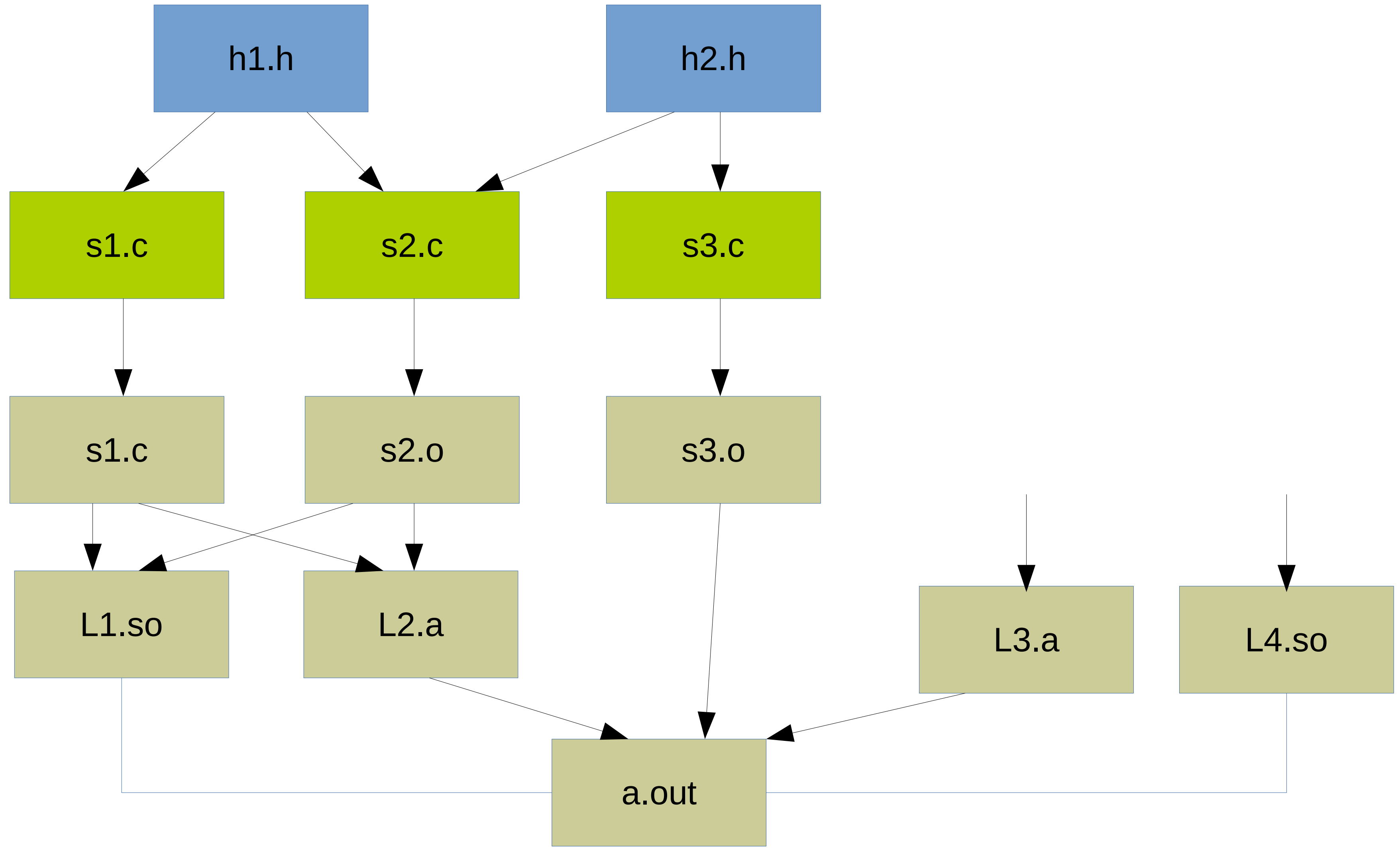
Imperative programming

2. Structure of C programs

Zoltán Porkoláb

Building the C programs



Building the C programs

- C programs consist of separate translation units
- Translation units should have .c suffix
- Translation units compiled separately
- Compilation takes several steps
  - Preprocessing
  - Tokenization
  - Parsing
  - Context sensitive analysis (type checking, ...)
  - Optimizations
  - Code generation
- Linking
  - Static linking
  - Dynamic linking

What is in a translation unit

- Comments
- Preprocessor directives
- C Tokens

Comments

```c
#include <stdio.h>

int main()
{
   printf( "hello world\n" );
   return 0;   /* not strictly necessary since C99 */
}
```

Comments

```c
#include <stdio.h>

int main()
{
   printf( "hello world\n" );
   return 0;   // not strictly necessary since C99
}
```

Comments

```c
#include <stdio.h>

/*
 * My first C program
 * with multi-line comment
 */

int main()
{
   printf( "hello world\n" );
   return 0;   // not strictly necessary since C99
}
```

Preprocessor directives

```c
#include <stdio.h>

int main()
{
   printf( "hello world\n" );
   return 0;
}
```

```
$ gcc -E hello.c
.....
$
```

Preprocessor directives

```c
#include <stdio.h>

#define MESSAGE "hello world\n"


int main()
{
   printf( MESSAGE );
   return 0;
}
```

```
$ gcc hello.c
$ ./a.out
hello world
$
```

Preprocessor directives

```
#include <stdio.h>

#define LONG_MESSAGE "Let me greet you dear friend \
with hello world\n"

int main()
{
   printf( MESSAGE );
   return 0;
}
```

```
$ gcc hello.c
$ ./a.out
Let me greet you dear friend with hello world
$
```

Preprocessor directives

```c
#include <stdio.h>
#ifdef SZIA
  #define MESSAGE "szia vilag"
#else
  #define MESSAGE "hello world"
#endif  /* SZIA */

int main()
{
   printf( MESSAGE "\n"); // "hello world" "\n"
   return 0;
}
```

```
$ gcc -DSZIA hello.c
$ ./a.out
szia vilag
$ gcc hello.c
$ ./a.out
hello world
```

Preprocessor directives

```
#ifndef MYHEADER_H
#define MYHEADER_H

/* header content */

#endif  /* MYHEADER_H */


#if DEBUG_LEVEL > 2
  fprintf(stderr, "file %s, line %d\n", __FILE__,__LINE__);
#endif


#ifdef __unix__
  #include <unistd.h>
#elif defined _Win32
  #include <windows.h>
#else
  #error Only UNIX and WINDOWS is supported
#end
```

C Tokens

```c
#include <stdio.h>

int main()
{
    printf( "hello world\n" );
    return 0;
}
```

C Tokens

```
#include <stdio.h>

int
      main
     (     )
                 {
   printf
(
        "hello world\n"
)
   ;
        return
     0
   ;
                 }
```

C Tokens

```c
#include <stdio.h>

int main()
{
   printf( "hello world\n" );
   return 0;
}
```

- Keywords
- Identifiers
- Literals / constants
- Operators
- Separators

Keywords

```
#include <stdio.h>

int main()
{
   printf( "hello world\n" );
   return 0;
}
```

C89: auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch union unsigned void volatile while

C99: inline restrict _Bool _Complex _Imaginary

C11: _Alignas _Alignof _Atomic _Generic _Noreturn _Static_assert _Thread_local

C23: alignas alignof bool constexpr false nullptr static_assert thread_local true typedef typeof typeof_unqual _BitInt _Decimal128 _Decimal32 _Decimal64

Identifiers

```c
#include <stdio.h>

int main()
{
   printf( "hello world\n" );
   return 0;
}
```

- Names of variables, functions and types
- Starts with letter and continuous with letter (including _) or digit
- Lower case and upper case characters are different
- Keywords must not be used, identifiers starting underscore are reserved
- Standard library names are reserved
- Compilers have limit for length (31 for external, 63 for internal names)

Identifier conventions

```
#define MACRO_NAMES_ARE_ALL_UPPERCASE 100

int CamelCaseNotation = 42;

int underscore_notation = 43;
```

- Use all uppercase names for MACRO identifiers
- Names are valuable resources, minimize their scope
- Choose long names for identifiers with larger scope
- Follow some naming convention (e.g. Hungarian notation from Charles Simonyi)

Literals

```
#include <stdio.h>

int main()
{
    printf("hello world\n");
    return 0;
}
```

- A literals has a **type** and a **value**
- Literal categories
  - Integral
  - Floating point
  - String

Integral types

type	min size (bit)	example
_Bool (C99) bool (C23)	1	`1 0 (true/false)`
char	8	`'x'   '\n'   '\377'`
signed char	8	`'\377'   ‘\xff’`
unsigned char	8	
short	16	
unsigned short	16	
int	16	`12  014  0xC`
unsigned int	16	`1u`
long	32	`1l`
unsigned long	32	`1uL  1UL`
long long (C99)	64	`99ll 100LL`
unsigned long long (C99)	64	`420uLL`

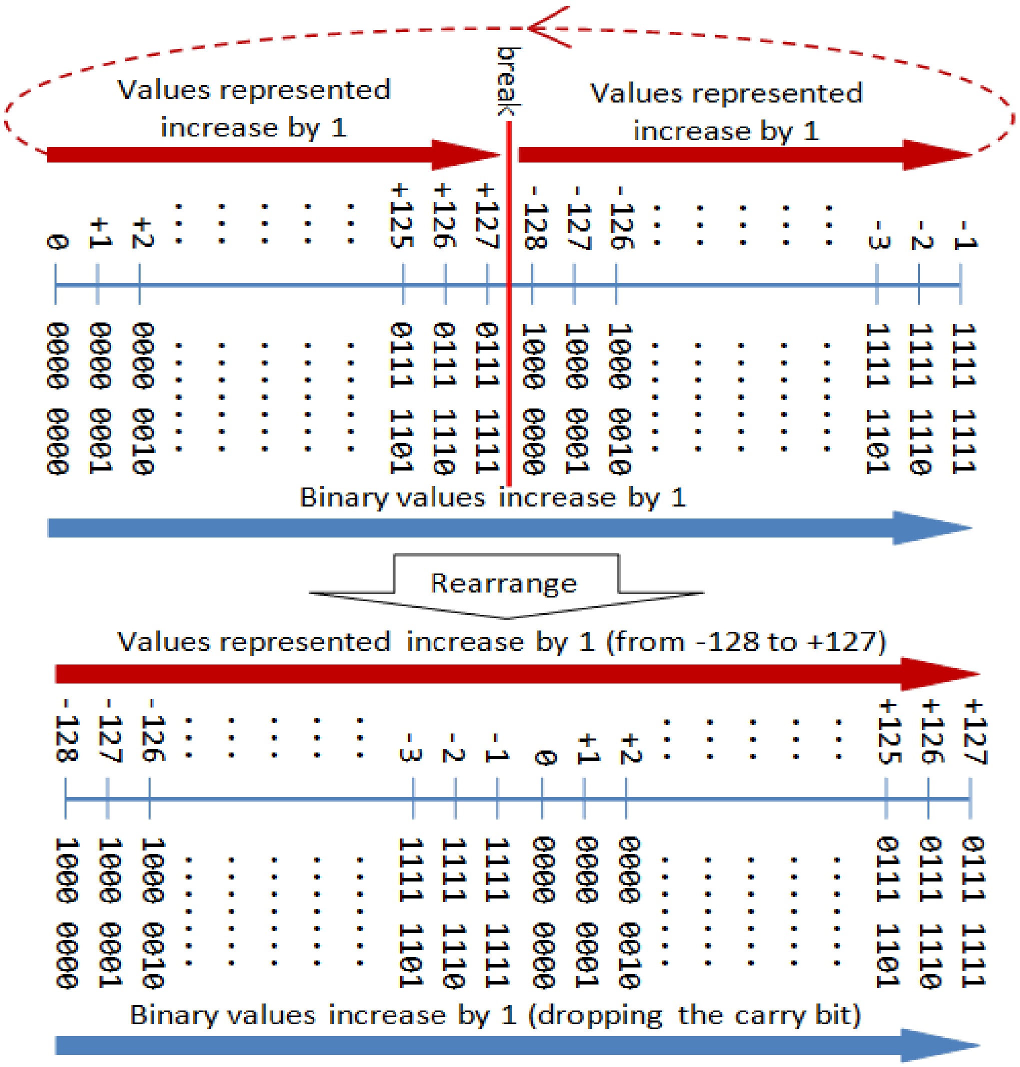
Integral types

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int)
  <= sizeof(long) <= sizeof(long long)

sizeof(int) == sizeof(signed int) == sizeof(unsigned int)
```

- The exact size of types are implementation defined
- The **minimum** size is defined by the standard
- The **sizeof** operator returns the size of the type in “bytes”
- The **signed** T and **unsigned** T types have the same size
- T and signed T are the same **except** char, signed char and unsigned char
- There is a keyword **void** to represent incomplete/empty type

Integral representation



2's Complement Representation

Usually 2's complement

Signed

- Most significant bit is sign bit
- Overflow is undefined behavior

Unsigned

- All bits are representing value
- Overflow works as 2^N modulo

```
int i = 1;
unsigned ui = 1;

i  -= 2; // i == -1
ui -= 2; // ui > 0  big
```

Character literals

```
char ch = 'a';

char quote           = '\'';
char double_quote    = '\"';
char qmark           = '\?';
char backslash       = '\\';
char bell            = '\a';
char backspace       = '\b';
char formfeed        = '\f';
char newline         = '\n';
char carriage_return = '\r';
char horizontal_tab  = '\t';
char vertical_tab    = '\v';

char oct             = '\377';
char hex             = '\xff';  //  hex ? 0
signed char shex     = '\xff';  // shex < 0
unsigned char uhex   = '\xff';  // uhex > 0
```

Boolean (C99)

```
_Bool  b = 0; // false
       b = 1; // true
```

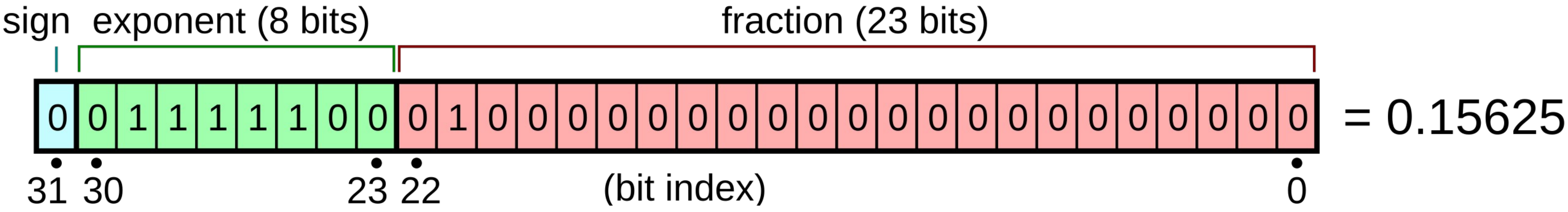
- All non-zero values are **true**
- **_Bool** is only since C99
- **bool**, **true**, **false** are provided as macro (C99-C23)
- **bool**, **true**, **false** are keywords (since C23)

```
#include <stdbool.h>   // before C23

bool b = false;
     b = true;
```

Floating point types

- Implemented with IEEE 754 standard
- Format: {+1|-1}*frac*2^{exp} where frac = 1.F
- Supported by hardware manufacturers



sign exponent (8 bits) fraction (23 bits)

0 0 1 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 = 0.15625

31 30 23 22 (bit index) 0

type	min size (bits)	IEEE	s+exp+frac
float	32	Single precision	1+23+8
double	64	Double precision	1+52+11
long double	128	Quadruple precision	1+64+15

Floating point literals

```
float  f        = 3.14f;
double d        = 3.14;
long double ld = 3.14l;

if ( d != NAN  &&  d != INFINITY )
  printf("d == %f", d);
```

- Floating point operations may overflow or underflow or rounding
- There are positive and negative infinity
- There are positive and negative zero
- Not a Number: **NAN**
- **Never use floating point numbers when you need precise values!**

String literals

```
printf("Hello world\n");
char *p  = "Hello world\n";
char *q  = "Hello world\n";
char a[] = "Hello world\n";

assert(sizeof("Hello world\n") == 13 && sizeof(a) == 13)

p[1] = 'a';  // undefined behavior
a[1] = 'a';  // OK, character a
```

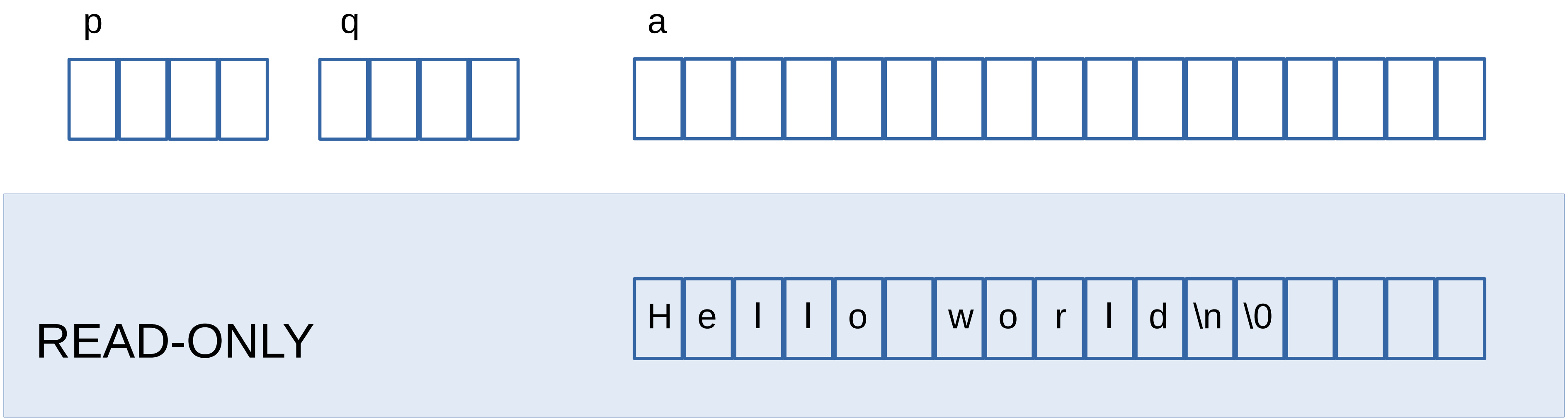
- String literals are strictly read-only! Attempting to modify is undefined behavior
- Type of a string literal is a character array (including terminator ‘\0’ char)
- Identical string literals may refer to the same memory address (p==q)
- String can be used to initialize character arrays (a[ ] = “Hello world\n”)
- Character arrays converted to char*

String literals

```
printf("Hello world\n");
char *p  = "Hello world\n";
char *q  = "Hello world\n";
char a[] = "Hello world\n";

assert(sizeof("Hello world\n") == 13 && sizeof(a) == 13)

p[1] = 'a';  // undefined behavior
a[1] = 'a';  // OK, character a
```



p

q

a

READ-ONLY

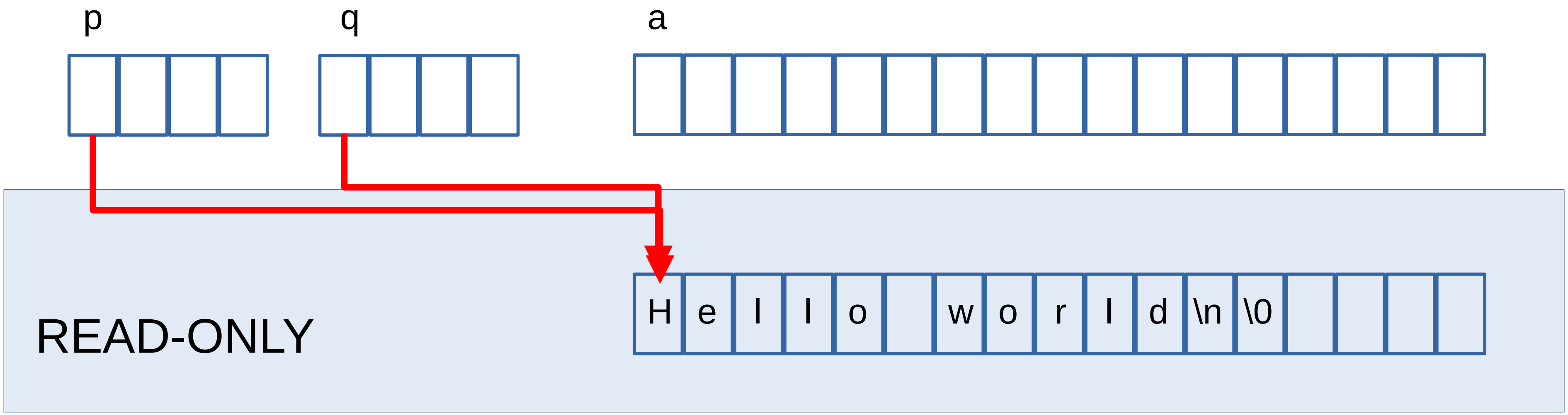
H	e	l	l	o		w	o	r	l	d	\n	\0				

String literals

```
printf("Hello world\n");
char *p  = "Hello world\n";
char *q  = "Hello world\n";
char a[] = "Hello world\n";

assert(sizeof("Hello world\n") == 13 && sizeof(a) == 13)

p[1] = 'a';  // undefined behavior
a[1] = 'a';  // OK, character a
```

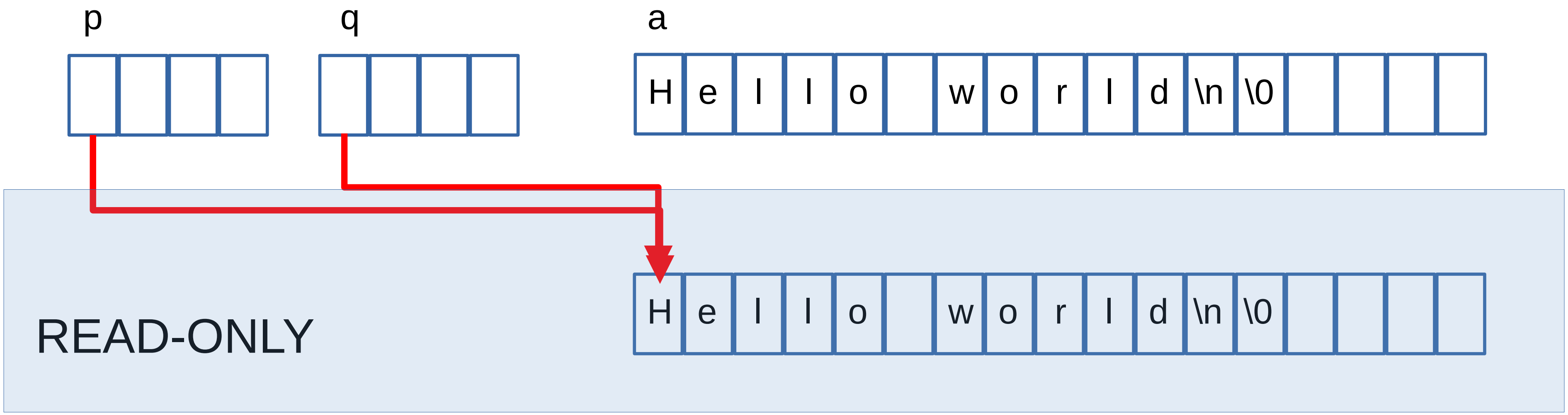


String literals

```
printf("Hello world\n");
char *p  = "Hello world\n";
char *q  = "Hello world\n";
char a[] = "Hello world\n";

assert(sizeof("Hello world\n") == 13 && sizeof(a) == 13)

p[1] = 'a';  // undefined behavior
a[1] = 'a';  // OK, character a
```



p

q

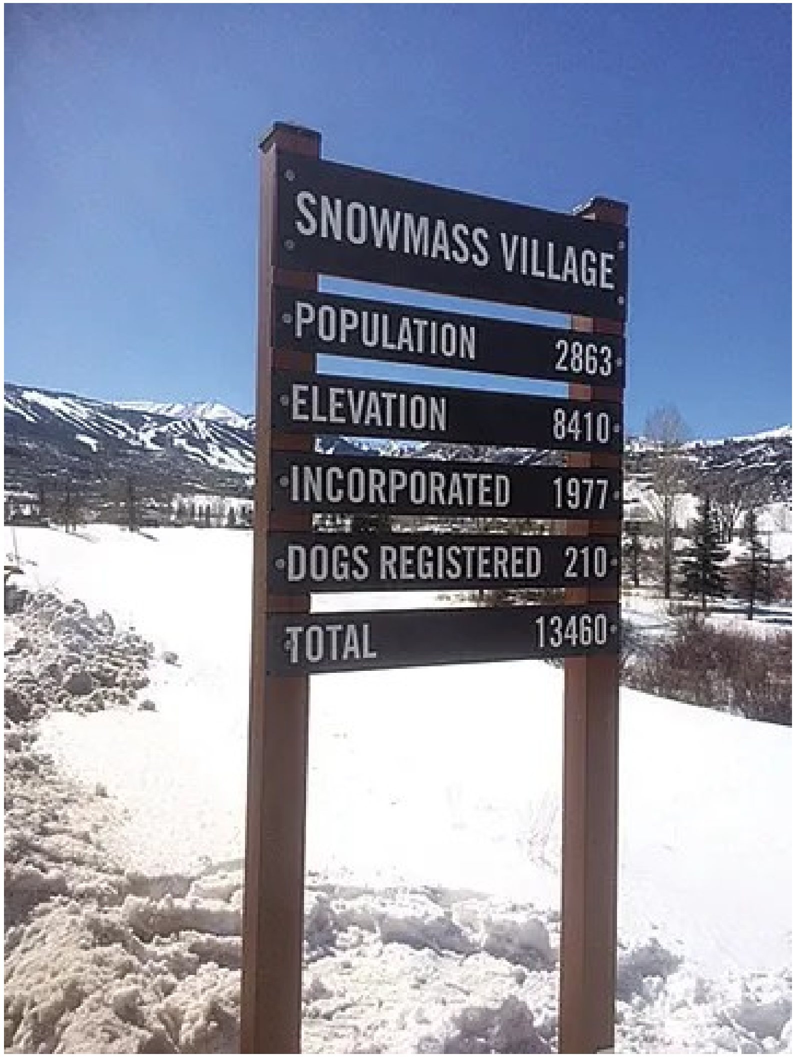
a

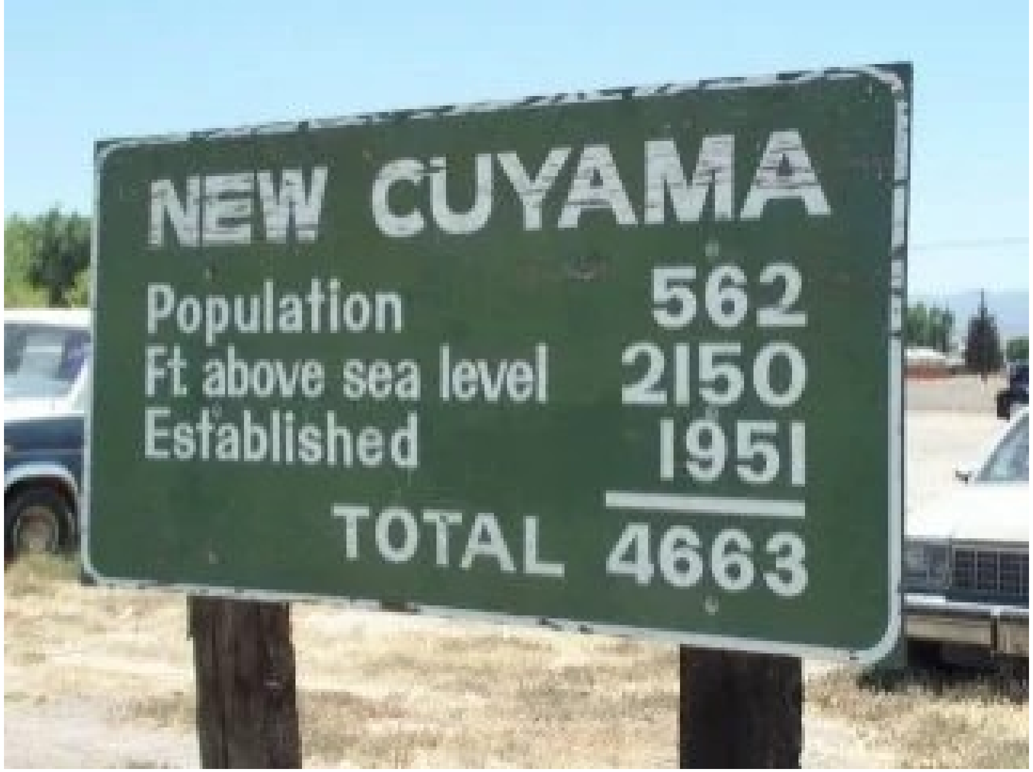
H e l l o   w o r l d \n \0

READ-ONLY

H e l l o   w o r l d \n \0

Why types are important?





Why types are important?

- How the bits in memory should be interpret
- What value the represent
- Which operations can be performed
- Protect against (some) programming errors
- Express the programmers intents
- Help to form abstractions
- Help to generate more effective code

Strong and weak typing

- Strong typing
  - Restricts which operations can be performed on a value of a type
- Weak typing
  - Less control, more automatic conversions

```perl
#!/usr/bin/perl
$x = 12 + "12";
print $x;

$ perl add.pl
24
```

```bash
#!/usr/bin/bash
if test $1 -gt $2
then
   echo $1 '>' $2;
fi

$ bash add.bash 12 "11"
12 > 11
```

Strong and weak typing

- Strong typing
  - Restricts which operations can be performed on a value of a type
- Weak typing
  - Less control, more automatic conversions

```
public class add
{
  public static void main(String[] args)
  {
    System.out.println(123+"123");
  }
}
```

Strong and weak typing

- Strong typing
  - Restricts which operations can be performed on a value of a type
- Weak typing
  - Less control, more automatic conversions

```java
public class add
{
  public static void main(String[] args)
  {
    System.out.println(123+"123");
  }
}
```

```
$ java add
123123
```

Strong and weak typing

- Strong typing
  - Restricts which operations can be performed on a value of a type
- Weak typing
  - Less control, more automatic conversions

```
public class add
{
  public static void main(String[] args)
  {
    System.out.println(new Integer(123).toString()+"123");
  }
}
```

```
$ java add
123123
```

Static and dynamic type system

- Static type system
  - Compiler checks type correctness in compile time
  - Every literal, variable, expression has a type known at compile time
  - The type does not change in run-time
- Dynamic type system
  - Run-time system checks type correctness in run-time
  - Values have type, not the variables
  - Values of different types can be assigned to the same variable

```
z = input("z= ")
x = "123"
if z > 2:
    x = 42
print("x=",x)
```

```
$
```

Static and dynamic type system

- Static type system
  - Compiler checks type correctness in compile time
  - Every literal, variable, expression has a type known at compile time
  - The type does not change in run-time
- Dynamic type system
  - Run-time system checks type correctness in run-time
  - Values have type, not the variables
  - Values of different types can be assigned to the same variable

```
z = input("z= ")    # always str
x = "123"
if int(z) > 2:      # converts string to int
    x = 42
print("x=",x)

$ python3 x.py
42
```

Fahrenheit to Celsius

```
#include <stdio.h>
int main()
{
  int fahr;
  for ( fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %d,\tCels = %d\n", fahr, 5/9*(fahr-32));
  }
  return 0;
}

$ gcc -ansi -pedantic -Wextra -std=c11 fahrenheit.c
```

Fahrenheit to Celsius

```
#include <stdio.h>
int main()
{
  int fahr;
  for ( fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %d,\tCels = %d\n", fahr, 5/9*(fahr-32));
  }
  return 0;
}

$ gcc -ansi -pedantic -Wextra -std=c11 fahrenheit.c
$ ./a.out
Fahr = -100,   Cels = 0
Fahr = 0,      Cels = 0
Fahr = 100,    Cels = 0
Fahr = 200,    Cels = 0
Fahr = 300,    Cels = 0
```

Fahrenheit to Celsius

```
#include <stdio.h>
int main()
{
  int fahr;
  for ( fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %d,\tCels = %d\n", fahr, 5./9.*(fahr-32));
  }
  return 0;
}
```

```
$ gcc -ansi -pedantic -Wextra -std=c11 fahrenheit.c
```

Fahrenheit to Celsius

```
#include <stdio.h>
int main()
{
  int fahr;
  for ( fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %d,\tCels = %d\n", fahr, 5./9.*(fahr-32));
  }
  return 0;
}

$ gcc -ansi -pedantic -Wextra -std=c11 fahrenheit.c
fahrenheit.c: In function ‘main’:
fahrenheit.c:17:5: warning: format ‘%d’ expects argument of
type ‘int’, but argument 3 has type ‘double’ [-Wformat=]
   printf( "Fahr = %d,\tCels = %d\n", fahr, 5./9.*(fahr-32));
```

Fahrenheit to Celsius

```c
#include <stdio.h>
int main()
{
  int fahr;
  for ( fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %d,\tCels = %d\n", fahr, 5./9.*(fahr-32));
  }
  return 0;
}
```

```
$ gcc -ansi -pedantic -Wextra -std=c11 fahrenheit.c
$ ./a.out
Fahr = -100,   Cels = 913552376
Fahr = 0,      Cels = -722576928
Fahr = 100,    Cels = -722576928
Fahr = 200,    Cels = -722576928
Fahr = 300,    Cels = -722576928
```

Fahrenheit to Celsius

```
#include <stdio.h>
int main()
{
  int fahr;
  for ( fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %d,\tCels = %f\n", fahr, 5./9.*(fahr-32));
  }
  return 0;
}
```

```
$ gcc -ansi -pedantic -Wextra -std=c11 fahrenheit.c
```

Fahrenheit to Celsius

```
#include <stdio.h>
int main()
{
  int fahr;
  for ( fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %d,\tCels = %f\n", fahr, 5./9.*(fahr-32));
  }
  return 0;
}

$ gcc -ansi -pedantic -Wextra -std=c11 fahrenheit.c
$ ./a.out
Fahr = -100,   Cels = -73.333333
Fahr = 0,      Cels = -17.777778
Fahr = 100,    Cels = 37.777778
Fahr = 200,    Cels = 93.333333
Fahr = 300,    Cels = 148.888889
```

Fahrenheit to Celsius

```
#include <stdio.h>
double fahr2cels(double f)
{
  return 5./9.*(f-32);
}
int main()
{
  for ( int fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %4d,\tCels = %7.2f\n", fahr, fahr2cels(fahr));
  }
  return 0;
}

$ gcc -ansi -pedantic -Wextra -std=c11 fahrenheit.c
```

Fahrenheit to Celsius

```
#include <stdio.h>
double fahr2cels(double f)
{
  return 5./9.*(f-32);
}
int main()
{
  for ( int fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %4d,\tCels = %7.2f\n", fahr, fahr2cels(fahr));
  }
  return 0;
}

$ gcc -ansi -pedantic -Wextra -std=c11 fahrenheit.c
$ ./a.out
Fahr = -100,   Cels =  -73.33
Fahr =    0,   Cels =  -17.78
Fahr =  100,   Cels =   37.78
Fahr =  200,   Cels =   93.33
Fahr =  300,   Cels =  148.89
```

Fahrenheit to Celsius

```
#include <stdio.h>

#define LOWER -100
#define UPPER  400
#define STEP   100

double fahr2cels(double f);

int main()
{
  for ( int fahr = LOWER; fahr <= UPPER; fahr += STEP )
  {
    printf("Fahr = %4d,\tCels = %7.2f\n",fahr,fahr2cels(fahr));
  }
  return 0;
}
double fahr2cels(double f)
{
 return 5./9.*(f-32);
}
```

Fahrenheit to Celsius

```c
#include <stdio.h>

double fahr2cels(double f);

int main()
{
  const int lower = -100;
  const int upper = 400;
  const int step  = 100;

  for ( int fahr = lower; fahr <= upper; fahr += step )
  {
    printf("Fahr = %4d,\tCels = %7.2f\n",fahr,fahr2cels(fahr));
  }
  return 0;
}
double fahr2cels(double f)
{
  return 5./9.*(f-32);
}
```