Imperative programming

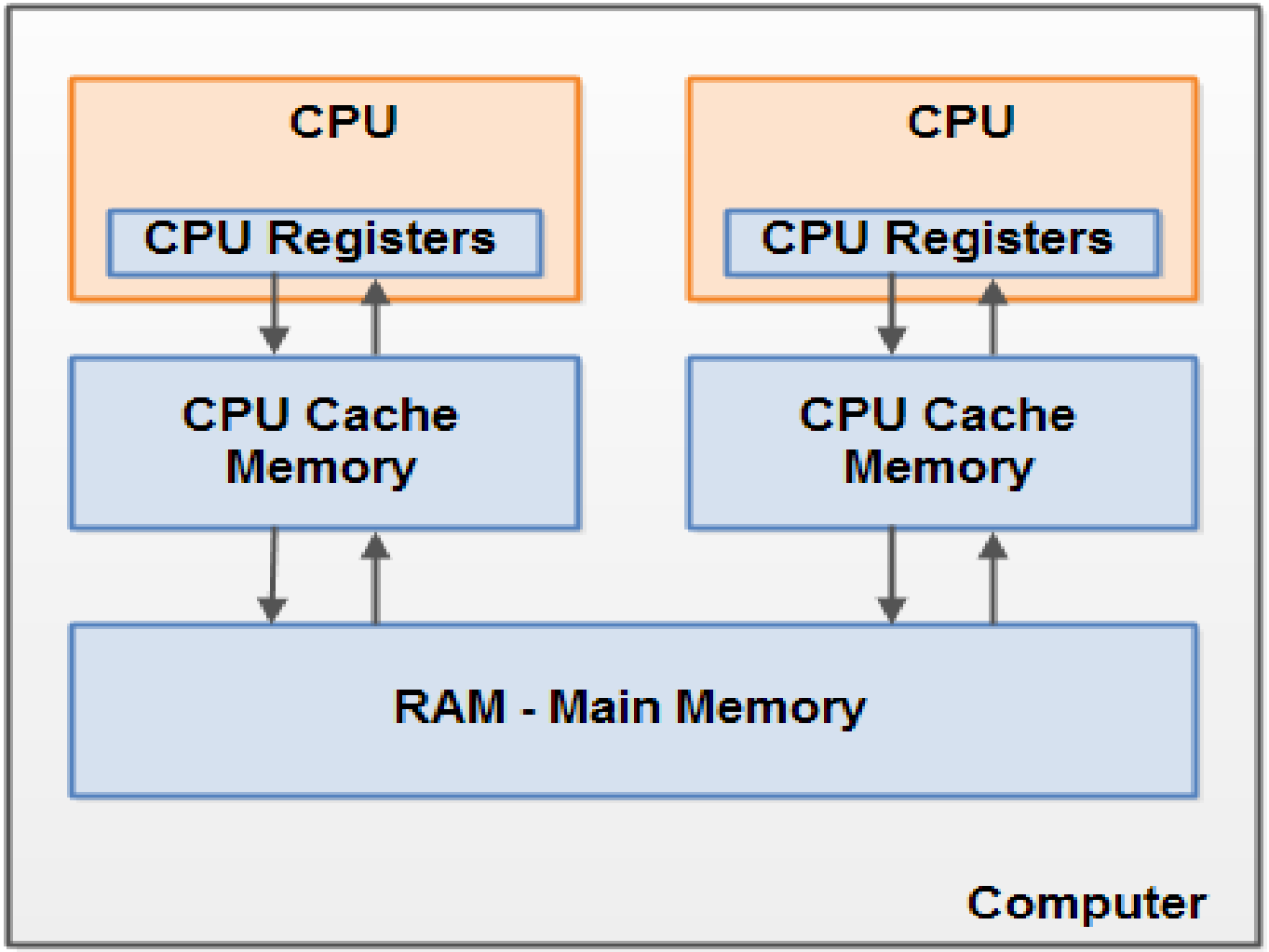
5. Memory, arrays, pointers

Zoltán Porkoláb

Memory

- In imperative programs statements modify the state of the program
- The state of the program is stored in the memory of the computer
- Reading/writing the main memory is (relatively) slow
- Waiting for the results of the memory I/O the processor is not working
- Multiple level of memory cache is used to speed up the execution
- Various algorithms utilize caching, e.g. cache prefetching
- There is a separate cache for instructions
- More from Ulrich Drepper: What Every Programmer Should Know About Memory

Modern hardware architecture



Modern hardware architecture

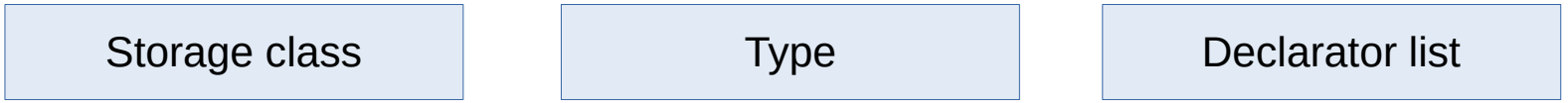
```
Latency Comparison Numbers (~2012)
----------------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5 ns
L2 cache reference                           7 ns
Mutex lock/unlock                           25 ns
Main memory reference                      100 ns
Compress 1K bytes with Zippy             3,000 ns        3 us
Send 1K bytes over 1 Gbps network       10,000 ns       10 us
Read 4K randomly from SSD*             150,000 ns      150 us
Read 1 MB sequentially from memory     250,000 ns      250 us
Round trip within same datacenter      500,000 ns      500 us
Read 1 MB sequentially from SSD*     1,000,000 ns    1,000 us    1 ms
Disk seek                           10,000,000 ns   10,000 us   10 ms
Read 1 MB sequentially from disk    20,000,000 ns   20,000 us   20 ms
Send packet CA->Netherlands->CA    150,000,000 ns  150,000 us  150 ms
```

https://blog.morizyun.com/computer-science/basic-latency-comparison-numbers.html

Declaration, definition

- Static type system: compiler should know the type of identifiers (variables, functions, types)
- **Declaration:** inform the compiler about the type of the identifier
- **Definition:** declaration +
  - Variables: allocate the memory
  - Functions: the body (statements) of the function
  - Types: the structure/components of the type
- Declarations and definitions can be **external** or **internal**
- **One Definition Rule (ODR):** every used identifier should have exactly one definition in the entire program
  - but may have multiple non-contradicting declarations

Declaration, definition



Declaration, definition

Storage class	Type	Declarator list
`extern`	`double`	`pi;`
`static`	`int`	`i, j, k;`
`static`	`short`	`t[20], s[5];`
~~`register`~~	`int`	`r;`
~~`auto`~~	`int`	`n;`
	`double`	`fahr2cels(double x);`
`static`	`int`	`f(void);`
	`int`	`*ptr, *qtr;`
	~~`int`~~`*`	`ptr, qtr; // int qtr;`

Definition

- Introducing an identifier with all necessary properties
- Functions should be externally defined

```
int glob = 1;  // global definition of int        = 1
int zero;      // global (tentative) definition  = 0
const double pi = 3.14;  // global double const  = 3.14

void f(void) // definition of function f without parameters
{
  int i;            // local int, uninitialized
  const int j = 2; // local int const  = 2

  int *ptr;       // pointer to int, unitialized
  int *qpr = &i; // pointer to int, points to i

  double d1[5]; // array of 5 double, unitialized
  double d2[2] = { 1.1, 2.2 };  // array of 2 double
  double d3[]  = { 1.1, 2.2 };  // same as d2
}
```

Definition

- Declarators can be recursive
- No array of functions or functions returning array

```
void f(void) // definition of function f returning int
{
  int i=42, j=24;   // int
  int *p = &i;      // pointer to int, points to i
  int ar1[2] = { 1, 2 };  // array of 2 ints

  int **pp = &p; // pointer to pointer to int, points to p
  int ar2[2][3] = {{1,2,3},{4,5,6}}; // array of 2x3 ints

  int *ptr_arr[2];
  int (*ptr_to)[2];
}
```

Definition

- Declarators can be recursive
- No array of functions or functions returning array

```
void f(void) // definition of function f returning int
{
  int i=42, j=24;   // int
  int *p = &i;      // pointer to int, points to i
  int ar1[2] = { 1, 2 };  // array of 2 ints

  int **pp = &p; // pointer to pointer to int, points to p
  int ar2[2][3] = {{1,2,3},{4,5,6}}; // array of 2x3 ints

  int *ptr_arr[2] = { &i, &j };// array of two pinters to int
  int (*ptr_to)[2];
}
```

Definition

- Declarators can be recursive
- No array of functions or functions returning array

```
void f(void) // definition of function f no param, no return
{
  int i=42, j=24;   // int
  int *p = &i;      // pointer to int, points to i
  int ar1[2] = { 1, 2 };  // array of 2 ints

  int **pp = &p; // pointer to pointer to int, points to p
  int ar2[2][3] = {{1,2,3},{4,5,6}}; // array of 2x3 ints

  int *ptr_arr[2] = { &i, &j };// array of two pinters to int
  int (*ptr_to)[2] = &ar1;     // pointer to array of 2 ints
}
```

Definition

- Declarators can be recursive
- No array of functions or functions returning array

```
void f(void) // definition of function f no param, no return
{
  int i=42, j=24;   // int
  int *p = &i;      // pointer to int, points to i
  int ar1[2] = { 1, 2 };  // array of 2 ints

  int **pp = &p; // pointer to pointer to int, points to p
  int ar2[2][3] = {{1,2,3},{4,5,6}}; // array of 2x3 ints

  int *ptr_arr[2] = { &i, &j };// array of two pinters to int
  int (*ptr_to)[2] = &ar1;     // pointer to array of 2 ints
}
char *getenv(const char *p) // definition of function
{                           //   with parameter const char *
  /* ... */                //   returning pointer to char
}
```

Definition

- Pointer to function
- Function name is the “pointer value”

```
double fahr2cels( double fahr)
{
  return 5./9.*(fahr-32);
}

double f(void)
{
  double (*funptr)(double) = fahr2cels; // pointer to function
  double x = 0.5;

  return (*funptr)(x);  // calls fahr2cels(0.5)

}
```

Definition

- Pointer to function
- Function name is the “pointer value”

```
double fahr2cels( double fahr)
{
  return 5./9.*(fahr-32);
}

double f(void)
{
  double (*funptr)(double) = fahr2cels; // pointer to function
  double x = 0.5;

  return (*funptr)(x);  // calls fahr2cels(0.5)
  return   funptr(x);   // calls fahr2cels(0.5)
}
```

Initialization

- Variables can be initialized when defined
- Static lifetime variables are zero initialized by default
- Constants must be initialized

```
int   i = 42;
int *ip = &i; // ip "points to" i

int arr1[10] = { 0,1,2,3,4,5,6,7,8,9 };
int arr2[10] = { 0,1,2,3,4,5,6,7,8 };  // arr2[9] is 0
int arr3[]   = { 0,1,2,3,4,5,6,7,8 };  // int arr3[9]

int arr4[10] = { 0,1,2,3,4,5,6,7,8,9,10 };  // Compile error

char welcome1[] = {'H','e','l','l','o','\0'}; // array
char welcome2[] = "Hello";                    // same as above
char  *welcome3 = "Hello";                    // pointer to read-only

extern double sin(double);        // declaration
double (*funcptr)(double) = sin; // funcptr "points to" sin()
```

Declaration

- Introducing an identifier with information enough to use
- If initialized, then **definition**

```
extern int glob;  // glob is int, defined somewhere else
extern int zero;  // zero is int, defined somewhere else

void f(void) // definition of function f without parameters
{
  extern int i;    // i is int, defined somewhere else
  extern int *ptr; // pointer to int, defined somewhere else

  extern int d1[5][6]; // array of 5x6 defined somewhere else
  extern int d2[][6];  // array of ?x6 defined somewhere else
  extern int d3[][];   // compiler error

  extern double fahr2cels(double fahr); // function decl
  extern double fahr2cels(double);      // function decl
}
```

Declaration

- Introducing an identifier with information enough to use
- If initialized, then **definition**

```
extern int glob;  // glob is int, defined somewhere else
extern int zero;  // zero is int, defined somewhere else

void f(void) // definition of function f without parameters
{
  extern int i;    // i is int, defined somewhere else
  extern int *ptr; // pointer to int, defined somewhere else

  extern int d1[5][6]; // array of 5x6 defined somewhere else
SAME
  extern int d2[][6];  // array of ?x6 defined somewhere else
  extern int d3[][];   // compiler error

  extern double fahr2cels(double fahr); // function decl
SAME
  extern double fahr2cels(double);      // function decl
}
```

Declaration

- Introducing an identifier with information enough to use
- If initialized, then **definition**

```
extern int glob;  // glob is int, defined somewhere else
extern int zero;  // zero is int, defined somewhere else

void f(void) // definition of function f without parameters
{
  extern int i;    // i is int, defined somewhere else
  extern int *ptr; // pointer to int, defined somewhere else

  extern int d1[5][6]; // array of 5x6 defined somewhere else
SAME
  extern int d2[][6];  // array of ?x6 defined somewhere else
  extern int d3[][];   // compiler error

  extern double fahr2cels(double fahr); // function decl
SAME
  extern double fahr2cels(double);      // function decl
}
```

One Definition Rule (ODR)

- **ODR**: (One Definition Rule)
  - The program must contain **exactly one** definition (...)

One Definition Rule (ODR)

- **ODR**: (One Definition Rule)
  - The program must contain **exactly one** definition (...)

```cpp
// main.cpp



int i = 42;
extern int answer();

int main()
{
   return answer();
}
```

```cpp
// a.cpp



extern int i;

int meaningOfLife()
{
    return i;
}
```

```cpp
// b.cpp



extern int meaningOfLife();

int answer()
{
    return meaningOfLife();
}
```

One Definition Rule (ODR)

```
// mol.h



extern int i;
extern int meaningOfLife();
extern int answer();
```

```
// main.cpp

#include "mol.h"

int i = 42;
extern int answer();

int main()
{
   return answer();
}
```

```
// a.cpp

#include "mol.h"

extern int i;

int meaningOfLife()
{
    return i;
}
```

```
// b.cpp

#include "mol.h"

extern int meaningOfLife();

int answer()
{
    return meaningOfLife();
}
```

One Definition Rule (ODR)

```
// mol.h

#ifndef MOL_H
#define MOL_H

extern int i;
extern int meaningOfLife();
extern int answer();

#endif // MOL_H
```

```
// main.cpp

#include "mol.h"

int i = 42;
extern int answer();

int main()
{
   return answer();
}
```

```
// a.cpp

#include "mol.h"

extern int i;

int meaningOfLife()
{
    return i;
}
```

```
// b.cpp

#include "mol.h"

extern int meaningOfLife();

int answer()
{
    return meaningOfLife();
}
```

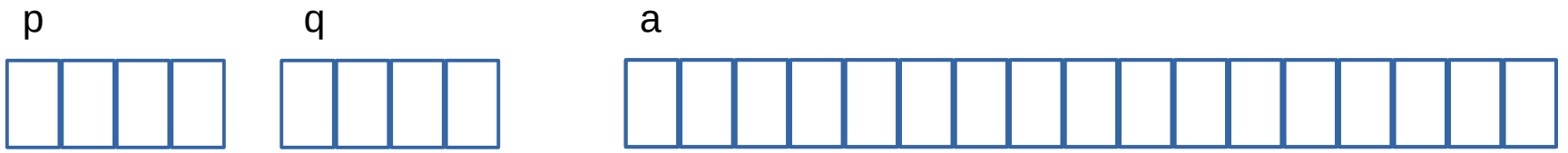
Arrays

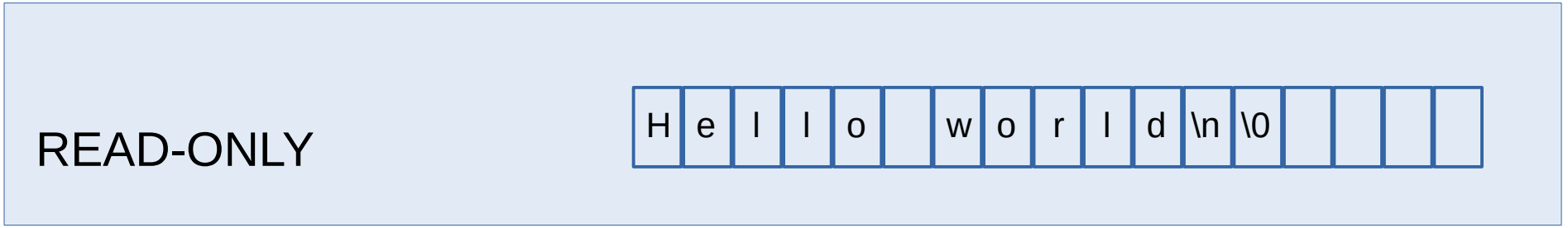
- Strictly continuous memory area, all elements are from the same type

```
void f(void)
{
  double da[5];   // array of 5 double, uninitialized
  int    ia1[]  = { 1, 2, 3 }; // array of 3 int
  int    ia2[3] = { 1, 2 };    // 3 int, ia2[2] == 0

  char s1[] = {'H','e','l','l','o','\0'}; // 6 char
  char s2[] = "Hello";          // 6 char, same as s2

  char *args[] = {"Hello", "world"}; // array of 2 char*
}
```

String literals (recap)

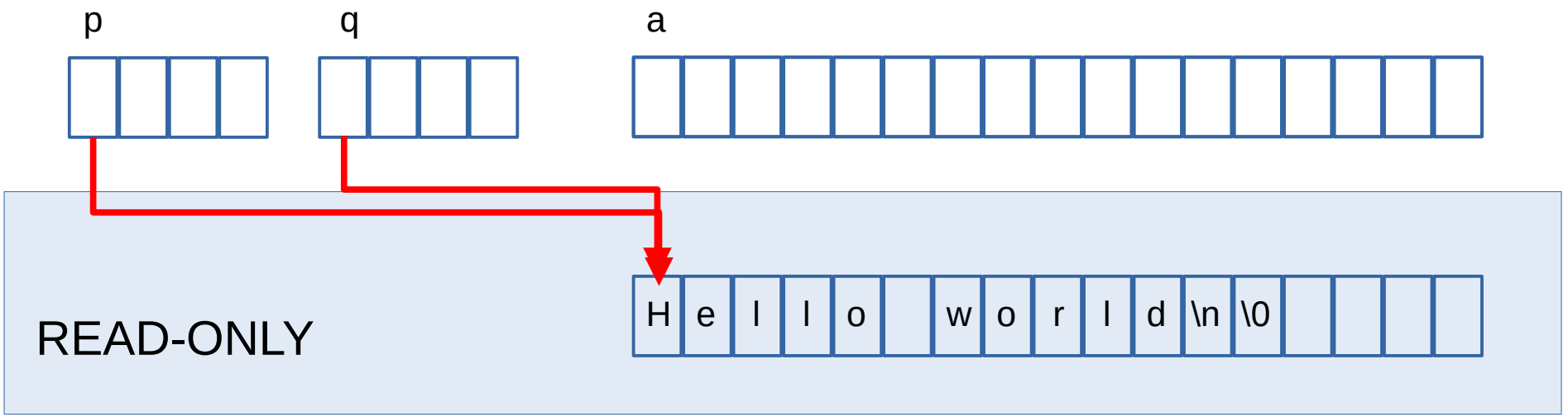
```
printf("Hello world\n");
char *p  = "Hello world\n";
char *q  = "Hello world\n";
char a[] = "Hello world\n";

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

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





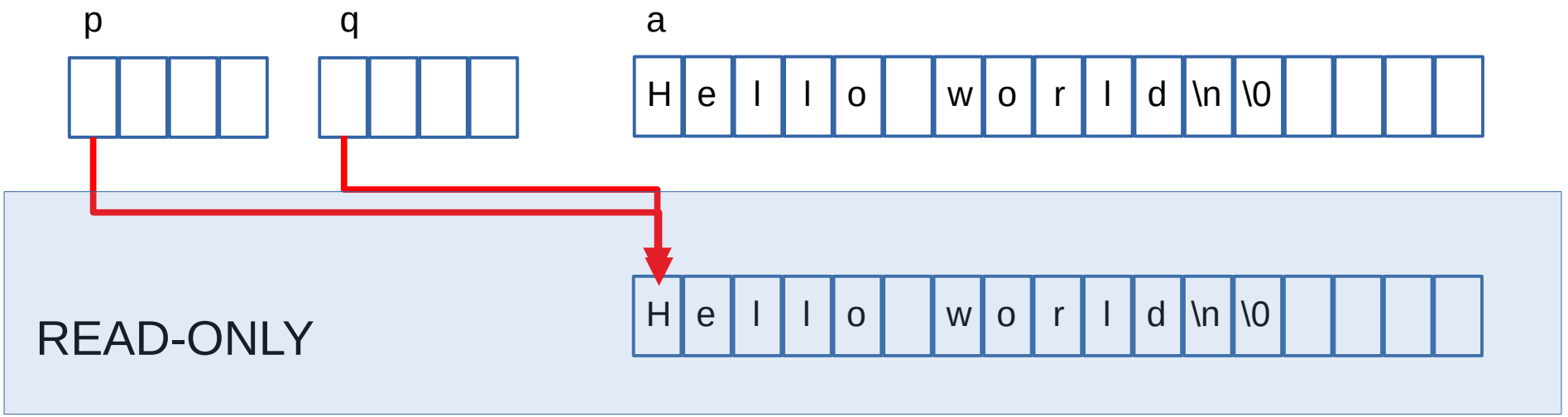
String literals (recap)

```
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
```



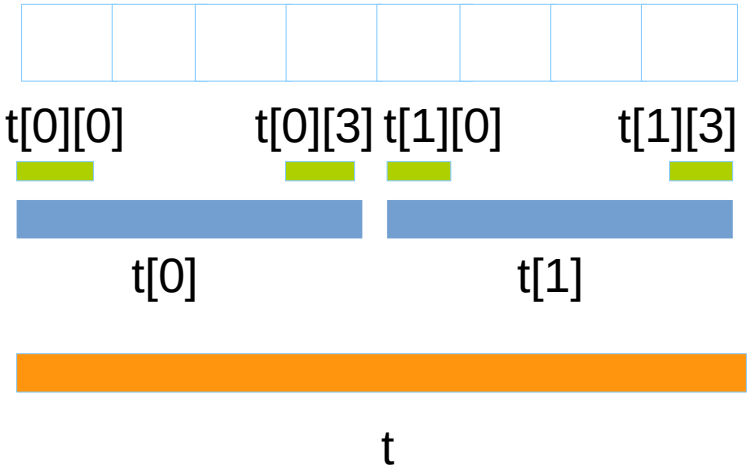
String literals (recap)

```
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
```



Arrays

- An array is a strictly continuous memory area.
- Arrays do not know their size, but: sizeof(t) / sizeof(t[0])
- Array names could be converted to pointer value to the first element.
- No multidimensional arrays. But there are arrays of arrays.
- No operations on arrays, only on array elements.



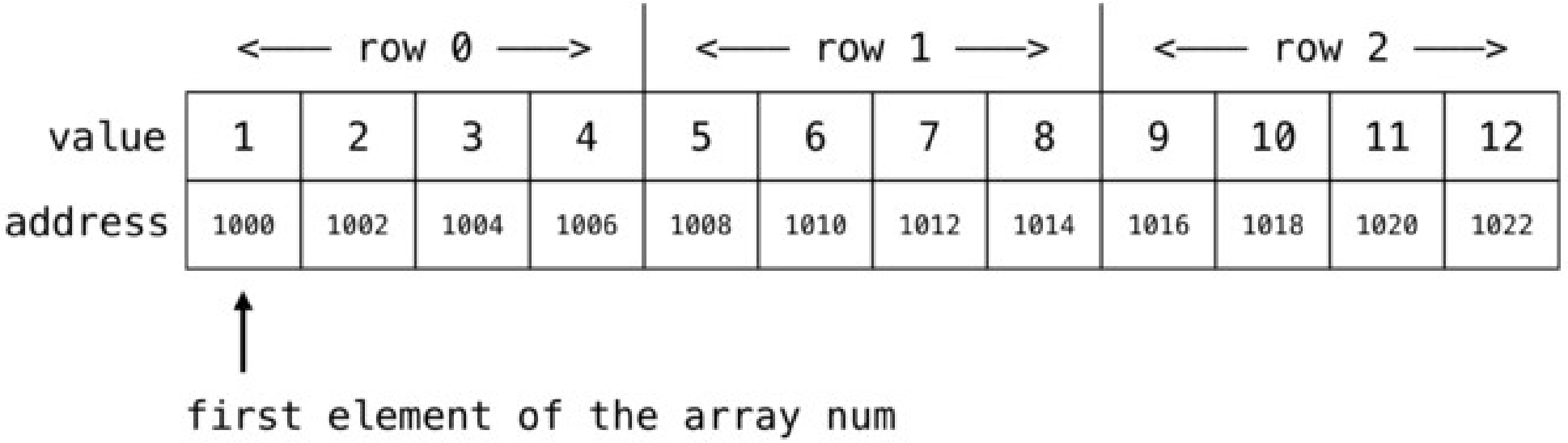

```
int t[2][4];

assert(sizeof(t) == 8*sizeof(int));
assert(sizeof(t[0] == 4*sizeof(int));
assert(sizeof(t[0][0])==sizeof(int));

t[0][1] = t[1][1];
// t[0] = t[1];  syntax error
```

Arrays

```
int num[3][4] = {
  {1, 2,  3,  4},
  {5, 6,  7,  8},
  {9, 10, 11, 12}
};
```

CLASSROOM

row-wise memory allocation



dyclassroom.com

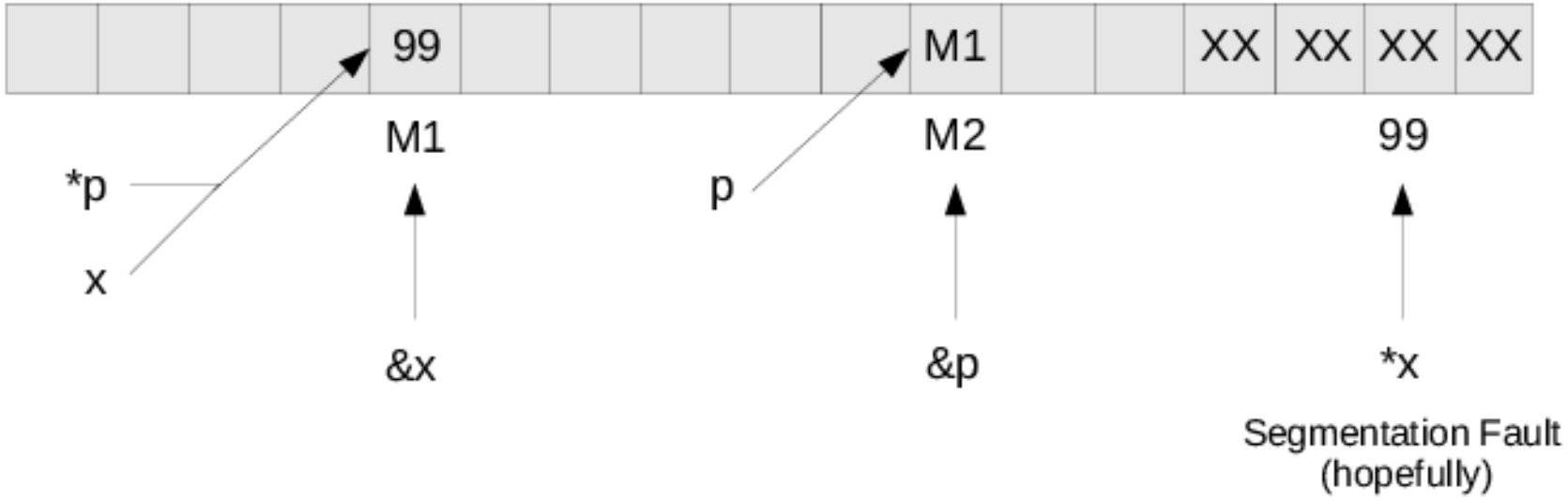
Pointers

C Code

Pointers in C

```
int x;
int *p;

x = 99;    //holds a value
p = &x;    //holds an address of a value
```

Memory



Pointers

- Pointer is a value that refers to a(n abstract) memory location
- Pointers can refer to any valid memory locations (unlike e.g. PASCAL)
- NULL is a universal null-pointer value (defined as a MACRO)
- Non-null pointers are considered as **true** value

```
int  i = 1;
int  j = 2;

int  *ip = &i;  /* ip points to variable i */
int  *jp = &j;  /* jp points to variable j */

if ( ip == jp )   { ... }  /* false */
if ( ip == NULL ) { ... }  /* false */
if ( 2 == *jp )   { ... }  /* true  */

ip = jp;    /* now ip also points to variable j */
if ( ip == jp )   { ... }  /* true  */
```

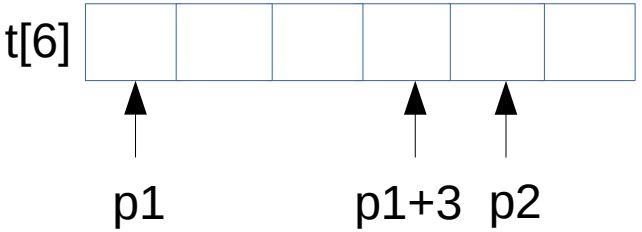
Pointers

- Pointer is a value that refers to a(n abstract) memory location
- Pointers can refer to any valid memory locations (unlike e.g. PASCAL)
- NULL is a universal null-pointer value (defined as a MACRO)
- Non-null pointers are considered as **true** value

```
char *ptr = (char *) malloc(1024);

if ( ptr )   // NULL != ptr
{
   // ptr != NULL here
}
```

Pointer arithmetics

- Pointers pointing to the same array can be compared by `<` `<=` `>` `>=`
- Integers can be added and subtracted from pointers
- Pointers pointing to the same array can be subtracted
- Pointer arithmetic depends on the pointed type!
- No pointer arithmetic on void *

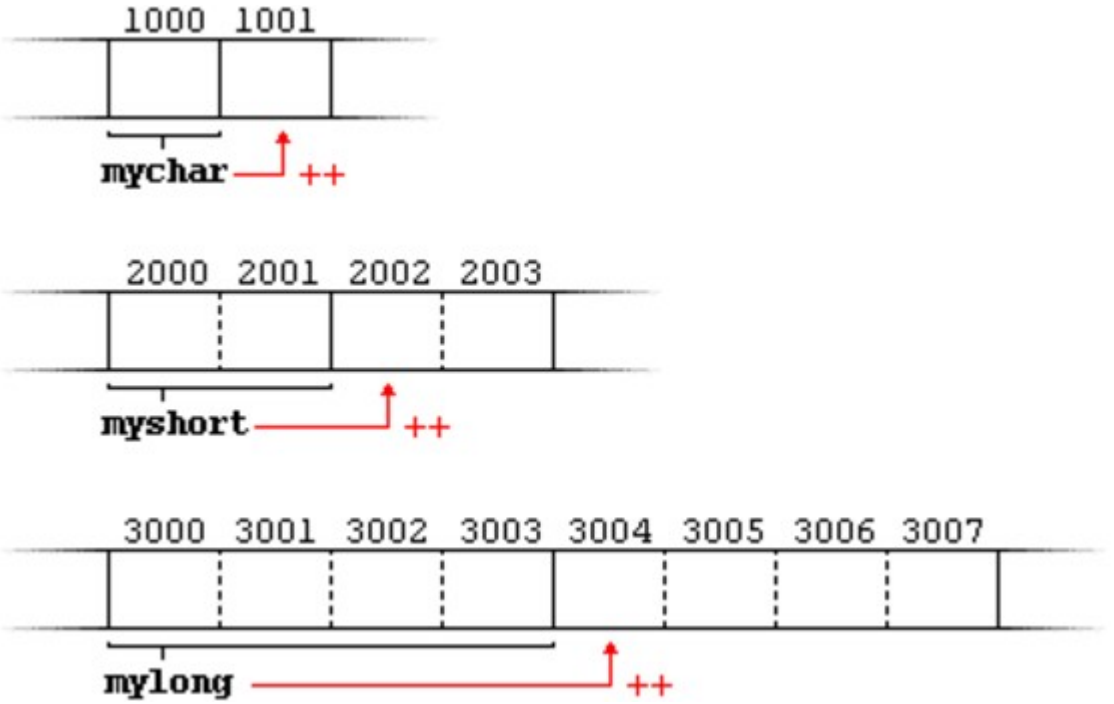



```
int t[6];
int *p1 = &t[0];
int *p2 = &t[4];

assert( &t[3] == p1+3 );
assert( p2 - p1 == 4 );
assert( p1 - p2 == -4 );
assert( p1 + 4 == p2 );
```

```
p = &t[0];
p = t;
p + k == &t[k]
*(p + k) == t[k]
```

Pointer arithmetics

```
char  ca[100];
short sa[100];
long  la[100];

char  *mychar  = ca;
short *myshort = sa;
long  *mylong  = la;

mychar++;
char ch = *mychar;

myshort++;
short s = *myshort;

mylong++;
long  l = *mylong;
```



Pointers and arrays

- Array names are converted to pointer to the first element
- Pointers and array names can be used similarly in expressions
- The index operator always defined via pointer arithmetics

```
int t[10];
int i = 3;

t[i] = 42;  // == *(t+i) = 42

int *p = t; // == &t[0], t is used as pointer to the first element

// if  &t[0] == t == p  then  t[i] == *(t+i) == *(p+i) == p[i]

p[i] = 42;  // == *(p+i)= 42, p used as array,
```

Pointers and arrays

- Array names are converted to pointer to the first element
- Pointers and array names can be used similarly in expressions
- The index operator always defined via pointer arithmetics

```
int t[10];
int i = 3;

t[i] = 42;  // == *(t+i) = 42

int *p = t; // == &t[0], t is used as pointer to the first element

// if  &t[0] == t == p  then  t[i] == *(t+i) == *(p+i) == p[i]

p[i] = 42;  // == *(p+i), p used as array,

printf("%c", "Hello"[2]); // == "Hello"[2] == *("Hello"+2) ==
printf("%c", 2["Hello"]); // == *(2+"Hello") == 2["Hello"]
```

Pointers and arrays

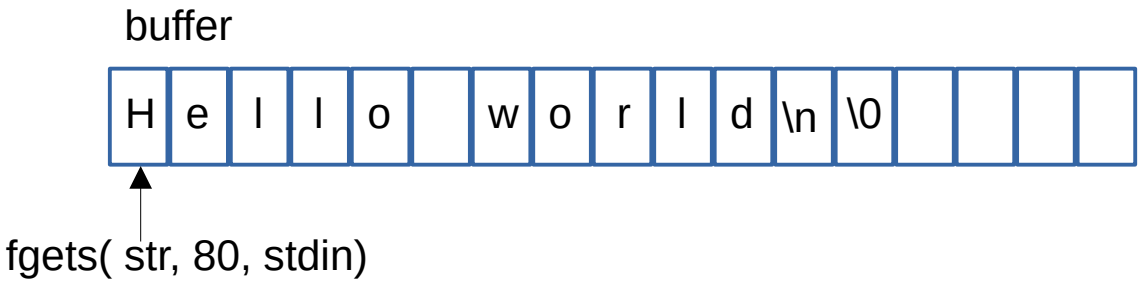
- Array names are converted to pointer to the first element
- Pointers and array names can be used similarly in expressions
- The index operator always defined via pointer arithmetics

```
int t[10];
int i = 3;

t[i] = 42;  // == *(t+i) = 42

int *p = t; // == &t[0], t is used as pointer to the first element

// if  &t[0] == t == p  then  t[i] == *(t+i) == *(p+i) == p[i]

p[i] = 42;  // == *(p+i), p used as array,

printf("%c", "Hello"[2]); // 2["Hello"] == *(2+"Hello") ==
printf("%c", 2["Hello"]); // == *(2+"Hello") == "Hello"[2]
```

But pointers ARE NOT EQUIVALENT TO arrays !!!

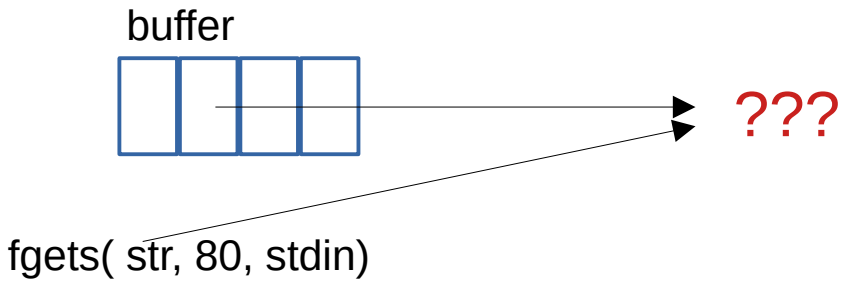
Pointers and arrays are different

```
#include <stdio.h> // char* fgets( char* str, int count, FILE* fp);
#define BUFSIZE 80

int main() // copy the standard input to standard output line by line
{
  char buffer[BUFSIZE];
  while ( fgets( buffer, BUFSIZE, stdin) ) // reads max BUFSIZE-1 char
  {                                        // places '\0' to the end
    fprintf(stdout, "%s", buffer);
  }
  return 0;
}
$ gcc copy.c
$ ./a.out
Hello world<ENTER>
```



Pointers and arrays are different

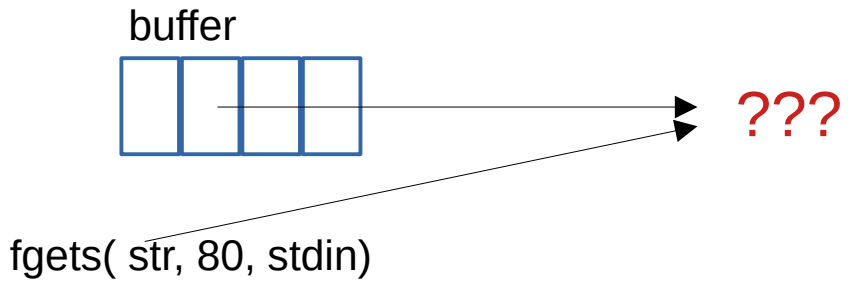
```
#include <stdio.h> // char* fgets( char* str, int count, FILE* fp);
#define BUFSIZE 80

int main() // copy the standard input to standard output line by line
{
  char *buffer;   // wrong, no space for characters to read
  while ( fgets( buffer, BUFSIZE, stdin) ) // reads max BUFSIZE-1 char
  {                                        // places '\0' to the end
    fprintf(stdout, "%s", buffer);
  }
  return 0;
}
$ gcc copy.c
$ ./a.out
Hello world<ENTER>
```



Pointers and arrays are different

```
#include <stdio.h> // char* fgets( char* str, int count, FILE* fp);
#define BUFSIZE 80

int main() // copy the standard input to standard output line by line
{
  char *buffer;   // wrong, no space for characters to read
  while ( fgets( buffer, BUFSIZE, stdin) ) // reads max BUFSIZE-1 char
  {                                        // places '\0' to the end
    fprintf(stdout, "%s", buffer);
  }
  return 0;
}
$ gcc copy.c
$ ./a.out
Hello world<ENTER>
```

Pointers ARE NOT EQUIVALENT TO arrays !!!



Pointers and arrays are different

```c
// s1.c
#include <stdio.h>

int t[] = {1,2,3};

void f( int *par)
{
  printf("%d",par[1]);
  printf("%d",t[1]);
}
int main()
{
  int *p = t;
  printf("%d",t[1]);
  printf("%d",p[1]);
  f(t);
  g(t);
}
```

```c
// s2.c
#include <stdio.h>

extern int *t;

void g( int *par)
{
  printf("%d",par[1]);
  printf("%d",t[1]);
  // the program crashes here
}
```

Variadic length array (VLA)

- Since C99 array size can be dynamic

```
void f()
{
  int n;
  printf("enter the size of the array: ");
  scanf("%d",&n);

  int t[n];  /* VLA – do not use */
  /* array elements: t[0]...t[n-1] */
}
```

- VLA proved to be unsafe and cause more security issues
- VLA is slower than the fixed size array
- It was never a standard and will not be in C++
- A paper to make Linux VLA-free: https://www.phoronix.com/news/Linux-Kills-The-VLA

Variadic length array (VLA)

- Since C99 array size can be dynamic

```
void f()
{
  int n;
  printf("enter the size of the array: ");
  scanf("%d",&n);

  int t[n]; /* VLA – do not use */
  /* array elements: t[0]...t[n-1] */
}
```

DO NOT USE VLA!

- VLA proved to be unsafe and cause more security issues
- VLA is slower than the fixed size array
- It was never a standard and will not be in C++
- A paper to make Linux VLA-free: