is a sequence of letters and digits**
 - The first character must be a letter
 - The underscore character `_` counts as a letter
 - Upper and lower case letters are different
- **Identifiers may have any length**
 - Usually, only the first 31 characters are significant
 - For macro names, only the first 63 characters are significant
- **Reserved keywords cannot be used as identifiers!**

Examples

counter

Valid: consists of letters

_Temp_variable_2

Valid: consists of letters and digits

1myVariable

Invalid: first character is not a letter

\$steps

Invalid: \$ is not allowed in C

continue

Invalid: reserved word

Reserved Keywords

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

Data Types

- Recall: Java has 8 basic data types which have fixed sizes

Data Type	Size (bytes)
boolean	1
byte	1
char	2
short	2
int	4
long	8
float	4
double	8

Data Types

- C data types:

Data Type	Size (bytes)	
<code>boolean</code>	1	
<code>byte</code>	1	
<code>char</code>	2-1	
<code>short (short int)</code>	2 Machine-dependent	Integral types
<code>int</code>	4 Machine-dependent	
<code>long (long int)</code>	8 Machine-dependent	
<code>long long (long long int)</code>	Machine-dependent	
<code>float</code>	4 Machine-dependent	Float types
<code>double</code>	8 Machine-dependent	
<code>long double</code>	16 Machine-dependent	

Data Type Size

- Sizes of different types
 - Use `sizeof()` to find out
 - Some of the types' sizes may vary from machine to machine
- The following rules are always guaranteed:
 - `sizeof(char) == 1`
 - `sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`
 - `sizeof(float) <= sizeof(double) <= sizeof(long double)`

Data Types

- Integer types can either be signed or unsigned

```
signed int var1;    // Signed integer
```

```
unsigned int var2;  // Unsigned integer
```

```
int var1;           // If signed or unsigned is not present, default is signed
```

Data Types

```
signed int var1;    // Signed integer
```

```
unsigned int var2;  // Unsigned integer
```

```
int var1;           // If signed or unsigned is not present, default is signed
```

The easiest way to remember it:

- **Signed integer**: can store **negative and positive** numbers.
- **Unsigned integer**: can store **only zero and positive** numbers.

Data Types

Signed integers

A normal int in C is a **signed integer** by default.

```
int age = 20;  
int temperature = -5;  
int score = 100;
```

For a typical **32-bit** int, the range is:

-2,147,483,648
to
2,147,483,647

A signed integer can contain:

Negative ← 0 → Positive

-3 -2 -1 0 1 2 3

Why isn't the positive side also 2,147,483,648?

Because one bit is effectively used to distinguish the negative/positive representation, and modern C implementations normally use **two's complement** representation.

Data Types

Unsigned integers

An unsigned integer cannot represent negative numbers.

```
unsigned int age = 20;  
unsigned int students = 500;
```

For a typical 32-bit unsigned integer:

0 to 4,294,967,295

Because unsigned integers don't need to represent negative values, **all the bits can be used for non-negative values.**

Data Types

Why would we use unsigned?

Suppose you're storing someone's age.

```
int age = 25;
```

But imagine you're storing the number of **bytes in a file**.
A file can't have -500 bytes

So an unsigned type can make conceptual sense:

```
unsigned int file_size = 500;
```

Other examples where negative values don't make sense include:

- number of students
- number of items
- memory size
- pixel values
- bit masks
- array sizes
- hardware registers

char Data Type

- unsigned char: 0 to 255; signed char: -128 to 127
- char is meant to hold 1 ASCII character
 - see <https://www.asciitable.com/>

	0	NUL		1	SOH		2	STX		3	ETX		4	EOT		5	ENQ		6	ACK		7	BEL	
	8	BS		9	HT		10	NL		11	VT		12	NP		13	CR		14	SO		15	SI	
	16	DLE		17	DC1		18	DC2		19	DC3		20	DC4		21	NAK		22	SYN		23	ETB	
	24	CAN		25	EM		26	SUB		27	ESC		28	FS		29	GS		30	RS		31	US	
	32	SP		33	!		34	"		35	#		36	\$		37	%		38	&		39	'	
	40	(		41	)		42	*		43	+		44	,		45	-		46	.		47	/	
	48	0		49	1		50	2		51	3		52	4		53	5		54	6		55	7	
	56	8		57	9		58	:		59	;		60	<		61	=		62	>		63	?	
	64	@		65	A		66	B		67	C		68	D		69	E		70	F		71	G	
	72	H		73	I		74	J		75	K		76	L		77	M		78	N		79	O	
	80	P		81	Q		82	R		83	S		84	T		85	U		86	V		87	W	
	88	X		89	Y		90	Z		91	[		92	\		93	]		94	^		95	_	
	96	`		97	a		98	b		99	c		100	d		101	e		102	f		103	g	
	104	h		105	i		106	j		107	k		108	l		109	m		110	n		111	o	
	112	p		113	q		114	r		115	s		116	t		117	u		118	v		119	w	
	120	x		121	y		122	z		123	{		124			125	}		126	~		127	DEL	

Example

01000001

What do you see?

- Interpreted as an integer: 65
- Interpreted as an ASCII character: 'A'

Variable Declaration

- Similar syntax to Java
- A variable must be declared before it can be used
- A variable may be initialised in its declaration
 - If a variable name is followed by an equals sign and an expression, the latter serves as an *initialiser*

```
int i = 0, j = 1, k = 2;  
char c = 'A';  
float f = 1.25;
```

- Possible initializers
 - Constant expressions
 - Run-time expressions

Constants and Literals

- Constants are **fixed values** that cannot be changed during a program's execution
- The fixed values themselves are called **literals**
- Literals
 - Integer
 - Floating Point
 - Character
 - *String*
 - *Enumeration*

Integer Literals

- Used for representing integer-valued constants
 - Can be written in decimal (no prefix), octal (prefix 0), or hexadecimal (prefix 0x)
 - Can have suffix that is a combination of U (unsigned) and L (long) in any order
 - No suffix means the literal is of type `int`

12345

Valid

12345u

Valid: unsigned

0xbeef

Valid: hexadecimal

081

Invalid: 8 is not a valid octal digit

0x123uu

Invalid: same suffix is repeated

Floating Point Literals

- Used for representing real-valued constants
 - Can be written in decimal form or exponential form
 - Can have suffix `f` (`float`) or `L` (`long double`)
 - No suffix means the literal is of type `double`

`3.1415`

Valid (decimal form)

`31415e-4`

Valid (exponential form)

`31415e-4L`

Valid: long double

`6.22e`

Invalid: incomplete exponent

`.e23`

Invalid: missing integer/fractional part

Character Literals

- Used for representing character constants
 - Enclosed in single quotes (')
 - Can be plain (single character) or escape (single character preceded by \)

```
'A'
```

Valid (plain character)

```
'\t'
```

Valid (escape character): tab

```
'Aa'
```

Invalid: multiple characters in single quotes

```
'\z'
```

Invalid: not a valid escape character

Escape sequences

Escape sequence	Character represented
\a	Alert (bell, alarm)
\b	Backspace
\f	Form feed (new page)
\n	New-line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\\	Backslash

Declaring Constants

- Constants can be declared using `const` qualifier or `#define` pre-processor
- Such named constants are also called **symbolic constants**

```
const float PI = 3.14;  
const int MAX = 12345;
```

```
#define PI 3.14  
#define MAX 12345
```

Type Casting

- Type casting is a way to convert a variable from one data type to another data type
- C performs automatic type casting (implicit type conversion)

```
int i = 2;  
double d = 2.5;  
i = (int)d;      // explicit type casting  
  
i = d;           // d is converted to an int  
                 // and then assigned to i
```

Operators

- Java and C share many of the built-in operators
 - Arithmetic
 - Assignment
 - Increment/decrement
 - Relational
 - Equality and logical
 - Bitwise
- C specific operators
 - Pointers and reference related operators (*, &, ->)
 - Others: sizeof

Operator Precedence

- Operator *precedence* determines the sequence in which operators in an expression are evaluated
- *Associativity* determines execution for operators of equal precedence
- Precedence can be overridden by explicit grouping using parenthesis: (and)

Operator Precedence Table (not complete)

Unary operators

Arithmetic
operators

Ternary operator

Assignment operators

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
? :	right to left
= += -= *= /= %= &= ^= = <<= >>=	right to left
,	left to right

Important Things to Remember

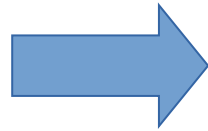
- `/` denotes integer division if both operands are of integral types
 - `5/2` evaluates to 2 (integer part is used, decimal part is truncated)
- `%` denotes modulo operation
 - `5%2` evaluates to 1 (the remainder after dividing 5 with 2)

Increment/decrement operators

`++` `--`

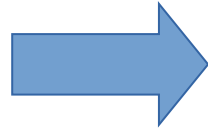
- Increase (`++`) or decrease (`--`) variable by 1
- Can be applied to variables, but not constants and ordinary expressions

`i++;`



`i = i + 1;`

`j--;`



`j = j - 1;`

- `++` and `--` are called *unary* operators because they operate on 1 operand

Increment/decrement operators

```
k++;  
counter--;
```

Valid if k and counter are variables of basic types

```
777++;  
(a + b*c)--;
```

Invalid

++ and -- can be used *postfix* or *prefix*:

```
a = b++;
```

Postfix: use the current value of b in the assignment, then increment b after the assignment

```
a = ++b;
```

Prefix: increment b first, then assign it to a

Increment/decrement operators

```
1.int a, b, c = 0;  
2.  
3.a = ++c;  
4.b = c++;
```

What are the values of a, b and c immediately after line 4?

Unary operators

Unary operators are operators that work on **only one operand** (one value or variable).

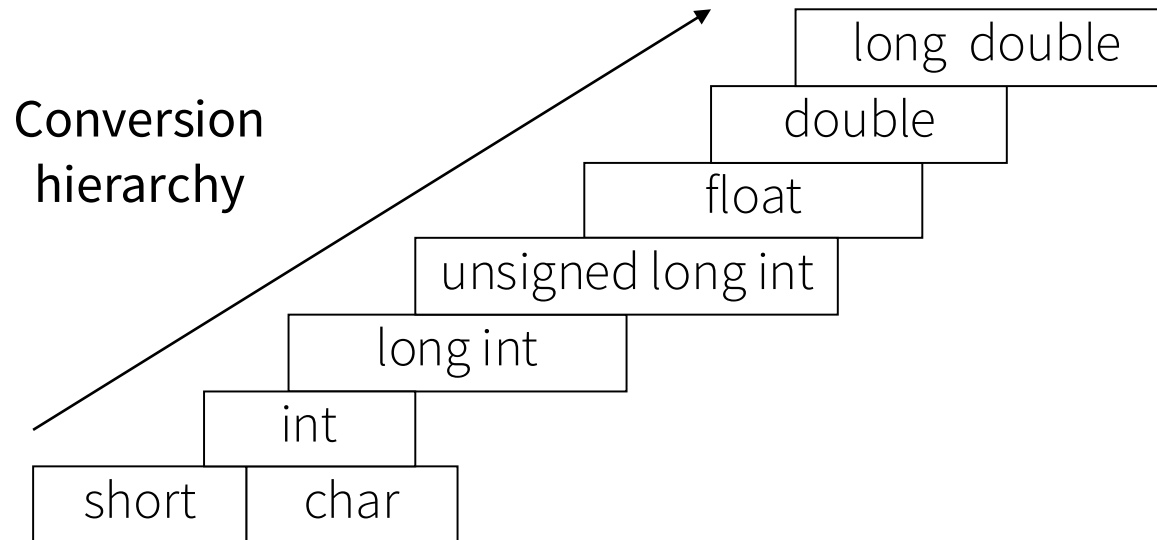
Operator	Name	Example	Meaning
+	Unary plus	+a	Positive value
-	Unary minus	-a	Changes the sign
++	Increment	++a	Increases by 1
--	Decrement	--a	Decreases by 1
!	Logical NOT	!a	Reverses true/false
~	Bitwise NOT	~a	Inverts the bits
sizeof	Size operator	sizeof(a)	Gives the size in bytes
&	Address-of	&a	Gets the memory address
*	Dereference	*p	Accesses the value at an address

True and false

- Unlike newer programming languages, C doesn't have native types for Boolean (logical *true* and *false*)
 - Zero (0) is used to denote false
 - Conceptually, one (1) is used to denote true
 - Any non-zero (positive and negative) value is also treated as true
- Relational, equality and logical operations evaluate to either true (1) or false (0)

“Conversion hierarchy”

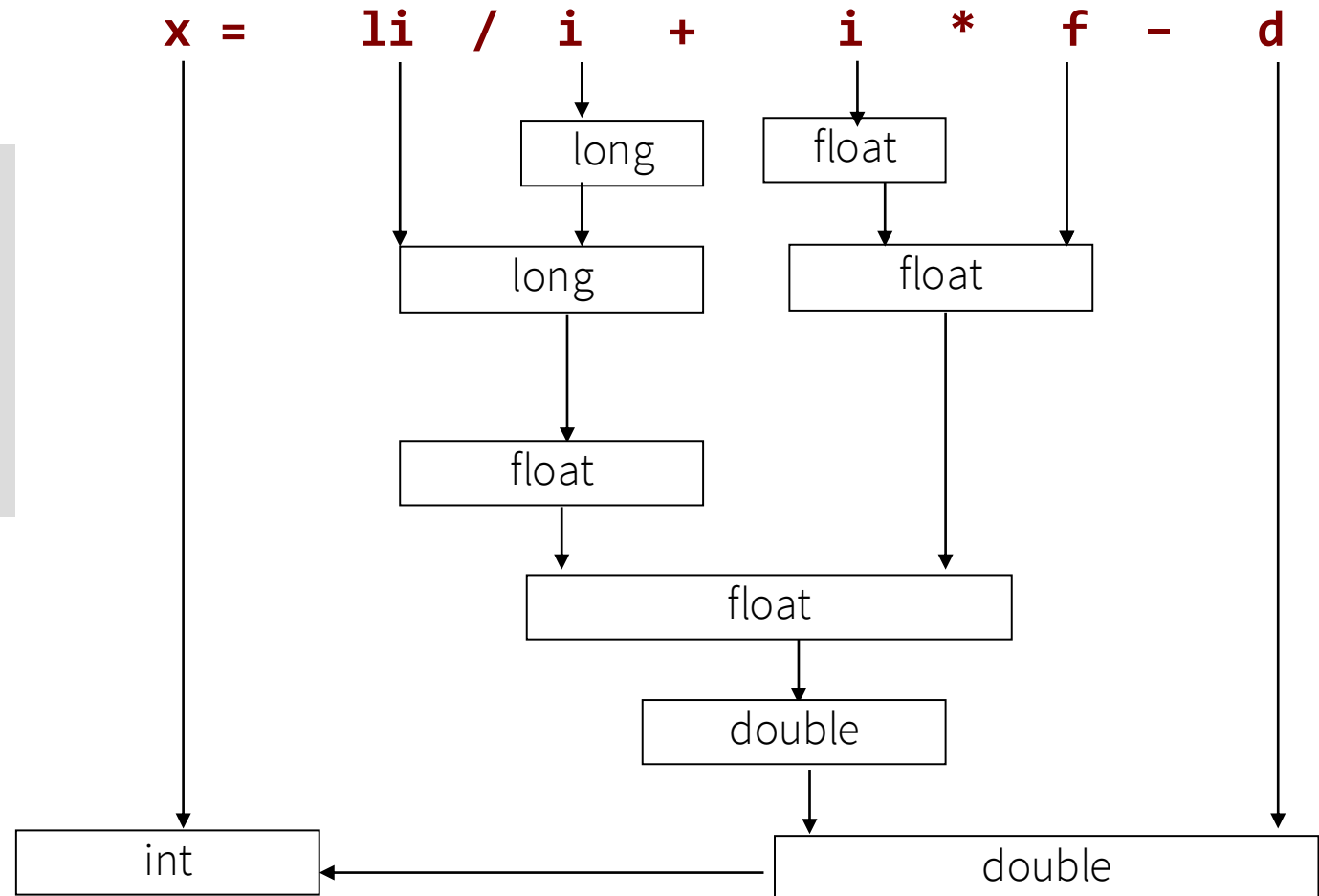
- What happens when operands have different types in an arithmetic expression?
 - **Implicit type conversion is performed:** compiler automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance



Implicit Type Conversion Example

Suppose:

```
int i, x;  
float f;  
double d;  
long int li;
```



The final result of the right hand side expression is converted to the type of the variable on the left of the assignment