Chapter 2:
C Data Types

References:

- Kernighan/Ritchie: The C Programming Language, 2nd Ed. [1988]
  Chapter 2: Types, Operators, and Expressions (pages 35–54)
  Appendix A: Reference Manual (pages 191–240)
  Appendix B2: Character Class Tests (p. 248). Appendix B11: Standard Library: Implementation-defined Limits (p. 257).
- Wirth: Algorithmen und Datenstrukturen [Teubner, 1983]
- Güting: Datenstrukturen und Algorithmen [Teubner, 1992]
- Ehrich/Gogolla/Lipeck: Algebraische Spezifikation abstrakter Datentypen [Teubner, 1989]
- `http://www.unicode.org`
- Petzold: Programming Windows, 5th Ed. [Microsoft Press, 1998]

Overview

1. Data Type Foundations

2. Basic C Types

3. Variables (L-Values)

4. Pointers, Arrays and Strings

5. Structures, Unions, Enumeration Types

Data Types (1)

- A data type is
  - a set of values together with
  - operations (functions) on that set.
- E.g. the set of real numbers together with the operations $+$, $-$, $*$, $/$.
- Often one has a system of data types, i.e. multiple named sets and operations that can have input and output values from different sets.

  E.g. < is an operation that takes two numbers and returns a boolean value (true or false).

Data Types (2)

- The "signature" of an operation specifies the input and output sets, e.g.:

$$\begin{array}{lcl} + & : & \text{integer} \times \text{integer} \longrightarrow \text{integer} \\ < & : & \text{integer} \times \text{integer} \longrightarrow \text{boolean} \\ \text{length} & : & \text{string} \longrightarrow \text{integer} \end{array}$$

- Operators can be overloaded, i.e. the same operation name can be used for two different functions that are defined on different input sets, e.g.:

$$\begin{array}{lcl} + & : & \text{integer} \times \text{integer} \longrightarrow \text{integer} \\ + & : & \text{real} \times \text{real} \longrightarrow \text{real} \end{array}$$

Implementations (1)

- Implementations of data types on computers are often limited.

  E.g. the set of integers might be restricted to -32 768..+32 767 (on old 16-bit machines) or to -2 147 483 648..+2 147 483 647 (32-bit).

- If the result of an operation falls outside the implemented subset, the result is undefined. E.g.:
  - The program is aborted with an error message.
  - The program continues to work, but with a wrong value.

    E.g. additing two large positive numbers gives a negative number.

Implementations (2)

- When programming, one has to be aware of such limitations. One must make sure that the user gets never a wrong result (maybe a clear error message).
- In case of the real numbers, also the precision is limited: Results of operations are rounded.
- E.g. if we assign `float x = 1.0/3.0;` and print `x`, we might get `0.333333343267440 8`.
- This might lead to the violation of common arithmetic laws, e.g. associativity.

Type Conversions (1)

- Sometimes the compiler introduces "invisible" operations that transform a value from one type into another type (type coercions).
- E.g. suppose we write $1 + 0.5$.
  There might be only two versions of $+$:

$$+ \; : \; \text{integer} \times \text{integer} \longrightarrow \text{integer}$$
$$+ \; : \; \text{real} \times \text{real} \longrightarrow \text{real}$$

- Most compilers will then automatically introduce an operation that converts the integer 1 into the real number 1.0 (very different internal representation).

Type Conversions (2)

- If there is no operation with a fitting signature and no automatic type conversions are possible, the compiler will report a type error.
- E.g. `1 + 'a'` makes little sense.
- In C this is actually legal, since `'a'` stands for the code of the letter `a`.

  So the result will be 98 on systems that use ASCII codes.
- Too many automatic conversions can be confusing and lead to undetected errors.

Overview

1. Data Type Foundations

2. Basic C Types

3. Variables (L-Values)

4. Pointers, Arrays and Strings

5. Structures, Unions, Enumeration Types

Basic C Types

- The basic data types of C are:
  - integers (various lengths, signed/unsigned)
  - characters (which are actually small integers)
  - floating point numbers (real values)
  - Enumeration types: see end of chapter.
  - `void`: Empty set of values (no value).
- C has (unfortunately) no boolean data type, but one can use integers for that purpose.

  0 means false, any other value means true.

Integer Types (1)

- The signed integer types of C are
  - `short int` (can be abbreviated as `short`)

    This is usually a 16-bit value: `-32 768..+32 767`.
    C guarantees that at least the range `-32767..+32767` is implemented.

  - `int`

    At least the interval `-32767..+32767` is implemented, but on modern machines `int` is often 32 bit. The standard C library has a file `limits.h` that contains constants like `INT_MIN` and `INT_MAX`.

  - `long int` (can be abbreviated as `long`)

    Often 32 bit value: `-2 147 483 648..+2 147 483 647`
    C guarantees that at least `-2147483647..+2147483647` is implemented.
    C also guarantees Values(short) $\subseteq$ Values(int) $\subseteq$ Values(long).
    Some modern implementations also have `long long int` (64 bit).

Integer Types (2)

- Each of the three types also has an unsigned variant, e.g. `unsigned short` may be `0..65535`.

  `unsigned int` can be abbreviated to `unsigned`.

- Use these types only if the larger range of values is required or for bit operations.

  If one uses this type in order to state that a variable value will always be non-negative, one might get compiler warnings for assignments of `unsigned int` to `int` (also, one does `unsigned` arithmetic, see below).

- One can also write, e.g. `signed long int`, but the three integer types are signed by default.

  `signed` is important for `char`, see below.

Internal Representation (1)

- Integers are usually represented in binary, i.e. as base-2 number (with the digits 0 and 1):

... 16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0
$\pm 2^{15}$	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

- E.g. $1 * 2^6 + 1 * 2^5 + 1 * 2^2 = 64 + 32 + 4 = 100$.
- `unsigned` integers with b bits can represent the range from 0 to

$$\sum_{i=0}^{b-1} 2^i = 2^b - 1.$$

Internal Representation (2)

- In C, all arithmetic (`+`, `-`, etc.) with `unsigned` values is officially modulo 2^b where b is the number of bits.

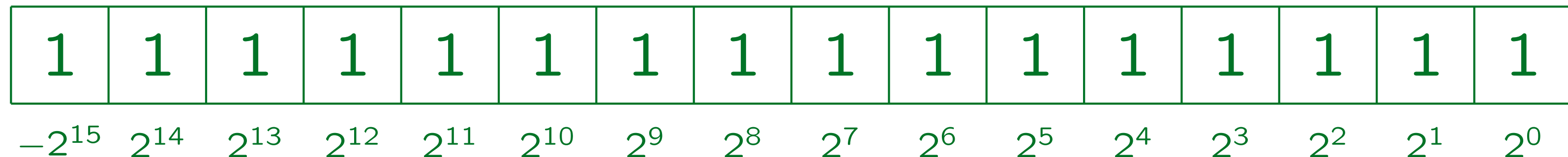
  It is by definition correct that e.g. $65\,535 + 1 = 0$ if the size is 16 bit. Unsigned arithmetic cannot overflow. If e.g. `-1` is converted to `unsigned`, the result is 65 535 (modulo 2^{16}, this is the same).

- Negative numbers are today usually represented as "two's complement", i.e. one complements all bits of the corresponding positive number and adds 1.

  C does not specify the encoding of negative numbers since there are (or were) machines that used a different encoding. If C guaranteed anything, implementations on those machines would be very inefficient. C also does not specify a behaviour for overflows (for signed numbers).

Internal Representation (3)

- Two's complement means that the first binary digit has the value -2^{b-1} (where b is the number of bits).
- E.g. the number -1 is encoded as all 1-bits:



- There is one more negative number than there are positive numbers. The range of values that is encoded with b bits is $-2^{b-1} \ \ldots \ 2^{b-1} - 1$.

Characters (1)

- The data type for characters is called `char` in C.
- Characters are treated like small numbers, e.g. in the range `-128..127` or `0..255`.

  Values of type `char` are today usually encoded in 8-bit bytes.
- So C really works with the character codes.

  The encoding is dependent on the machine. Today it is often the ASCII encoding, but there are still machines with different encodings. Many different encodings of national characters like ä, ö, ü are still in use, although there is some hope that in future the ISO Latin 1 Code (8859/1) will be used more universally (it is a superset of ASCII). The standard C library has a file `ctype.h` with functions (macros) like `isalpha(`c`)`, `isupper(`c`)`, `islower(`c`)`, `isdigit(`c`)`, `isspace(`c`)`.

Characters (2)

- C automatically transforms `char` into `int` values, but it is machine-dependent whether the result can be negative or not.

  One can use the types `unsigned char` or `signed char` to request a specific transformation. C guarantees the range `-127..+127` for signed characters and `0..255` for unsigned characters (i.e. `char` has at least 8 bit).

- Of course, one should try to write programs that are portable, i.e. that do not depend on the specific machine or compiler.

  E.g. they should not assume that characters are encoded in ASCII or that characters are signed/unsigned.

ASCII Codes

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

Extended Character Sets

- Asian languages need more than 256 characters.
- Also, Unicode will be used more often. This standard tries to encode all alphabets of the world.

  It needs at least 16 bit (there are already extensions to 32 bit). Windows NT uses internally Unicode with 2 bytes for every character (in memory, on the disk there are more compact encodings).

- The standard library has a file `stddef.h` that declares a type `wchar_t` (e.g. 16 bit). Constants of this type are written `L'a'` (wide character constants).

  Of course, there are also wide character strings: `L"abc"`. Some systems also have a file `wchar.h`.

Arithmetic Operations (1)

- The operations `+`, `-`, `*`, `/` can be used on integers and floating point numbers.
- All are overloaded with the signature $T \times T \rightarrow T$, where T is one of the following types: **`int`**, **`unsigned int`**, **`long`**, **`unsigned long`**, **`float`**, **`double`**, **`long double`**.
- If the two operands have different types, the type that comes earlier in the list is converted to the one that comes later in the list.

  Exception: If the two operands have types `long` and `unsigned int`, they are converted to `long` if `long` has more bits than `unsigned int` or `unsigned long` otherwise (in order not to lose unsigned values).

Arithmetic Operations (2)

- Arithmetic is not done on `short` and `char` values, these are first converted to `int`.

  Again with the exception that if `short` and `int` are the same size, `unsigned short` is converted to `unsigned int`.

- `+` and `-` also have the signature $T \rightarrow T$, i.e. they are not only a binary operation, but can also be used as a monadic operation.

  E.g. `-5` is understood as the monadic `-` applied to the number `5`. The monadic `+` does nothing (identity mapping).

Arithmetic Operations (3)

- If the division / is applied to two integer values, the result will be an integer (integer division).
- If both operands are positive, C always rounds down (truncates), e.g. $5/3 = 1$.
- The operator % gives the remainder of the division, e.g. $5 \% 3 = 2$.

  The operands must be of integral type (i.e. `int`, `unsigned int`, `long`, `unsigned long`). For positive operands, `A % B` is non-negative and smaller than `B`. If one of the operands is negative, C only guarantees that the absolute value of `A % B` is smaller than the absolute value of `B`.

- C guarantees that `A = (A/B)*B + A%B`.

Arithmetic Operations (4)

- The result of / and % is undefined if the right operand is zero ("division by zero").
- It is quite likely (but not guaranteed) that the computer hardware will treat this as serious error: It will jump to an "exception handler" that will then terminate the program.

  It might be possible to declare one's own routine for this event.
- This is a case of a "runtime error" that the compiler cannot detect and that might depend on the input.

Shift Operations

- The result of `A << B` is `A` left-shifted `B` bits.

  A << B is $A * 2^B$. On many machines, the shift is faster than the multiplication. The compiler should automatically replace a multiplication by a shift if one of the multiplication operands is a constant.

- The result of `A >> B` is `A` right-shifted `B` bits.

  A >> B is $A \,/\, 2^B$ if A is not negative. Otherwise the result depends on the machine (the sign bit might or might not be propagated).

- Both shift operations have signatures of the form $T \times S \rightarrow T$, where T and S are integral types.

- The right operand (`B`) must be not negative.

Bit Operations (1)

- One can do boolean operations on every bit of the integer types.
- E.g. 32 boolean values can be encoded in a single **`unsigned long`** value.

  I.e. the **`unsigned long`** can be seen as an array of boolean values.
- One can then do operations like "and" and "or" very efficiently on the entire set of boolean values.

  E.g. 32 "and"-operations in parallel with a single machine instruction.

Bit Operations (2)

- C has the binary operators `&` (bit-and), `|` (bit-or), `^` (bit-xor) and the monadic operator `~` (bit-not) for working on the single bits of an integer value:

A	B	A & B	A | B	A ^ B	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

- All have the signature $T \times T \rightarrow T$, where T is one of: **int**, **unsigned int**, **long**, **unsigned long**.

  char and **short** are (at least conceptionally) first converted to **int**.

Comparison Operators

- The six comparison operators are: `==` (equal-to), `!=` (not-equal-to), `<` (less-than), `<=` (less-than-or-equal), `>` (greater-than), `>=` (greater-than-or-equal).

  Comparison operators are also called relational operators.

- All have the signature $T \times T \longrightarrow$ **`int`**.

  T can be one of `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`. In addition, it is possible to compare pointers, see below.

- The result of the comparison is `1` if the relation holds and `0` if it does not hold.

Logical Operators (1)

- C has two logical operators: `&&` (and) and `||` (or).
- `A && B` is `1` if `A` and `B` are both unequal to zero, otherwise the result is `0`.

  C treats everything unequal to `0` as true, and `0` as false. So this really means that `A && B` is true iff `A` and `B` are both true, otherwise (at least one of `A` and `B` is false) `A && B` is false. C guarantees that the comparison and logical operators return only `0` and `1`, but on input it is more liberal, e.g. `3 && -27` would be legal.

- `A || B` is `0` if `A` and `B` are both zero, otherwise it is `1`.

  I.e. `A || B` is true if at least one of `A` and `B` is true. Otherwise (`A` and `B` are both false) `A || B` is false.

Logical Operators (2)

- C guarantees that the two logical operators first evaluate the left operand and then the right operand only if it is still needed to determine the result value.

  For `A && B`, if `A` is 0, the result is 0, and `B` is not evaluated.
  For `A || B`, if `A` is not zero, the result is 1, and `B` is not evaluated.

- This legalizes conditions like `A == 0 || B / A == 0`.

  In other languages, e.g. Pascal, the compiler writer is free to evaluate the right condition even if the left one is already true (e.g. he/she could evaluate the right one first). This would then give a runtime error ("division by zero"). This can create subtle portability problems.

Overview

1. Data Type Foundations
2. Basic C Types
3. Variables (L-Values)
4. Pointers, Arrays and Strings
5. Structures, Unions, Enumeration Types

Variables (1)

- A variable is a storage space (container) for a value.
- The two basic operations of a variable are:
  - Assign a value to the variable.

    I.e. put that value into the container.
  - Get the current value of the variable.

    One actually gets a copy of the value. The value remains in the variable until another value is assigned.
- Variables are typed: Each variable can store only values of a certain type.

  That type is defined in the declaration of the variable. Variables must be declared before they can be used.

Variables (2)

- One must first assign a value before one can retrieve the current value of the variable.

  This is the responsibility of the programmer: The computer hardware cannot detect this error, it will simply return garbage. Of course, the compiler can generate code that checks for the initialization of variables, but that will be quite costly (slows down the program).

- A variable X of type T can be used as an operand of an operation that requires type T — the compiler will automatically take the value of X.

  So here X can be used like a constant of type T.

Variables (3)

- C calls a variable of type T an lvalue of type T.

  Because it is a value that can appear on the left side of an assignment.

- It is possible to understand variables of type T as having the type `lvalue`(T).

  This is my personal interpretation of the standard. No such type is explicitly introduced.

- A value of type `lvalue`(T) is automatically converted into a value of type T whereever this is required.

- This is done by looking up the value currently stored in the variable.

Assignment Operator (1)

- The assignment operator is written in C as "`=`".

  This is a source for possible errors, since the test for equality is written as ==, which is slightly unusual. The reason is that an assignment is used very often (more often than an equality test), therefore it was considered as important that the assignment needs as few as possible keystrokes. C is always very compact.

- It is overloaded with the types $\texttt{lvalue}(T) \times S \rightarrow T$ where T and S are an arithmetic type, a pointer type or a structure/union type (see below).

- S and T must be assignment-compatible.

  For structure/union and pointer types, this basically means that $S = T$, although there are certian exceptions for pointers (see below).

Assignment Operator (2)

- Arithmetic types are: `char`, `signed char`, `unsigned char`, `short`, `unsigned short int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double` and enumeration types (see below).
- All arithmetic types are assignment-compatible:
  If S and T are different, the value of type S is converted to type T.

  It is bad style to assign e.g. a `long` value to an `int` variable, since there might be `long` values that exceed the range of `int` values. In this case the result is undefined. Some compilers might issue warnings, but the assignment is legal. One should use an explicit type-cast (see below).

Assignment Operator (3)

- Unlike many other languages (e.g. Pascal), the assignment has a value in C, namely the value that is stored in the variable.
- The assignment can be part of a larger expression.

  E.g. the following is quite typical:

  ```
  while((c = getc(stdin)) != EOF)
          /* process character c */;
  ```

  This assigns the next input character to the variable `c` and then tests whether it is `EOF` (end-of-file).

  As mentioned before, C is always very compact.

Assignment Operator (4)

- This is a bit dangerous, since e.g. the following is legal C:

```
if(n = 0) ...;
```

- This assigns `0` to the variable `n`. The value of the assignment is `0`, i.e. false, thus the dependent statement "..." is never executed.
- Since this is a very common error, modern compilers will issue a warning.

  One should always switch to the highest warning level and try to get a clean compile without any warnings.

Abbreviated Assignments (1)

- `++X` is an abbreviation for `X = X+1`.
- To "increment a variable is to add one to it. Therefore `++` is called the increment operator.

  Adding one to a variable is a very common operation. Since C is very compact, it has a special notation for this case.
- As usual, the value of the assignment is the assigned value.
- E.g. if `X` had the value 3 before the `++X`, it will have the value 4 afterwards, and 4 is also the value the entire operation `++X`.

Abbreviated Assignments (2)

- There is also a variant `X++`. This increments `X`, too, but the value of `X++` is the value of `X` before it was incremented.

  This is easy to remember: If you write ++ before the variable, it is incremented before its value is taken. If you write ++ after the variable, it is incremented after its value is taken.

- If the value of the `++X` and `X++` is not used (i.e. it is only executed for the purpose of incrementing the variable), there is no difference between the two variants.

Abbreviated Assignments (3)

- There are also decrement operators `--X` and `X--` that subtract one from the variable.
- `X += Y` is an abbreviation for `X = X + Y`.
- E.g. `++X` and `X += 1` are equivalent.
- In the same way, there are abbreviated assignments for all binary operators: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`.

Exercise

What does the following program print?

```
 (1)   #include <stdio.h>
 (2)   int main()
 (3)   {   int i, n, m;
 (4)       n = 5;
 (5)       m = 2;
 (6)       n += m;
 (7)       printf("%d %d\n", n++, ++m);
 (8)       printf("%d\n", n << 2);
 (9)       printf("%d\n", n * m);
(10)       printf("%d\n", i);
(11)       return(0);
(12)   }
```

Implementation: Memory (1)

- The computer has main memory (RAM) that is a modifiable mapping from addresses to values.
- The smallest addressable unit is usually an 8-bit value (a byte).
- A typical size of RAM for a PC today is 128 MB.

  Larger machines might have several GB.

- Then the main memory (at a particular point in time) can be understood as a mapping

$$M\colon \texttt{[0..134\,217\,727]} \rightarrow \texttt{[0..255]}$$

Implementation: Memory (2)

- Since the mapping is modifiable, M describes only a state of the main memory at a given point in time.
- Storing the value v at address a gives the mapping M':

$$M'(n) := \begin{cases} M(n) & \text{if } n \neq a \\ v & \text{if } n = a \end{cases}$$

- This mapping M' replaces the current mapping M, i.e. the main memory implements only one such mapping at each point in time.

Implementation: Memory (3)

- The compiler implements a variable x of type `char` by allocating a particular address a for that variable.
- Then the assignment of a value v to x generates an instruction that stores v at a.
- If the value of variable x is accessed in the program, the value currently stored at address a is retrieved.

  If no value was previously assigned to x, the value currently stored at a cannot be predicted. Many operating systems initialize the main memory used by a program to 0 before it can access it. However, the compiler will reuse the same address a for different variables that have non-overlapping lifetimes (see below).

Implementation: Memory (4)

- Depending on its type, a variable might need more than one byte of storage space.
- E.g. a variable of type `int` needs 4 bytes on a 32-bit machine.
- The compiler will allocate 4 consecutive bytes in memory.
- The variable still has a single address a, but the bytes at addresses $a+1$, $a+2$, and $a+3$ are also reserved for that variable.

Implementation: Memory (4)

- Modern CPUs have instructions that can directly work with 32-bit quantities.
- However, it might be a requirement that the start address a can be evenly divided by 4 ("alignment").

  The reason is that the RAM actually consists of 32-bit "words" (or even 64-bit double-words). So the address is internally split into a part that addresses a word and a part that selects one or more bytes in that word. Although each single byte can be addressed, it is not possible to access bytes from different words in a single memory cycle. Even if the machine can execute 32-bit accesses at addresses that are not multiples of 4, they will be slower (they require two memory cycles).
- The compiler respects these requirements.

Implementation: Memory (5)

- Machines have different byte orders.
- If a 32-bit quantity is stored in the bytes with addresses a, $a+1$, $a+2$, $a+3$, it is not clear which byte will contain which 8 bit parts.
- E.g. if the number `0x44332211` is stored in a 32-bit value at address a, the contents of the single bytes will be as follows:

Adress:	$a+0$	$a+1$	$a+2$	$a+3$
Contents (Sparc):	0x44	0x33	0x22	0x11
Contents (Intel):	0x11	0x22	0x33	0x44

`sizeof` Operator (1)

- C has an operator `sizeof` to determine the number of bytes needed for a type/value.
- One can use it in two ways:
  - With a type as argument, e.g. `sizeof(int)` will return 4 on 32-bit machines.

    Some types are excluded: Functions, bit-fields, incomplete types.
  - With an expression as argument, e.g. `sizeof(s)` will return 2 on many machines if `s` is declared as a variable of type `short`.

    The expression is not evaluated, the compiler determines only its type (important if the expression contains assignments).

sizeof Operator (2)

- By definition, `sizeof(char)` always returns `1`.
- The result of the `sizeof` operator has type `size_t` declared in the standard library file `stddef.h`.

  Today, it is often `unsigned long`. In earlier compilers, it was `unsigned int` or `int`. Often, variables could not be larger than 64 KB, therefore a 16-bit `unsigned int` was sufficient. Since the compilers differed in this aspect, the simplest solution was to declare a type name in the standard library. Portable programs should use `size_t`.

- The `sizeof` operator is e.g. needed when memory is dynamically allocated at runtime.

  The library procedure `malloc` requires as input the number of bytes needed (see below).

Overview

1. Data Type Foundations

2. Basic C Types

3. Variables (L-Values)

4. Pointers, Arrays and Strings

5. Structures, Unions, Enumeration Types

Pointers (1)

- An object of type $\texttt{pointer}(T)$ contains the memory address of a variable of type T.

  The notation $\texttt{pointer}(T)$ is only used in this theoretical description of the type system. In C, a variable of pointer type is declared in the form `char *p;`

- Pointer variables can be dereferenced, i.e. one can determine the variable to which it points.
- This is done with the prefix operator "`*`" in C, e.g. `*p` is the variable to which `p` points.
- `*` has the signature $\texttt{pointer}(T) \rightarrow \texttt{lvalue}(T)$.

Pointers (2)

- E.g. this is a legal assignment: `*p = 'a';`
- Since `p` is itself a variable, it really has the type `lvalue(pointer(char))`, but the compiler automatically transforms `lvalue`(T) $\rightarrow$ T (as usual).
- In C, one can also determine the memory address of a variable. This is done with the prefix operator `&`. It has the signature `lvalue`(T) $\rightarrow$ `pointer`(T).
- E.g. if `c` is a variable of type `char`, then `&c` is a pointer to `c` (its address). It has the type `pointer(char)`.

Pointers (3)

- Pointers are normally typed in C, and one can assign a value of type `pointer`(T) only to a variable of the same pointer type.
- However, the special typeless pointer type `void*` (`pointer(void)`) is allowed on the left and right side of assigments where the other side contains a possible different pointer type.

  E.g. library procedure responsible for memory allocation (`malloc`) is declared to have the return type `void*`. In this way, its result can be assigned to any pointer variable, no matter of what type.

NIL Pointer

- A pointer variable can also be assigned the number 0, but this must be explicitly written as a constant on the right side of the assignment.

  I.e. the compiler must be able to verify that this is really the number 0 and not any other integer.

- The numeric value 0 is the NIL pointer or NULL pointer that points to nowhere.

  It is used e.g. to mark the end of a linked list — the pointer to the next list element is then 0 (NIL).

- It is illegal to dereference a NIL pointer, i.e. `*p` generates a runtime error (exception) when `p` is NIL.

Exercise

What does this program print?

```
 (1)   #include <stdio.h>
 (2)   int main()
 (3)   {   int n;
 (4)       int *p, *q;    /* pointers to int */
 (5)       n = 5;
 (6)       p = &n;
 (7)       (*p)++;
 (8)       q = p;
 (9)       *q = *p * 2;
(10)       printf("%d %d\n", n, *q);
(11)       return(0);
(12)   }
```

Type Casts (1)

- C permits nearly arbitrary explicit type conversions.

  They are called casts. In medicine, "cast" means a bandage for broken legs etc.

- The prefix operator (T) has the signature $S \rightarrow T$. E.g. "`(float)5`" is `5.0`.

- In particular, one can convert
  - any arithmetic type into any arithmetic type,
  - any pointer type into any pointer type,
  - any pointer type into sufficiently long integers and vice versa.

Type Casts (2)

- The conversion of `pointer`(T) into `pointer`(S) can cause a runtime error if S has stricter alignment requirements than T.

  A conversion to `pointer(char)` and `pointer(void)` is safe. It is also guaranteed that the pointer can be converted back to its original type, and the result is exactly the same pointer.

- Since pointers are internally memory addresses, it is also natural that they can be converted to numbers.

  However, C does not state which integer type is "sufficiently long". On 32-bit machines, `long` or `unsigned long` should be right. One should define a type name that can easily be changed.

Type Casts (3)

- Casts allow the programmer to switch the “C” type checking off. This is of course dangerous.
- But sometimes, this makes useful tricks possible.
- E.g., in Pascal, **`read/write/new`** are hardwired into the language.

  The programmer cannot define similar procedures, since they would not have unique argument types.

- In C, **`scanf/printf/malloc`** are library procedures, not part of the core language.

  Programmers can define their one versions.

const (1)

- Variables can be declared as not modifiable, e.g.

  const float pi = 3.141593;

- The type `const`(T) is automatically converted to T whereever this is required.
- Therefore, `const`(T) behaves like T except that it is possible to determine the memory address of such an object. The operator `&` has also the signature

  `const`$(T) \rightarrow$ `pointer_to_const`(T).

const (2)

- `pointer_to_const`(T) is identical to `pointer`(T), except that `*` has the signature

  `pointer_to_const`$(T) \rightarrow$ `const`(T).

- `lvalue(pointer_to_const`(T)`)` vs. `const(pointer`(T)`)`:
  - In the first case, the pointer itself can be modified, but not the location it points to.

    Supported in C. The declaration is written: `const char *p;`
  - In the second case, the pointer itself cannot be modified, but the location it points to can.

    Supported only in C++ (?). Declaration: `char *const p;`

const (3)

- A value of type `pointer`(T) can be assigned to a variable of type `pointer_to_const`(T), since this can do no harm.

  One only restricts the possible accesses via the pointer to read-only accesses.

- Of course, the opposite is forbidden: A value of type `pointer_to_const`(T), cannot be assigned to a variable of type `pointer`(T).

  Then one would get write access to a read-only memory area.

Arrays (1)

- An array of type T with size n is an object that contains n objects of type T.
- E.g. if `A` is a variable of type $\texttt{array}(\texttt{char}, 20)$, there are actually 20 variables of type `char`.

  In C, such an array is declared as `char A[20];`

- The single variables are referenced as `A[0]`, `A[1]`, . . . , `A[18]`, `A[19]`.
- Mathematically, an object of type $\texttt{array}(T, n)$ is a mapping $\texttt{[0..}n-1\texttt{]} \rightarrow \texttt{T}$.

Arrays (2)

- In C, the index ranges always from 0 to $n-1$, where n is the size of the array (number of elements).

  In contrast to Pascal, it is not possible to specify the lower boundery of the index range.

- For a variable of type **array**(T, n), the compiler allocates n times the space required for a variable of type T.

  So if the array starts at address a, and each element needs b bytes, the first array element has address a, the second has address $a + b$, the third address $a + 2 * b$ and so on. C guarantees that the `sizeof` operator for an array of n elements returns n times the size of a single element.

Arrays (3)

- The index of an array access can be computed.
- E.g. `a[i]` and `a[i+j-1]` are legal, where `i` and `j` must be variables of integral type.

  Integral type means `char`, `short`, `int`, `long` (signed and unsigned) and enumeration types.

- `a[...]` can also be the left side of an assignment.
- Thus, `_[_]` has the signature

  $$\texttt{array}(T, n) \times S \rightarrow \texttt{lvalue}(T),$$

  where S is an integral type.

Arrays (4)

- It is forbidden that the array index violates the boundaries. I.e. `a[i]` is only legal if `i` has a value between 0 and $n-1$, where n is the size of the array.
- By default, most C compilers do not generate runtime checks for the index boundaries.

  E.g. one might get the value of another variable that happens to be stored near to the array.

- This is especially dangerous for assignments to the array, e.g. `a[-1] = 0` will probably overwrite important data in memory.

Arrays (5)

- It is a common error not to check input sizes.
- E.g. the program might ask the user to enter a name and read input characters into a array of size 80.
- A malicious user might enter a very long string.
- If the program does not count the input characters it will overwrite other data in memory.
- Hackers have used this trick many times.

  They might be able to overwrite the return address of a procedure e.g. with an address in the array where they can put any code they want.

Exercise

What does this program print?

```
 (1)   #include <stdio.h>
 (2)   int main()
 (3)   {   int a[3];
 (4)       int n;
 (5)       n = 2;
 (6)       a[n] = 0;
 (7)       a[a[2]] = 5;
 (8)       n--;
 (9)       a[n] = a[0] + a[2] - 3;
(10)       printf("%d %d %d\n", a[0], a[1], a[2]);
(11)       return(0);
(12)   }
```

Multi-Dimensional Arrays

- C has no multi-dimensional arrays (i.e. arrays with more than one index).
- But the array elements can themselves be arrays.
- E.g. the declaration

```
int a[5][7];
```

  means that a has the type

  array(**array**(**int**, 7), 5),

  i.e. a consists of 5 arrays, each of which consists of 7 integers.

Arrays and Pointers (1)

- The types array(T, n) and pointer(T) are equivalent.

  But there is no `lvalue(array(...))` in C. Arrays themselves cannot be the goal of an assignment, pointers can.

- The difference between declaring `p` as a pointer to `char` and `a` as a `char`-array of size 80 is
  - For `p`, space is reserved to hold a memory address (e.g. 4 bytes). `p` is not initialized.
  - For `a`, space is reserved to hold 80 characters (80 bytes). The name `a` then stands for the starting address of that storage space.

Arrays and Pointers (2)

- Difference between arrays and pointers, continued:
  - The memory address to which `p` points can be changed, i.e. assignments to `p` are legal.

    `p` has the type `lvalue`(`pointer`(char)).
  - Assignments to `a` itself are not legal, (i.e. the memory address is constant), but assignments to `a[0]` etc. are legal.

    `a`'s type is only equivalent to `pointer`(char) (without `lvalue`).
- The assignment `p = a` is legal: `p` will then point to the starting address of `a` (i.e. to the first element).

Pointer Arithmetic (1)

- If `p` is a pointer that points to the first element of an array `a`, then `p+1` points to the second element, `p+2` points to the third element, and so on.

  In general, if `p` points to any element in an array, then `p+1` points to the next element, `p+2` to the next but one, and so on. Furthermore, `p-1` points to the previous element, `p-2` to the previous but one, etc.

- This also holds if the elements of `a` are not characters, but need more than one byte.
- If `p` has the type `pointer`(T), then `p+i` actually computes the address $\mathtt{p} + \mathtt{i} * b$ where $b = \mathtt{sizeof}(T)$.

Pointer Arithmetic (2)

- E.g. if `p` is a pointer to `int` with value 1000 and `int` is 4 bytes, then `p+1` has the value 1004, and `p+2` has the value 1008, and so on.
- Thus, in addition to the signature $T \times T \to T$ for arithmetic types T, `+` and `-` also have the signature

$$\texttt{pointer}(T) \times S \to \texttt{pointer}(T),$$

where S is an integral type.

- In C, the expressions `a[i]` and `*(a+i)` are equivalent.

Actually, pointer arithmetic is commutative, and one can write `i[a]`.

Pointer Arithmetic (3)

- The result of pointer arithmetic is undefined if the array boundaries are violated.

  I.e. the machine may or may not check that the resulting address remains in the same array. Also, compilers for earlier Intel processors supported pointer arithmetic only within a “segment” that could be at most 64 KB large. Of course, arrays were also restricted to this size.

- However, the C standard explicitly legalizes the address a $+ n$, where n is the size of the array (i.e. just beyond the array boundaries).

  Of course, one cannot assign anything to that address, but one can use it in comparisons (see below).

Pointer Arithmetic (4)

- C also permits to subtract two pointers in order to get the difference of the array indices.

  Of course, both pointers must point somewhere into the same array (or just beyond it), or else the result is undefined.

- The - operator has also the following signature:

  $$\texttt{pointer}(T) \times \texttt{pointer}(T) \rightarrow \texttt{ptrdiff_t}.$$

  The type `ptrdiff_t` is declared in `stddef.h`. It is today usually `long`.

- E.g. if `p` points to `a[5]` and `q` points to `a[2]`, then `p-q` is 3.

Pointer Arithmetic (5)

- Again, the size of the element type of the array does not matter.

  E.g. if `a` is an array of `int`, the difference of the memory addresses will probably be 12, but the compiler automatically generates code to divide the difference by `sizeof(int)`.

- In summary, it is possible to compute back and forth between array indices and pointers:
  - If `a` is an array, then `p = a+i` points to `a[i]`.
  - If `p` is a pointer pointing somewhere into the array `a`, then `i = p-a` is the index of the element to which `p` points.

Pointer Comparisons (1)

- If `p` and `q` point somewhere into the same array (or just beyond it), one can compare them with `==`, `!=`, `<`, `<=`, `>`, `>=`.

  More generally, any pointers into the same variable can be compared.

- So these operators also have the signature

$$\texttt{pointer}(T) \times \texttt{pointer}(T) \rightarrow \texttt{int}.$$

- The result is the same as if one would compare the corresponding array indices (0 for false, 1 for true).

  The comparison is done directly on the memory addresses, but the mapping from array indices to memory addresses is order-preserving.

Pointer Comparisons (2)

- For `==` and `!=`, the right side of the comparison can also be the number `0` (the NIL pointer).

  Again, the compiler must be able to check that the right side is the number `0` and not an arbitrary integer.

- Furthermore, the type `pointer(void)` is legal on one or both sides of the comparison with `==` and `!=`.

  The reason for the restriction that the compared pointers must point into the same variable is that on some machines, memory addresses are represented as sum of a segment base address and an offset within the segment. The C compiler is then free to compare only the offsets.

Exercise

What does this program print?

```
 (1)   #include <stdio.h>
 (2)   int main()
 (3)   {   int a[3];
 (4)       int *p;
 (5)       p = a + 1;
 (6)       *p++ = 1;    /* This means *(p++) = 1 */
 (7)       if(p == &(a[2]))
 (8)           *p = 2;
 (9)       *a = 0;
(10)       printf("%d %d %d\n", a[0], a[1], a[2]);
(11)       return(0);
(12)   }
```

Character Strings (1)

- A character string, e.g. `"abc"` is represented in C as an array of characters.
- The end of the string is marked by a null byte after the characters of the string.
- The string `"abc"` is represented in memory as

Address	Contents
s	`'a'`
$s+1$	`'b'`
$s+2$	`'c'`
$s+3$	`'\0'`

Character Strings (2)

- In C, a character string constant like `"abc"` has the type `pointer_to_const(char)`.
- E.g. the following is legal:

```
(1)   #include <stdio.h>
(2)   int main()
(3)   {   const char *s;
(4)       s = "abc";
(5)       s++;
(6)       printf("%c%c\n", *s, s[1]);
(7)       return(0);
(8)   }
```

Character Strings (3)

- The following program demonstrates two possible errors (violations of `const`):

```
 (1)   #include <stdio.h>
 (2)   int main()
 (3)   {   const char *s;
 (4)       char *p;
 (5)       s = "abc";
 (6)       *s = 'X'; /* ERROR */
 (7)       p = s;    /* ERROR */
 (8)       *p = 'X';
 (9)       return(0);
(10)   }
```

Character Strings (4)

- Only string constants have these restrictions.

  But one can also build strings in character arrays:

```
 (1)   #include <stdio.h>
 (2)   int main()
 (3)   {  char a[4];   char *p;
 (4)      a[0]='A'; a[1]='B'; a[2]='C'; a[3]=0;
 (5)      *a = 'X';
 (6)      p = a;
 (7)      *++p = 'Y';
 (8)      printf("%s\n", a);
 (9)      return(0);
(10)   }
```

Character Strings (5)

- Note that an array for a character string of n characters must have size $n+1$ (an additional null byte is needed to mark the end of the string).
- Since strings are arrays, e.g. the following is legal: `"abc"[i]`. The value of `i` must be between `0` and `3`.
- It is only a convention that strings are terminated in C with a null byte. C guarantees this for string constants and many library functions use it.

  But it is also possible to work with strings as an array of characters plus an integer that contains the length (needed for some interfaces).

Character Strings (6)

- Arrays cannot be assigned or compared in a single instruction. However, the standard C library has many functions for working with strings, e.g.:
  - `strlen`(s): Number of characters in s.

    This does not include the null byte, e.g. `strlen("abc")` is 3.
  - `strcpy`(s, t): Creates a copy of t in the array s.

    s must have at least the size `strlen`(t) + 1. If the array is too small, other data in memory is destroyed. `strcpy` returns s (seldom useful).
  - `strcat`(s, t): Concatenate a copy of t to the string already stored in s.

    I.e. s is modified. The array containing s must be large enough.

Character Strings (7)

- Standard library functions for strings, continued:
  - `strcmp`(s, t): Compares strings s and t.

    The return value is slightly unusual: `strcmp` returns `0` if the two strings are equal, a positive value if s comes alphabetically behind t, and a negative value if s comes before t. If interpreted as a boolean value, the result is false if the two strings are equal.
  - `strchr`(s, c): Returns a pointer to the first occurrence of the character c in s (NIL if not found).
  - Plus many more functions, see the manual.
- String functions are declared in `string.h`.

Pointers vs. Indices (1)

- In C, it is often slightly more efficient to work with pointers than with indices.
- The reason is that array accesses `a[i]` need an addition and often also a multiplication to compute the address $a + i * b$ where b is the size of the array element type.
- By using a pointer, one can directly maintain the address of the array element.

  Good optimizers of Pascal compilers introduce such pointers internally. Since the programmer can work with pointers explicitly in C, there is much less pressure on C compiler vendors to implement this feature.

Pointers vs. Indices (2)

- E.g. this version of `strlen` uses array indexes:

```
(1)   int strlen(const char *s)
(2)   {
(3)       int i = 0;
(4)       while(s[i] != 0)
(5)           i++;
(6)       return(i);
(7)   }
```

- At least for long strings, it is probably slightly slower than the version with pointers shown below.

  Besides incrementing `i`, an addition is needed to compute the address of `s[i]` in each turn through the loop.

Pointers vs. Indices (3)

- This version only increments a pointer while the array is searched.

```
(1)   int strlen(const char *s)
(2)   {
(3)       const char *p = s;
(4)       while(*p)
(5)           p++;
(6)       return(p - s);
(7)   }
```

- Only at the very end, a single subtraction is needed.

Overview

1. Data Type Foundations
2. Basic C Types
3. Variables (L-Values)
4. Pointers, Arrays and Strings
5. Structures, Unions, Enumeration Types

Structures (1)

- As an array, an object of a structure type can be seen as composition of several simpler values.
- However, in contrast to an array,
  - the components are selected by name,

    and not by a number (index) as in an array,
  - the components can have different types,

    whereas in an array, all components have the same type.
- In Pascal, structures are called records. Structures also correspond to tuples (rows) in relational databases.

Structures (2)

- An object of type

$$\texttt{struct}(N, A_1\colon T_1, \ldots, A_n\colon T_n)$$

  has the component selector functions $\texttt{.}A_i$

$$\texttt{struct}(N, A_1\colon T_1, \ldots, A_n\colon T_n) \longrightarrow T_i$$

  that return the i-th component of the structure.

- N is the name of the structure (see below).
- If applied to a variable of structure type, $\texttt{.}A_i$ gives:

$$\texttt{lvalue}(\texttt{struct}(N, A_1\colon T_1, \ldots, A_n\colon T_n)) \longrightarrow \texttt{lvalue}(T_i).$$

  Exception: Array components cannot give lvalues.

Structures (3)

- E.g. the following is a legal C program:

```
(1)   #include <stdio.h>
(2)   int main()
(3)   {   struct {float x; float y;} point;
(4)       point.x = 1.0;
(5)       point.y = point.x;
(6)       printf("%f %f\n", point.x, point.y);
(7)       return(0);
(8)   }
```

- In C, the official name for components is members.

  Other languages call them “fields” or “attributes”.

Structures (4)

- Structures can get names ("tags") if in order to declare several variables with the same structure type.

  Of course, one can also use a `typedef` declaraction, which is a more general mechanism to give names to types (see a later chapter).

- E.g. consider the following declarations:

```
(1)   struct {float x; float y;} p1, p2;
(2)   struct {float x; float y;} p3;
(3)   struct xy_coord {float x; float y;} p4;
(4)   struct xy_coord p5;
```

- Here the name `xy_coord` is assigned to the kind of structure declared in line (3). It is reused in line (4).

Structures (5)

- C does not look inside structures to check whether two variables have the same type. It only checks whether the structures have the same name.

  Of course, each name can be declared only once (in the same block).

- In the example, `p1` and `p2` have the same type, since they are declared together.

- `p3` has a different type. C only notices that a new structure is declared (no explicit name: new name).

- The type of `p4` is again different, but `p5` has the same type as `p4`.

Structures (6)

- Structures can be assigned to structure variables of the same type. (But `==` and `!=` cannot be used.)

  Thus, `=` has also the signature `lvalue`$(T) \times T \rightarrow T$, where T is a structure type. This was not included in the original version of C, since here it might happen that a simple assignment has to be implemented as many machine instructions. In contrast, entire arrays still cannot be assigned (unless they are part of a structure). Maybe, the reason was that small structures can be assigned in a single instruction (so the compiler can generate more efficient code than the programmer, who could only assign the single components). Also, assignments of entire arrays have the problem that they could be easily intermixed with assigning only pointers. Structures are also useful as function values.

- One often works with pointers to structures. Therefore, C declares `p->A` as a shorthand for `(*p).A`.

Implementation (1)

- The compiler implements a structure by reserving space for each of its components in memory.
- Because of the alignment restrictions, there might be gaps between the components. E.g. consider

```
struct chess_square { char col; int row; } s;
```

- The variable needs 8 bytes (for 32-bit machines). Let the address of the structure variable be 1000:

Implementation (2)

- The difference of the address of a component and the address of the structure is called the offset of the component.
- E.g. if `p` is a pointer to a structure "`chess_square`", the compiler will implement "`p->row = 2`" by taking the address stored in `p`, adding the offset 4 and storing 2 in the resulting address.
- C guarantees that the first component has offset 0 and that the offsets of components are monotonically increasing in the sequence of their declaration.

Incomplete Types

- C permits references to structures that are not yet defined if the size of the structure is not needed.
- The typical case is to define a pointer to a structure. All pointers have the same size.

```
(1)   struct list {
(2)       int data;
(3)       struct list *next; /* pointer */
(4)   };
```

- In line (3), the type `struct list` is not yet defined.

  One can even reference structures that are not yet mentioned at all.

Unions (1)

- A `union` type looks in C very similar to a `struct` type, however only one of the components can be used at the same time:

```
union int_or_char { int i; char c; } x;
```

- I.e. an assignment to a component A_i overwrites the values of all components A_j.

  The compiler reserves only $\max(b_i)$ bytes, where b_i is the number of bytes needed by component A_i. All components have offset 0.

- Instead of components, one better says "variants".

Unions (2)

- The programer is responsible for reading only the value of that component/variant that was last assigned a value.

  It is not an error to read a different component/variant, but the result is implementation-defined: It is one way to interpret the bit-pattern of a value of one type as a value of another type.

- The concept is similar to variant records in Pascal.

  By nesting in union in a structure, the programmer can himself/herself manage a flag that shows the currently used variant of the union.

Enumeration Types (1)

- Enumeration types are defined by explicitly enumerating all possible values:

```
enum colors { white, black, red, green, blue,
              yellow } x;
enum months { jan, feb, mar, apr, may, jun,
              jul, aug, sep, oct, nov, dec };
```

- An enumeration type is very similar to `int` and the elements of the enumeration type are basically constants of type `int`.

  The compiler may choose to implement it with `char` or `short`.

Enumeration Types (2)

- C guarantees that the first constant gets the value `0`, the second the value `1`, and so on.
- It is also possible to assign explicit `int` values:

```
enum months { jan = 1, feb = 2, mar, apr, ... }
```

  C always assigns the next enumeration constant the value of the previous one plus 1, so in the example `mar` has value 3.

- Most compilers do not generate runtime checks that ensure that an enumeration type variable really only contains a value of that type.

Enumeration Types (3)

- C and C++ do an automatic conversion from an enumeration type to `int`.

  Enumeration constants can be used in places where normally `int` constants are required, e.g. for array indexes and array sizes.

- C also automatically converts from `int` to any enumeration type, but this is not legal in C++.

  One must use an explicit cast. Also operators like `++` must be explicitly defined for enumeration types, since they are new types.

- In C and C++, there is no automatic conversion from one enumeration type to another enumeration type.

Enumeration Types (4)

- Exercise: Which statements are legal in C? In C++?

```
(1)   enum colors {red, green, blue} c;
(2)   enum months {jan, feb, mar, ...} m;
(3)   int a[blue+1];
(4)   c = red;
(5)   c++;
(6)   m = 2;
(7)   m = c;
(8)   a[c] = 3;
(9)   m = c + 1;
```

In C, "`c+1`" still has the type "`enum colors`".