Introduction to Programming

Chapter 2

Data types

Basic definitions

Data type, variable, declaration, assignment, initialization

Variables

- The computer's RAM (Read Access Memory) consists of millions of successive memory cells that are used to store data
- A **variable** is a memory location with a given name
- Naming variables:
  - can contain letters, digits, and underscore characters _
  - must begin with either a letter or the underscore character
  - must not be a C++ keyword
- Examples:
  - `count`
  - `sum`
  - `_x`
  - `sum_2`
- The value of a variable is the content of its memory location

C++ keywords and reserved words

alignas
alignof
and
and_eq
asm
atomic_cancel
atomic_commit
atomic_noexcept
auto
bitand
bitor
bool
break
case
catch
char
char8_t
char16_t
char32_t
class
compl
concept
const
consteval
constexpr
constinit
const_cast
continue
co_await
co_return
co_yield
decltype
default
delete
do
double
dynamic_cast
else
enum
explicit
export
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
noexcept
not
not_eq
nullptr
operator
or
or_eq
private
protected
public
reflexpr
register
reinterpret_cast
requires
return
short
signed
sizeof
static
static_assert
static_cast
struct
switch
synchronized
template
this
thread_local
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while
xor
xor_eq

Data types

A data type is set of values and set of operations on those values.

type	set of values	examples of values	examples of operations
`int`	integers	`17` `12345`	add, subtract, divide, multiply
`double`	floating-point numbers	`3.14159` `7.034e23`	add, subtract, divide, multiply
`char`	characters	`'a'` `'$'`	compare
`std::string`	sequence of characters	`"Hello World!"` `"C++ is fun"`	concatenate
`bool`	truth values	`true` `false`	and, or, not

Variable Declaration

- Data types define the kind of data a variable can hold
- To use a variable, you need to **declare** its data type and name

```
int age;            // Declares an integer variable named 'age’
double price;       // Declares a double variable named 'price’
char initial;       // Declares a character variable named 'initial’
bool isReady;       // Declares a boolean variable named 'isReady'
```

- We can declare multiple names on the same line:

```
int a, b, c;        // Declares three variable of type int
```

Assignment

- After declaring a variable, you can **assign** a value to it.
- Use assignment operator '='

```
age = 25;          // Assigns the value 25 to the 'age' variable
price = 9.5;       // Assigns the value 9.99 to the 'price' variable
initial = 'J';     // Assigns the character 'J' to the 'initial' variable
isReady = true;    // Assigns the value 'true' to the 'isReady' variable
```

- Right-hand side could be an expression

```
double dp = 0.75*price; // initialize 'dp' variable with 25% discount price
```

Initialization

- While declaring a variable, you can also **initialize** it.

```
int x = 123;          // traditional way
int y{ 123 };         // modern C++
int y = { 123 };      // = is optional with {…}
```

Basic data types

int, double, char, bool, std::string

Data type for computing with integers: `int`

Implementation dependent size: 4 bytes on most systems nowadays

values	integers between -2^{31} and $2^{31}-1$
typical literals	`1234` `99` `0` `1000000`
operations	add subtract multiply divide remainder
operators	`+` `-` `*` `/` `%`

Important note:

only 2^{32} different int values

↑

not quite the same as integers

Example of int operations

expression	value	comment
`5 + 3`	8	
`5 - 3`	2	
`5 * 3`	15	
`5 / 3`	1	drop fractional part
`5 % 3`	2	remainder
`1 / 0`		runtime error

Precedence

expression	value	comment
`3 * 5 - 2`	13	* has precedence
`3 + 5 / 2`	5	/ has precedence
`3 - 5 - 2`	−4	left associative
`( 3 - 5 ) - 2`	−4	better style

Example of computing with integers

```
#include<iostream>


int main() {
    int a, b;
    std::cin >> a >> b;
    int sum = a + b;
    int prod = a * b;
    int quot = a / b;
    int rem = a % b;
    std::cout << a << " + " << b << " = " << sum << std::endl;
    std::cout << a << " * " << b << " = " << prod << std::endl;
    std::cout << a << " / " << b << " = " << quot << std::endl;
    std::cout << a << " % " << b << " = " << rem << std::endl;
}
```

```
➢ g++ intops.cpp

➢ a
1234 99
1234 + 99 = 1333
1234 * 99 = 122166
1234 / 99 = 12
1234 % 99 = 46
```

user input

Data type for computing with floating point numbers: `double`

values	real numbers
typical literals	`3.14159 2.0 1.4142135623730951 6.022e23`
operations	add subtract multiply divide
operators	+ - * /

$6.022 * 10^{23}$

Typical `double` values are *approximations*

Examples:
- no `double` value for π
- no `double` value for $\sqrt{2}$
- no `double` value for $1/3$

Example of `double` operations

expression	value
`3.141 + .03`	3.171
`3.141 - .03`	3.111
`6.02e23/2`	3.01e23
`5.0 / 3.0`	1.6666666666666667
`sqrt(2.0)`	1.4142135623730951

Special values

expression	value
`1.0 / 0.0`	**inf**
`sqrt(-1.0)`	**nan**

Example: *quadratic equation*

From algebra: the roots of $x^2 + bx + c$ are $\frac{-b \pm \sqrt{b^2 - 4c}}{2}$

```
#include<iostream>
#include<cmath>
int main() {
    double b, c;
    std::cin >> b >> c; // input coefficients
    // Calculate roots of x*x + b*x + c.
    double discriminant = b*b - 4.0*c;
    double d = sqrt(discriminant);
    double root1 = (-b + d) / 2.0;
    double root2 = (-b - d) / 2.0;
    // Print them out.
    std::cout << root1 << " " << root2 << std::endl;
}
```

```
➢ g++ quadratic.cpp

➢ a
-3.0 2.0          x^2 − 3x + 2
2 1

➢ a
-1.0 -1.0         x^2 − x − 1
1.61803 -0.618034
```

Data type for handling characters: `char`

values	ASCII characters
typical literals	`'H'` `'3'` `'#'` `'\n'`
operations	increment
operators	++

ASCII representation:

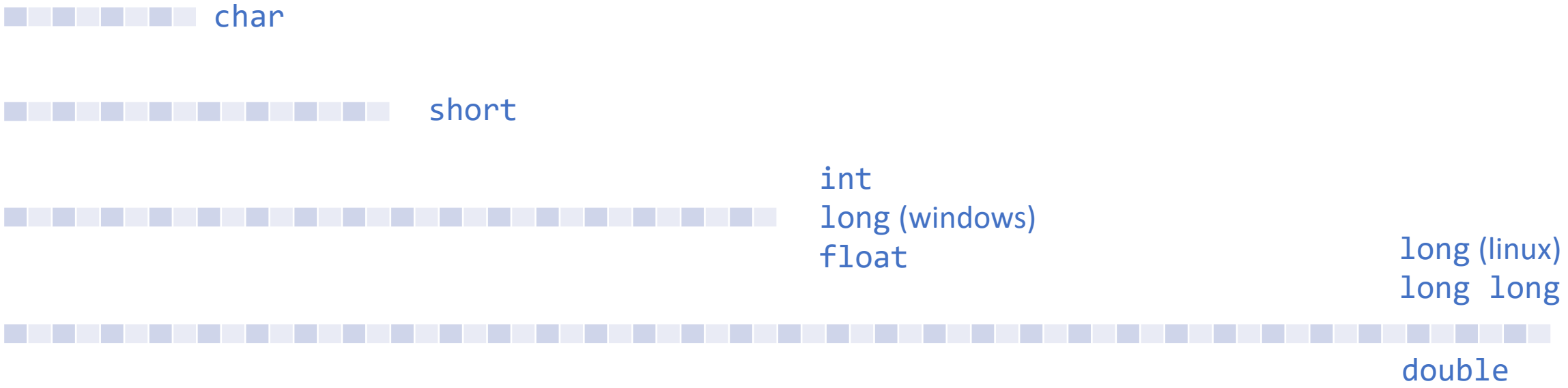
Each character is assigned a unique numeric value (0 to 127)
First 32 are control codes (non-printable)
Remaining 96 character-codes are representable characters
Example: ASCII value of `'A'` is 65, `'a'` is 97, '0' is 48, and so on

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Other built-in numeric types

Why different numeric types?

- Tradeoff between memory use and range for integers.
- Tradeoff between memory use and precision for real numbers.



Other built-in numeric types

Group	Type names*	Notes on size / precision	Usual size (in bytes)
Character types	`char`	Exactly one byte in size. At least 8 bits.	1
	`char16_t`	Not smaller than char. At least 16 bits.	2
	`char32_t`	Not smaller than char16_t. At least 32 bits.	4
	`wchar_t`	Can represent the largest supported character set.	2 / 4
Integer types (signed)	`signed char`	Same size as char. At least 8 bits.	1
	`signed short int`	Not smaller than char. At least 16 bits.	2
	`signed int`	Not smaller than short. At least 16 bits.	4
	`signed long int`	Not smaller than int. At least 32 bits.	4 / 8
	`signed long long int`	Not smaller than long. At least 64 bits.	8
Integer types (unsigned)	`unsigned char`	(same size as their signed counterparts)	
	`unsigned short int`		
	`unsigned int`		
	`unsigned long int`		
	`unsigned long long int`		
Floating-point types	`float`		4
	`double`	Precision not less than float	8
	`long double`	Precision not less than double	16

Excerpts from C++ standard library

```
#include <cmath>
```

`double fabs(double a)`	absolute value of a
`double fmax(double a, double b)`	maximum of a and b
`double fmin(double, double)`	minimum of a and b
`double sin(double theta)`	sine function
`double cos(double theta)`	cosine function
`double tan(double theta)`	tangent function
`double exp(double a)`	exponential (e^a)
`double log(double a)`	natural logarithm ($\log_e a$ or $\ln a$)
`double pow(double a, double b)`	raise a to the bth power (a^b)
`long lround(double a)`	round to the nearest integer
`double sqrt(double a)`	square root of a

Excerpts from C++ standard library

```
#include <numbers>
```

`double std::numbers::e`	value of e (constant)
`double std::numbers::pi`	value of π (constant)

```
#include <cstdlib>
```

`int rand()`	generate a pseudo-random integer between 0 and RAND_MAX
`void srand(unsigned seed)`	seeds the pseudo-random number generator

Data type for computing with true and false: `bool`

values	true	false	
typical literals	`true`	`false`	
operations	and	or	not
operators	&&	||	!

logical not

a	true	false
!a	false	true

and operator &&

&&		a	
		`true`	`false`
b	`true`	`true`	`false`
	`false`	`false`	`false`

or operator ||

||		a	
		`true`	`false`
b	`true`	`true`	`true`
	`false`	`true`	`false`

Typical usage: Control logic and flow of a program (stay tuned)

Comparison operators

Fundamental operations that are defined for each primitive type allow us to compare values.

- Operands: two expressions of the same type.
- Result: a value of type `bool`

operator	*meaning*	`true`	`false`
==	equal	2 == 2	2 == 3
!=	not equal	3 != 2	2 != 2
<	less than	2 < 13	2 < 2
<=	less than or equal	2 <= 2	3 <= 2
>	greater than	13 > 2	2 > 13
>=	greater than or equal	3 >= 2	2 >= 3

typical double values are approximations so beware of == comparison

Examples	*no-negative discriminant?*	`( b*b - 4.0*a*c ) >= 0.0`
	beginning of century	`( year % 100 ) == 0`
	legal month	`( month >= 1 ) && ( month <= 12 )`

Example of computing with `bool`: leap year test

Q. Is a given year a leap year?

A. Yes if either (i) divisible by 400 or (ii) divisible by 4 but not 100.

```
#include<iostream>

int main() {
    int year;
    std::cin >> year;
    bool isLeapYear;

    // divisible by 4 but not 100
    isLeapYear = (year % 4 == 0) && (year % 100 != 0);

    // or divisible by 400
    isLeapYear = isLeapYear || (year % 400 == 0);

    std::cout << std::boolalpha << isLeapYear;
}
```

enable printing of bool values as true/false instead on 1/0

Data type for computing with strings: `std::string`

values	sequence of characters[1]
typical literals[2]	`"Hello, "` `"1 "` `" * "`
operations	concatenation
operators	+

[1] infinite many possible values

[2] `std::string` can be constructed from these literals

character interpretation depends on the context

```
std::string s; // create empty string
```

Ex 1: plus sign

```
s + "1234" + " + " + "99"
```

character

operator operator operator

Examples of std::string operations (concatenation)

```
std::string s1 { "Hello, " };
std::string s2 { s1 + "ITP class" };
std::string s3 = s2 + '\n';
std::cout << s3;
```

Ex 2: spaces

```
s + "1234" + " + " + " + "99"
```

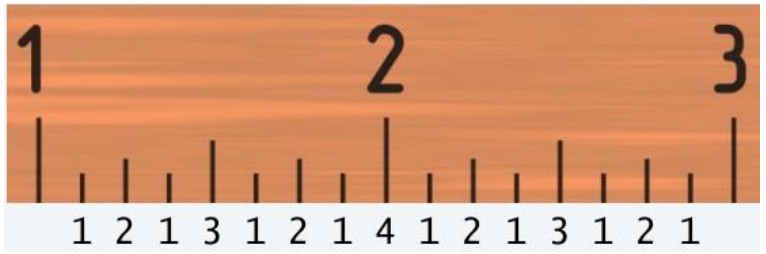
white space white space

space characters

Example of computing with strings: *subdivisions of a ruler*

```
#include<iostream>
#include<string>

int main() {
    std::string ruler1 = "1";
    std::string ruler2 = ruler1 + " 2 " + ruler1;
    std::string ruler3 = ruler2 + " 3 " + ruler2;
    std::string ruler4 = ruler3 + " 4 " + ruler3;
    std::cout << ruler4;
}
```



	ruler1	ruler2	ruler3	ruler4
	undeclared	undeclared	undeclared	undeclared
ruler1 = "1";	1	undeclared	undeclared	undeclared
ruler2 = ruler1 + " 2 " + ruler1;	1	1 2 1	undeclared	undeclared
ruler3 = ruler2 + " 3 " + ruler2;	1	1 2 1	1 2 1 3 1 2 1	undeclared
ruler4 = ruler3 + " 4 " + ruler3;	1	1 2 1	1 2 1 3 1 2 1	1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

Type conversions

Type checking

Types of variables involved in data-type operations always must match the definitions

The C++ compiler is your friend: it checks for type errors in your code

```
int main() {
    int x {3.5};
}
```

```
➢ g++ main.cpp
  main.cpp:2:12: error: narrowing conversion of '3.5e+0' from 'double' to 'int'
     2 |     int x {3.5};
       |            ^~~
```

When appropriate, we often convert a value from one type to another to make types match

Type conversion with built-in types

Type conversion is an essential aspect of programming.

Automatic

- Convert char to integer
- Make numeric types match, try not to lose precision

expression	type	value
'A'+1	int	66
11 * 0.25	double	2.75

Explicitly defined for function call

lround(2.71828)	long	3
std::to_string(123)	std::string	"123"

Cast for values that belong to multiple types.

- Ex: small integers can be short, int or long.
- Ex: double values can be truncated to int values.

(int) 2.71828	int	2
(int) lround(2.71828)	int	3
11 * (int) 0.25	int	0

Pay attention to the type of your data ← Type conversion can give counterintuitive results but gets easier to understand with practice