

# *Formal Methods for C*

*Seminar – Summer Semester 2014*

Daniel Dietsch, Sergio Feo Arenis, Marius Greitschus, **Bernd Westphal**

# Content

---

- Brief history
- Comments
- Declarations and Scopes
  - Variables
  - Expressions and Statements
  - Functions
  - Scopes
- Pointers
- Dynamic Storage & Storage Duration
- Storage Class Specifiers
- Strings and I/O
- Tools & Modules
- Formal Methods for C
- Common Errors

# *Function Pointers*

# Functions in the System's Memory

```
1 void f() { return; }
```

the compiler chose to store the machine code of 'f' at memory cell with address 0x1001

0x1000	0x1001 RET	0x1002	0x1003	...				
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F	
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017	
				...				

# Calling Functions

---


```
1 void f() { return; }  
2 f();
```

# Calling Functions

```
1 void f() { return; }
2 f();
```

calling **'f'** means  
machine op CALL with  
address of callee (here:  
**f**, address 0x1001)

0x1000	0x1001 RET	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A	0x100B CALL	0x100C 0x10	0x100D 0x01	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017
				...			



# *A Pointer to 'f' (16-bit Architecture)*

---

```
1 void f() { return; }  
2 f();  
3 void (*p) () = &f;
```

# A Pointer to 'f' (16-bit Architecture)

```
1 void f() { return; }
2 f();
3 void (*p) () = &f;
```

'p' is a variable which stores the address of a function (here: of 'f')

...							
0x1000	0x1001 RET	0x1002	0x1003	0x1004 0x10	0x1005 0x01	0x1006	0x1007
0x1008	0x1009	0x100A	0x100B CALL	0x100C 0x10	0x100D 0x01	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017
				...			



# Dereference Function Pointers

```
1 void f() { return; }
2 f();
3 void (*p) () = &f;
4 (*p)();
```

				...				
0x1000	0x1001 RET	0x1002	0x1003	0x1004 0x10	0x1005 0x01	0x1006	0x1007	
0x1008	0x1009	0x100A	0x100B CALL	0x100C 0x10	0x100D 0x01	0x100E	0x100F	
0x1010 MOV	0x1011 0x10	0x1012 0x04	0x1013 R	0x1014 CALL	0x1015 R	0x1016	0x1017	
				...				

# Dereference Function Pointers

```
1 void f() { return; }
2 f();
3 void (*p) () = &f;
4 (*p)();
```

calling via **'p'** means read  
value of **p** (here: 0x1001)  
into register *R*...

0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
	RET			0x10	0x01		
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
			CALL	0x10	0x01		
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017
MOV	0x10	0x04	<i>R</i>	CALL	<i>R</i>		
				...			

# Dereference Function Pointers

```
1 void f() { return; }
2 f();
3 void (*p) () = &f;
4 (*p)();
```

calling via '**p**' means read value of **p** (here: 0x1001) into register *R*...

...and then machine op CALL with *R*, calls address stored in *R*

0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
	RET			0x10	0x01		
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
			CALL	0x10	0x01		
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017
MOV	0x10	0x04	<i>R</i>	CALL	<i>R</i>		
				...			

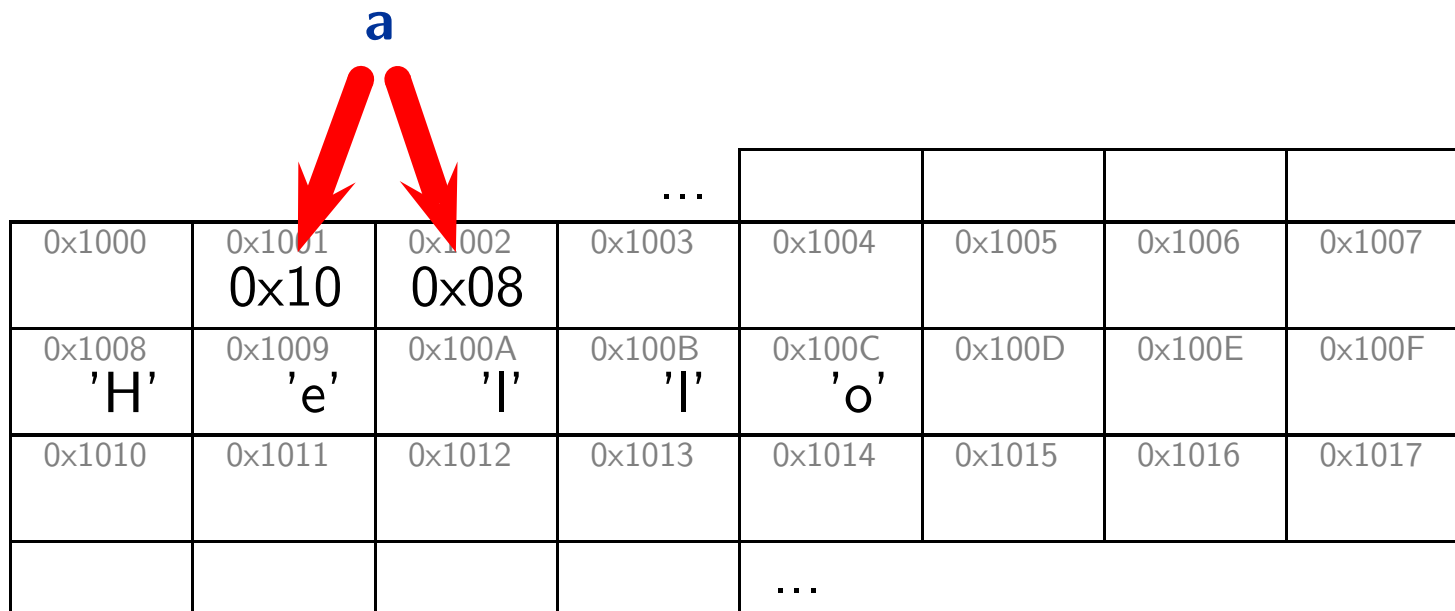
# *Pointers vs. Arrays*

# Arrays

reserve some space  
for 5 **chars**...

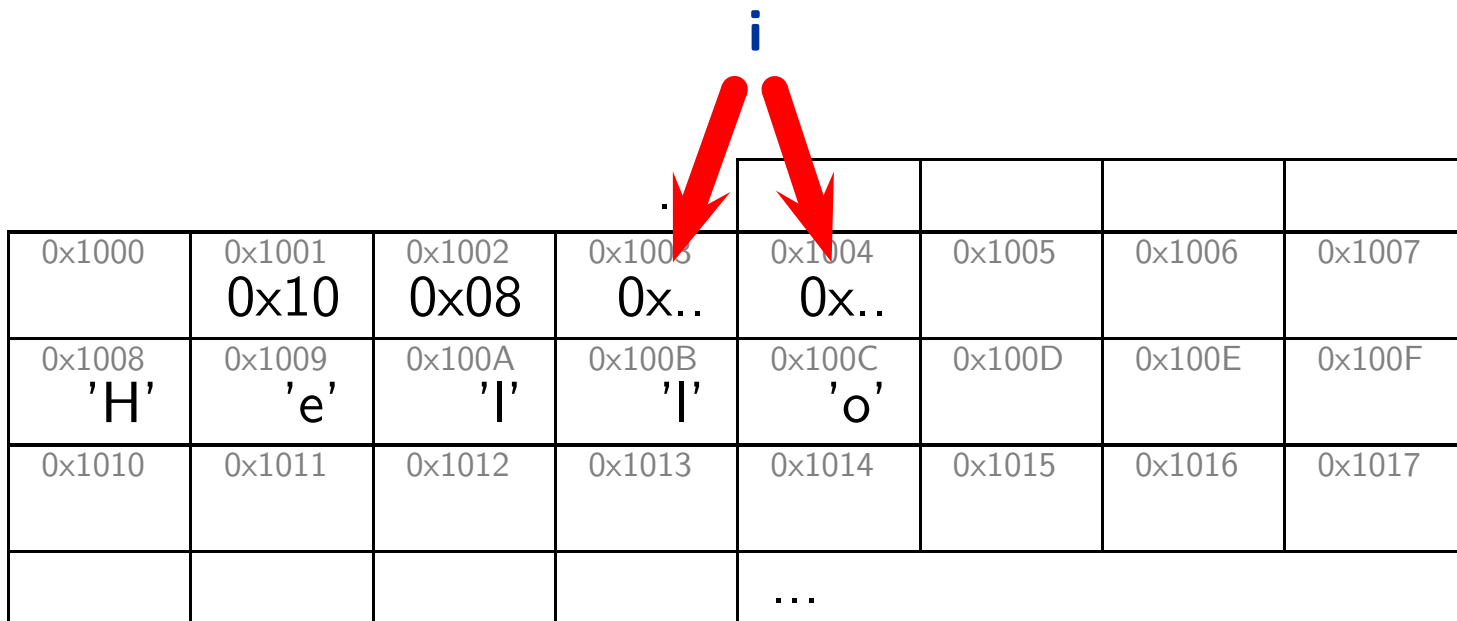
```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
```

...and let **a** point  
to that space



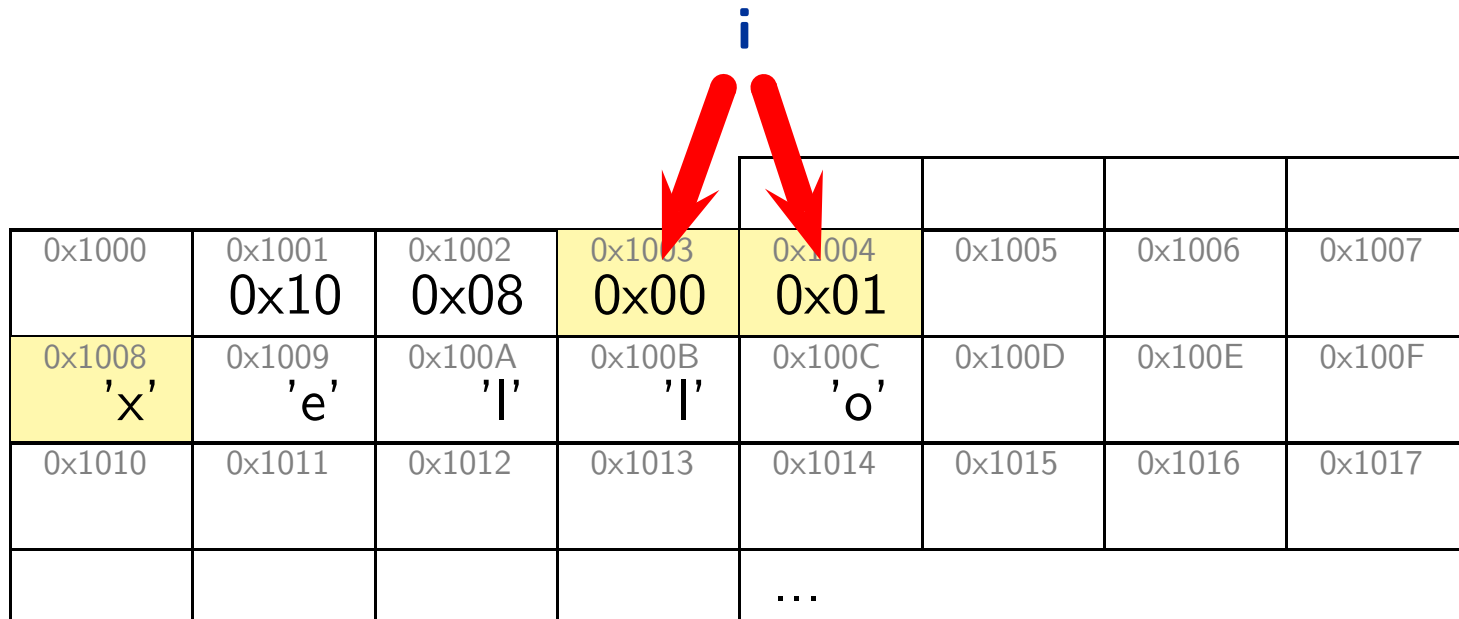
# Arrays

```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 int i;
3 for (i = 0; i < 5; ++i)
4     a[i] = 'x';
```



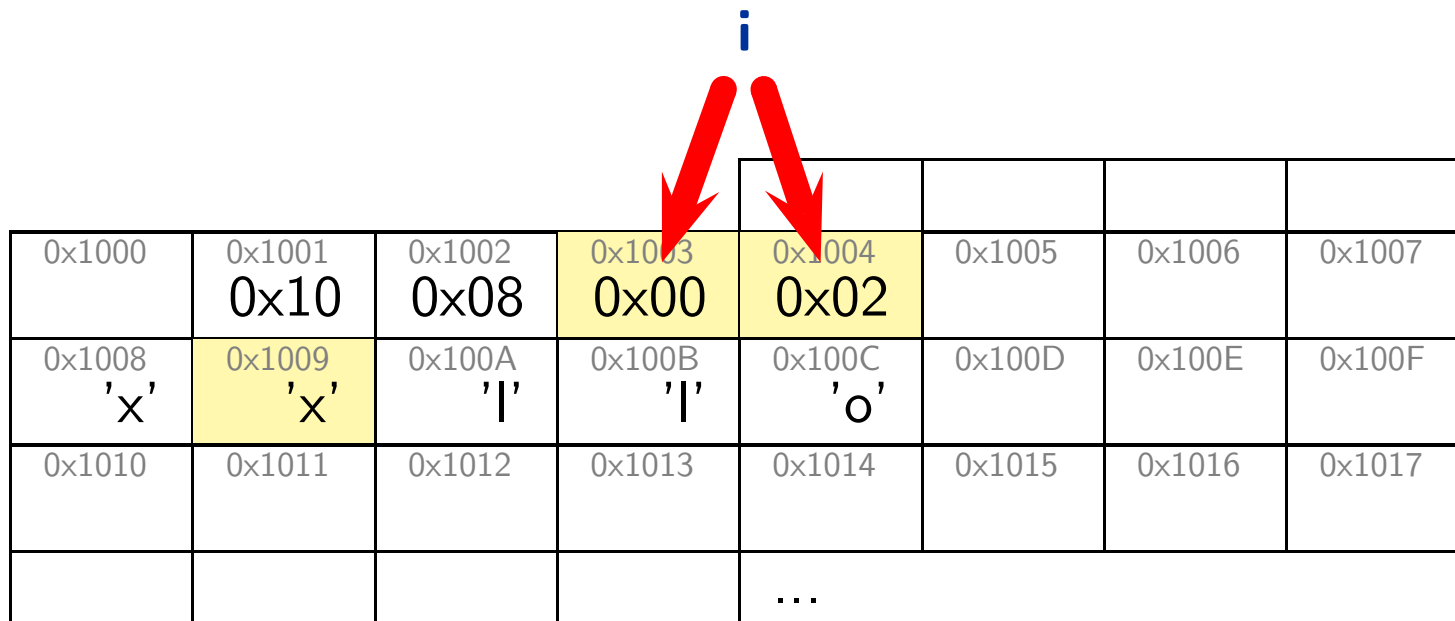
# Arrays

```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };  
2 int i;  
3 for (i = 0; i < 5; ++i)  
4   a[i] = 'x';
```



# Arrays

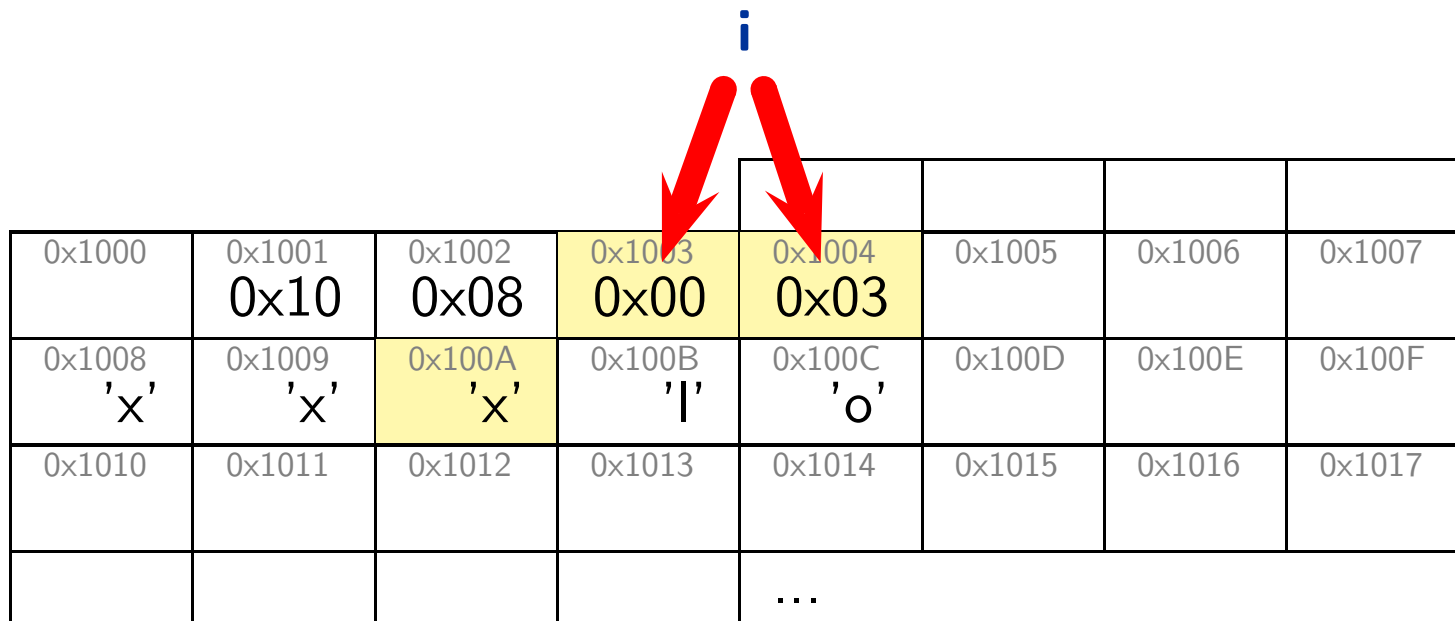
```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 int i;
3 for (i = 0; i < 5; ++i)
4     a[i] = 'x';
```





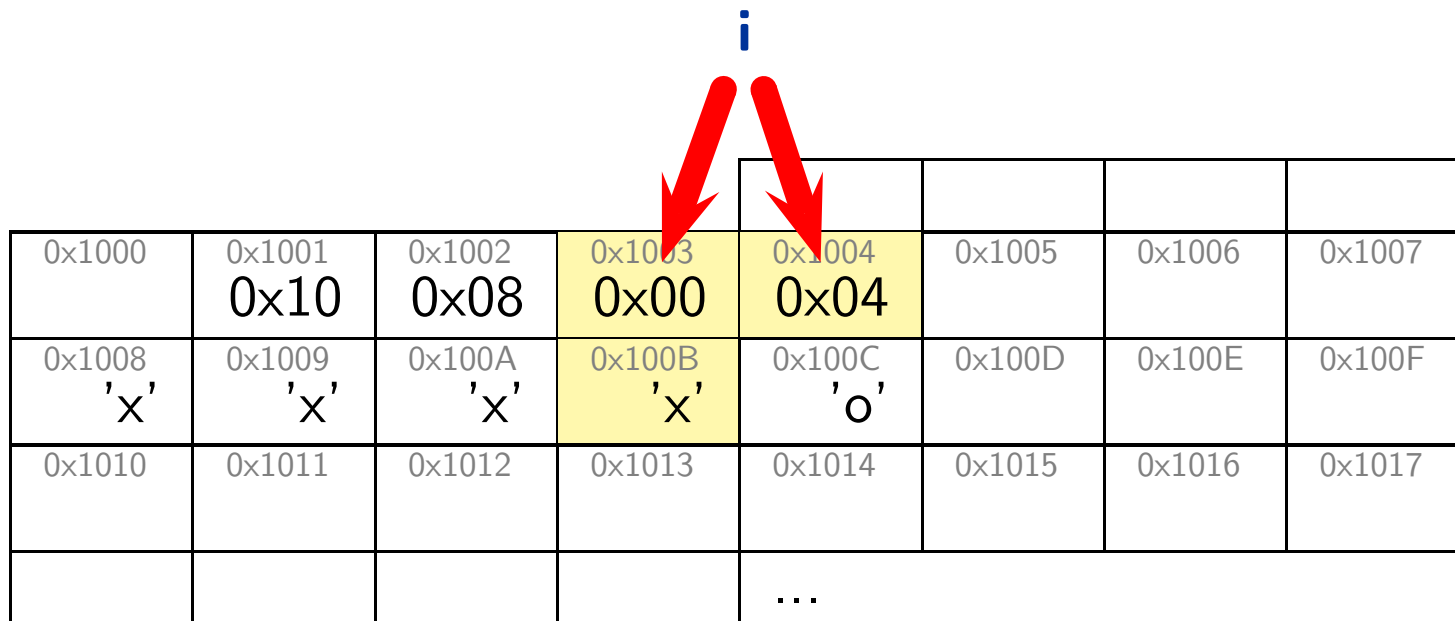
# Arrays

```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 int i;
3 for (i = 0; i < 5; ++i)
4     a[i] = 'x';
```



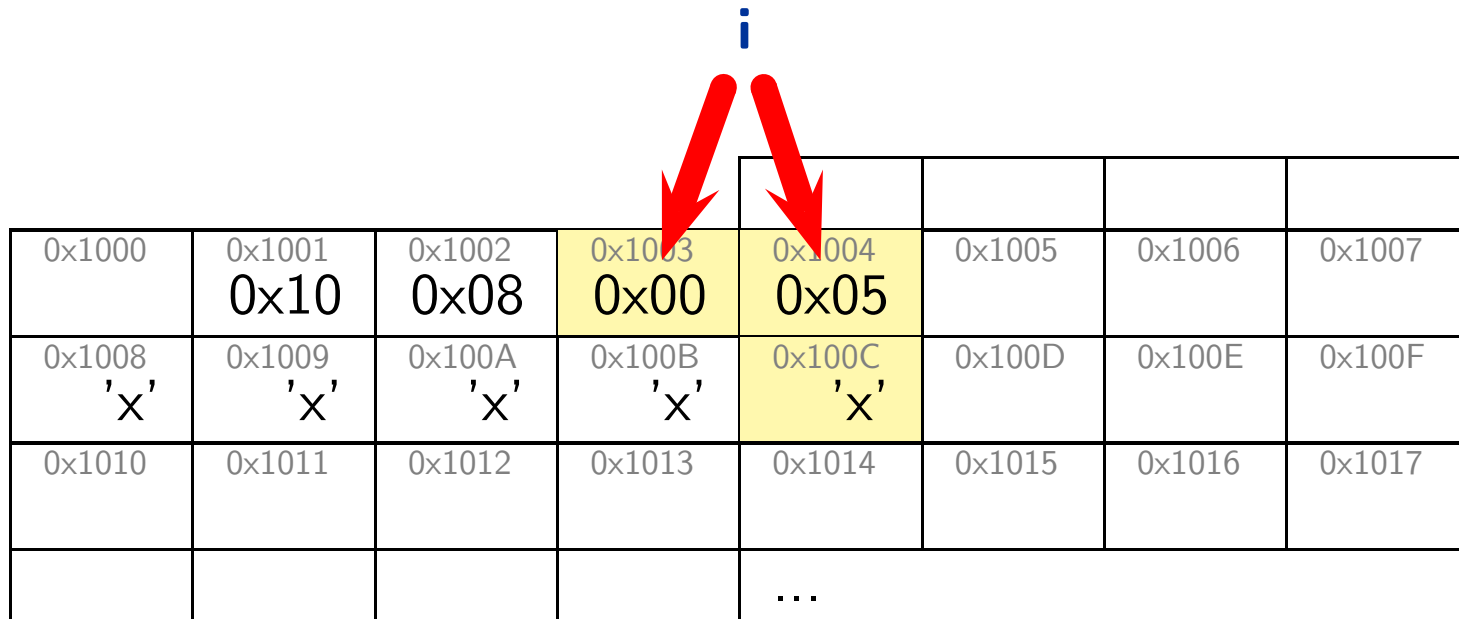
# Arrays

```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 int i;
3 for (i = 0; i < 5; ++i)
4     a[i] = 'x';
```



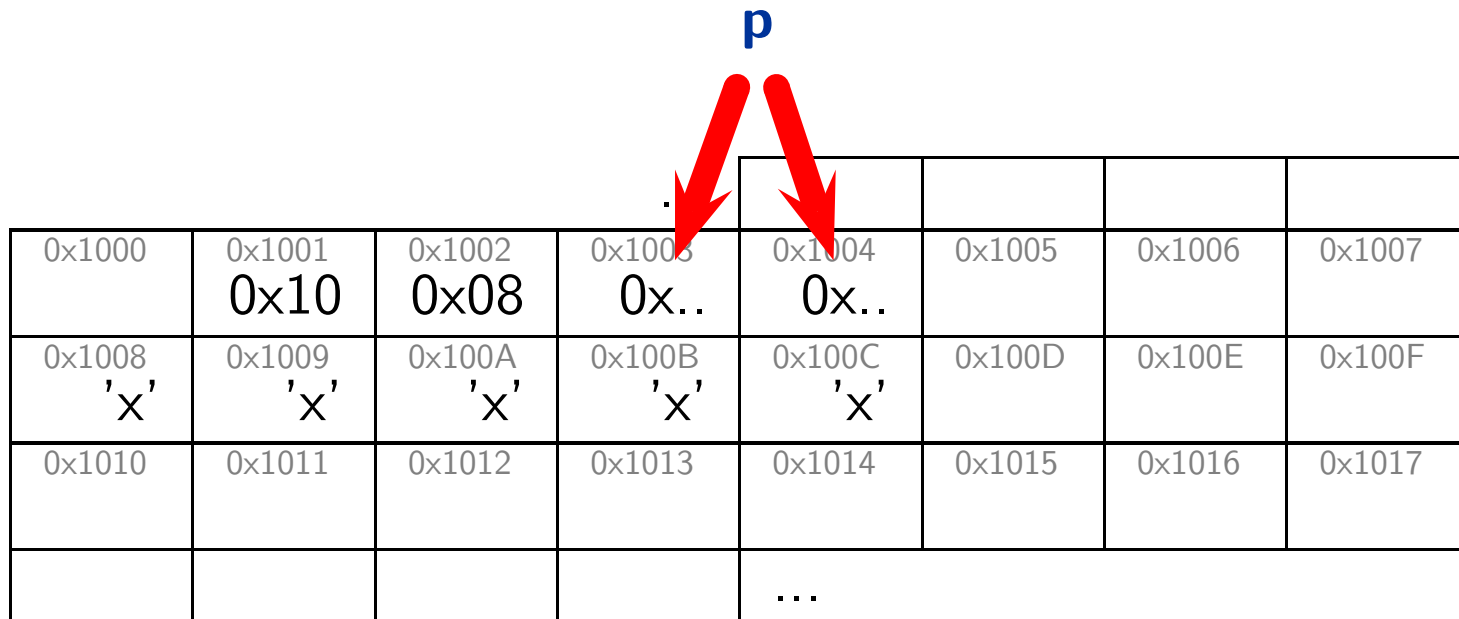
# Arrays

```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 int i;
3 for (i = 0; i < 5; ++i)
4     a[i] = 'x';
```



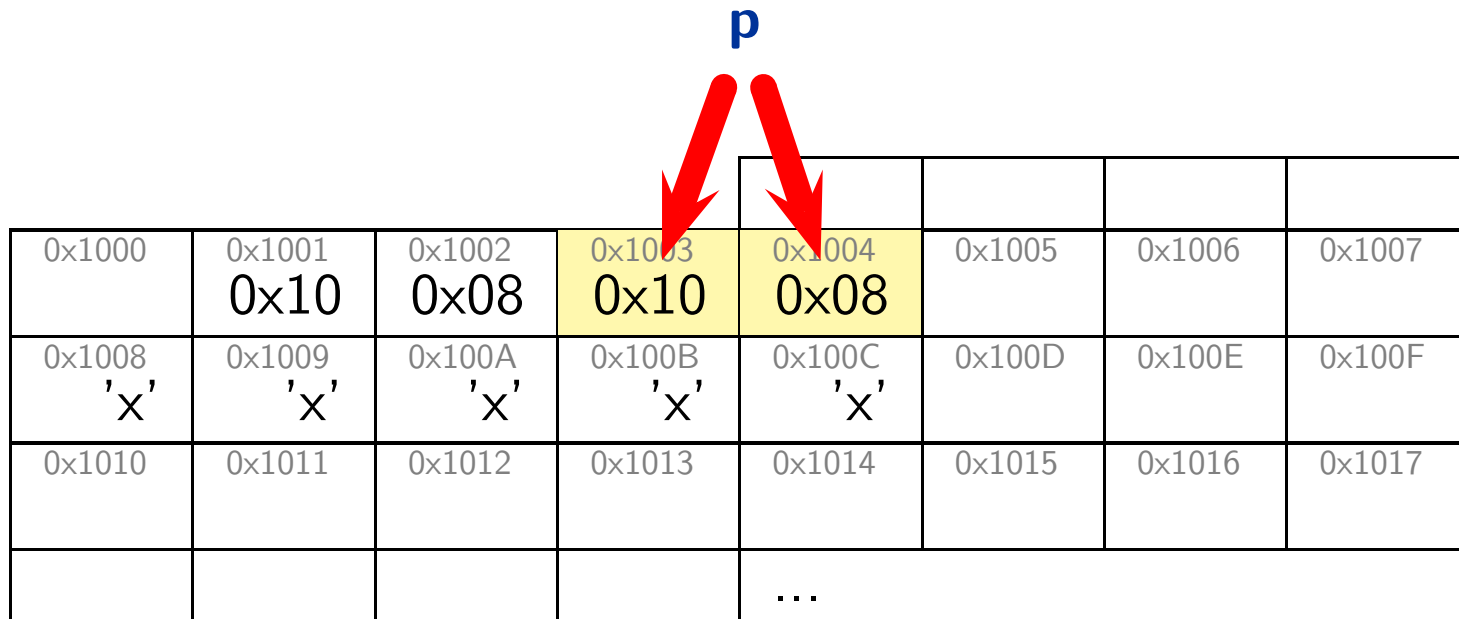
# Arrays vs. Pointers

```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };  
2 char* p = a; // not &a !  
3 for (int i = 0; i < 5; ++i, ++p)  
4     *p = 'o';
```



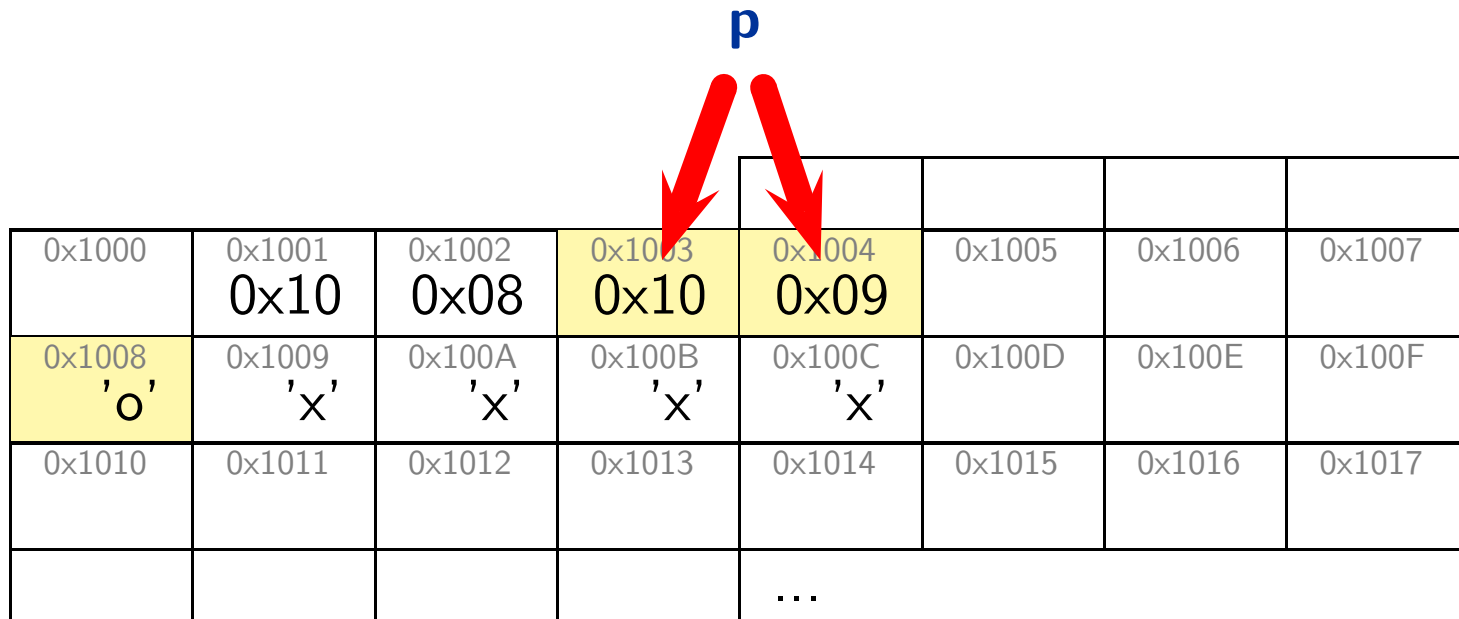
# Arrays vs. Pointers

```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };  
2 char* p = a; // not &a !  
3 for (int i = 0; i < 5; ++i, ++p)  
4     *p = 'o';
```



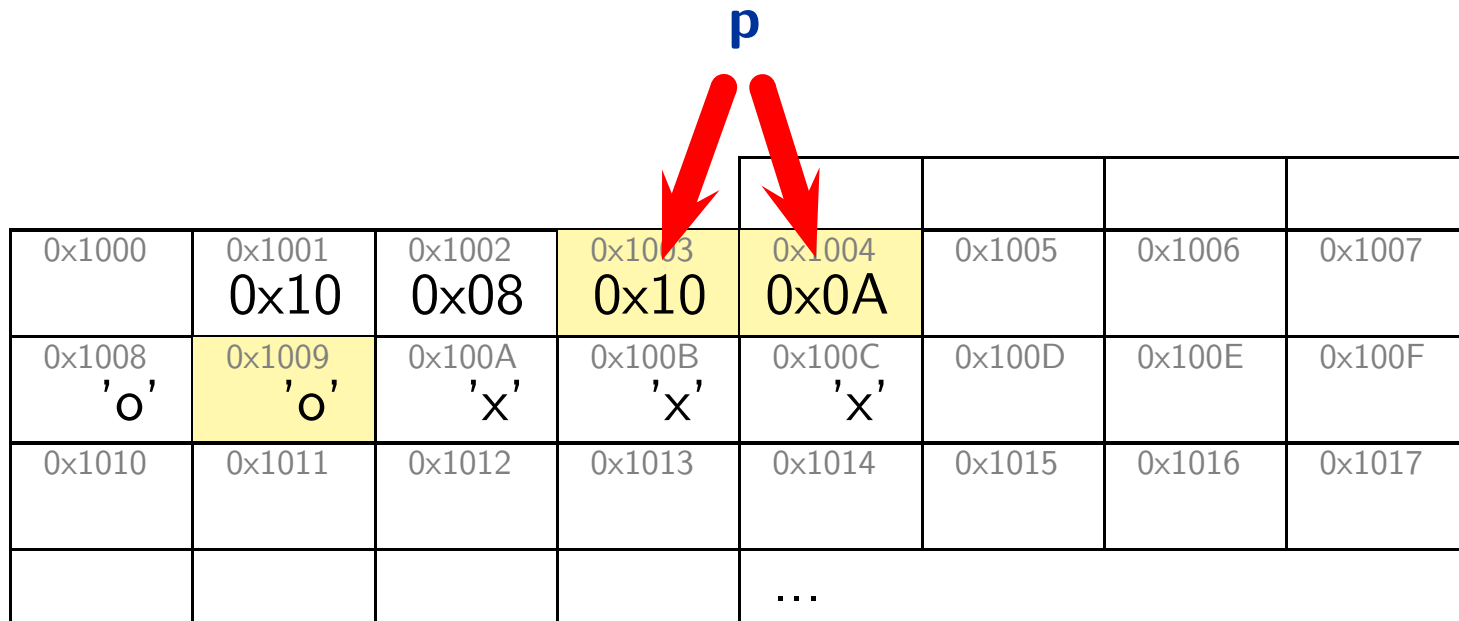
# Arrays vs. Pointers

```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 char* p = a; // not &a !
3 for (int i = 0; i < 5; ++i, ++p)
4     *p = 'o';
```



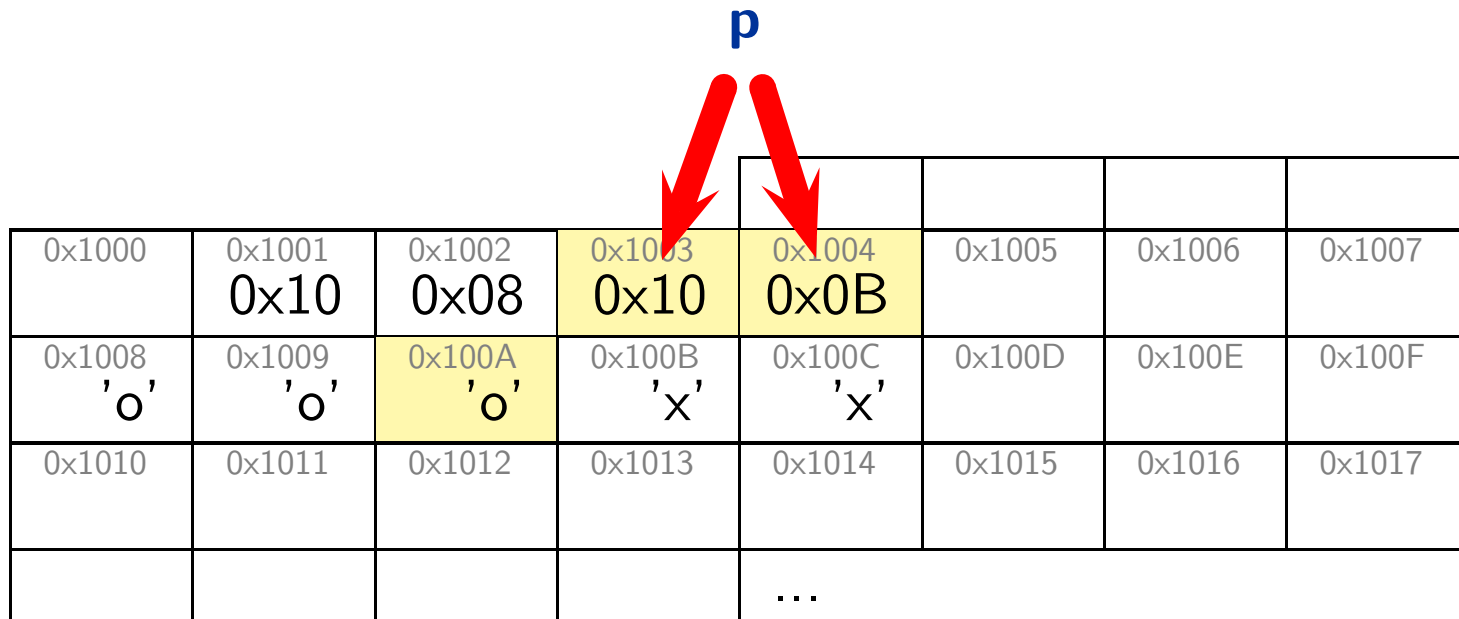
# Arrays vs. Pointers

```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 char* p = a; // not &a !
3 for (int i = 0; i < 5; ++i, ++p)
4     *p = 'o';
```



# Arrays vs. Pointers

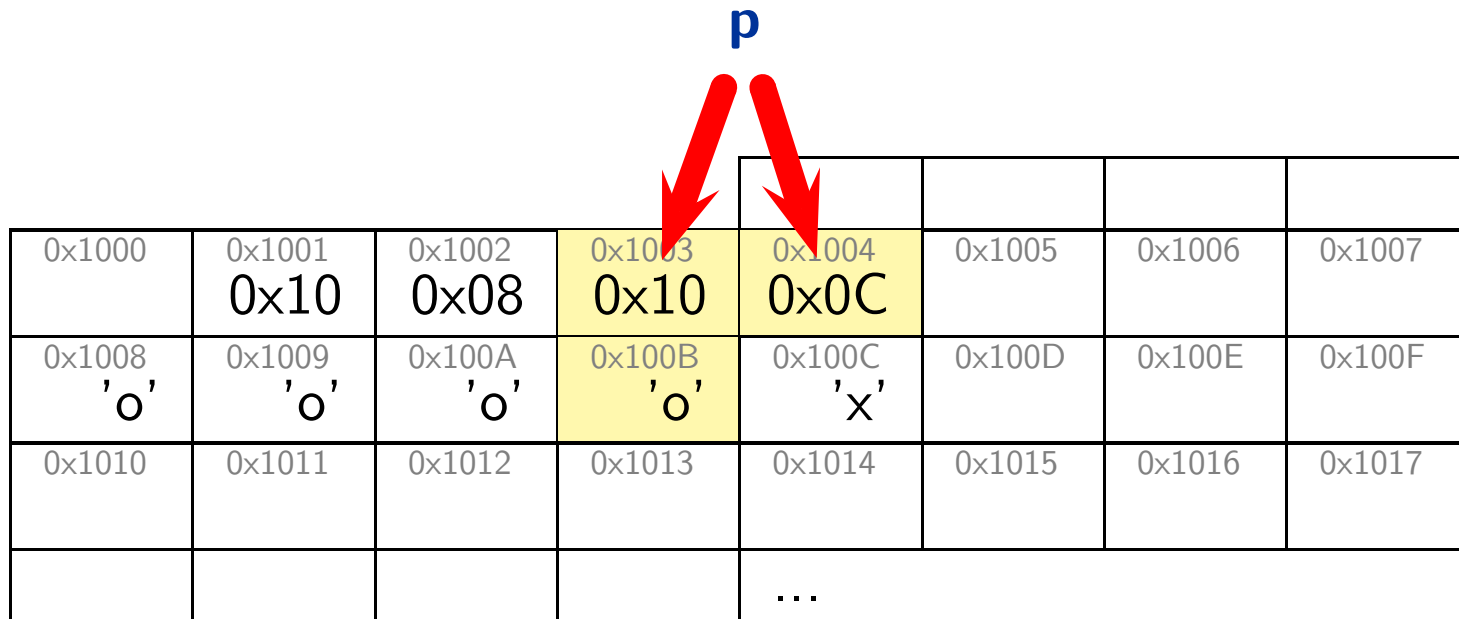
```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 char* p = a; // not &a !
3 for (int i = 0; i < 5; ++i, ++p)
4     *p = 'o';
```





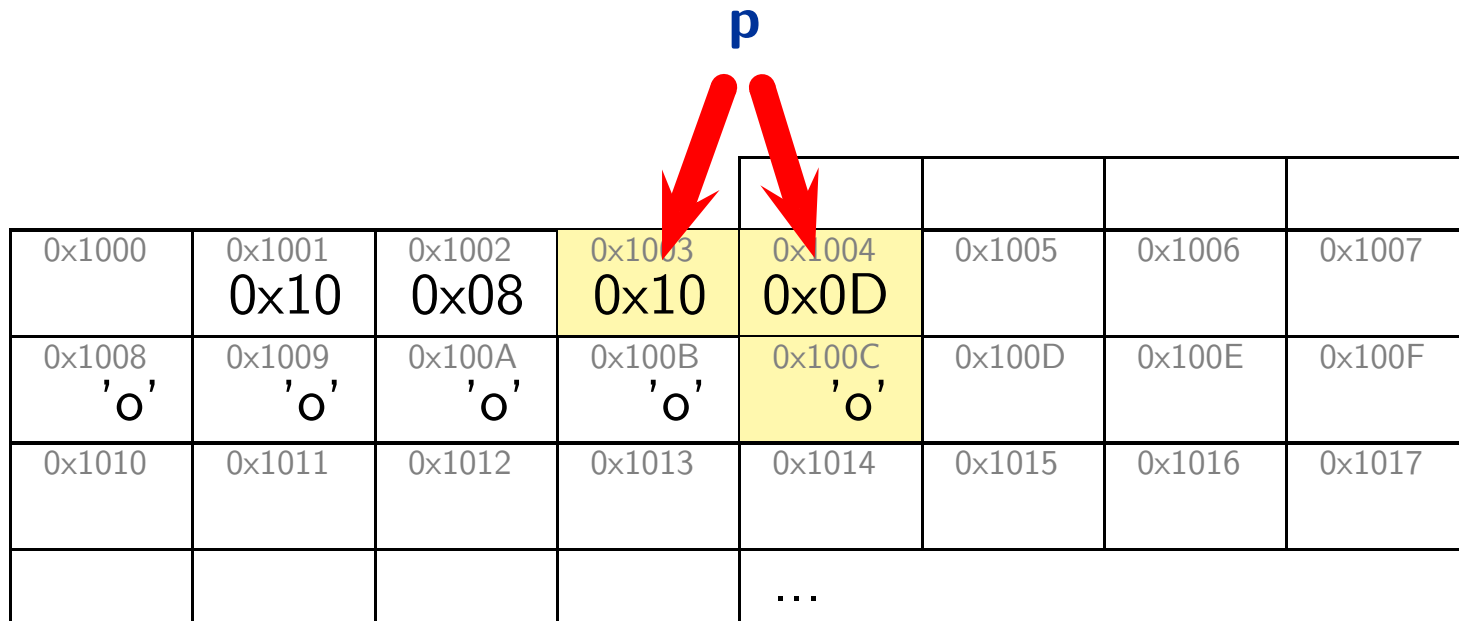
# Arrays vs. Pointers

```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 char* p = a; // not &a !
3 for (int i = 0; i < 5; ++i, ++p)
4     *p = 'o';
```



# Arrays vs. Pointers

```
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 char* p = a; // not &a !
3 for (int i = 0; i < 5; ++i, ++p)
4     *p = 'o';
```

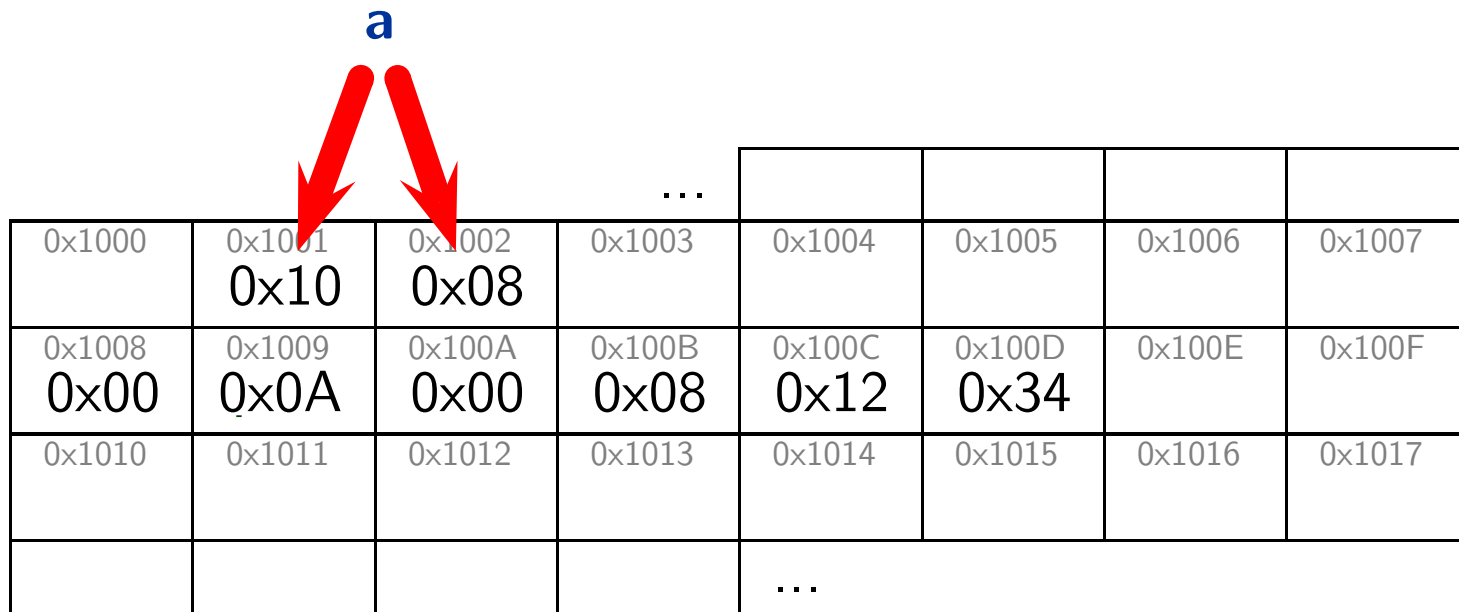


# Integer Arrays

reserve some space  
for 3 **ints**...

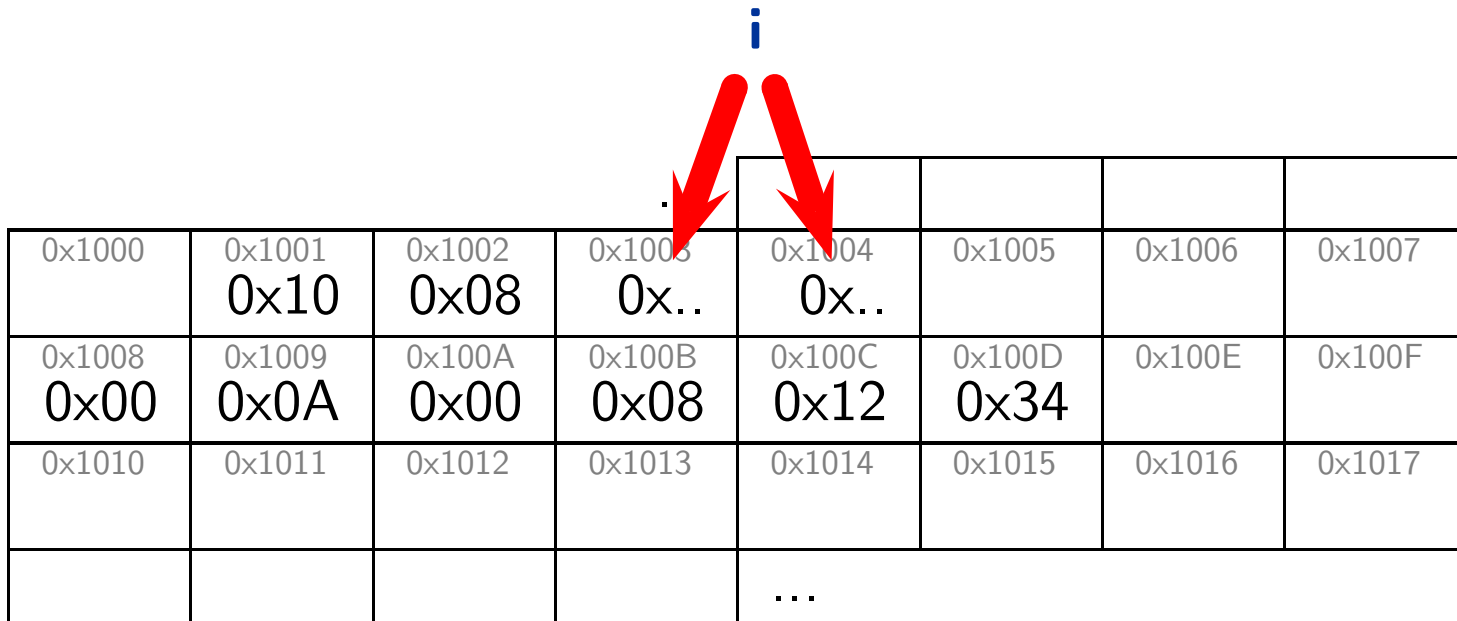
```
1 int a[3] = { 10, 010, 0x1234 };
```

...and let **a** point  
to that space



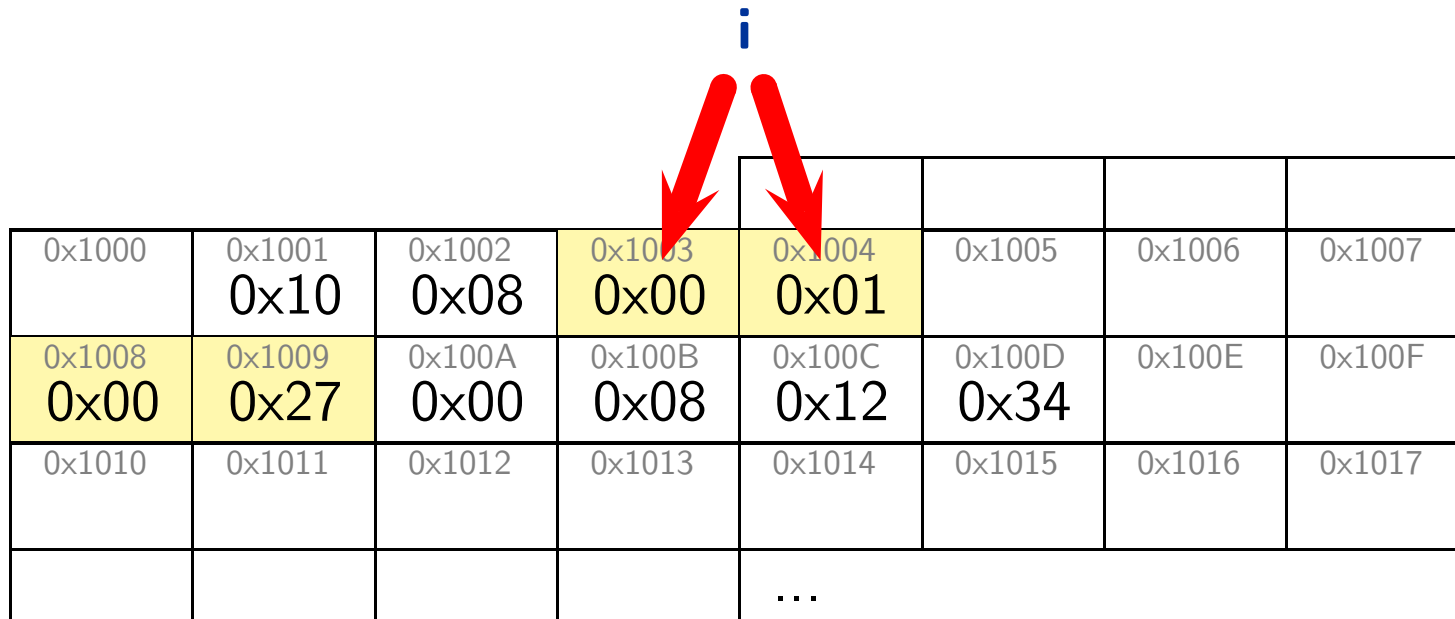
# Integer Arrays

```
1  int a[3] = { 10, 010, 0x1234 };
2  int i;
3  for (i = 0; i < 3; ++i)
4      a[i] = 0x27;
```



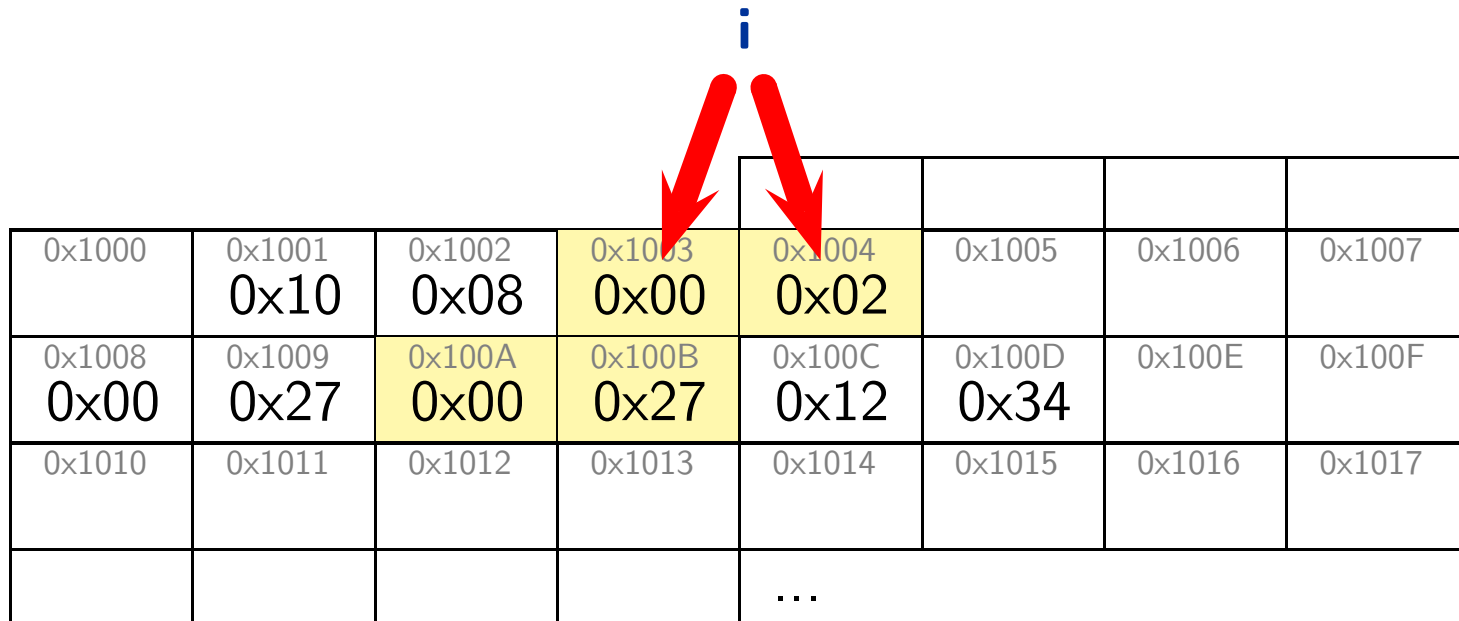
# Integer Arrays

```
1  int a[3] = { 10, 010, 0x1234 };
2  int i;
3  for (i = 0; i < 3; ++i)
4  a[i] = 0x27;
```



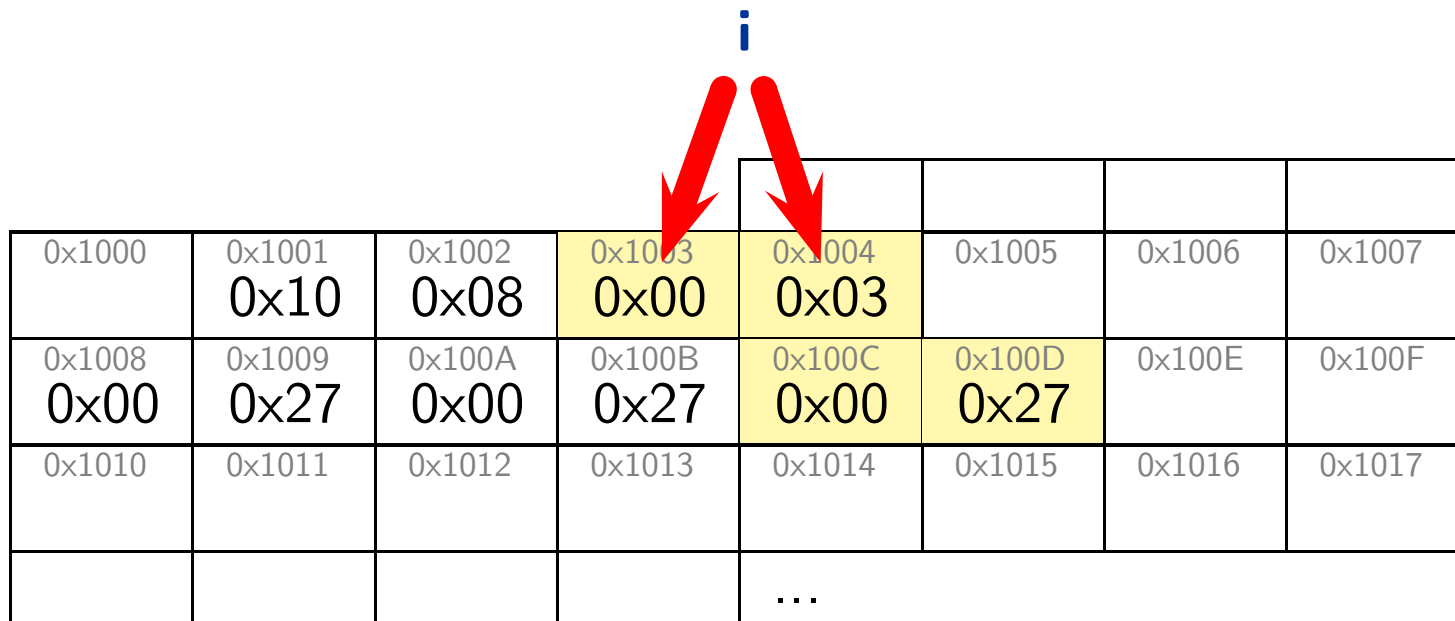
# Integer Arrays

```
1  int a[3] = { 10, 010, 0x1234 };
2  int i;
3  for (i = 0; i < 3; ++i)
4      a[i] = 0x27;
```



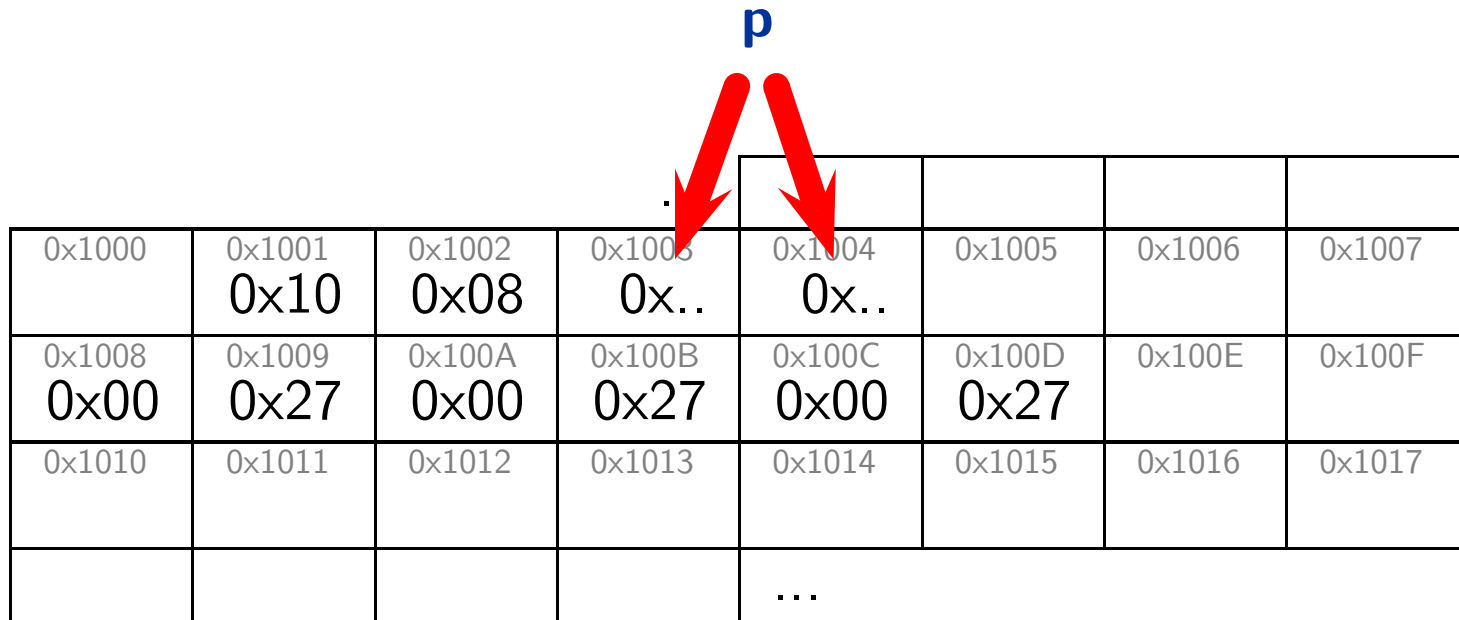
# Integer Arrays

```
1  int a[3] = { 10, 010, 0x1234 };
2  int i;
3  for (i = 0; i < 3; ++i)
4      a[i] = 0x27;
```



# Integer Arrays vs. Pointers

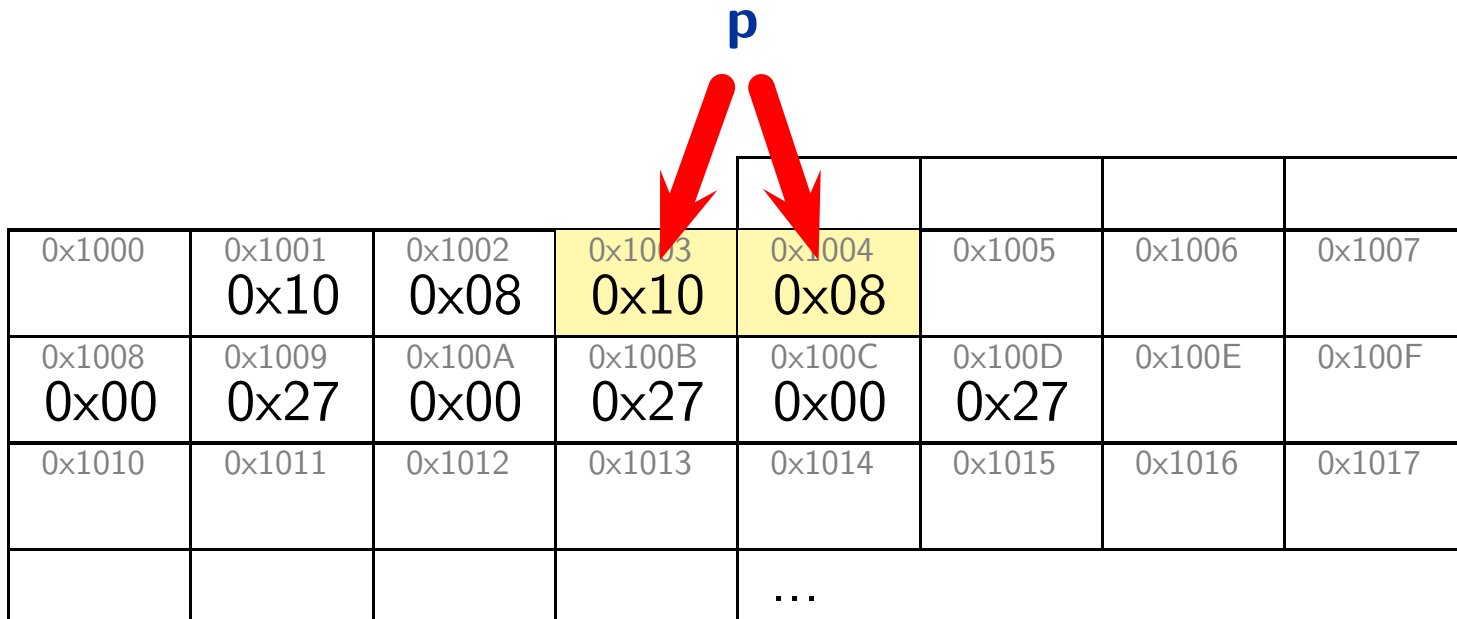
```
1  int a[3] = { 10, 010, 0x1234 };
2  int* p = a;
3  for (int i = 0; i < 3; ++i)
4      *p = 0x3421;
```





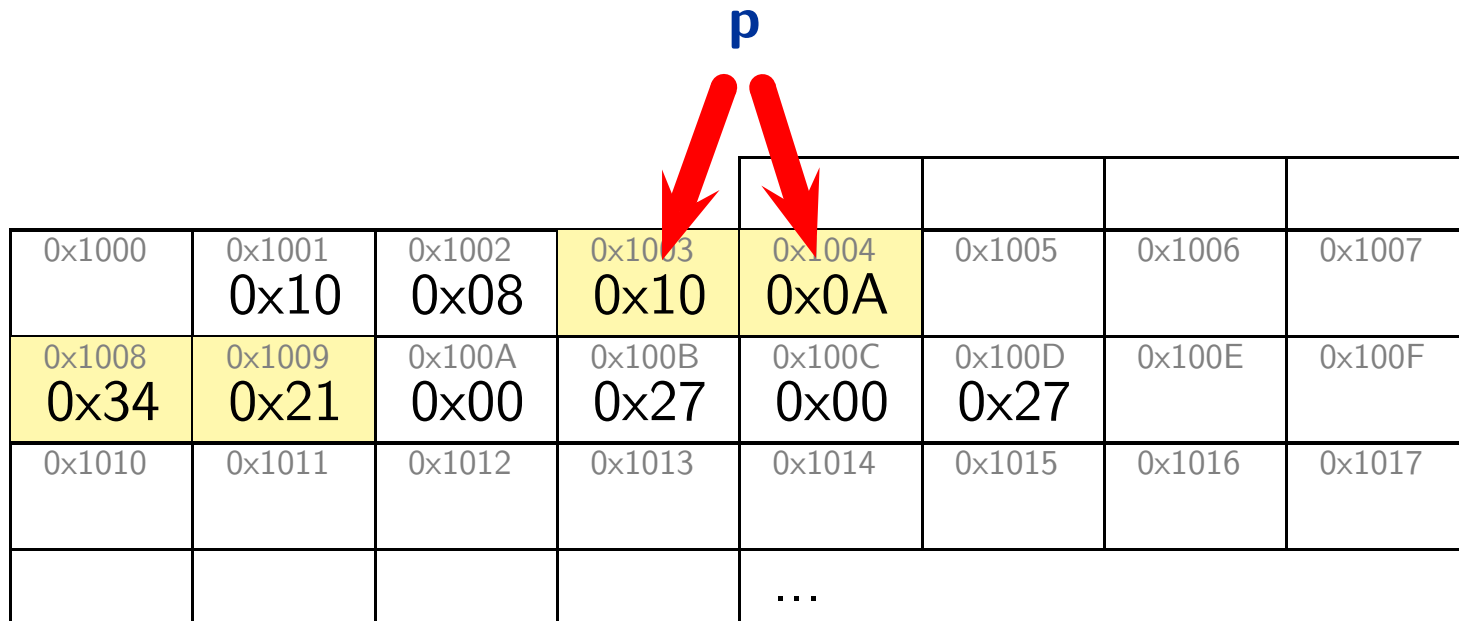
# Integer Arrays vs. Pointers

```
1  int a[3] = { 10, 010, 0x1234 };
2  int* p = a;
3  for (int i = 0; i < 3; ++p)
4      *p = 0x3421;
```



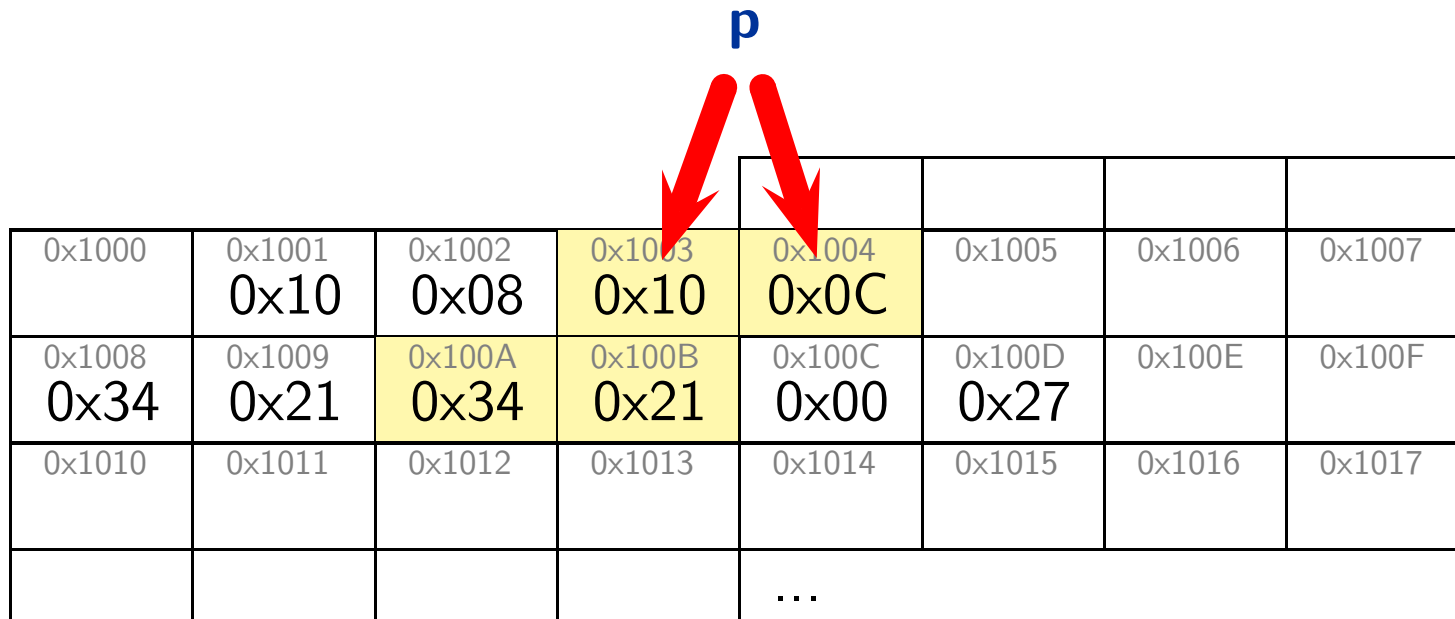
# Integer Arrays vs. Pointers

```
1  int a[3] = { 10, 010, 0x1234 };
2  int* p = a;
3  for (int i = 0; i < 3; ++i)
4  *p = 0x3421;
```



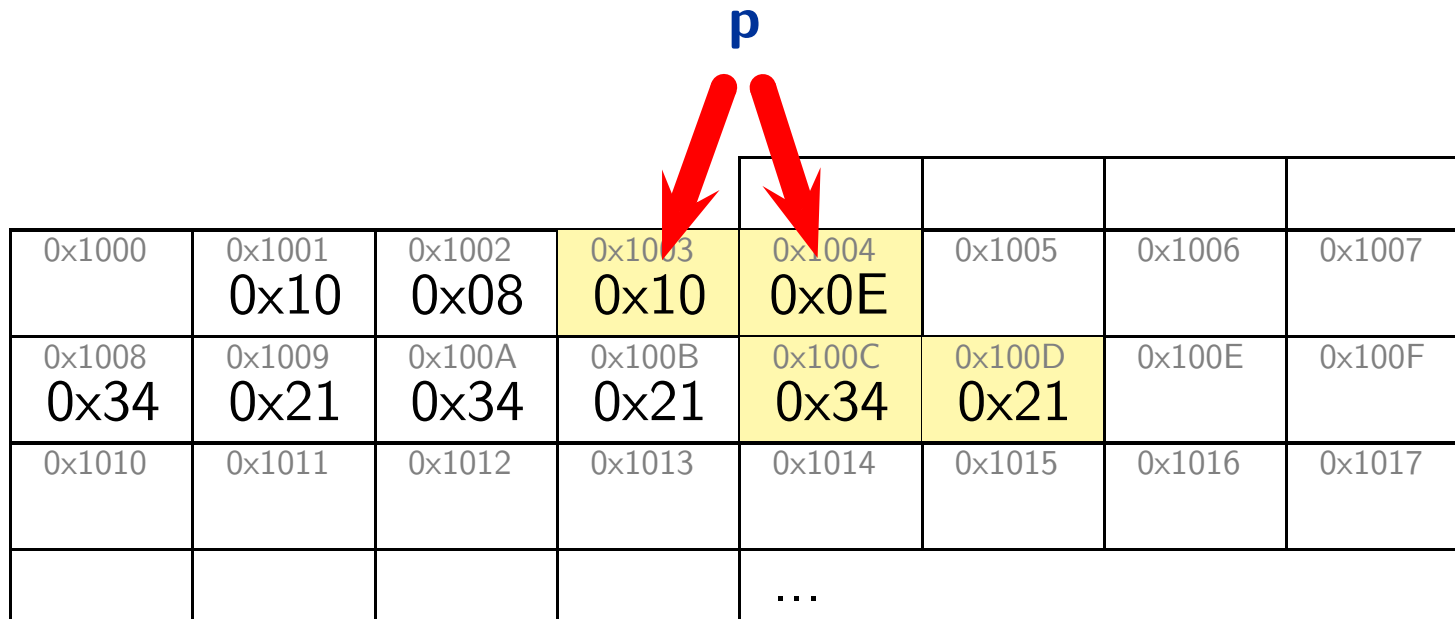
# Integer Arrays vs. Pointers

```
1  int a[3] = { 10, 010, 0x1234 };
2  int* p = a;
3  for (int i = 0; i < 3; ++p)
4      *p = 0x3421;
```



# Integer Arrays vs. Pointers

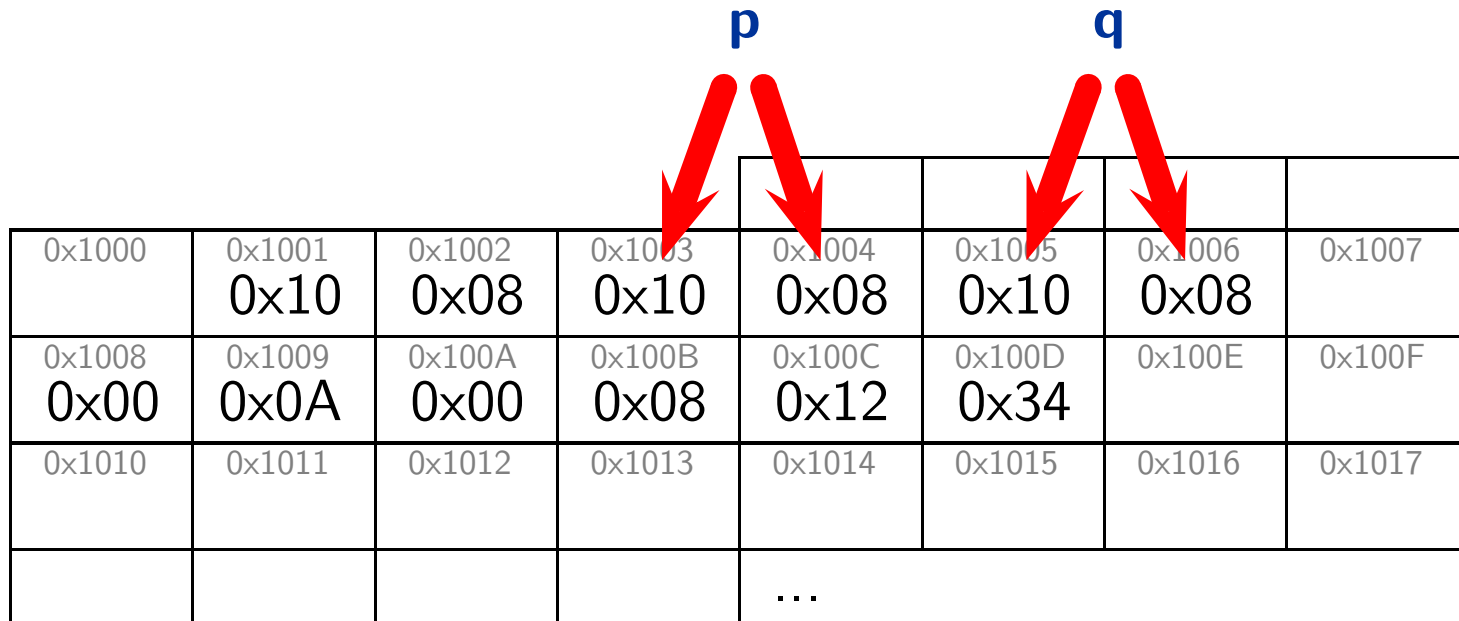
```
1  int a[3] = { 10, 010, 0x1234 };
2  int* p = a;
3  for (int i = 0; i < 3; ++p)
4      *p = 0x3421;
```



## *Pointers to 'void', Pointer Arithmetic*

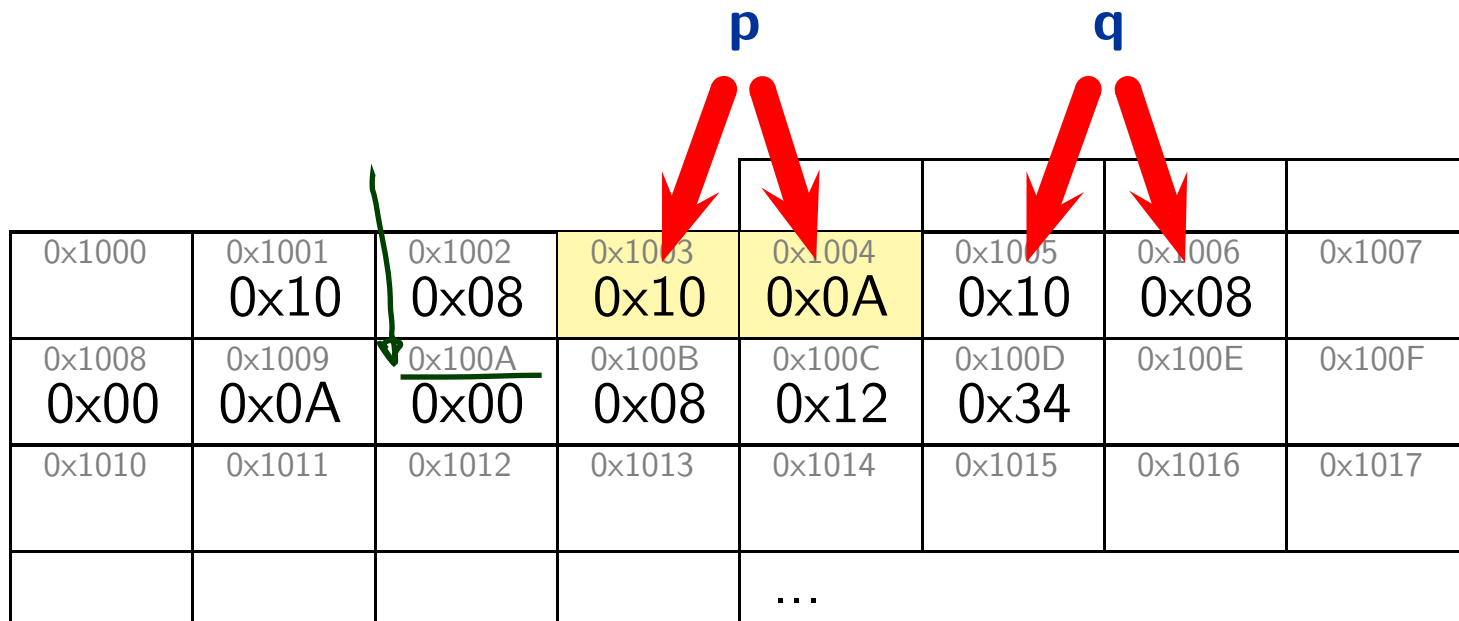
# Pointer to 'void'

```
1  int[3] a = { 10, 010, 0x1234 };
2  int* p = a;
3  void* q = a;
4  for (int i = 0; i < 3; ++i) {
5      p++;
6      q++;
7  }
```



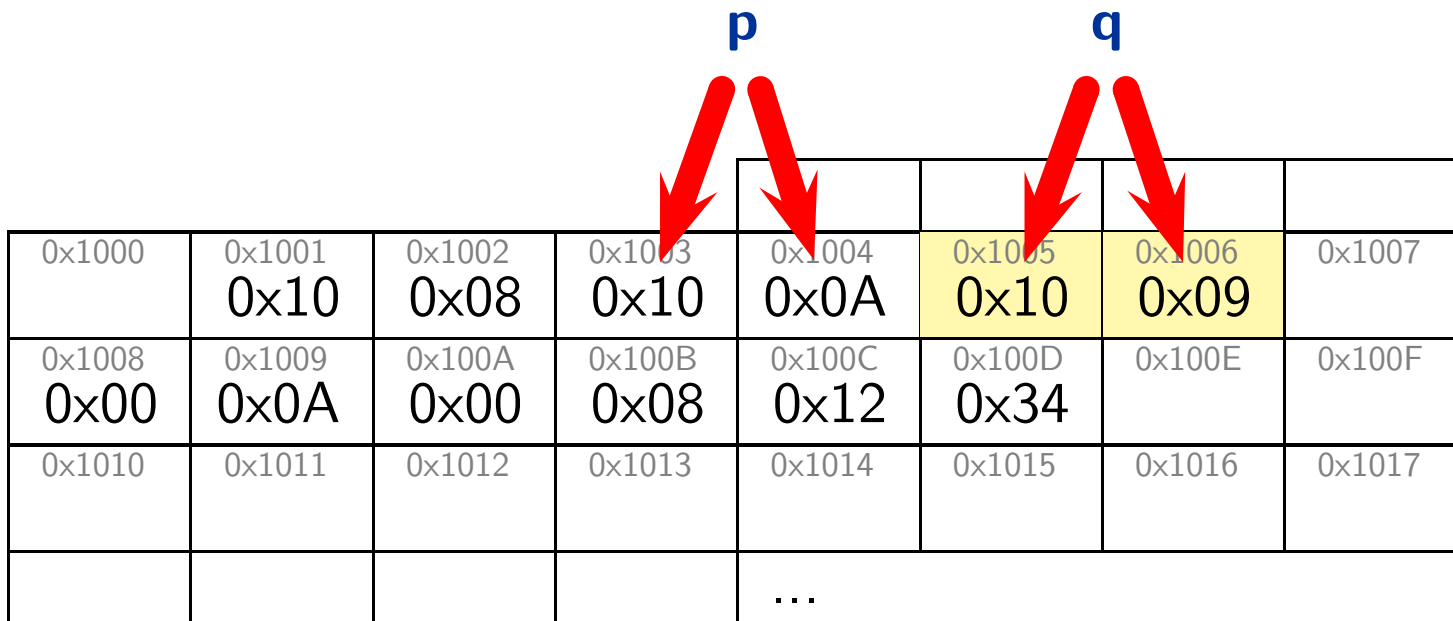
# Pointer to 'void'

```
1  int[3] a = { 10, 010, 0x1234 };
2  int* p = a;
3  void* q = a;
4  for (int i = 0; i < 3; ++i) {
5      p++;
6      q++;
7  }
```



# Pointer to 'void'

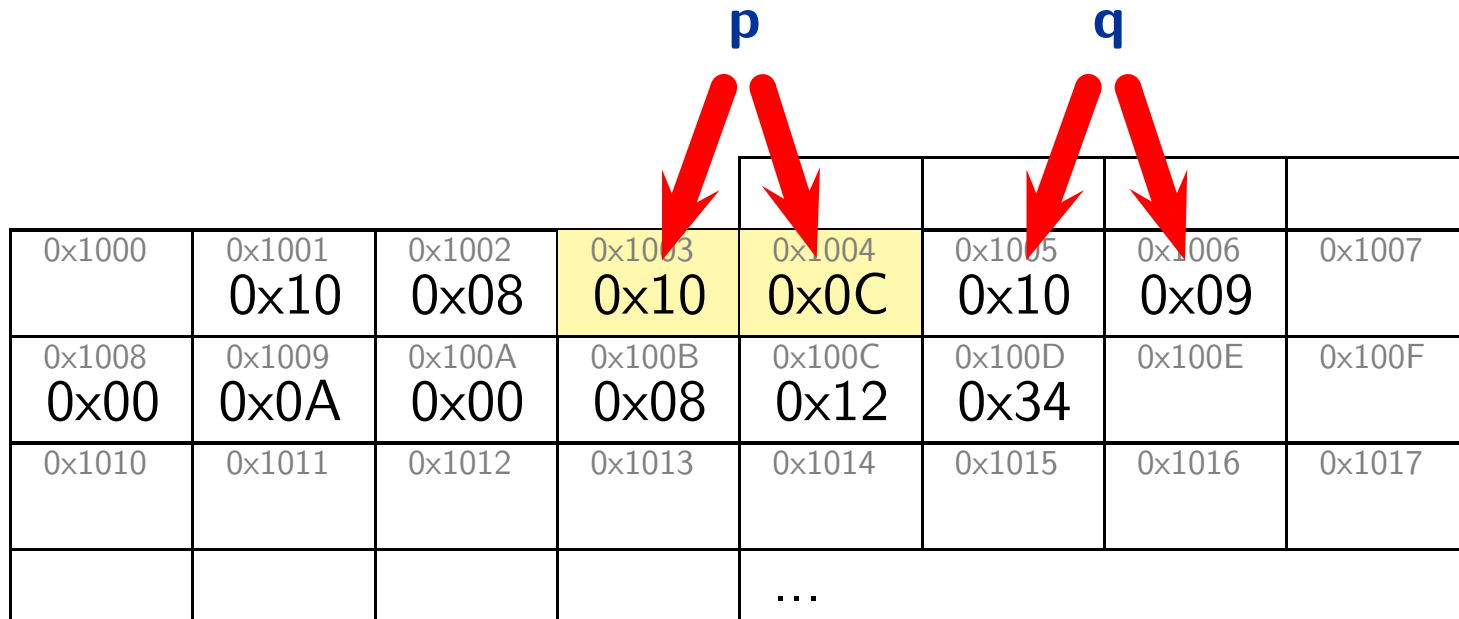
```
1  int[3] a = { 10, 010, 0x1234 };
2  int* p = a;
3  void* q = a;
4  for (int i = 0; i < 3; ++i) {
5      p++;
6      q++;
7  }
```





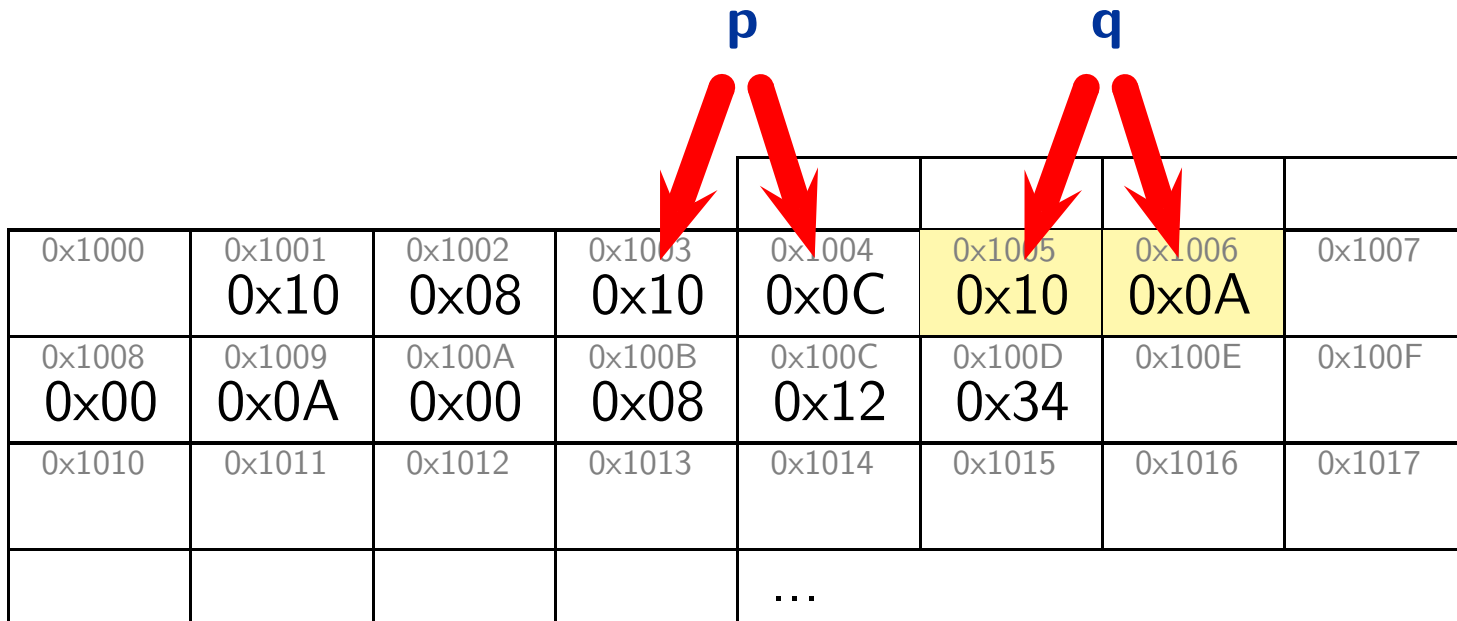
# Pointer to 'void'

```
1  int[3] a = { 10, 010, 0x1234 };
2  int* p = a;
3  void* q = a;
4  for (int i = 0; i < 3; ++i) {
5      p++;
6      q++;
7  }
```



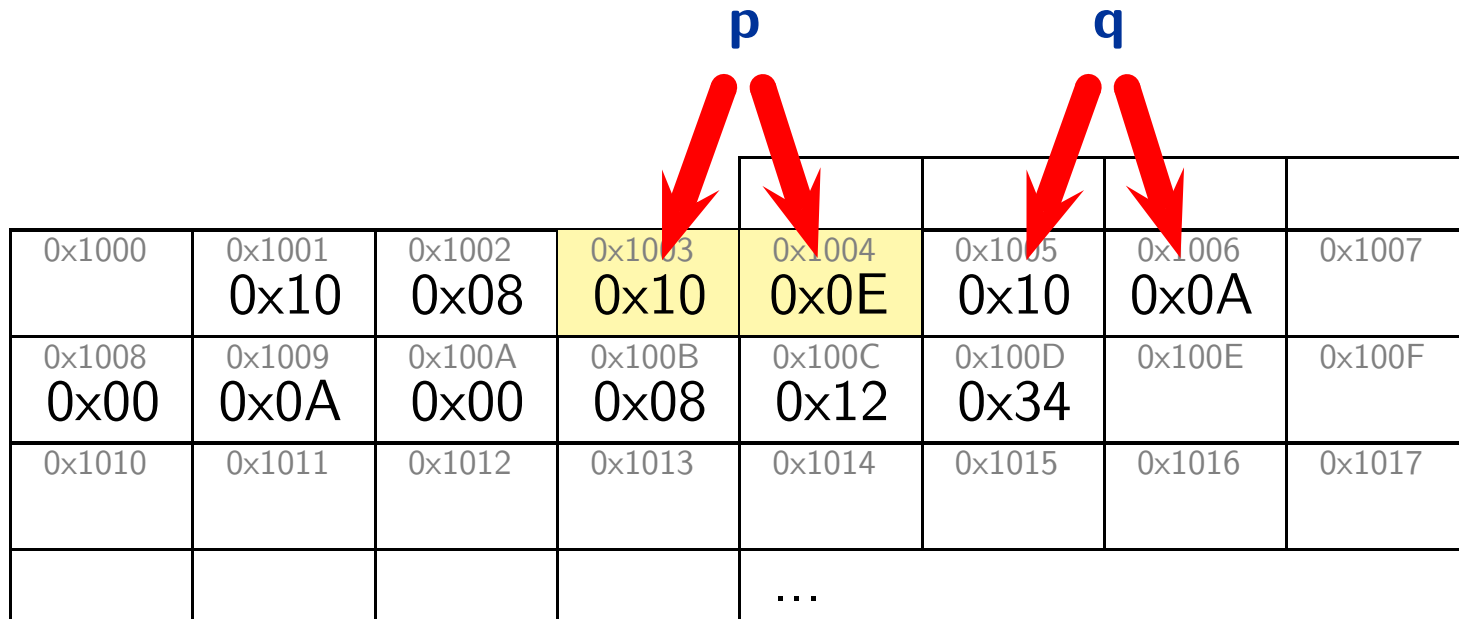
# Pointer to 'void'

```
1  int[3] a = { 10, 010, 0x1234 };
2  int* p = a;
3  void* q = a;
4  for (int i = 0; i < 3; ++i) {
5      p++;
6      q++;
7  }
```



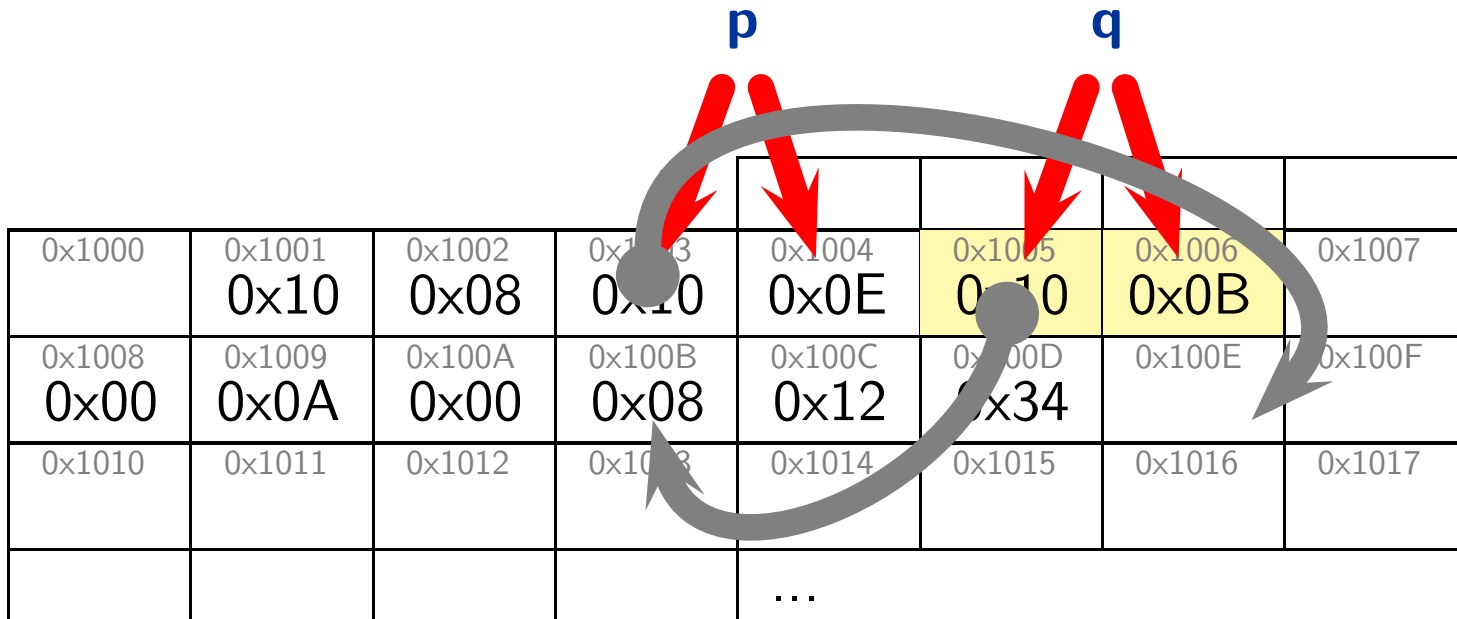
# Pointer to 'void'

```
1  int[3] a = { 10, 010, 0x1234 };
2  int* p = a;
3  void* q = a;
4  for (int i = 0; i < 3; ++i) {
5      p++;
6      q++;
7  }
```



# Pointer to 'void'

```
1  int[3] a = { 10, 010, 0x1234 };
2  int* p = a;
3  void* q = a;
4  for (int i = 0; i < 3; ++i) {
5      p++;
6      q++;
7  }
```



# Pointers: Observation

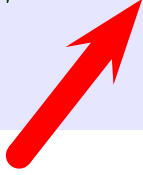
---

- A variable of pointer type just **stores an address**.
- So do variables of **array type**.
- Pointers can point to a certain type, or to **void**.
- “A pointer to void shall have the same representation and alignment requirements as a pointer to a character type.” (6.2.5.26)
- The effect of “incrementing” a pointer depends on the type pointed to.

```
1  int a[2];
2  int* p = a;
3  ++p; // points to a[1]
4  void* q = a;
5  q += sizeof(int); // points to a[1]
6  ++q; // may point into the middle
```

# Pointer Arithmetic

```
1  int[3] a = { 10, 010, 0x1234 }, i = 0;
2
3  int* p = a; // not &a !
4
5  if (a[0] == *p) i++; ✓
6  if (a[1] == *(p+1)) i++; ✓
7  if (a[2] == *(p+2)) i++; ✓
8
9  if (&(a[2]) - p == 2) i++; ✓
10
11 void* q = a;
12
13 if (a[2] == *((int*)(q + (2 * sizeof(int))))) i++; ✓
14
15 // i == 5
```

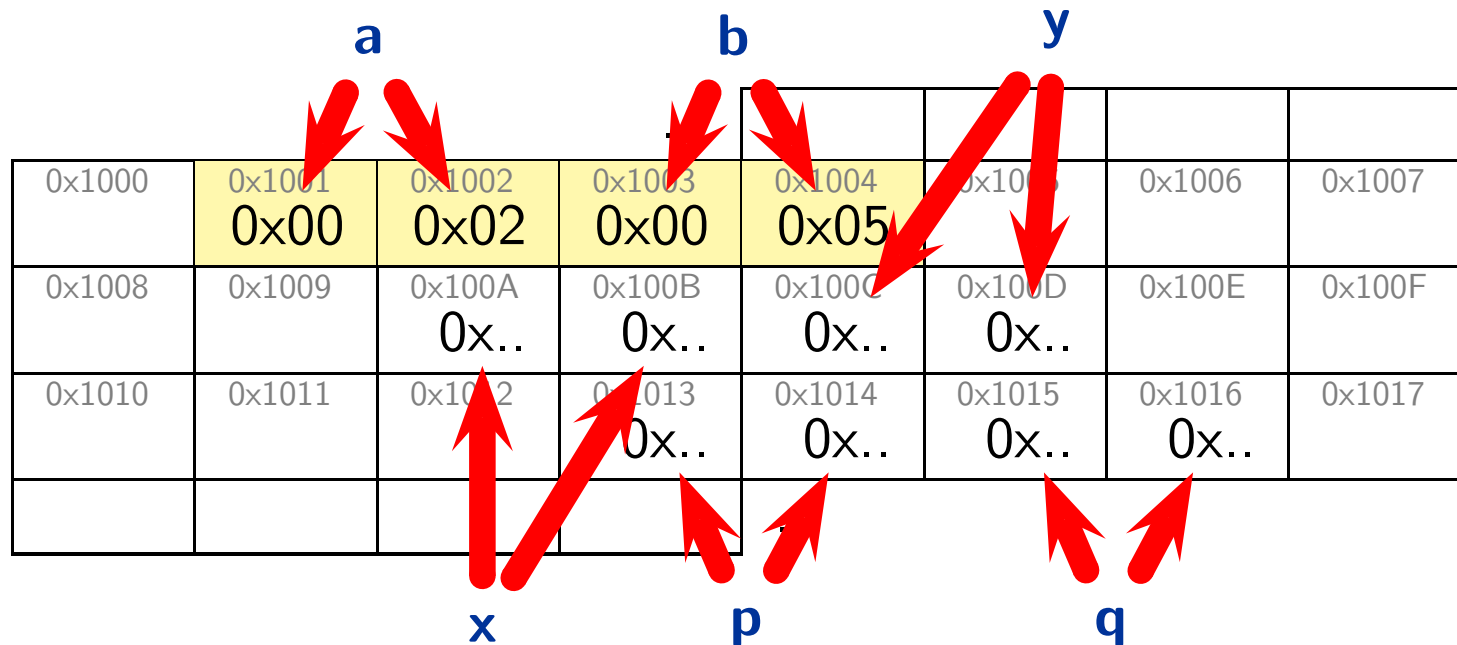


**void** as such does not have values, we need to **cast** 'q' here... note: **void\*** can be casted to everything

# *Pointers for Call By Reference*

# Call By Reference with Pointers

```
1 void f( int x, int y ) {
2   x++, y++;
3 }
4 void g( int* p, int* q ) {
5   (*p)++, (*q)++;
6 }
7 int a = 2, b = 5;
8 f( a, b );
9 g( &a, &b );
```





# Call By Reference with Pointers




```
1 void f( int x, int y ) {  
2     x++, y++;  
3 }  
4 void g( int* p, int* q ) {  
5     (*p)++, (*q)++;  
6 }  
7 int a = 2, b = 5;  
8 f( a, b );  
9 g( &a, &b );
```

				...				
0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007	
	0x00	0x02	0x00	0x05				
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F	
		0x00	0x02	0x00	0x05			
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017	
			0x..	0x..	0x..	0x..		
				...				

# Call By Reference with Pointers

```
1 void f( int x, int y ) {  
2     x++, y++;  
3 }  
4 void g( int* p, int* q ) {  
5     (*p)++, (*q)++;  
6 }  
7 int a = 2, b = 5;  
8 f( a, b );  
9 g( &a, &b );
```



				...				
0x1000	0x1001 0x00	0x1002 0x02	0x1003 0x00	0x1004 0x05	0x1005	0x1006	0x1007	
0x1008	0x1009	0x100A 0x00	0x100B 0x03	0x100C 0x00	0x100D 0x06	0x100E	0x100F	
0x1010	0x1011	0x1012	0x1013 0x..	0x1014 0x..	0x1015 0x..	0x1016 0x..	0x1017	
				...				

# Call By Reference with Pointers

```
1 void f( int x, int y ) {
2     x++, y++;
3 }
4 void g( int* p, int* q ) {
5     (*p)++, (*q)++;
6 }
7 int a = 2, b = 5;
8 f( a, b );
9 g( &a, &b );
```



			...				
0x1000	0x1001 0x00	0x1002 0x02	0x1003 0x00	0x1004 0x05	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A 0x..	0x100B 0x..	0x100C 0x..	0x100D 0x..	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 0x..	0x1014 0x..	0x1015 0x..	0x1016 0x..	0x1017
				...			

# Call By Reference with Pointers

```
1 void f( int x, int y ) {  
2     x++, y++;  
3 }  
4 void g( int* p, int* q ) {  
5     (*p)++, (*q)++;  
6 }  
7 int a = 2, b = 5;  
8 f( a, b );  
9 g( &a, &b );
```



			...				
0x1000	0x1001 0x00	0x1002 0x02	0x1003 0x00	0x1004 0x05	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A 0x..	0x100B 0x..	0x100C 0x..	0x100D 0x..	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 0x10	0x1014 0x01	0x1015 0x10	0x1016 0x03	0x1017
				...			

# Call By Reference with Pointers

```
1 void f( int x, int y ) {  
2     x++, y++;  
3 }  
4 void g( int* p, int* q ) {  
5     (*p)++, (*q)++;  
6 }  
7 int a = 2, b = 5;  
8 f( a, b );  
9 g( &a, &b );
```



			...				
0x1000	0x1001 0x00	0x1002 0x03	0x1003 0x00	0x1004 0x06	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A 0x..	0x100B 0x..	0x100C 0x..	0x100D 0x..	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 0x10	0x1014 0x01	0x1015 0x10	0x1016 0x03	0x1017
				...			

# Call By Reference with Pointers

```
1 void f( int x, int y ) {
2     x++, y++;
3 }
4 void g( int* p, int* q ) {
5     (*p)++, (*q)++;
6 }
7 int a = 2, b = 5;
8 f( a, b );
9 g( &a, &b );
```



				...				
0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007	
	0x00	0x03	0x00	0x06				
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F	
		0x..	0x..	0x..	0x..			
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017	
			0x..	0x..	0x..	0x..		
				...				

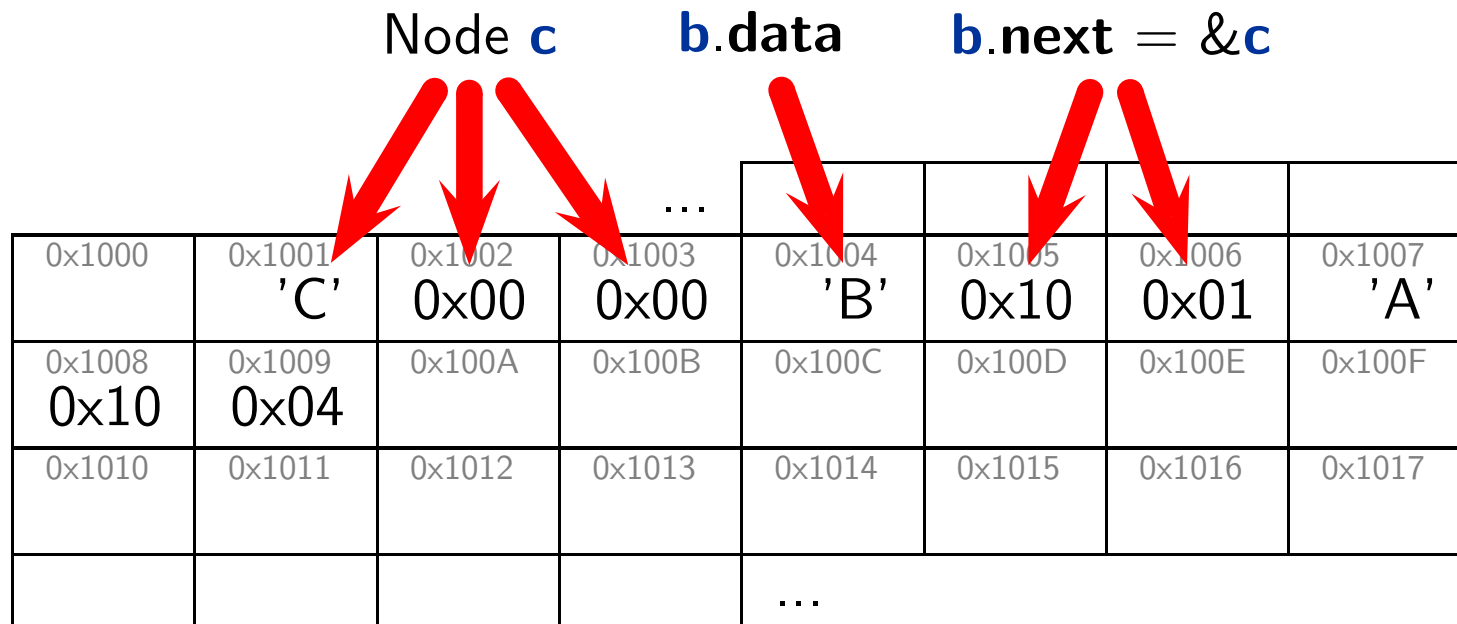
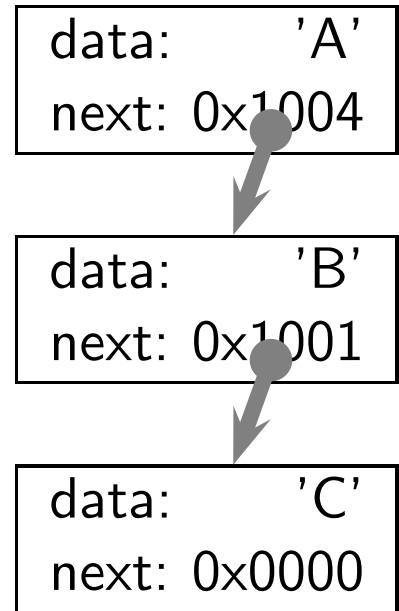
# *Dynamic Storage & Storage Duration*

# *Dynamic Storage Allocation*



# A Linked List

```
1 typedef struct Node {
2     char data;
3     struct Node* next;
4 } Node;
5
6 Node c = { 'C', 0 };
7 Node b = { 'B', &c };
8 Node a = { 'A', &b };
```



# Dynamic Storage Allocation

```
1 typedef struct Node {
2     char data; struct Node* next; } Node;
3
4 Node *head = 0, *hlp;
5
6 void insert( char d ) {
7     hlp = (Node*) malloc( sizeof(Node) );
8     hlp->data = d;
9     hlp->next = head;
10    head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```

*"mem. allocate"*

allocate some space for a **Node**, return its address;  
may fail ("out of memory"),  
*malloc(3)* yields 0 then

*was page reference section*

...							
0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017
				...			

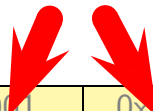
# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```



**head**

0x1000	0x1001	0x1002	...	0x1004	0x1005	0x1006	0x1007
	0x00	0x00					
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017
				...			



# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```



			...				
0x1000	0x1001 <b>0x00</b>	0x1002 <b>0x00</b>	0x1003	0x1004	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017
				...			

# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```



hlp

0x1000	0x1001 0x00	0x1002 0x00	0x1003 0x10	0x1004 0x13	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 '..'	0x1014 0x..	0x1015 0x..	0x1016	0x1017
				...			

# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```



			...				
0x1000	0x1001 <b>0x00</b>	0x1002 <b>0x00</b>	0x1003 <b>0x10</b>	0x1004 <b>0x13</b>	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x..</b>	0x1015 <b>0x..</b>	0x1016	0x1017
				...			

# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```



			...				
0x1000	0x1001 <b>0x00</b>	0x1002 <b>0x00</b>	0x1003 <b>0x10</b>	0x1004 <b>0x13</b>	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	<b>0x1014 0x00</b>	<b>0x1015 0x00</b>	0x1016	0x1017
				...			

# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```



			...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x13</b>	0x1003 <b>0x10</b>	0x1004 <b>0x13</b>	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 'C'	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017
				...			



# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```

data: 'C'
next: 0x0000

			...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x13</b>	0x1003 <b>0x10</b>	0x1004 <b>0x13</b>	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 'C'	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017
				...			



# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```



data: 'C'
next: 0x0000

			...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x13</b>	0x1003 <b>0x10</b>	0x1004 <b>0x13</b>	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 'C'	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017
				...			

# Dynamic Storage Allocation

```

1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );

```



data:	'C'
next:	0x0000

				...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x13</b>	0x1003 <b>0x10</b>	0x1004 <b>0x08</b>	0x1005	0x1006	0x1007	
0x1008 '..'	0x1009 <b>0x..</b>	0x100A <b>0x..</b>	0x100B	0x100C	0x100D	0x100E	0x100F	
0x1010	0x1011	0x1012	0x1013 'C'	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017	
				...				

# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```



data: 'C'
next: 0x0000

			...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x13</b>	0x1003 <b>0x10</b>	0x1004 <b>0x08</b>	0x1005	0x1006	0x1007
0x1008 <b>'B'</b>	0x1009 <b>0x..</b>	0x100A <b>0x..</b>	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017
				...			

# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```



data: 'C'
next: 0x0000

			...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x13</b>	0x1003 <b>0x10</b>	0x1004 <b>0x08</b>	0x1005	0x1006	0x1007
0x1008 <b>'B'</b>	0x1009 <b>0x10</b>	0x100A <b>0x13</b>	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017
				...			

# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```

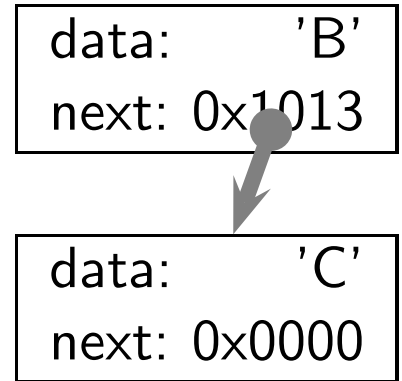


data: 'C'
next: 0x0000

			...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x08</b>	0x1003 <b>0x10</b>	0x1004 <b>0x08</b>	0x1005	0x1006	0x1007
0x1008 <b>'B'</b>	0x1009 <b>0x10</b>	0x100A <b>0x13</b>	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017
				...			

# Dynamic Storage Allocation

```
1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );
```



0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017
				...			

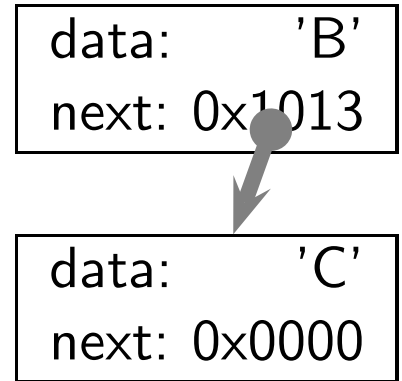
Memory layout showing nodes for 'B' and 'C'. Node 'B' is at 0x1009 with next pointer 0x1013. Node 'C' is at 0x1013 with next pointer 0x0000.

# Dynamic Storage Allocation

```

1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );

```



			...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x08</b>	0x1003 <b>0x10</b>	0x1004 <b>0x08</b>	0x1005	0x1006	0x1007
0x1008 <b>'B'</b>	0x1009 <b>0x10</b>	0x100A <b>0x13</b>	0x100B	0x100C	0x100D	0x100E	0x100F
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017
				...			

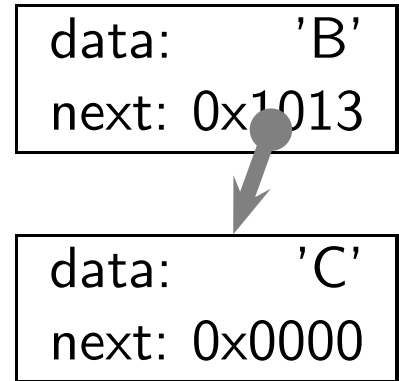


# Dynamic Storage Allocation

```

1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );

```



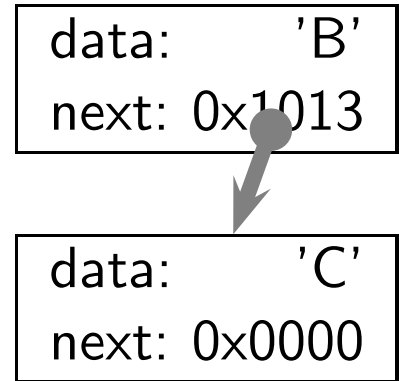
			...				
0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
	0x10	0x08	0x10	0x0D			
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
'B'	0x10	0x13			'..'	0x..	0x..
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017
			'C'	0x00	0x00		
				...			

# Dynamic Storage Allocation

```

1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );

```



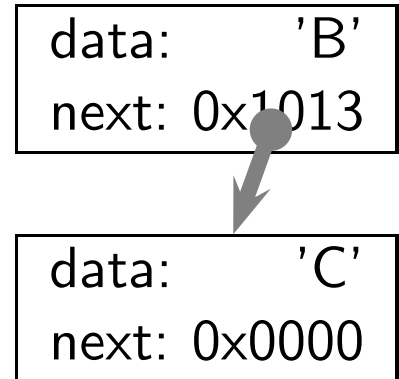
			...				
0x1000	0x1001 0x10	0x1002 0x08	0x1003 0x10	0x1004 0x0D	0x1005	0x1006	0x1007
0x1008 'B'	0x1009 0x10	0x100A 0x13	0x100B	0x100C	0x100D 'A'	0x100E 0x..	0x100F 0x..
0x1010	0x1011	0x1012	0x1013 'C'	0x1014 0x00	0x1015 0x00	0x1016	0x1017
				...			

# Dynamic Storage Allocation

```

1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );

```



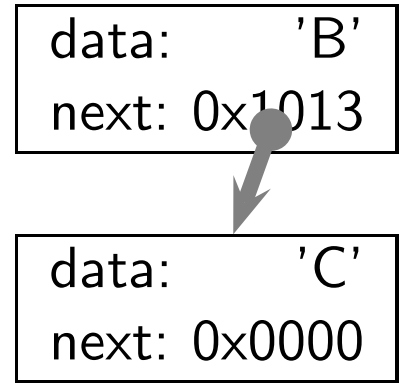
			...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x08</b>	0x1003 <b>0x10</b>	0x1004 <b>0x0D</b>	0x1005	0x1006	0x1007
0x1008 <b>'B'</b>	0x1009 <b>0x10</b>	0x100A <b>0x13</b>	0x100B	0x100C	0x100D <b>'A'</b>	0x100E <b>0x10</b>	0x100F <b>0x08</b>
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017
				...			

# Dynamic Storage Allocation

```

1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*)malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );

```



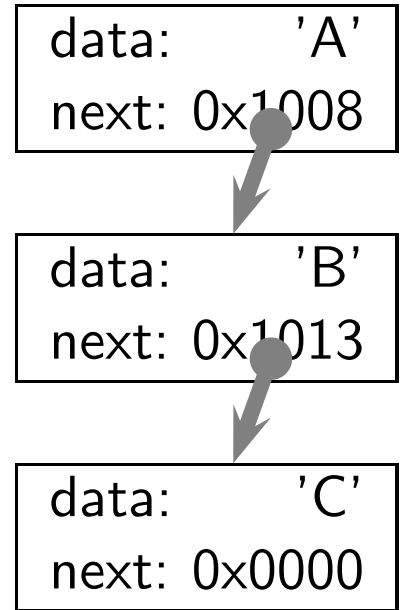
			...				
0x1000	0x1001 0x10	0x1002 0x0D	0x1003 0x10	0x1004 0x0D	0x1005	0x1006	0x1007
0x1008 'B'	0x1009 0x10	0x100A 0x13	0x100B	0x100C	0x100D 'A'	0x100E 0x10	0x100F 0x08
0x1010	0x1011	0x1012	0x1013 'C'	0x1014 0x00	0x1015 0x00	0x1016	0x1017
				...			

# Dynamic Storage Allocation

```

1  typedef struct Node {
2      char data; struct Node* next; } Node;
3
4  Node *head = 0, *hlp;
5
6  void insert( char d ) {
7      hlp = (Node*) malloc( sizeof(Node) );
8      hlp->data = d;
9      hlp->next = head;
10     head = hlp;
11 }
12
13 insert( 'C' );
14 insert( 'B' );
15 insert( 'A' );

```



			...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x0D</b>	0x1003 <b>0x10</b>	0x1004 <b>0x0D</b>	0x1005	0x1006	0x1007
0x1008 <b>'B'</b>	0x1009 <b>0x10</b>	0x100A <b>0x13</b>	0x100B	0x100C	0x100D <b>'A'</b>	0x100E <b>0x10</b>	0x100F <b>0x08</b>
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017
				...			

# Dynamic Storage Management

---

Dynamic Storage Allocation:

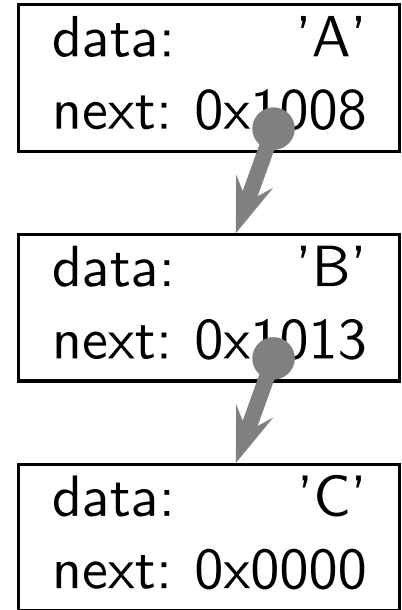
- void\* **malloc**( **size\_t** size );  
“[...] allocates **size** bytes and returns a pointer to the allocated memory.  
**The memory is not initialized.** [...]”
- “On error, [this function] returns NULL.”
- void **free**( void\* ptr )  
“[...] frees the memory space pointed to by **ptr**, which **must** have been returned by a previous call to malloc(), [...].”
- “Otherwise, or if **free(ptr)** has already been called before, **undefined behavior** occurs.”
- “If **ptr** is **NULL**, no operation is performed.”
- **No garbage collection!**  
Management of dynamic storage is **responsibility of the programmer**.  
Unaccessible, not free'd memory is called **memory leak**.

# Dynamic Storage Management Example

```

1 void remove() {
2     if (hlp = head) {
3         head = hlp->next;
4         free(hlp);
5     }
6 }
7 insert( 'C' ); insert( 'B' ); insert( 'A' );
8 remove();
9 insert( 'X' );

```



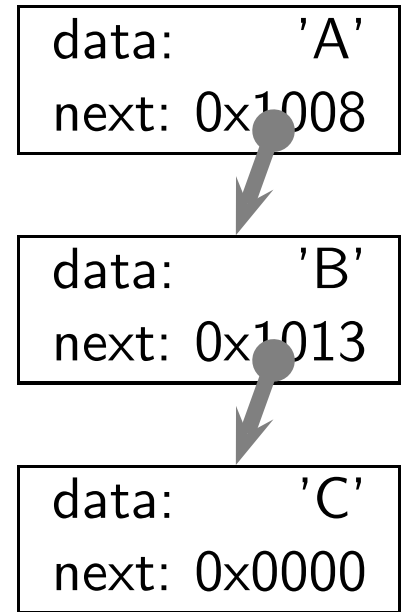
				...				
0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007	
	<b>0x10</b>	<b>0x0D</b>	<b>0x10</b>	<b>0x0D</b>				
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F	
<b>'B'</b>	<b>0x10</b>	<b>0x13</b>			<b>'A'</b>	<b>0x10</b>	<b>0x08</b>	
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017	
			<b>'C'</b>	<b>0x00</b>	<b>0x00</b>			
				...				

# Dynamic Storage Management Example

```

1 void remove() {
2     if (hlp = head) {
3         head = hlp->next;
4         free(hlp);
5     }
6 }
7 insert( 'C' ); insert( 'B' ); insert( 'A' );
8 remove();
9 insert( 'X' );

```



				...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x0D</b>	<b>0x1003</b> <b>0x10</b>	<b>0x1004</b> <b>0x0D</b>	0x1005	0x1006	0x1007	
0x1008 <b>'B'</b>	0x1009 <b>0x10</b>	0x100A <b>0x13</b>	0x100B	0x100C	0x100D <b>'A'</b>	0x100E <b>0x10</b>	0x100F <b>0x08</b>	
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017	
				...				

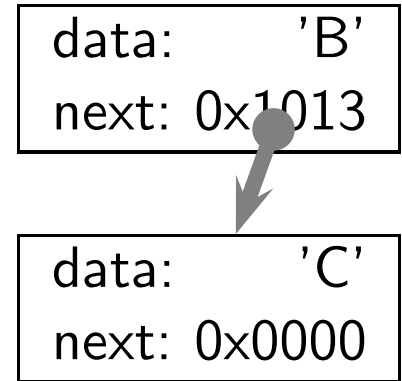


# Dynamic Storage Management Example

```

1 void remove() {
2     if (hlp = head) {
3         head = hlp->next;
4         free(hlp);
5     }
6 }
7 insert( 'C' ); insert( 'B' ); insert( 'A' );
8 remove();
9 insert( 'X' );

```



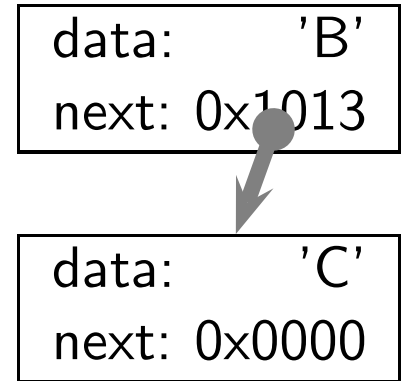
				...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x08</b>	0x1003 <b>0x10</b>	0x1004 <b>0x0D</b>	0x1005	0x1006	0x1007	
0x1008 <b>'B'</b>	0x1009 <b>0x10</b>	0x100A <b>0x13</b>	0x100B	0x100C	0x100D <b>'A'</b>	0x100E <b>0x10</b>	0x100F <b>0x08</b>	
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017	
				...				

# Dynamic Storage Management Example

```

1 void remove() {
2     if (hlp = head) {
3         head = hlp->next;
4         free(hlp);
5     }
6 }
7 insert( 'C' ); insert( 'B' ); insert( 'A' );
8 remove();
9 insert( 'X' );

```



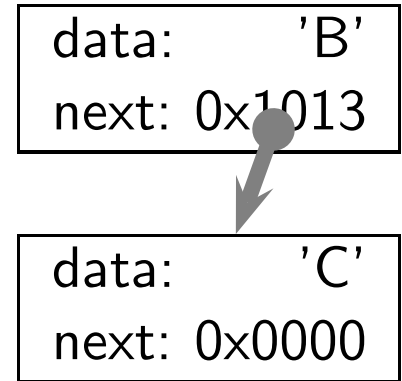
				...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x08</b>	0x1003 <b>0x10</b>	0x1004 <b>0x0D</b>	0x1005	0x1006	0x1007	
0x1008 <b>'B'</b>	0x1009 <b>0x10</b>	0x100A <b>0x13</b>	0x100B	0x100C	0x100D <b>'A'</b>	0x100E <b>0x10</b>	0x100F <b>0x08</b>	
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017	
				...				

# Dynamic Storage Management Example

```

1 void remove() {
2     if (hlp = head) {
3         head = hlp->next;
4         free(hlp);
5     }
6 }
7 insert( 'C' ); insert( 'B' ); insert( 'A' );
8 remove();
9 insert( 'X' );

```



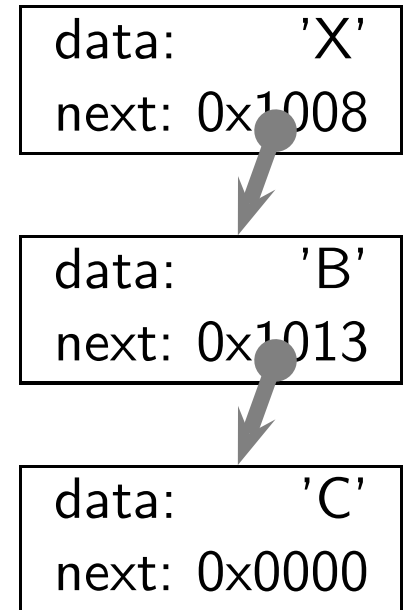
...							
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x08</b>	0x1003 <b>0x10</b>	0x1004 <b>0x0D</b>	0x1005	0x1006	0x1007
0x1008 <b>'B'</b>	0x1009 <b>0x10</b>	0x100A <b>0x13</b>	0x100B	0x100C	0x100D <b>'A'</b>	0x100E <b>0x10</b>	0x100F <b>0x08</b>
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017
				...			

# Dynamic Storage Management Example

```

1 void remove() {
2     if (hlp = head) {
3         head = hlp->next;
4         free(hlp);
5     }
6 }
7 insert( 'C' ); insert( 'B' ); insert( 'A' );
8 remove();
9 insert( 'X' );

```



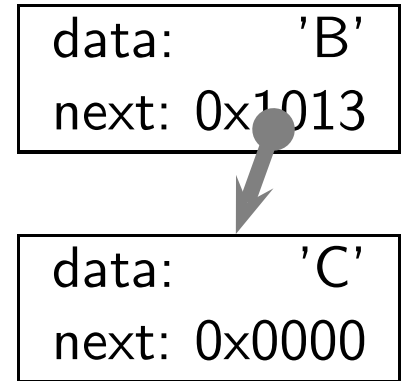
0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
	0x10	0x0D	0x10	0x0D			
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E	0x100F
'B'	0x10	0x13			'X'	0x10	0x08
0x1010	0x1011	0x1012	0x1013	0x1014	0x1015	0x1016	0x1017
			'C'	0x00	0x00		
				...			

# Dynamic Storage Management Example

```

1 void remove() {
2     if (hlp = head) {
3         head = hlp->next;
4         free(hlp);
5     }
6 }
7 insert( 'C' ); insert( 'B' ); insert( 'A' );
8 remove();
9 insert( 'X' );

```



				...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x08</b>	0x1003 <b>0x10</b>	0x1004 <b>0x0D</b>	0x1005	0x1006	0x1007	
0x1008 <b>'B'</b>	0x1009 <b>0x10</b>	0x100A <b>0x13</b>	0x100B	0x100C	0x100D <b>'A'</b>	0x100E <b>0x10</b>	0x100F <b>0x08</b>	
0x1010 <b>'X'</b>	0x1011 <b>0x10</b>	0x1012 <b>0x08</b>	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017	
				...				

# Dynamic Linked List Iteration

```
1 Node* find( char d ) {
2   hlp = head;
3   while (hlp) {
4     if (hlp->data == d)
5       break;
6     hlp = hlp->next;
7   }
8   return hlp;
9 }
10 insert( 'C' ); insert( 'B' ); insert( 'A' );
11 find( 'B' ); // yields 0x1008
12 find( 'O' ); // yields 0x0000, aka. NULL
```

				...				
0x1000	0x1001 <b>0x10</b>	0x1002 <b>0x0D</b>	0x1003 <b>0x10</b>	0x1004 <b>0x0D</b>	0x1005	0x1006	0x1007	
0x1008 <b>'B'</b>	0x1009 <b>0x10</b>	0x100A <b>0x13</b>	0x100B	0x100C	0x100D <b>'A'</b>	0x100E <b>0x10</b>	0x100F <b>0x08</b>	
0x1010	0x1011	0x1012	0x1013 <b>'C'</b>	0x1014 <b>0x00</b>	0x1015 <b>0x00</b>	0x1016	0x1017	
				...				

# Pointers to Struct/Union — ‘.’ vs. ‘->’

```
1  typedef struct {
2      int x;
3      int y;
4  } coordinate;
5
6  coordinate pos = { 13, 27 };
7
8  coordinate* p = &pos;
9
10 int tmp;  tmp = pos.x;
11
12 tmp = (*p).x;
13 (*p).x = (*p).y;
14 (*p).y = tmp;
15
16 tmp = p->x;
17 p->x = p->y;
18 p->y = tmp;
```

## *Storage Duration of Objects*



# Storage Duration of Objects (6.2.4)

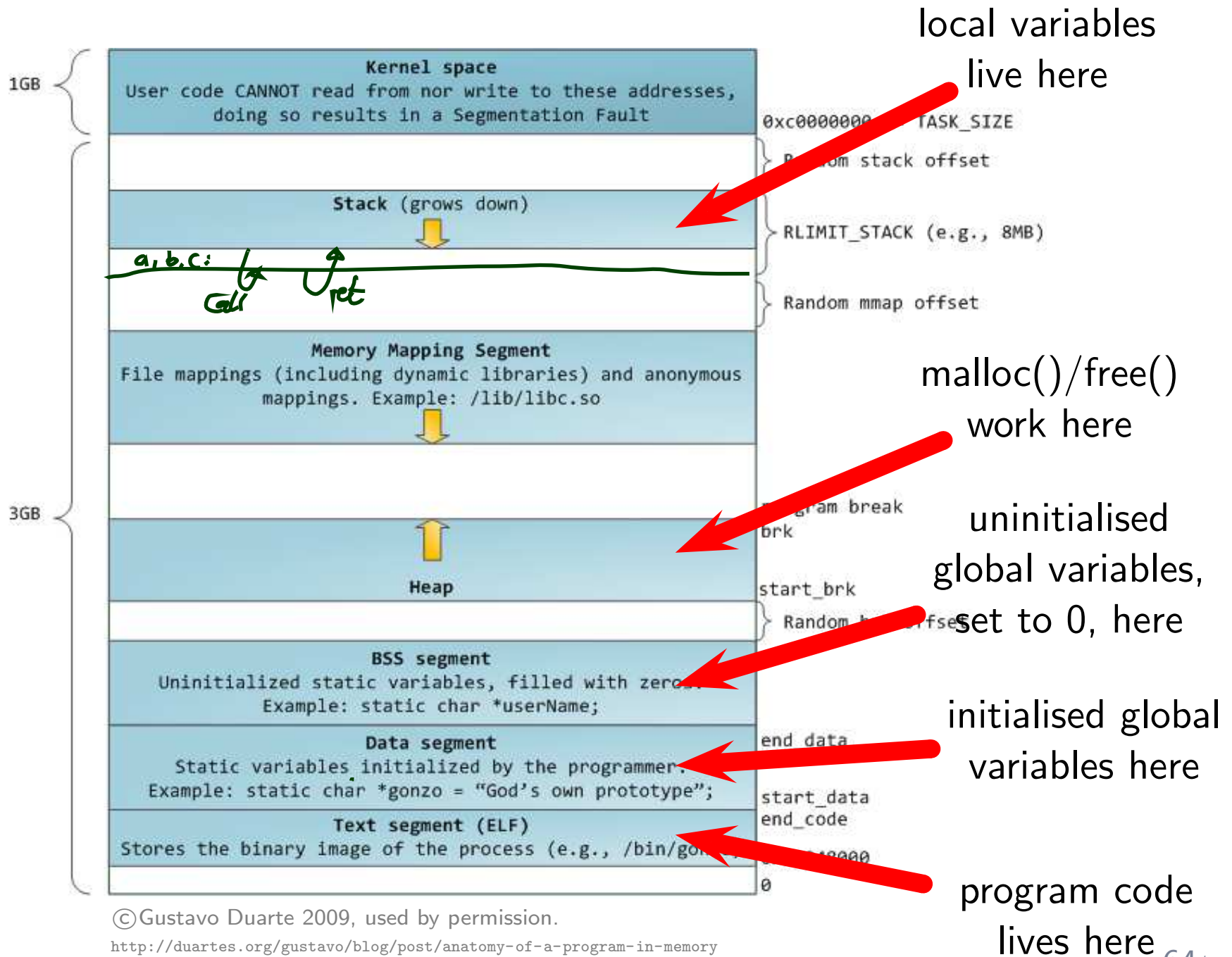
---

- **“static”** – e.g. variables in program scope:
  - live from program start to end
  - if not explicitly initialized, set to 0. (6.7.8)
- **“automatic”** – non-static variables in local scope:
  - live from block entry to exit
  - not automatically initialised: “initial value [...] is indeterminate”
- **“allocated”** – dynamic objects:
  - live from **malloc** to **free**
  - not automatically initialised

“If an object is referred to outside of its lifetime, **the behavior is undefined.**

The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.”

# Example: Anatomy of a Linux Program in Memory



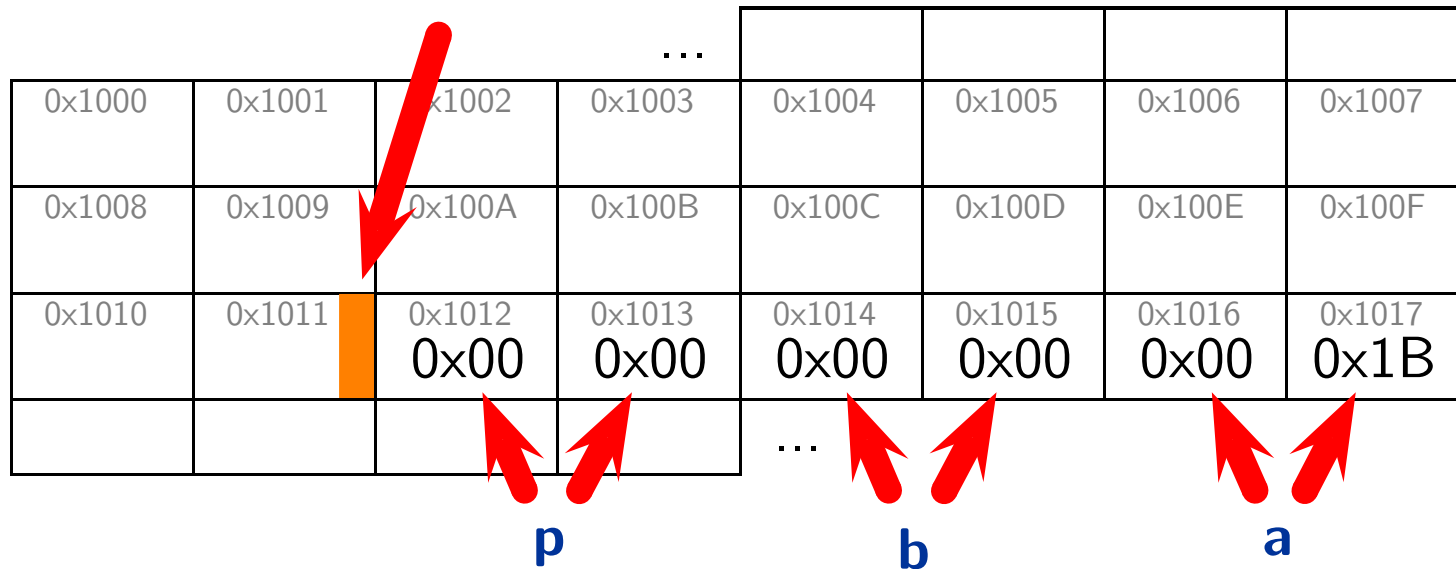
©Gustavo Duarte 2009, used by permission.

<http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory>

# Storage Duration “Automatic” (Simplified)

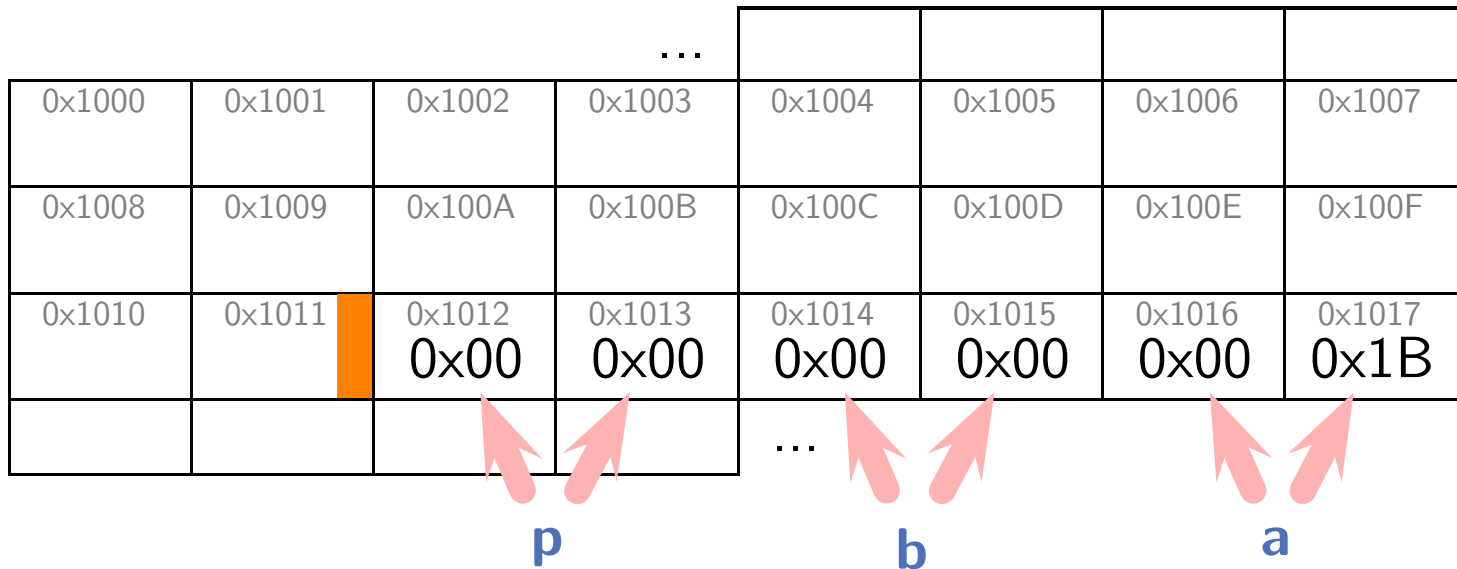
```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```

stack pointer – stack ends at  
0x1012 in this case; stack grows  
downwards (to smaller addr.)



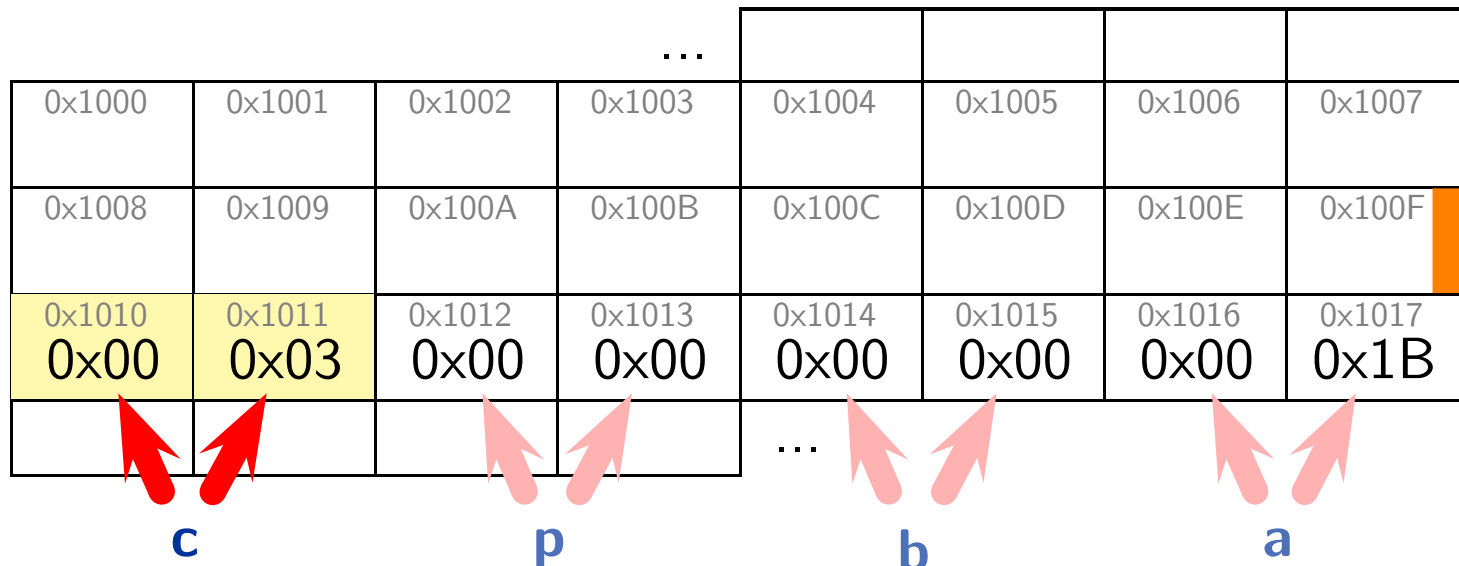
# Storage Duration “Automatic” (Simplified)

```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```



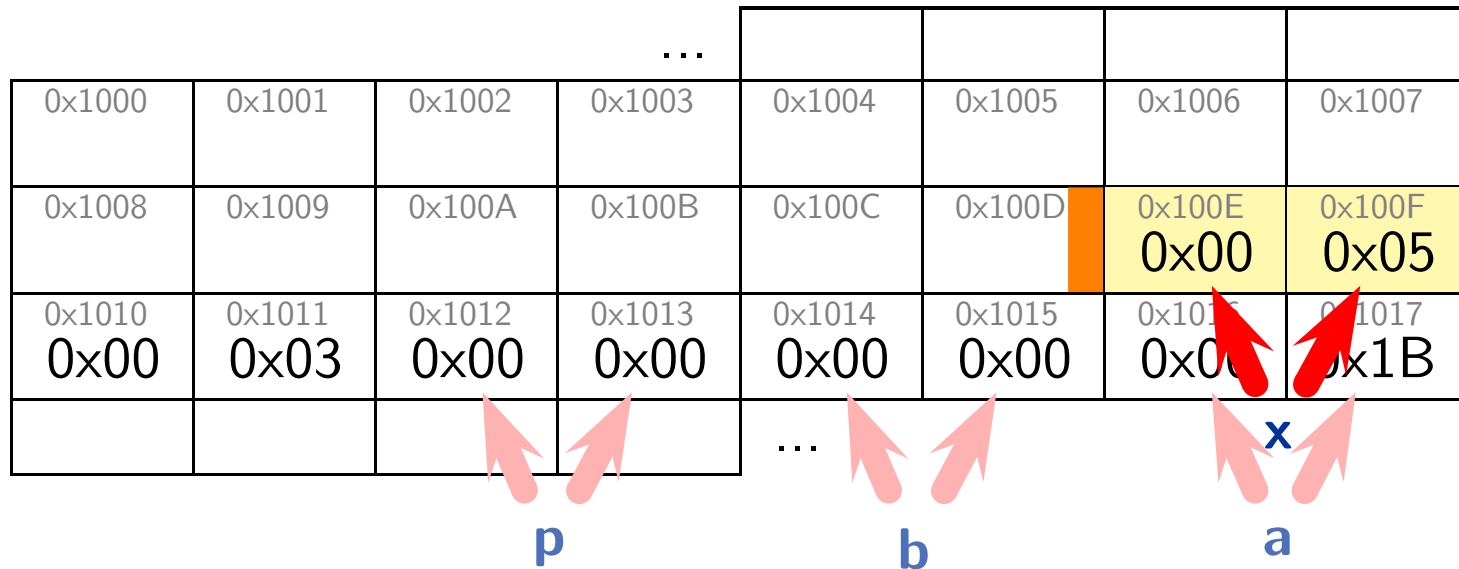
# Storage Duration “Automatic” (Simplified)

```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```



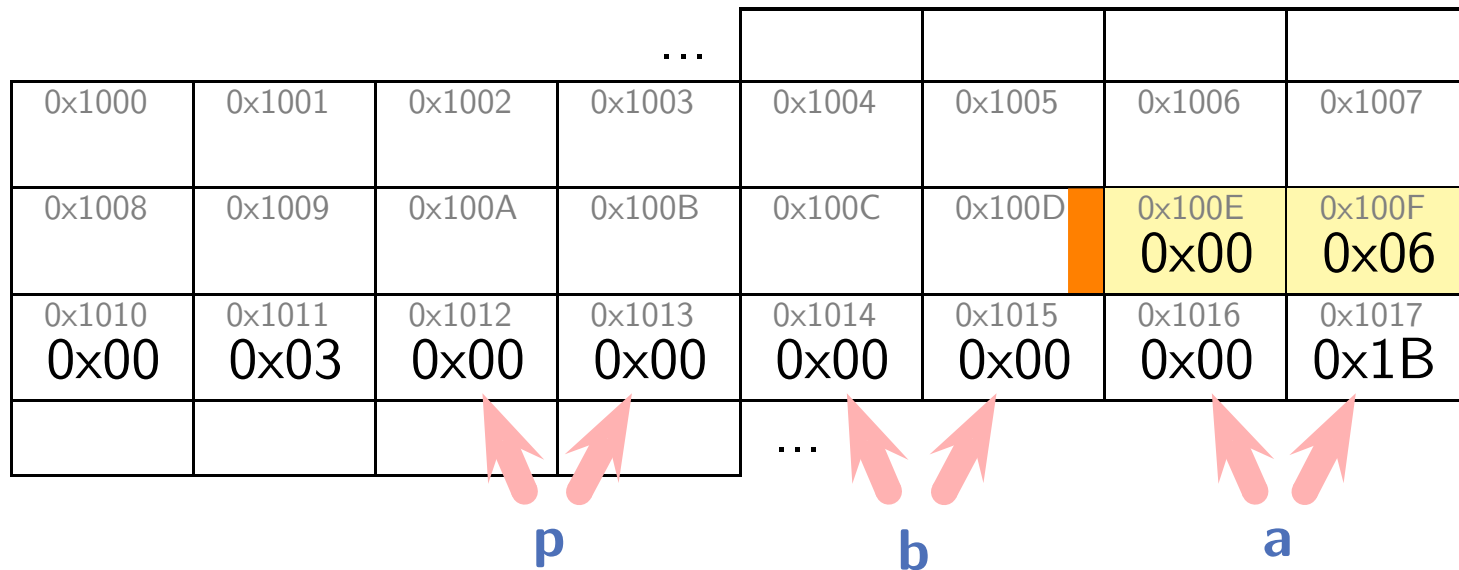
# Storage Duration “Automatic” (Simplified)

```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```



# Storage Duration “Automatic” (Simplified)

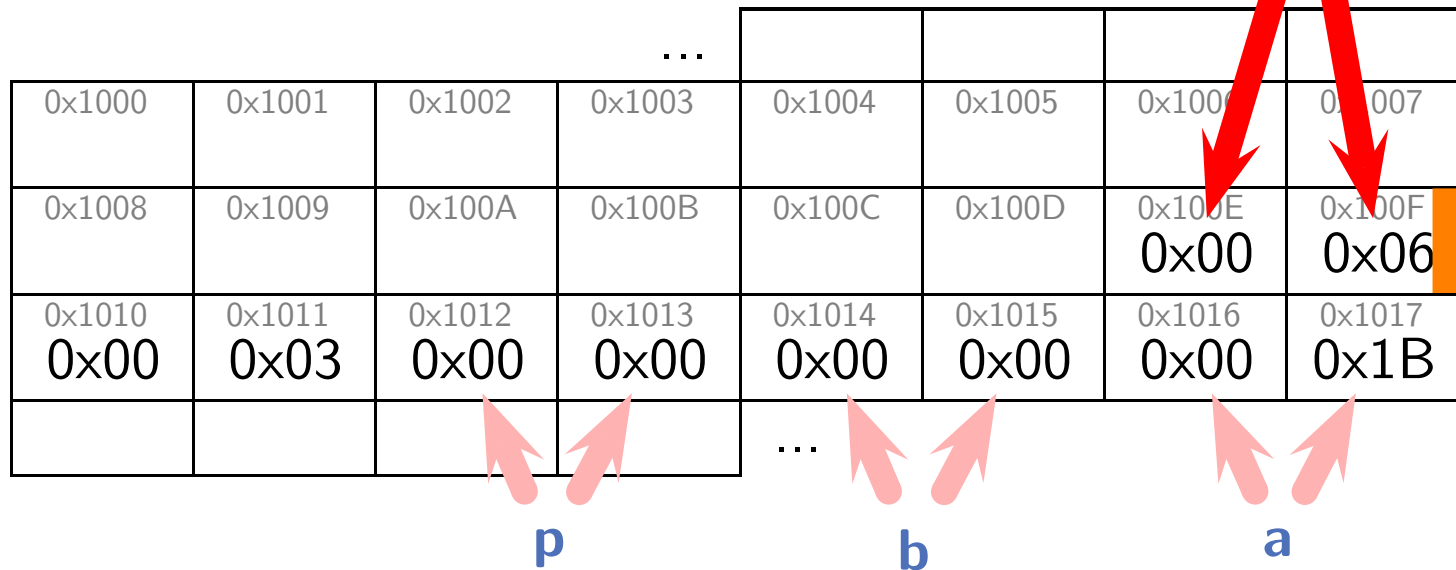
```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```



# Storage Duration “Automatic” (Simplified)

```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```

x no longer alive!

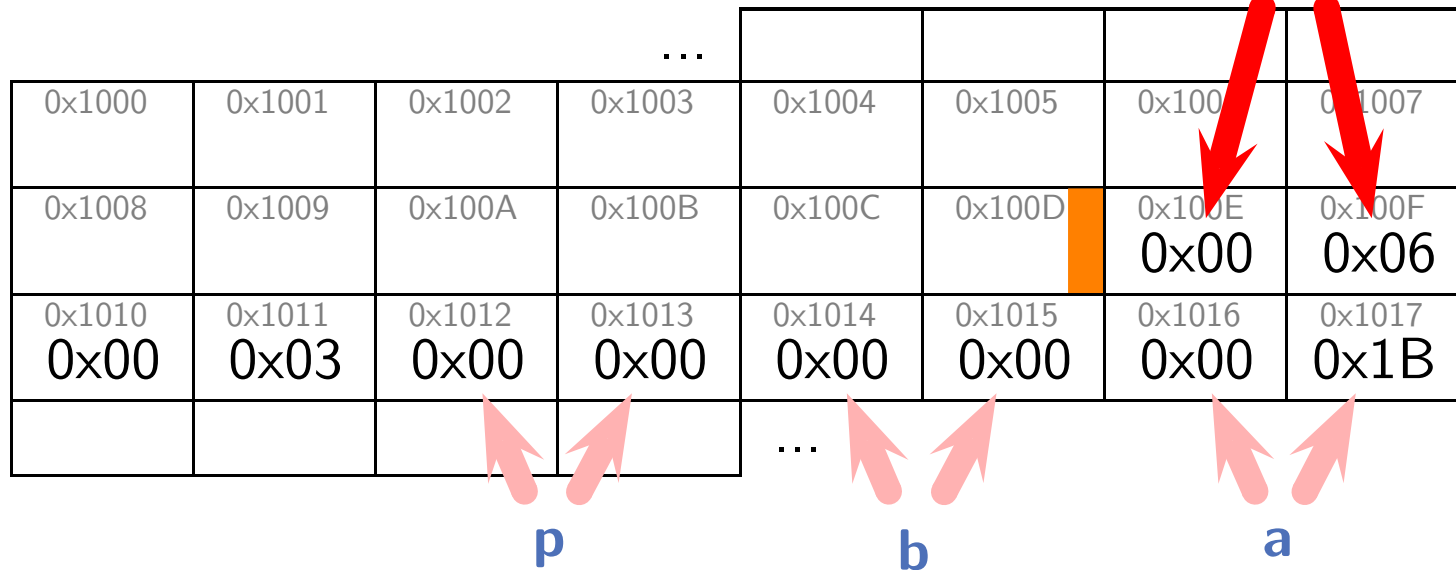




# Storage Duration “Automatic” (Simplified)

```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```

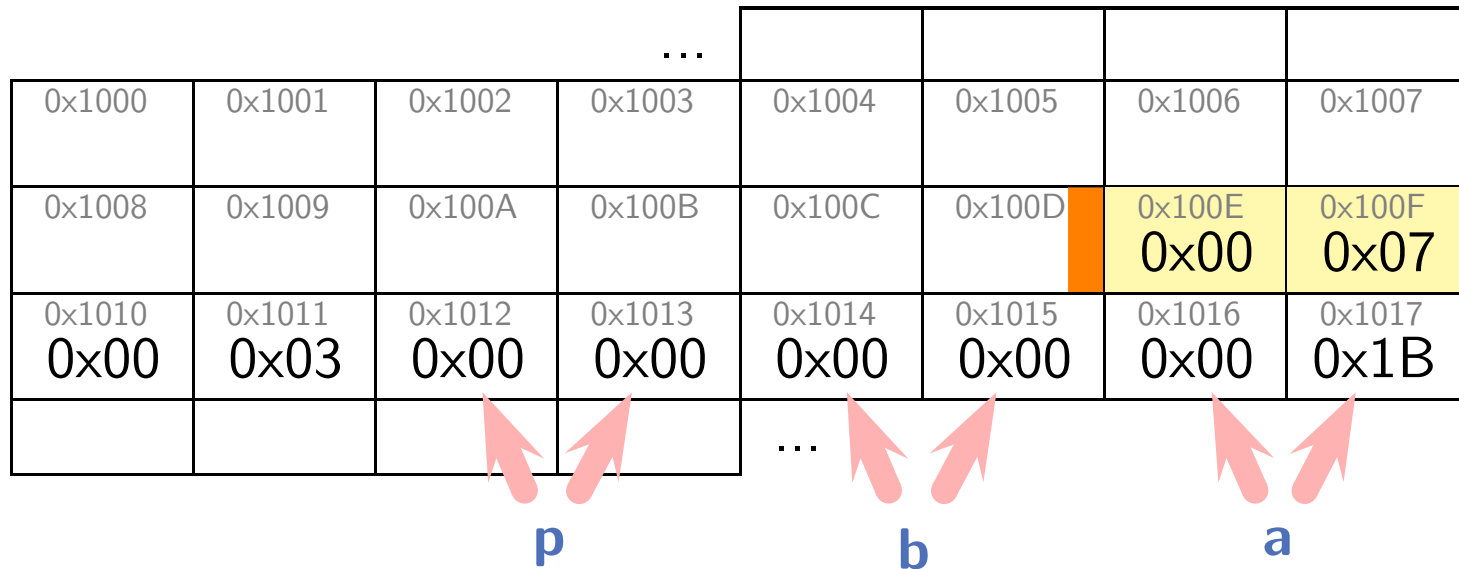
(now) **y** – not explicitly initialised, thus initial value is indeterminate



# Storage Duration “Automatic” (Simplified)



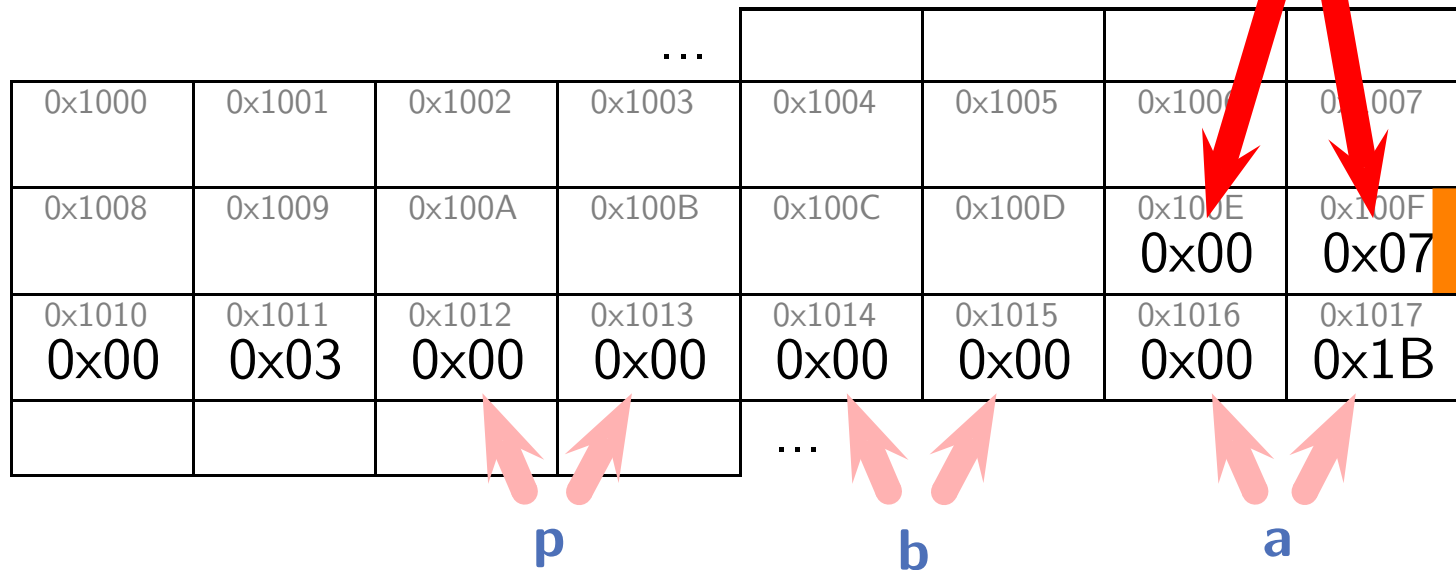
```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```



# Storage Duration “Automatic” (Simplified)

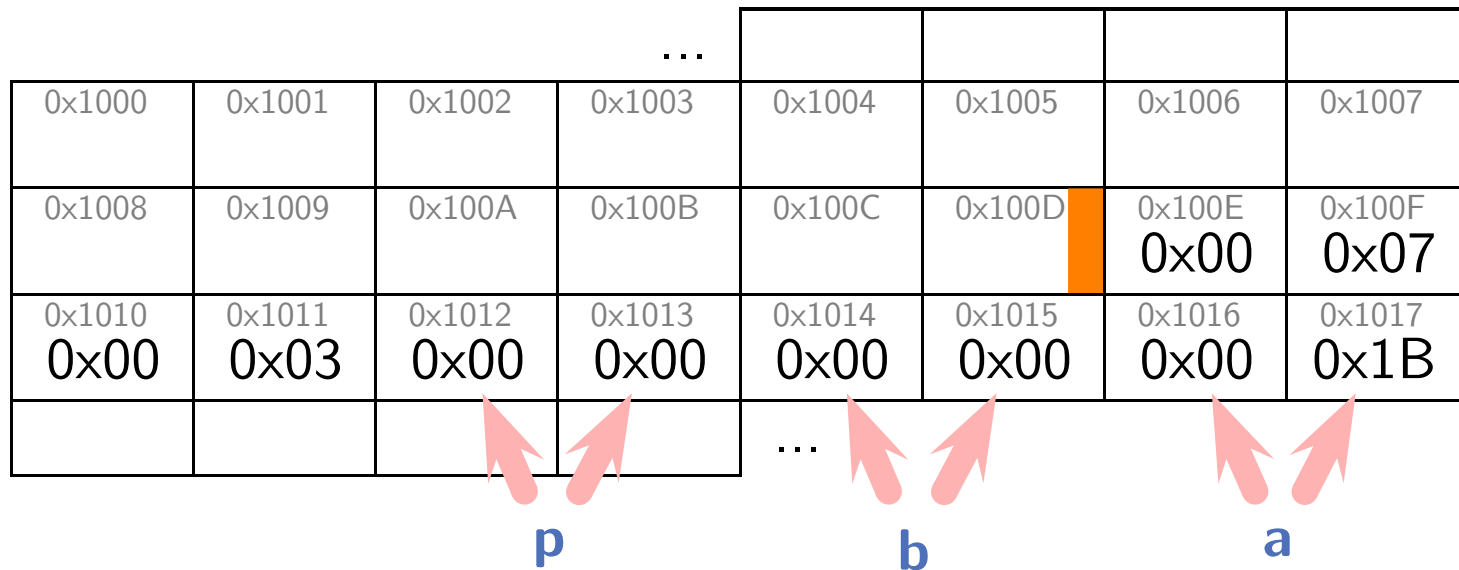
```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```

**y** no longer alive!



# Storage Duration “Automatic” (Simplified)

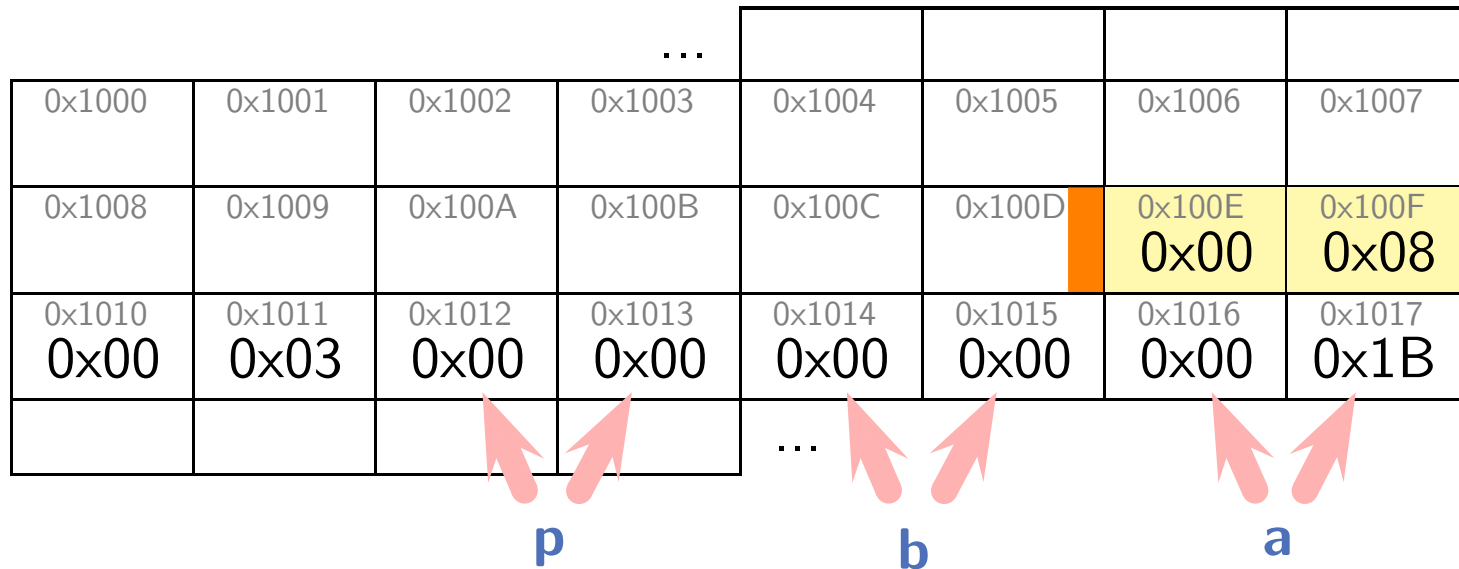
```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```



# Storage Duration “Automatic” (Simplified)




```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```




# Storage Duration “Automatic” (Simplified)

```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```

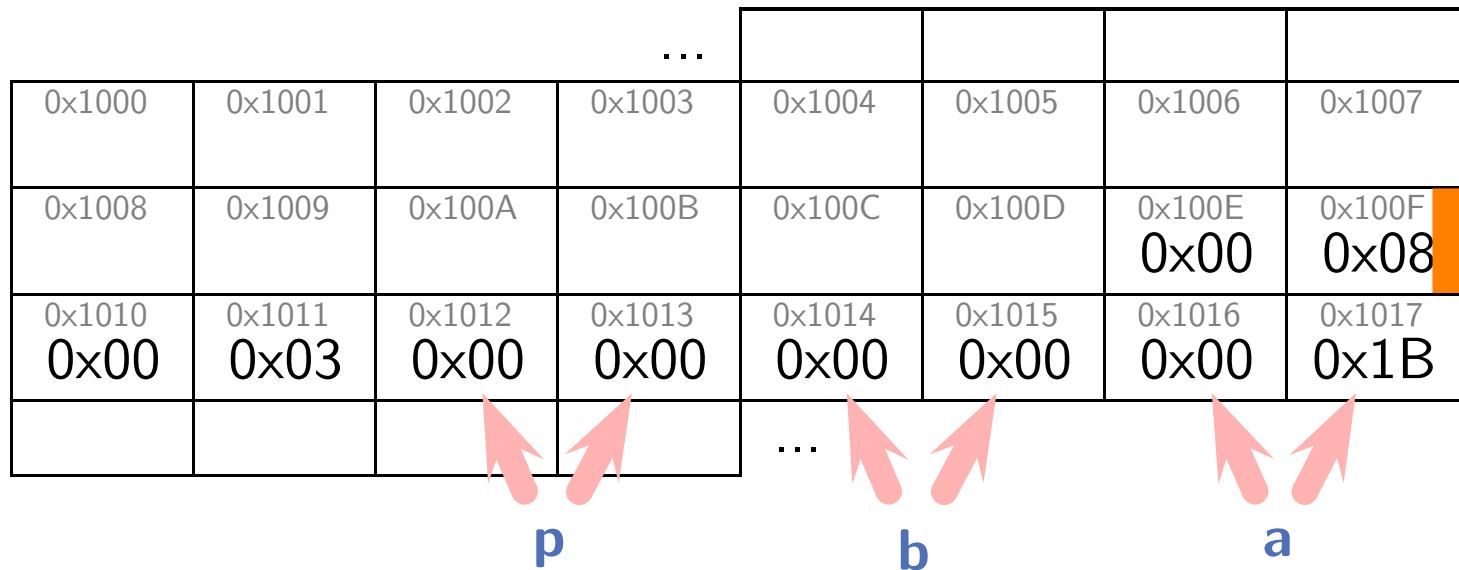



				...				
0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007	
0x1008	0x1009	0x100A	0x100B	0x100C	0x100D	0x100E 0x00	0x100F 0x08	
0x1010 0x00	0x1011 0x03	0x1012 0x00	0x1013 0x00	0x1014 0x00	0x1015 0x00	0x1016 0x00	0x1017 0x1B	
				...				

  
**p**                      **b**                      **a**

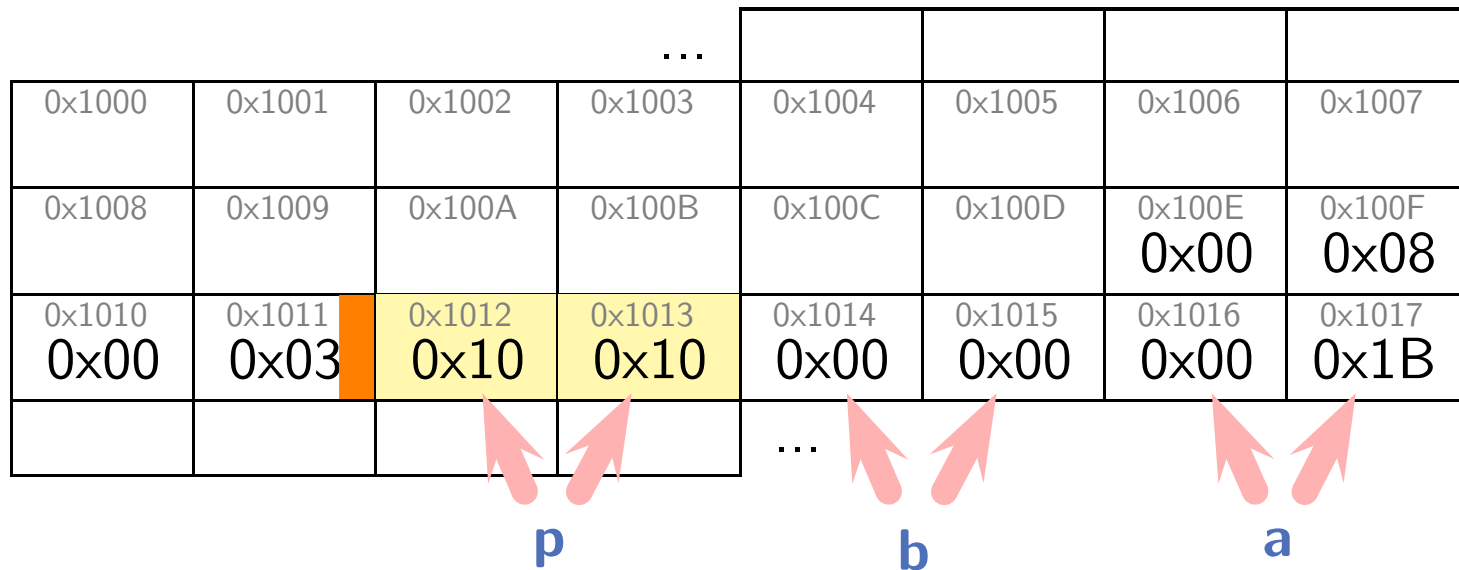
# Storage Duration “Automatic” (Simplified)

```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```



# Storage Duration “Automatic” (Simplified)

```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```



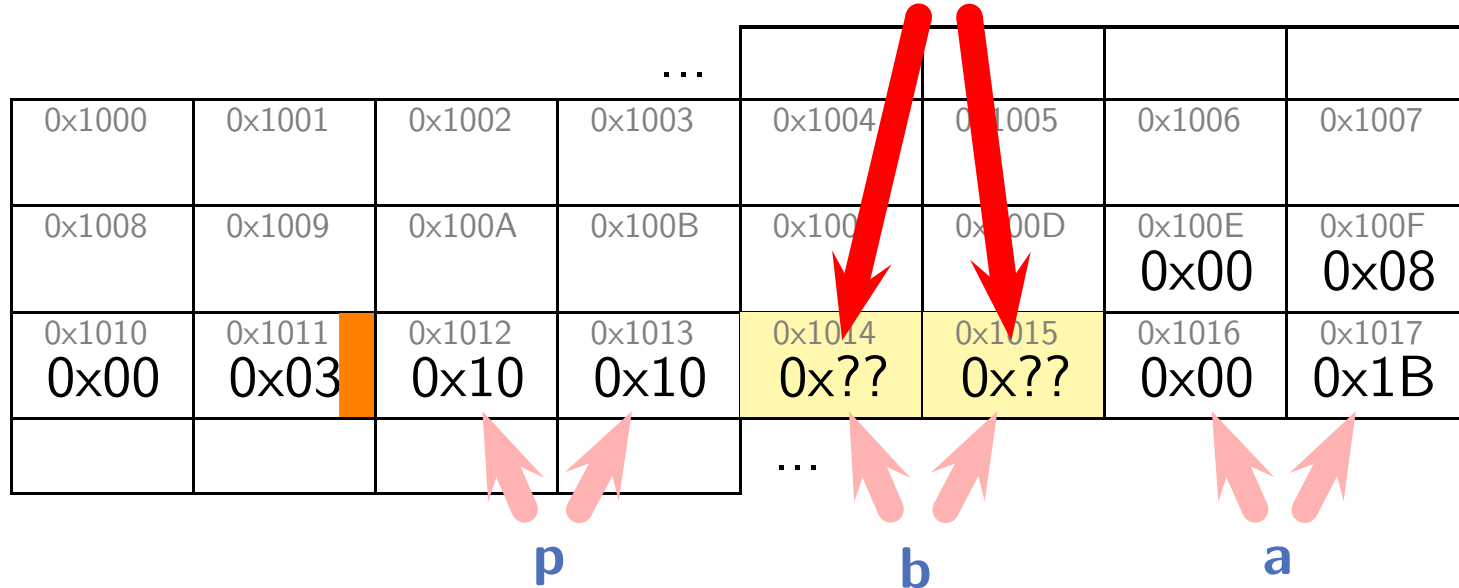


# Storage Duration “Automatic” (Simplified)

```
1 void h() { int y; y++; }
2 void g() { int x = 5; x++; }
3 int* f() { int c = 3; g(); h(); h(); return &c; }
4
5 int a = 27, b, *p;
6 p = f();
7 b = *p;
```



**p** refers to a non-alive object, the behavior is undefined (everything **may happen**, from ‘crash’ to ‘ignore’).



# *Storage Classes and Qualifiers*

## *Storage Class Specifiers (6.7.1)*

# Storage Class Specifiers (6.7.1)

```
1  typedef char letter;  
2  
3  extern int x;  
4  extern int f();  
5  
6  || static int x; // two uses! (-> later)  
7  || static int f();  
8  
9      int  
10 auto x; // "historic"  
11  
12 register y; // "historic"  
      int
```

# Storage Class Specifiers: *extern* (6.7.1)

---

```
1 // not _defined_ here, "imported" ...
2 //
3 extern int x;
4 extern void f();
5
6 // declared _and_ defined here, "exported" ...
7 //
8 int y;
9
10 int g() {
11     x = y = 27;
12     f();
13 }
```

- → modules, linking (later)
- usually only *extern* in headers (later)

# Storage Class Specifiers: *static* (6.7.1)

---

```
1 // declared _and_ defined here ,
2 // _not_ "exported" ...
3 //
4 static int x;
5 static void g();
6
7
8 int f() {
9     static int a = 0;
10    a++;
11    printf( "%s\n" , a );
12 }
13
14 f(); f(); f(); // yields 1, 2, 3
```

## *Qualifiers (6.7.3)*

# Qualifiers (6.7.3)

---

```
1  int x;  
2  
3  const int y;  
4  
5  volatile int z;  
6  
7  int* restrict p; // aliasing  
8  
9  
10 const volatile int a;
```

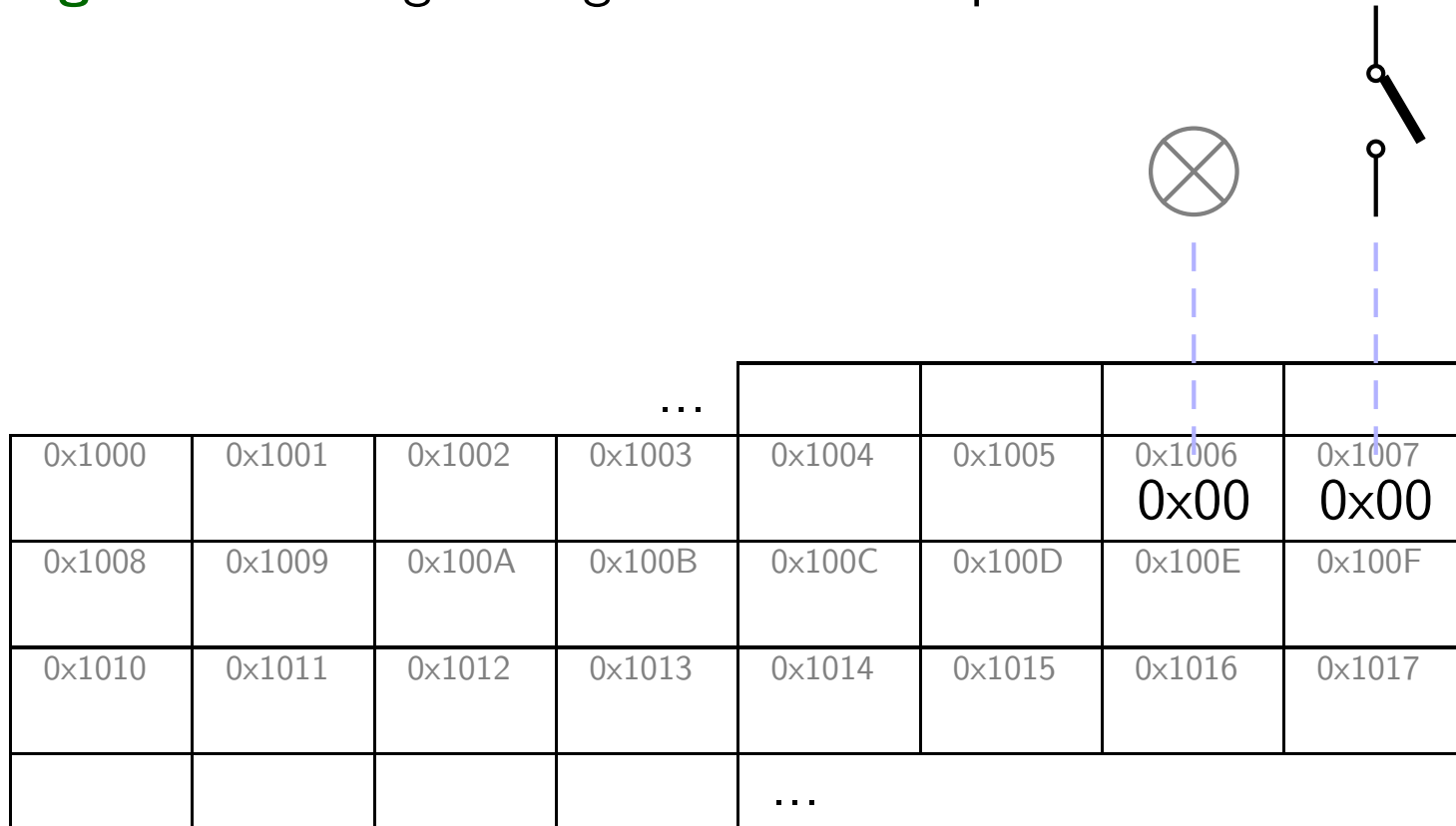
## restrict:

- “[... lengthy formal definition ...]”
- “[...] If these requirements are not met, then the behavior is **undefined**.”
- → use **extremely carefully** (i.e. if in doubt, not at all)



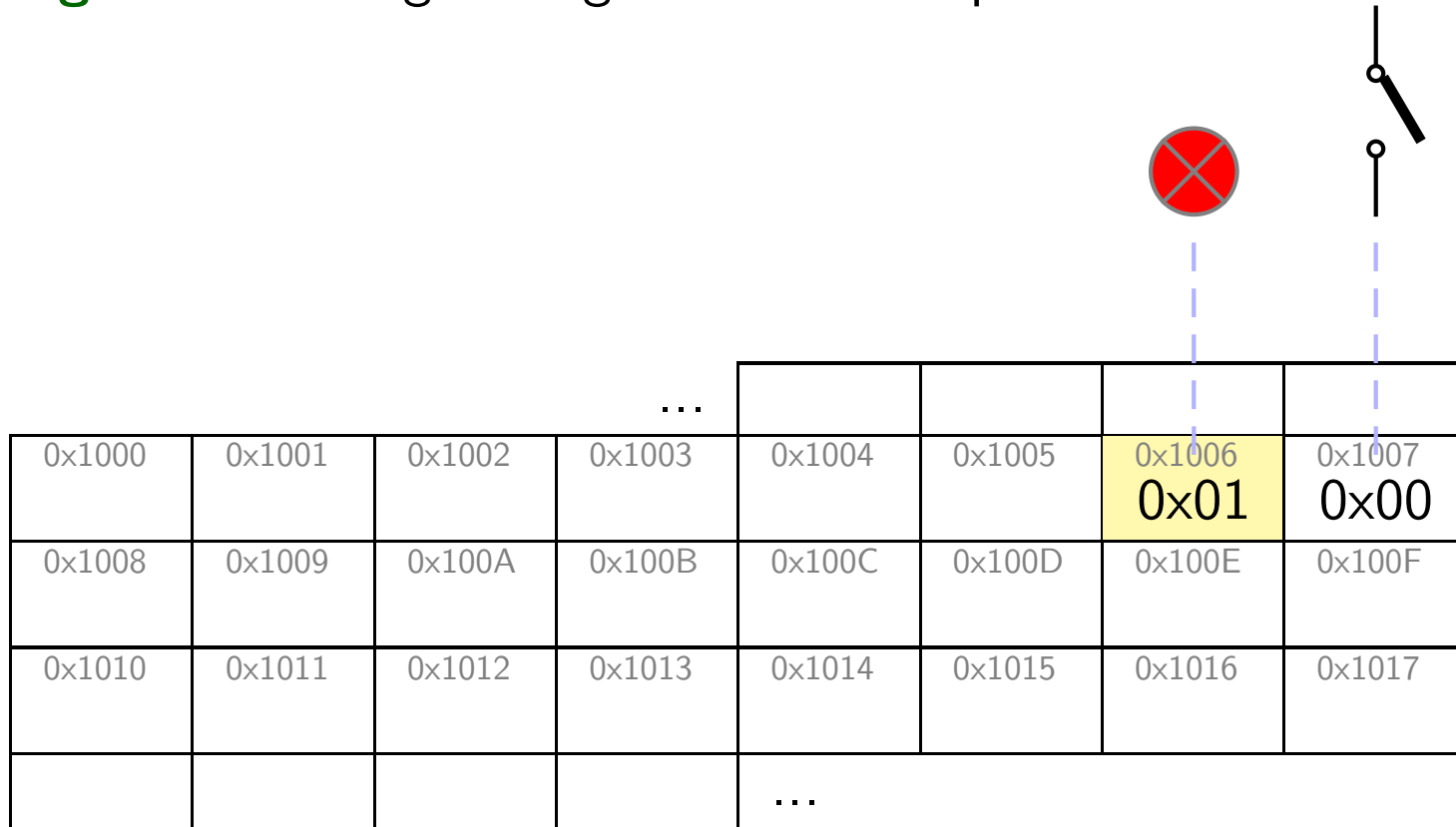
# Excursion: Memory Mapped I/O

- **Intuition:** some memory addresses are wired to hardware
  - **writing** to the address causes a pin to change logical value
  - **reading** the address gives logical value of a pin



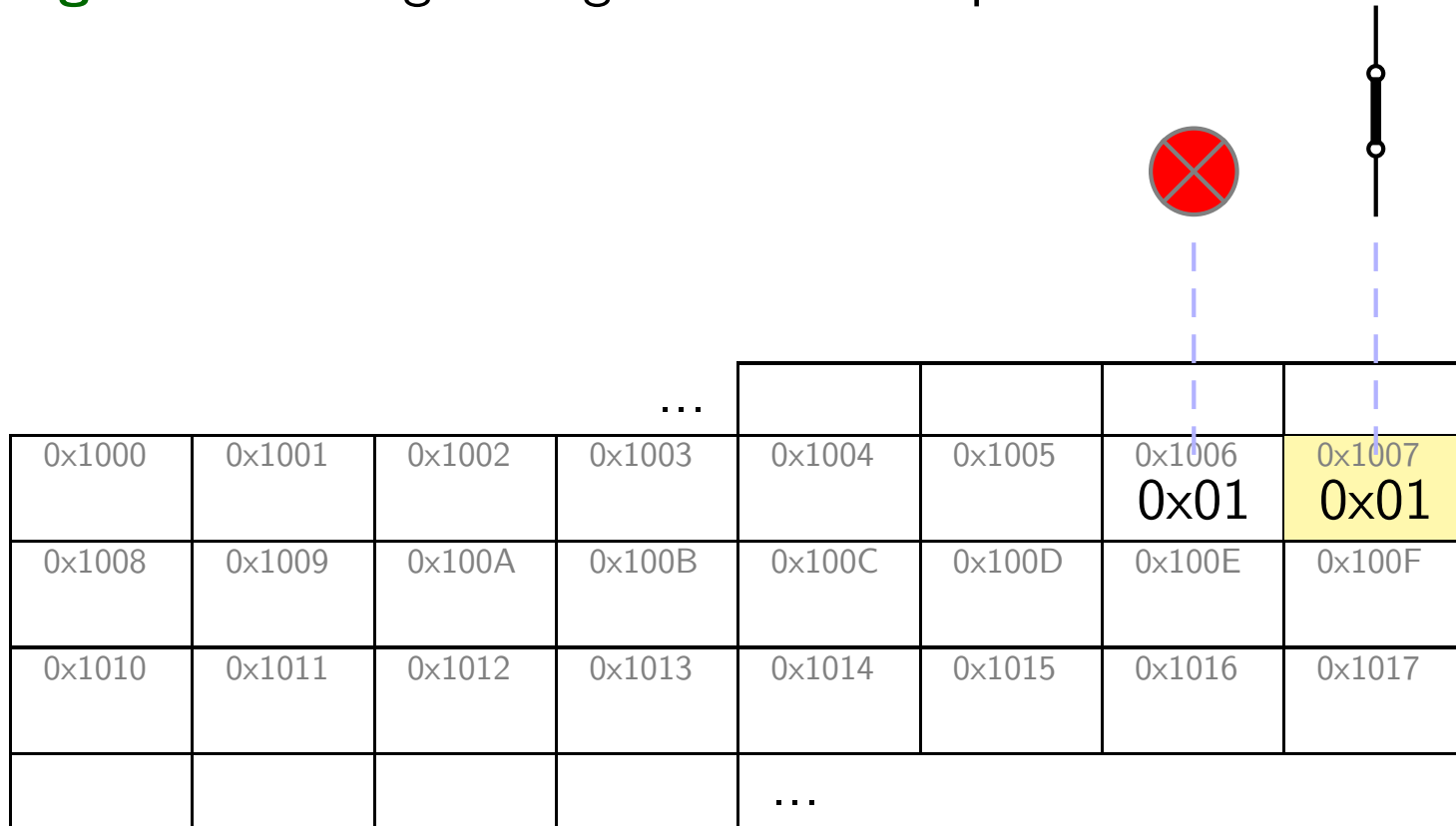
# Excursion: Memory Mapped I/O

- **Intuition:** some memory addresses are wired to hardware
  - **writing** to the address causes a pin to change logical value
  - **reading** the address gives logical value of a pin



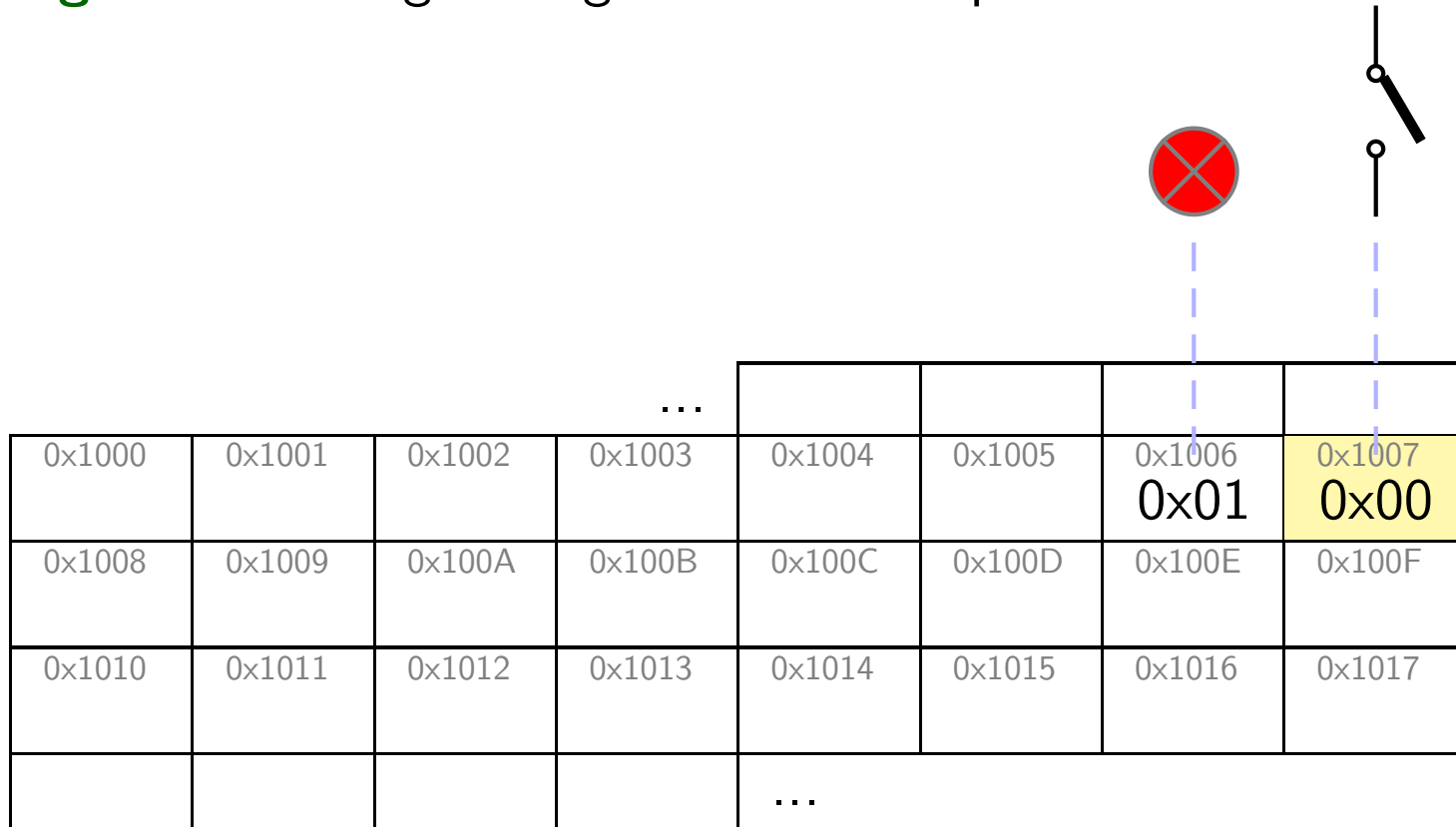
# Excursion: Memory Mapped I/O

- **Intuition:** some memory addresses are wired to hardware
  - **writing** to the address causes a pin to change logical value
  - **reading** the address gives logical value of a pin



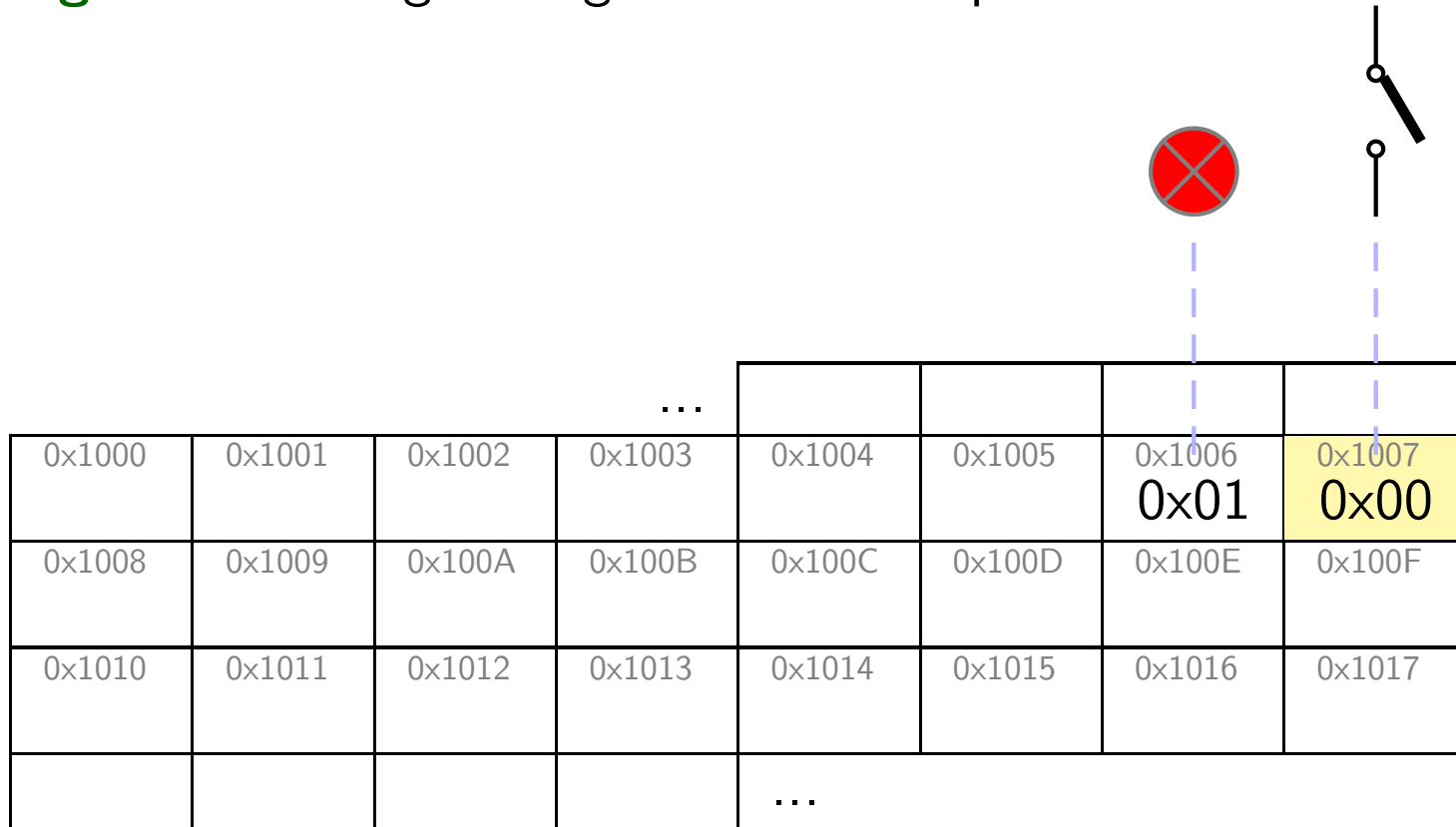
# Excursion: Memory Mapped I/O

- **Intuition:** some memory addresses are wired to hardware
  - **writing** to the address causes a pin to change logical value
  - **reading** the address gives logical value of a pin



# Excursion: Memory Mapped I/O

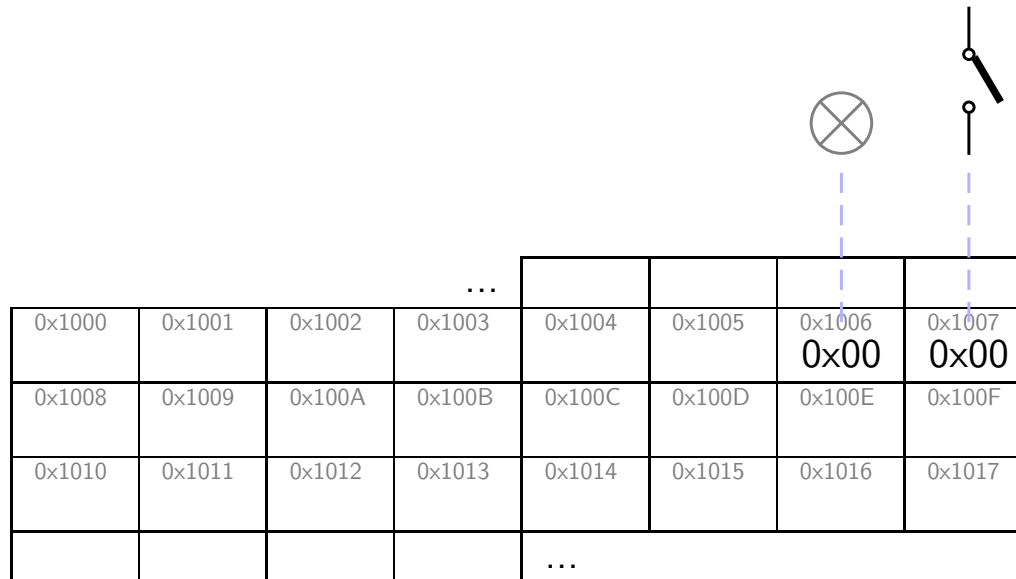
- **Intuition:** some memory addresses are wired to hardware
  - **writing** to the address causes a pin to change logical value
  - **reading** the address gives logical value of a pin



- The compiler does not know, “**memory is memory**”.

# Qualifiers: *volatile* (6.7.3)

```
1  volatile char* out = 0x1006;  
2  volatile char* in  = 0x1007;  
3  
4  *out = 0x01; // switch lamp on  
5  
6  if (*in & 0x01) { /* ... */ }  
7  
8  if ((*in & 0x01) && (*in & 0x01)) { /* ... */ }
```



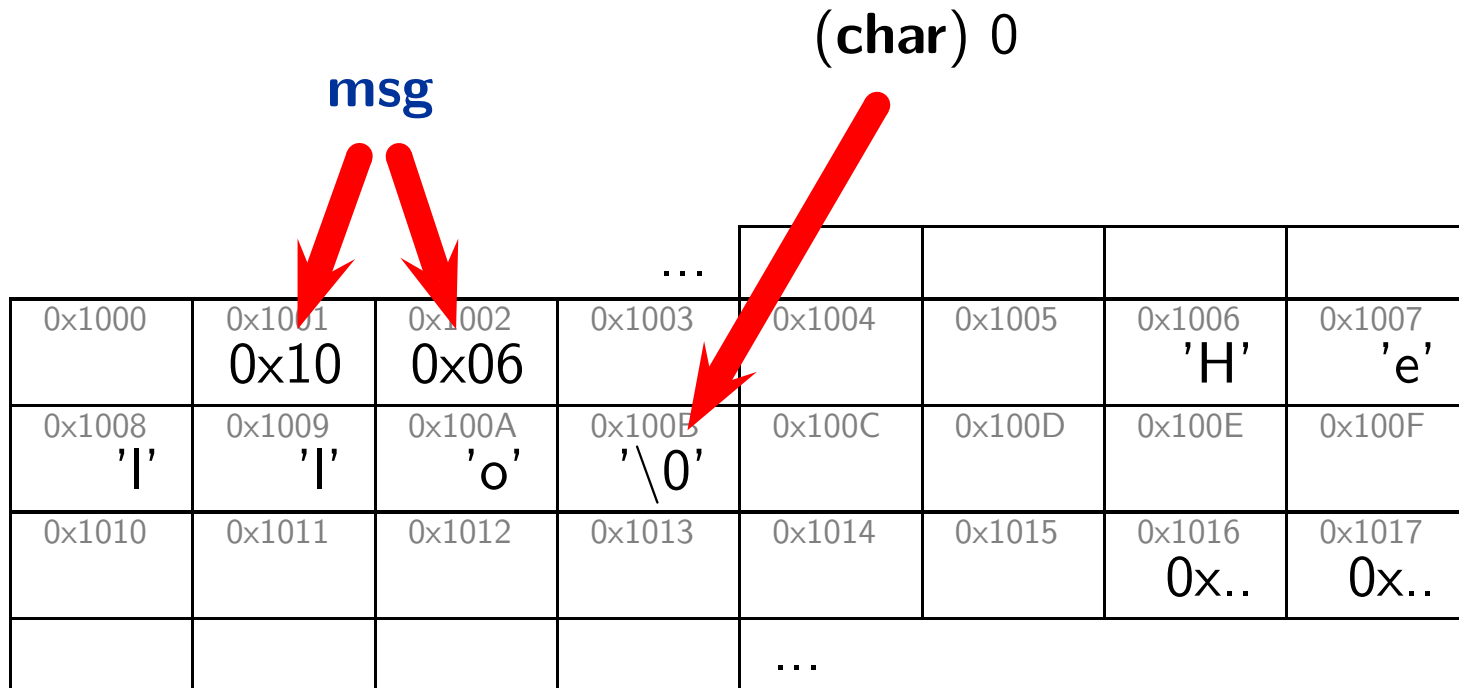
# *Strings & Input/Output*

# *Strings*



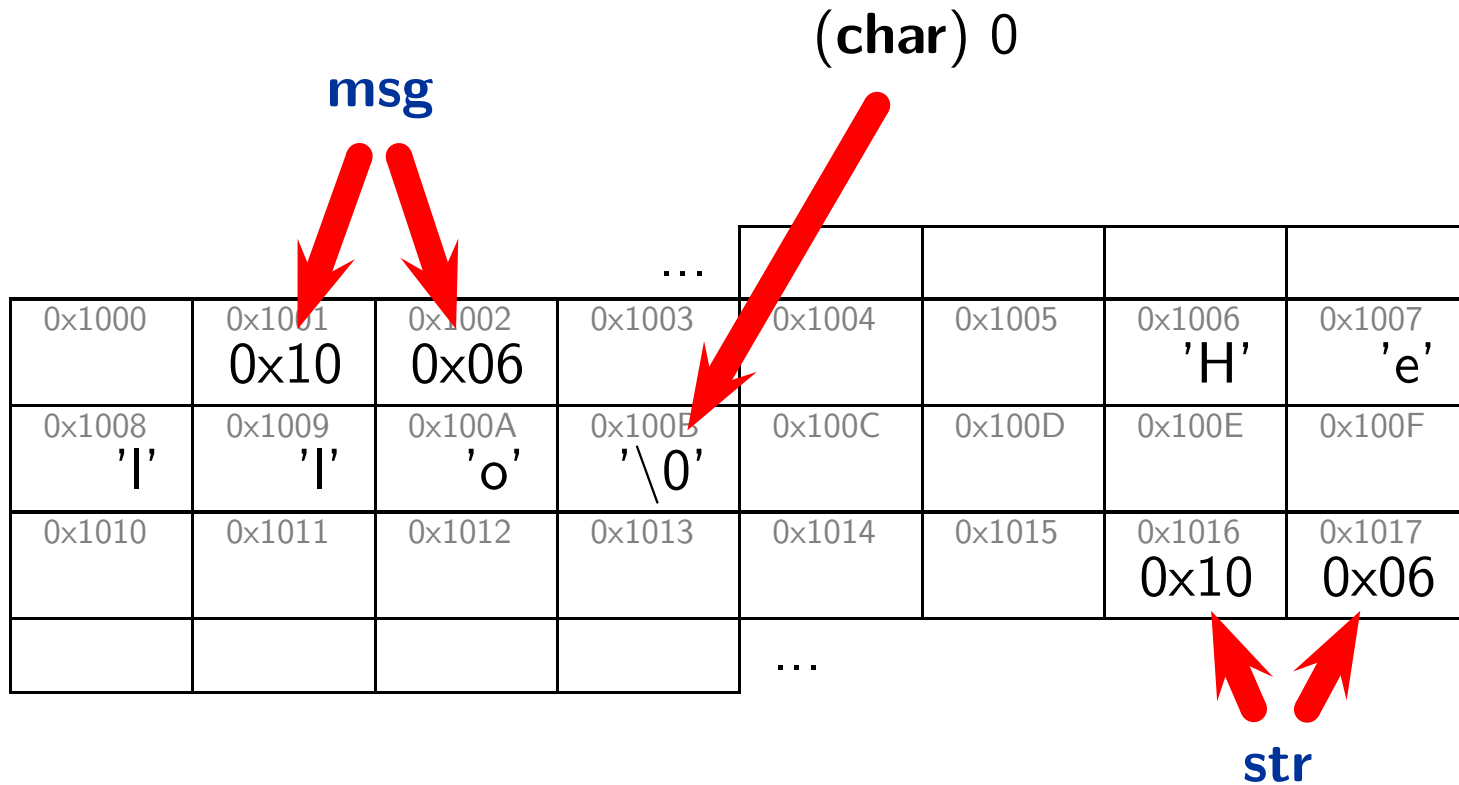

# Strings are 0-Terminated char Arrays

➔  
1 `char* msg = "Hello";`  
2 `char* str = msg;`



# Strings are 0-Terminated char Arrays

```
1 char* msg = "Hello";  
2 char* str = msg;
```



# String Manipulation (Annex B)

---

**# include**<string.h>

provides among others:

- size\_t **strlen**( const char\* s )  
“[...] calculates length of string s, excluding **the terminating null byte** ('\0’).”
- int **strcmp**( const char\* s1, const char\* s2 )  
“[...] compares the two strings s1 and s2.  
It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.”
- char\* **strcpy**( char\* s1, const char\* s2 )  
“The strcpy() function copies the string pointed to by s2, including **the terminating null byte** ('\0’), to the buffer pointed to by s1.”
- char\* **strncpy**( char\* s1, const char\* s2, size\_t n )

**None of these functions allocates memory!**

Allocate and copy: (not C99, but POSIX)

- char\* **strdup**( const char\* s )

# *Input/Output*

# Printing

---

```
1 #include <stdio.h>
2
3 printf( "%s %i %f\n", "Hello", 27, 3.14 );
```

formatted  
string  
integer  
(or d)  
float  
newline



# *Tools & Modules*

# Hello, Again

---

```
1  #include <stdio.h>
2
3  int g( int x ) { return x/2; }
4
5  int f() { return g(1); }
6
7  int main() {
8      printf( "Hello World.\n" );
9      return f();
10 }
```

- % gcc helloworld.c
- % ls
- a.out helloworld.c
- % ./a.out
- Hello World.
- %

# Zoom In: Preprocessing, Compiling, Linking

---

```
1  #include <stdio.h>
2
3  int g( int x ) { return x/2; }
4
5  int f() { return g(1); }
6
7  int main() {
8      printf( "Hello World.\n" );
9      return f();
10 }
```

- % gcc -E helloworld.c > helloworld.i
- % gcc -c -o helloworld.o helloworld.c
- % ld -o helloworld [...] helloworld.o [...]
- % ./helloworld
- Hello World.
- %



# Modules

---

```
1 #include <stdio.h>
2
3 int g( int x ) {
4     return x/2;
5 }
6
7 int f() {
8     return g(1);
9 }
10
11 int main() {
12     printf( "Hello _World.\n" );
13     return f();
14 }
```

# Modules

---

```
1  #include <stdio.h>
2
3  int g( int x ) {
4      return x/2;
5  }
6
7  int f() {
8      return g(1);
9  }
10
11 int main() {
12     printf( "Hello _World.\n" );
13     return f();
14 }
```

Split into:

- .h (header): declarations
- .c: definitions, use headers to “import” declarations

# Modules

```
1 #include <stdio.h>
2
3 int g( int x ) {
4     return x/2;
5 }
6
7 int f() {
8     return g(1);
9 }
10
11 int main() {
12     printf( "Hello World.\n" );
13     return f();
14 }
```

g.h

```
1 #ifndef G_H
2 #define G_H
3
4 extern int
5     g( int x );
6 #endif
```

f.h

```
1 #ifndef F_H
2 #define F_H
3
4 extern int
5     f();
6 #endif
```

g.c

```
1 #include "g.h"
2
3 int g( int x ) {
4     return x/2;
5 }
```

f.c

```
1 #include "g.h"
2 #include "f.h"
3
4 int f() {
5     return g(1);
6 }
```

Split into:

- .h (header): declarations
- .c: definitions, use headers to “import” declarations

helloworld.c

```
1 #include <stdio.h>
2 #include "f.h"
3
4 int main() {
5     printf( "Hello World.\n" );
6     return f();
7 }
```

# Modules At Work

## preprocess & compile:

- `% gcc -c g.c f.c \`  
`helloworld.c`
- `% ls *.o`
- `f.o g.o helloworld.o`

## link:

- `% gcc g.o f.o helloworld.o`

## execute:

- `% ./a.out`
- Hello World.

`g.h`

```
1 #ifndef G_H
2 #define G_H
3
4 extern int
5     g( int x );
6 #endif
```

`f.h`

```
1 #ifndef F_H
2 #define F_H
3
4 extern int
5     f();
6 #endif
```

`g.c`

```
1 #include "g.h"
2
3 int g( int x ) {
4     return x/2;
5 }
```

`f.c`

```
1 #include "g.h"
2 #include "f.h"
3
4 int f() {
5     return g(1);
6 }
```

`helloworld.c`

```
1 #include <stdio.h>
2 #include "f.h"
3
4 int main() {
5     printf( "Hello World.\n" );
6     return f();
7 }
```

# Modules At Work

## preprocess & compile:

- % gcc -c g.c f.c \  
helloworld.c
- % ls \*.o
- f.o g.o helloworld.o

## link:

- % gcc g.o f.o helloworld.o

## execute:

- % ./a.out
- Hello World.

g.h

```
1 #ifndef G_H
2 #define G_H
3
4 extern int
5     g( int x );
6 #endif
```

f.h

```
1 #ifndef F_H
2 #define F_H
3
4 extern int
5     f();
6 #endif
```

g.c

```
1 #include "g.h"
2
3 int g( int x ) {
4     return x/2;
5 }
```

f.c

```
1 #include "g.h"
2 #include "f.h"
3
4 int f() {
5     return g(1);
6 }
```

helloworld.c

```
1 #include <stdio.h>
2 #include "f.h"
3
4 int main() {
5     printf( "Hi!\n" );
6     return f();
7 }
```

# Modules At Work

## preprocess & compile:

- % gcc -c g.c f.c \ helloworld.c
- % ls \*.o
- f.o g.o helloworld.o

## link:

- % gcc g.o f.o helloworld.o

## execute:

- % ./a.out
- Hello World.

## fix and re-build:

- % gcc -c helloworld.c
- % gcc g.o f.o helloworld.o
- % ./a.out
- Hi!

### g.h

```
1 #ifndef G_H
2 #define G_H
3
4 extern int
5     g( int x );
6 #endif
```

### f.h

```
1 #ifndef F_H
2 #define F_H
3
4 extern int
5     f();
6 #endif
```

### g.c

```
1 #include "g.h"
2
3 int g( int x ) {
4     return x/2;
5 }
```

### f.c

```
1 #include "g.h"
2 #include "f.h"
3
4 int f() {
5     return g(1);
6 }
```

### helloworld.c

```
1 #include <stdio.h>
2 #include "f.h"
3
4 int main() {
5     printf( "Hi!\n" );
6     return f();
7 }
```

# Preprocessing

## helloworld.c

```
1 #include <stdio.h>
2 #include "f.h"
3
4 int main() {
5     printf( " Hello World.\n" );
6     return f();
7 }
```

- % gcc -E helloworld.c  
-o helloworld.i

## helloworld.i

```
1 # 1 " helloworld.c"
2 # 1 "<command-line>"
3 # 1 " helloworld.c"
4 # 1 "/usr/include/stdio.h" 1 3 4
5 # 28 "/usr/include/stdio.h" 3 4
6 # 1 "/usr/include/features.h" 1 3 4
7 # 323 "/usr/include/features.h" 3 4
8 # 1 "/usr/include/x86_64-linux-gnu/bits/predefs.h" 1 3 4
9 # 324 "/usr/include/features.h" 2 3 4
10 # 356 "/usr/include/features.h" 3 4
11 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
12 # 359 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
13 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
14 # 360 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
15 # 357 "/usr/include/features.h" 2 3 4
16 # 388 "/usr/include/features.h" 3 4
17 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
18
19
20
21 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
22 # 5 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
23
24 [...]
25
26 extern int ftrylockfile (FILE *__stream) __attribute__((__nothrow__));
27
28
29 extern void funlockfile (FILE *__stream) __attribute__((__nothrow__));
30 # 936 "/usr/include/stdio.h" 3 4
31
32 # 2 " helloworld.c" 2
33 # 1 "f.h" 1
34
35
36
37 extern int
38     f();
39 # 3 " helloworld.c" 2
40
41 int main() {
42     printf( " Hello World.\n" );
43     return f();
44 }
```

# Preprocessing Directives (6.10)

---

```
1  #include <stdio.h>
2  #include "battery.h"
3
4  #define PI 3.1415
5
6  #define DEBUG
7  #ifdef DEBUG
8      fprintf( stderr , "honk\n" );
9  #endif
10
11 #if __GNUC_  _ >= 3
12 # define __pure          __attribute_  _ ((pure))
13 #else
14 # define __pure          /* no pure */
15 #endif
16
17 extern int f() __pure;
```



# Linking

---

**provides:** int g(int)

**needs:** ./.  
g.o

**provides:** int f()

**needs:** int g(int)

f.o

**provides** int main()

**needs:**

int f(int)

int printf(const char\*,...)

helloworld.o

**provides:**

int printf(const char\*,...)

...

**needs:**

...

libc.a

# Linking

---

```
provides: int g(int)
```

```
needs: ./.
```

```
g.o
```

```
provides: int f()
```

```
needs: int g(int)
```

```
f.o
```

```
provides int main()
```

```
needs:
```

```
int f(int)
```

```
int printf(const char*,...)
```

```
helloworld.o
```

```
provides:
```

```
int printf(const char*,...)
```

```
...
```

```
needs:
```

```
...
```

```
libc.a
```

# Linking

---

**provides:** int g(int)

**needs:** ./.  
g.o

**provides:** int f()

**needs:** int g(int)

f.o

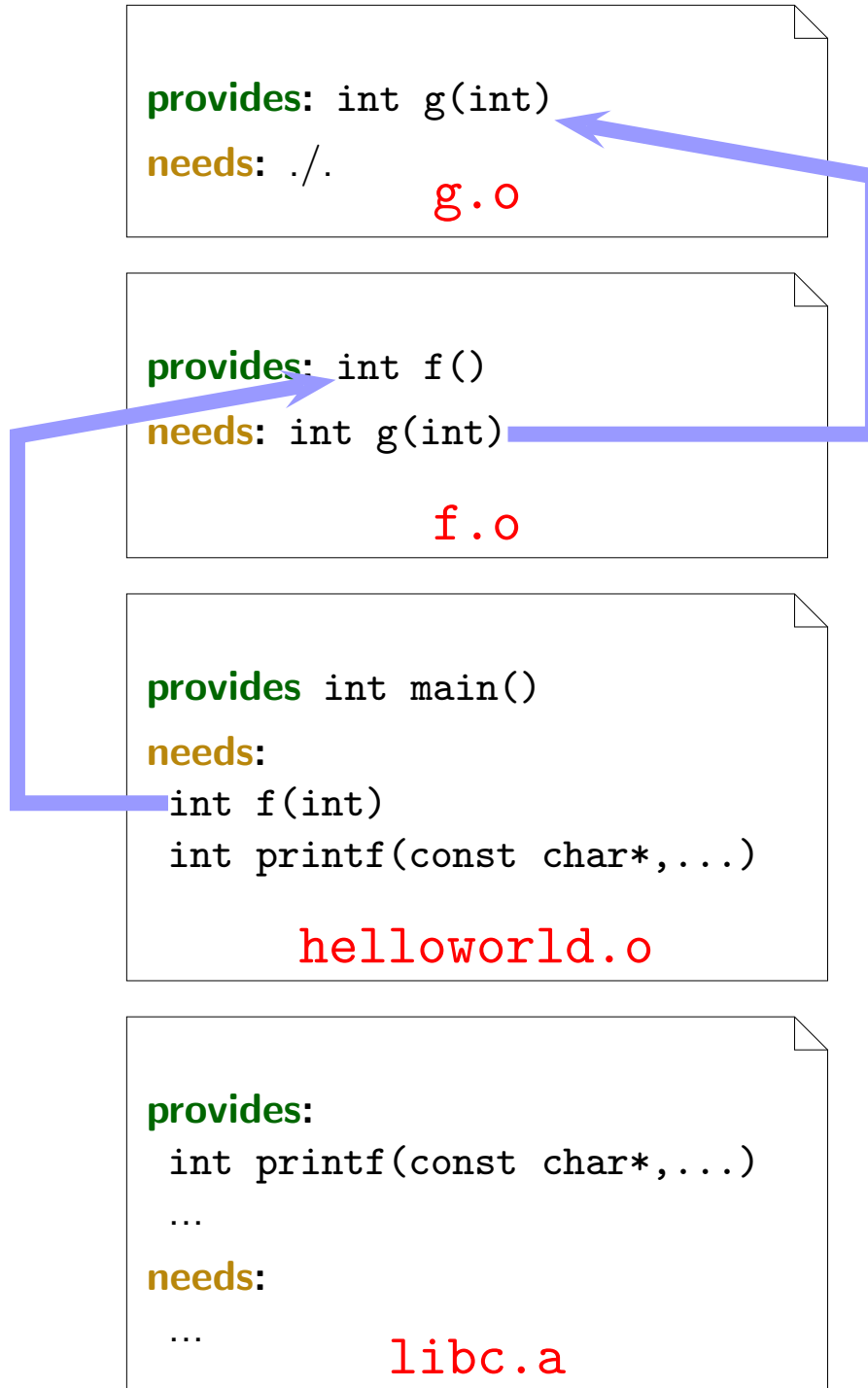
**provides** int main()

**needs:**  
int f(int)  
int printf(const char\*,...)

helloworld.o

**provides:**  
int printf(const char\*,...)  
...

**needs:**  
...  
libc.a



# Linking

---

```
provides: int g(int)
```

```
needs: ./.  
g.o
```

```
provides: int f()
```

```
needs: int g(int)  
f.o
```

```
provides int main()
```

```
needs:  
int f(int)  
int printf(const char*,...)
```

```
helloworld.o
```

```
provides:
```

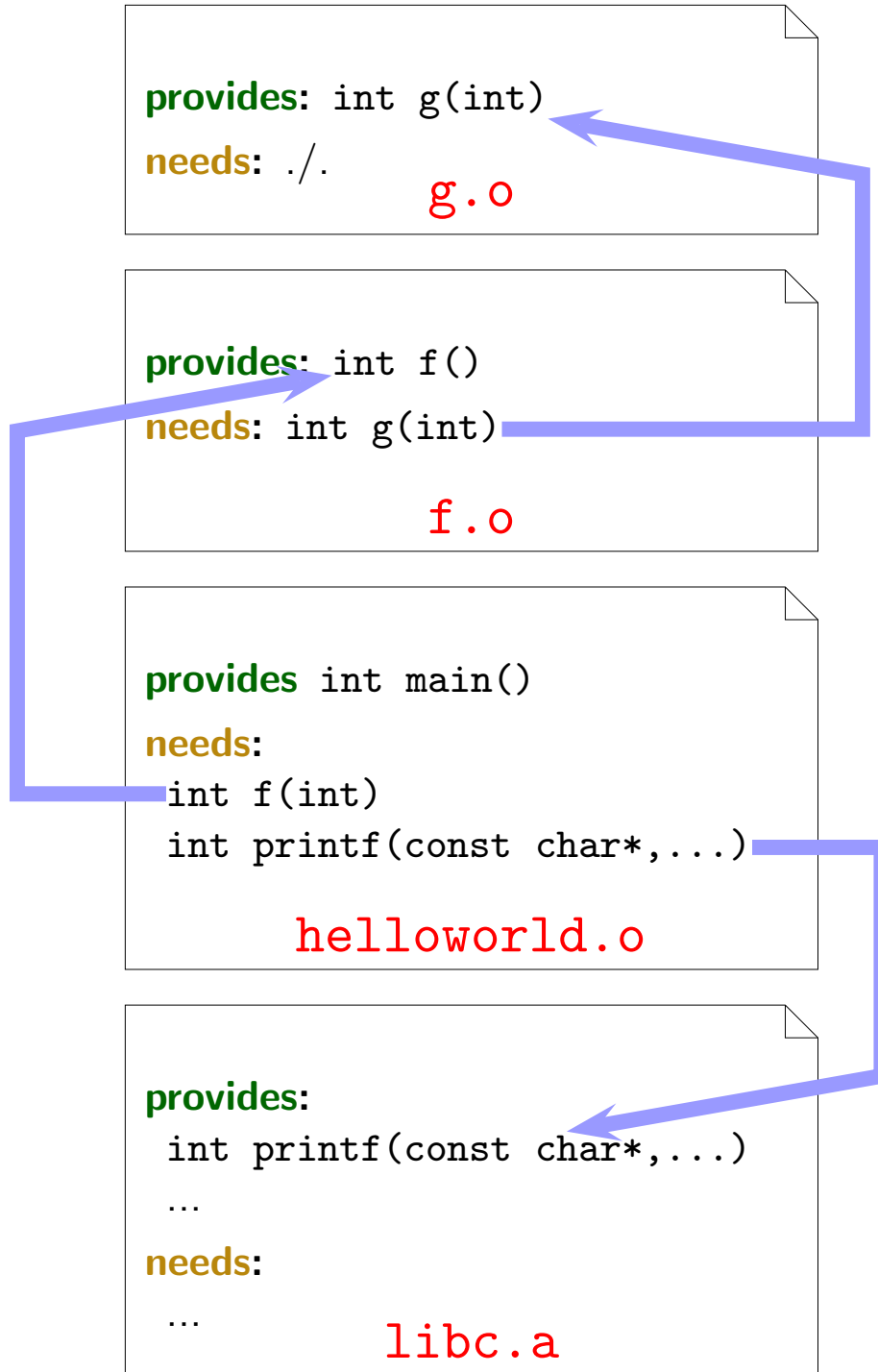
```
int printf(const char*,...)
```

```
...
```

```
needs:
```

```
...
```

```
libc.a
```



# Linking

**provides:** int g(int)

**needs:** ./.  
g.o

**provides:** int f()

**needs:** int g(int)  
f.o

**provides** int main()

**needs:**  
int f(int)  
int printf(const char\*,...)

helloworld.o

**provides:**

int printf(const char\*,...)

...

**needs:**

...

libc.a

a.out

# Compiler

---

`gcc [OPTION]... infile...`

**-E** – preprocess only

**-c** – compile only, don't link

**Example:** `gcc -c main.c` — produces `main.o`

**-o outfile** – write output to **outfile**

**Example:** `gcc -c -o x.o main.c` — produces `x.o`

**-g** – add debug information

**-W, -Wall, ...** – enable warnings

**-I dir** – add **dir** to **include path** for searching headers

**-L dir** – add **dir** to **library path** for searching libraries

**-D macro[=defn]** – define **macro** (to **defn**)

**Example:** `gcc -DDEBUG -DMAGICNUMBER=27`

**-l library** link against `liblibrary.{a,so}`, order matters

**Example:** `gcc a.o b.o main.o -lxy`

→ cf. `man gcc`

# *gdb(1), ddd(1), nm(1), make(1)*

---

- **Command Line Debugger:**

```
gdb a.out [core]
```

- **GUI Debugger:**

```
ddd a.out [core]
```

(works best with debugging information compiled in (`gcc -g`))

- **Inspect Object Files:**

```
nm a.o
```

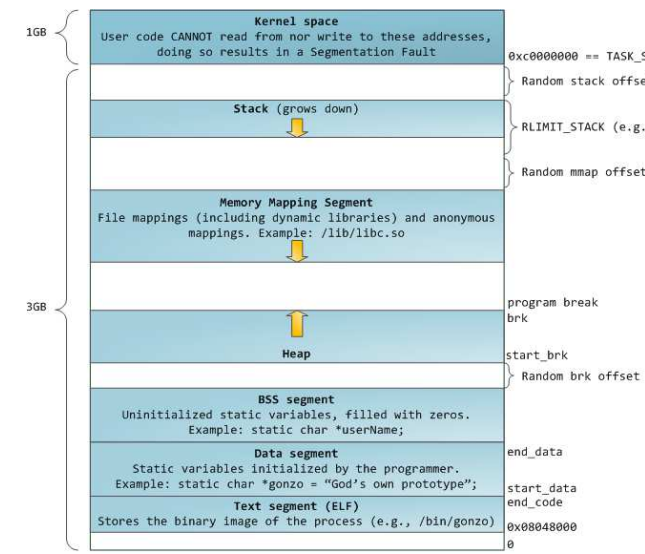
- **Build Utility:**

```
make
```

See battery controller exercise for an example.

# Core Dumps

- **Recall:** Anatomy of a Linux Program in Memory
- **Core dump:** (basically) this memory written to a file.

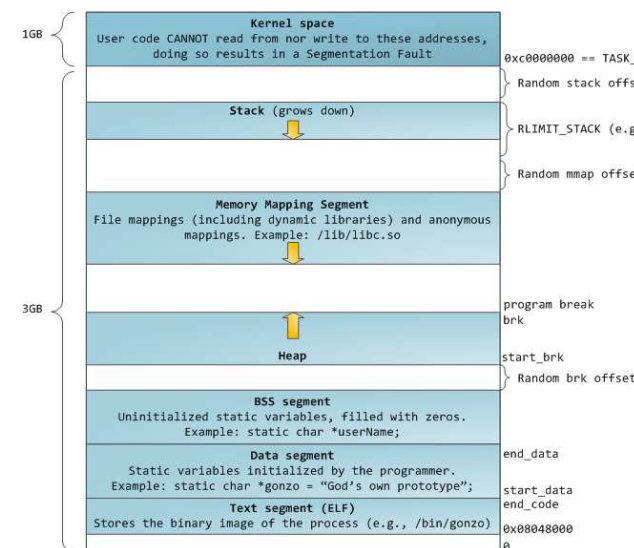


©Gustavo Duarte 2009, used by permission.  
<http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory>



# Core Dumps

- **Recall:** Anatomy of a Linux Program in Memory
- **Core dump:** (basically) this memory written to a file.



©Gustavo Duarte 2009, used by permission.  
<http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory>

```
1 int main() {
2     int *p;
3     *p = 27;
4     return 0;
5 }
```

```
1 % gcc -g core.c
2 % limit coredumpsize
3 coredumpsize 0 kbytes
4 % limit coredumpsize 1g
5 % ./a.out
6 Segmentation fault (core dumped)
7 % ls -lh core
8 -rw-rw-r-- 1 user user 232K Feb 29 11:11 core
9 % gdb a.out core
10 GNU gdb (GDB) 7.4.1-debian
11 [...]
12 Core was generated by './a.out'.
13 Program terminated with signal 11, Segmentation fault.
14 #0 0x0000000004004b4 in main() at core.c:3
15 3 *p = 27;
16 (gdb) p p
17 $1 = (int *) 0x0
18 (gdb) q
```

# *Formal Methods for C*

# *Correctness and Requirements*

# Correctness

---

- Correctness is defined **with respect to** a specification.
- A program (function, ...) is **correct** (wrt. specification  $\varphi$ )  
**if and only if** it satisfies  $\varphi$ .
- Definition of “satisfies”: **in a minute**.

## Examples:

- $\varphi_1$ : the return value is 10 divided by parameter (if parameter not 0)
- $\varphi_2$ : the value of variable  $x$  is “always” strictly greater than 3
- $\varphi_3$ : the value of  $i$  increases in each loop iteration
- ...

# Common Patterns

---

- **State Invariants:**

“at **this** program point, the value of  $p$  must not be NULL”

“at **all** program points, the value of  $p$  must not be NULL”

(cf. **sequence points** (Annex C))

- **Data Invariants:**

“the value of  $n$  must be the length of  $s$ ”

- **(Function) Pre/Post Conditions:**

Pre-Condition: the parameter must not be 0

Post-Condition: the return value is 10 divided by the parameter

- **Loop Invariants:**

“the value of  $i$  is between 0 and array length minus 1”

*Poor Man's Requirements Specification  
aka. How to Formalize Requirements in C?*

# Diagnosics (7.2)

---

```
1 #include <assert.h>
2 void assert( /* scalar */ expression );
```

# Diagnostics (7.2)

---

```
1 #include <assert.h>
2 void assert( /* scalar */ expression );
```

- “The assert macro puts diagnostic tests into programs; [...]

When it is executed, if `expression` (which shall have a scalar type) is false (that is, compares equal to 0), the assert macro

- writes information about the particular call that failed [...] on the standard error stream in an implementation-defined format.
- It then calls the **abort** function.”



# Diagnostics (7.2)

---

```
1 #include <assert.h>
2 void assert( /* scalar */ expression );
```

- “The assert macro puts diagnostic tests into programs; [...]

When it is executed, if **expression** (which shall have a scalar type) is false (that is, compares equal to 0), the assert macro

- writes information about the particular call that failed [...] on the standard error stream in an implementation-defined format.
- It then calls the **abort** function.”

Pitfall:

- If macro **NDEBUG** is **defined** when including **<assert.h>**, **expression is not evaluated** (thus should be side-effect free).

# *abort* (7.20.4.1)

---

```
1 #include <stdlib.h>
2
3 void abort();
```

- “The abort function causes abnormal program termination to occur, unless [...]”
- [...] An implementation-defined form of the status unsuccessful termination is returned to the host environment by means of the function call `raise(SIGABRT)`.”

(→ Core Dumps)

# Common Patterns with `assert`

---

- **State Invariants:**

“at **this** program point, the value of  $p$  must not be NULL”

“at **all** program points, the value of  $p$  must not be NULL”

(cf. **sequence points** (Annex C))

- **Data Invariants:**

“the value of  $n$  must be the length of  $s$ ”

- **(Function) Pre/Post Conditions:**

Pre-Condition: the parameter must not be 0

Post-Condition: the return value is 10 divided by the parameter

- **Loop Invariants:**

“the value of  $i$  is between 0 and array length minus 1”

# State Invariants with `<assert.h>`

---

```
1 void f() {
2     int* p = (int*)malloc(sizeof(int));
3
4     if (!p)
5         return;
6
7     assert(p); // assume p is valid from here
8     // ...
9 }
10
11 void g() {
12     Node* p = find('a');
13
14     assert(p); // we inserted 'a' before
15     // ...
16 }
```

# Data Invariants with `<assert.h>`

---

```
1  typedef struct {
2      char* s;
3      int n;
4  } str;
5
6  str* construct( char* s ) {
7      str* x = (str*)malloc( sizeof(str) );
8      // ...
9      assert( (x->s == NULL && x->n == -1)
10             || (x->n == strlen( x->s ) ) );
11 }
```

# Pre/Post Conditions with `<assert.h>`

---

```
1  int f( int x ) {
2      assert( x != 0 ); // pre-condition
3
4      int r = 10/x;
5
6      assert( r == 10/x ); // post-condition
7
8      return r;
9  }
```

# Loop Invariants with `<assert.h>`

```
1 void f( int a[], int n ) {
2     int i = 0;
3
4     // holds before the loop
5     assert( 0 <= i && i <= n );
6     assert( i < 1 || a[i-1] == 0 );
7
8     while ( i < n ) {
9         // holds before each iteration
10        assert( 0 <= i && i <= n );
11        assert( i < 1 || a[i-1] == 0 );
12
13        a[i++] = 0;
14    }
15    // holds after exiting the loop
16    assert( 0 <= i && i <= n );
17    assert( i < 1 || a[i-1] == 0 );
18
19    return;
20 }
```

# Old Variables, Ghost Variables

---

```
1 void xorSwap( unsigned int* a, unsigned int* b ) {
2 #ifndef NDEBUG
3     unsigned int *old_a = a, *old_b = b;
4 #endif
5     assert( a && b ); assert( a != b ); // pre-condition
6
7     *a = *a + *b;
8     *b = *a - *b;
9     *a = *a - *b;
10
11     assert( *a == *old_b && *b == *old_a ); // post-con-
12     assert( a == old_a && b == old_b ); // dition
13 }
```



# Outlook

---

- Some verification tools simply verify for each assert statement:  
When executed, `expression` is not false.
- Some verification tools support sophisticated requirements specification languages like ACSL with explicit support for
  - pre/post conditions
  - ghost variables, old values
  - data invariants
  - loop invariants
  - ...

# *Dependable Verification (Jackson)*

# *Dependability*

---

- “**The program has been verified.**” tells us

# *Dependability*

---

- “The program has been verified.” tells us **not very much**.

# *Dependability*

---

- “**The program has been verified.**” tells us **not very much**.
- One wants to know (and should state):

# *Dependability*

---

- “**The program has been verified.**” tells us **not very much**.
- One wants to know (and should state):
  - **Which specifications** have been considered?

# Dependability

---

- “**The program has been verified.**” tells us **not very much**.
- One wants to know (and should state):
  - **Which specifications** have been considered?
  - Under **which assumptions** was the verification conducted?
    - Platform assumptions: finite words (size?), mathematical integers, ...
    - Environment assumptions, input values, ...

Assumptions are often implicit, “**in the tool**”!

# Dependability

---

- “**The program has been verified.**” tells us **not very much**.
- One wants to know (and should state):
  - **Which specifications** have been considered?
  - Under **which assumptions** was the verification conducted?
    - Platform assumptions: finite words (size?), mathematical integers, ...
    - Environment assumptions, input values, ...

Assumptions are often implicit, “**in the tool**”!
- And **what does verification mean** after all?
  - In some contexts: **testing**.
  - In some contexts: **review**.
  - In some contexts: **model-checking** procedure.  
(“We verified the program!” – “What did the tool say?” – “Verification failed.”)
  - In some contexts: **model-checking tool claims correctness**.



# *Common Errors*

# *Distinguish*

---

Most **generic errors** boil down to:

- specified but **unwanted behaviour**,  
e.g. under/overflows
- **initialisation issues**  
e.g. automatic block scope objects
- **unspecified behaviour** (J.1)  
e.g. order of evaluation in some cases
- **undefined behaviour** (J.2)
- **implementation defined behaviour** (J.3)

# Conformance (4)

---

- “A program that is
  - correct in all other aspects,
  - operating on correct data,
  - containing **unspecified behavior**

shall be a correct program and act in accordance with 5.1.2.3. (Program Execution)

- A conforming program is one that is acceptable to a conforming implementation.
- Strictly conforming programs are intended to be maximally portable among conforming implementations.
- An implementation [of C, a compiler] shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.

# *Over- and Underflows*

# Over- and Underflows, Casting

---

- Not specific to C...

```
1 void f( short a, int b ) {
2     a = b; // typing ok, but...
3 }
4
5 short a; // provisioning, implicit cast
6 if (++a < 0) { /* no */ }
7
8 if (++i > MAX_INT) {
9     /* no */ }
10
11
12 int e = 0;
13
14 void set_error() { e++; }
15 void clear_error() { e = 0; }
16
17 void g() { if (e) { /* ... */ } }
```

## *Initialisation (6.7.8)*

# Initialisation (6.7.8)

---

- “If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.”

```
1 void f() {  
2     int a;  
3  
4     printf( "%i\n" , a ); // surprise ...  
5 }
```

## *Unspecified Behaviour (J.1)*



# Unspecified Behaviour (J.1)

---

Each implementation (of a compiler) documents how the choice is made.

## For example

- whether two string literals result in distinct arrays (6.4.5)
- the order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2)
- the layout of storage for function parameters (6.9.1)
- the result of rounding when the value is out of range (7.12.9.5, ...)
- the order and contiguity of storage allocated by successive calls to `malloc` (7.20.3)
- etc. pp.

```
1  char a[] = "hello", b[] = "hello"; // a == b?  
2  
3  i = 0; f( ++i, ++i, ++i ); // f(1,2,3)?  
4  
5  int g() { int a, b; } // &a > &b ?  
6  
7  int* p = malloc( sizeof(int) );  
8  int* q = malloc( sizeof(int) ); // q > p?
```

## *Undefined Behaviour (J.2)*

## *Undefined Behaviour (3.4.3)*

---

“Behaviour, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.”

“**Possible undefined behaviour ranges from**

- ignoring the situation completely with **unpredictable results**,
- to behaving during **translation or program execution** in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message),
- to terminating a **translation or execution** (with the issuance of a diagnostic message).”

“An example of undefined behaviour is the behaviour on **integer overflow**.”

# *Undefined Behaviour (J.2)*

---

## **More examples:**

- an identifier [...] contains an invalid multibyte character (5.2.1.2)

# *Undefined Behaviour (J.2)*

---

## **More examples:**

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
- an object is referred to outside of its lifetime (6.2.4)

# *Undefined Behaviour (J.2)*

---

## **More examples:**

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
- an object is referred to outside of its lifetime (6.2.4)
- the value of a pointer to an object whose lifetime has ended is used (6.2.4)

# *Undefined Behaviour (J.2)*

---

## **More examples:**

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
- an object is referred to outside of its lifetime (6.2.4)
- the value of a pointer to an object whose lifetime has ended is used (6.2.4)
- conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4)

# *Undefined Behaviour (J.2)*

---

## **More examples:**

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
- an object is referred to outside of its lifetime (6.2.4)
- the value of a pointer to an object whose lifetime has ended is used (6.2.4)
- conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4)
- conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3)



# *Undefined Behaviour (J.2)*

---

## **More examples:**

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
- an object is referred to outside of its lifetime (6.2.4)
- the value of a pointer to an object whose lifetime has ended is used (6.2.4)
- conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4)
- conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3)
- the program attempts to modify a string literal (6.4.5)

# *Undefined Behaviour (J.2)*

---

## **More examples:**

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
- an object is referred to outside of its lifetime (6.2.4)
- the value of a pointer to an object whose lifetime has ended is used (6.2.4)
- conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4)
- conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3)
- the program attempts to modify a string literal (6.4.5)
- an exceptional condition occurs during the evaluation of an expression (6.5)

# *Undefined Behaviour (J.2)*

---

## **More examples:**

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
- an object is referred to outside of its lifetime (6.2.4)
- the value of a pointer to an object whose lifetime has ended is used (6.2.4)
- conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4)
- conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3)
- the program attempts to modify a string literal (6.4.5)
- an exceptional condition occurs during the evaluation of an expression (6.5)
- the value of the second operand of the / or % operator is zero (6.5.5)

# *Undefined Behaviour (J.2)*

---

## **More examples:**

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
- an object is referred to outside of its lifetime (6.2.4)
- the value of a pointer to an object whose lifetime has ended is used (6.2.4)
- conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4)
- conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3)
- the program attempts to modify a string literal (6.4.5)
- an exceptional condition occurs during the evaluation of an expression (6.5)
- the value of the second operand of the / or % operator is zero (6.5.5)
- pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6)

# *Undefined Behaviour (J.2)*

---

## **More examples:**

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
- an object is referred to outside of its lifetime (6.2.4)
- the value of a pointer to an object whose lifetime has ended is used (6.2.4)
- conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4)
- conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3)
- the program attempts to modify a string literal (6.4.5)
- an exceptional condition occurs during the evaluation of an expression (6.5)
- the value of the second operand of the / or % operator is zero (6.5.5)
- pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6)
- An array subscript is out of range [...] (6.5.6)

# Undefined Behaviour (J.2)

---

## More examples:

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
- an object is referred to outside of its lifetime (6.2.4)
- the value of a pointer to an object whose lifetime has ended is used (6.2.4)
- conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4)
- conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3)
- the program attempts to modify a string literal (6.4.5)
- an exceptional condition occurs during the evaluation of an expression (6.5)
- the value of the second operand of the / or % operator is zero (6.5.5)
- pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6)
- An array subscript is out of range [...] (6.5.6)
- the program removes the definition of a macro whose name begins with an underscore and either an uppercase letter or another underscore (7.1.3)

# Undefined Behaviour (J.2)

---

## More examples:

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
- an object is referred to outside of its lifetime (6.2.4)
- the value of a pointer to an object whose lifetime has ended is used (6.2.4)
- conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4)
- conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3)
- the program attempts to modify a string literal (6.4.5)
- an exceptional condition occurs during the evaluation of an expression (6.5)
- the value of the second operand of the / or % operator is zero (6.5.5)
- pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6)
- An array subscript is out of range [...] (6.5.6)
- the program removes the definition of a macro whose name begins with an underscore and either an uppercase letter or another underscore (7.1.3)
- etc. pp.

# *Null-Pointer*

---

```
1  int main() {  
2      int *p;  
3      *p = 27;  
4      return 0;  
5  }
```



# Null-Pointer

---

```
1  int main() {  
2      int *p;  
3      *p = 27;  
4      return 0;  
5  }
```

- “An integer constant expression with the value 0, or such an expression cast to type `void*`, is called a **null pointer constant**. [...]”
- “The macro **NULL** is defined in `<stddef.h>` (and other headers) as a null pointer constant; see 7.17.”
- “Among the invalid values for dereferencing a pointer by the unary `*` operator are a null pointer, [...]” (6.5.3.2)

# Segmentation Violation

---

```
1  int main() {
2      int *p = (int*)0x12345678;
3      *p = 27;
4
5      *((int*)((void*)p) + 1) = 13;
6      return 0;
7  }
```

# Segmentation Violation

---

```
1  int main() {
2      int *p = (int*)0x12345678;
3      *p = 27;
4
5      *((int*)((void*)p) + 1) = 13;
6      return 0;
7  }
```

- Modern operating systems provide **memory protection**.
- Accessing memory which the process is not allowed to access is observed by the operating system.
- Typically an instance of “accessing an object outside its lifetime”.
- **But:** other way round does not hold, accessing an object outside its lifetime does not imply a segmentation violation.

# Segmentation Violation

---

```
1  int main() {
2      int *p = (int*)0x12345678;
3      *p = 27;
4
5      *((int*)((void*)p) + 1) = 13;
6      return 0;
7  }
```

- Modern operating systems provide **memory protection**.
- Accessing memory which the process is not allowed to access is observed by the operating system.
- Typically an instance of “accessing an object outside its lifetime”.
- **But:** other way round does not hold, accessing an object outside its lifetime does not imply a segmentation violation.
- Some platforms (e.g. SPARC): unaligned memory access, i.e. outside word boundaries, not supported by hardware (“bus error”).  
Operating system notifies process, default handler: terminate, dump core.

## *Implementation-Defined Behaviour (J.3)*

# *Implementation-Defined Behaviour (J.3)*

---

“A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:”

# *Implementation-Defined Behaviour (J.3)*

---

“A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:”

- J.3.2 Environment, e.g.  
The set of signals, their semantics, and their default handling (7.14).

# *Implementation-Defined Behaviour (J.3)*

---

“A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:”

- J.3.2 Environment, e.g.  
The set of signals, their semantics, and their default handling (7.14).
- J.3.3 Identifiers, e.g.  
The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).



# *Implementation-Defined Behaviour (J.3)*

---

“A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:”

- J.3.2 Environment, e.g.  
The set of signals, their semantics, and their default handling (7.14).
- J.3.3 Identifiers, e.g.  
The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).
- J.3.4 Characters, e.g.  
The number of bits in a byte (3.6).

# *Implementation-Defined Behaviour (J.3)*

---

“A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:”

- J.3.2 Environment, e.g.  
The set of signals, their semantics, and their default handling (7.14).
- J.3.3 Identifiers, e.g.  
The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).
- J.3.4 Characters, e.g.  
The number of bits in a byte (3.6).
- J.3.5 Integers, e.g.  
Any extended integer types that exist in the implementation (6.2.5).

# *Implementation-Defined Behaviour (J.3)*

---

“A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:”

- J.3.2 Environment, e.g.  
The set of signals, their semantics, and their default handling (7.14).
- J.3.3 Identifiers, e.g.  
The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).
- J.3.4 Characters, e.g.  
The number of bits in a byte (3.6).
- J.3.5 Integers, e.g.  
Any extended integer types that exist in the implementation (6.2.5).
- J.3.6 Floating Point, e.g.  
The accuracy of the floating-point operations [...] (5.2.4.2.2).

# Implementation-Defined Behaviour (J.3)

---

“A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:”

- J.3.2 Environment, e.g.  
The set of signals, their semantics, and their default handling (7.14).
- J.3.3 Identifiers, e.g.  
The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).
- J.3.4 Characters, e.g.  
The number of bits in a byte (3.6).
- J.3.5 Integers, e.g.  
Any extended integer types that exist in the implementation (6.2.5).
- J.3.6 Floating Point, e.g.  
The accuracy of the floating-point operations [...] (5.2.4.2.2).
- J.3.7 Arrays and Pointers, e.g.  
The result of converting a pointer to an integer or vice versa (6.3.2.3).

# Implementation-Defined Behaviour (J.3)

---

“A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:”

- J.3.2 Environment, e.g.  
The set of signals, their semantics, and their default handling (7.14).
- J.3.3 Identifiers, e.g.  
The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).
- J.3.4 Characters, e.g.  
The number of bits in a byte (3.6).
- J.3.5 Integers, e.g.  
Any extended integer types that exist in the implementation (6.2.5).
- J.3.6 Floating Point, e.g.  
The accuracy of the floating-point operations [...] (5.2.4.2.2).
- J.3.7 Arrays and Pointers, e.g.  
The result of converting a pointer to an integer or vice versa (6.3.2.3).
- etc. pp.

# *Locale and Common Extensions (J.4, J.5)*

---

- J.4 Locale-specific behaviour
- J.5 Common extensions
  - “The following extensions are widely used in many systems, but are not portable to all implementations.”

# *References*

---

[ISO, 1999] ISO (1999). Programming languages – C. Technical Report ISO/IEC 9899:1999, ISO. Second edition, 1999-12-01.