Dynamic Memory

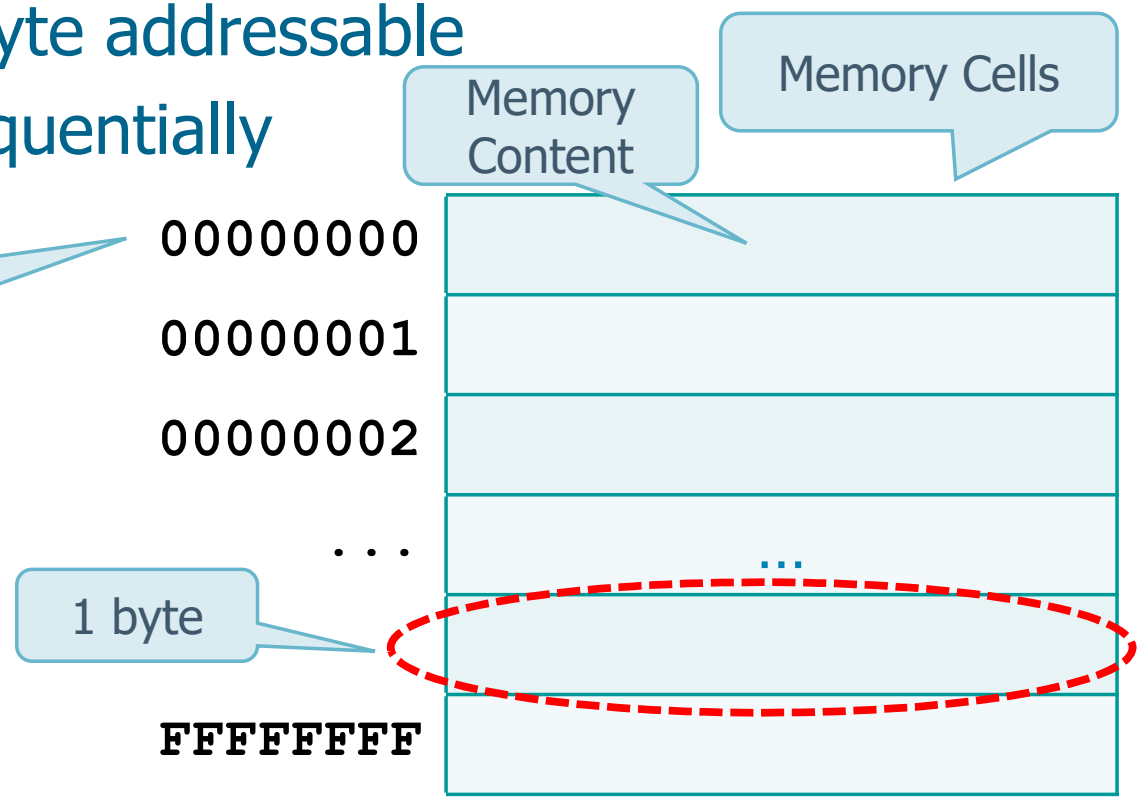
Memory and Pointers

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Memory

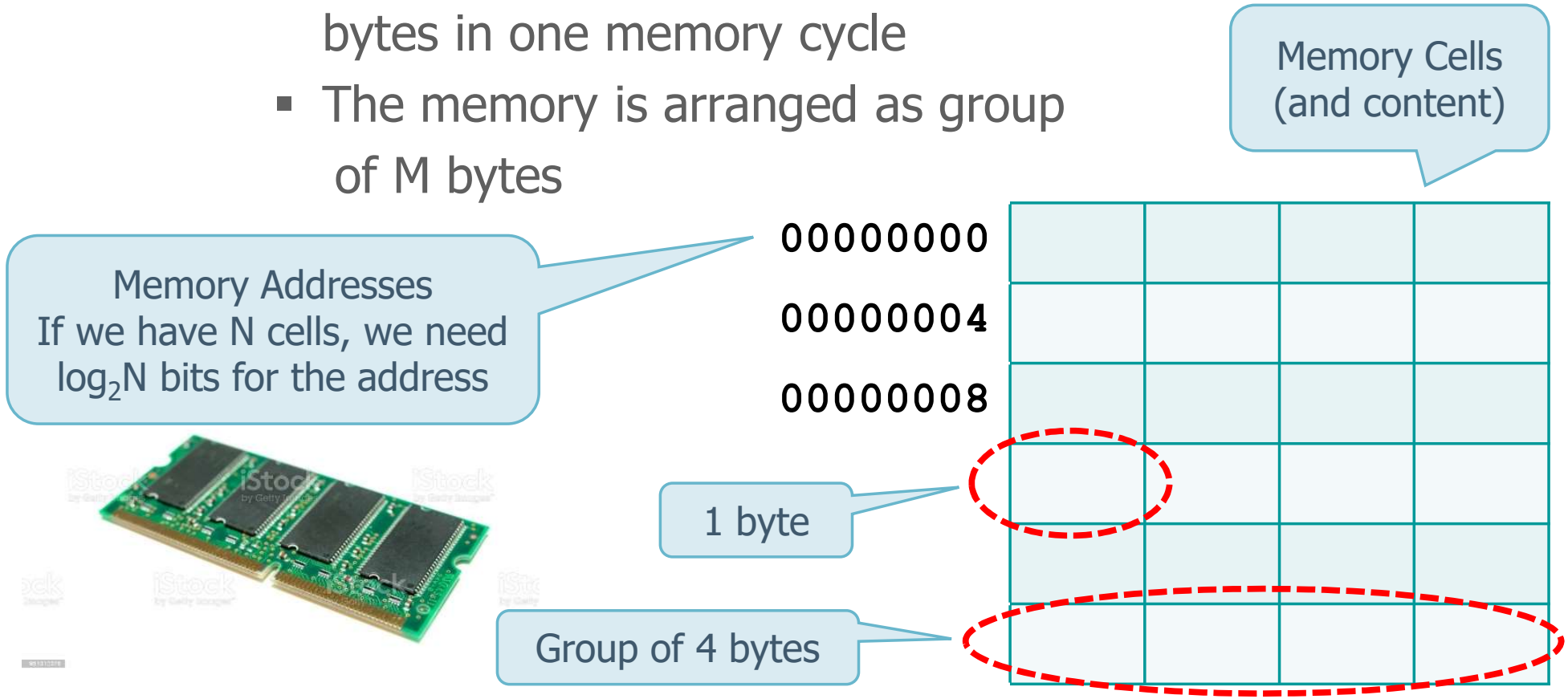
- A computer memory (RAM) can be seen as a sequence of cells
  - A physical memory address is a reference to a specific memory cell
  - The memory is byte addressable and arranged sequentially

Memory Addresses
If we have N cells, we need $\log_2 N$ bits for the address



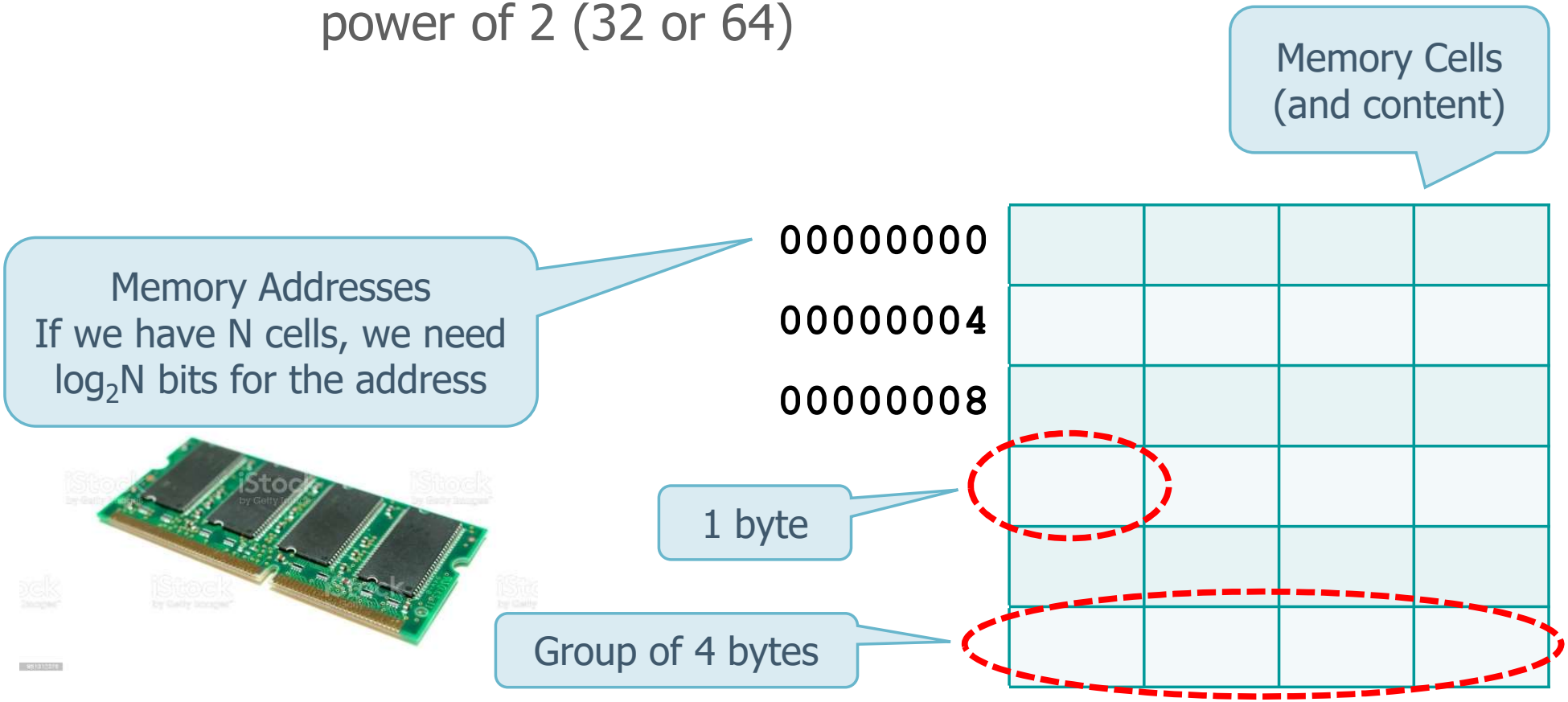
Memory

- If the memory was arranged as cells of one byte width, the processor would need to issue 4 memory read cycles to fetch an integer
  - It is more economical to read/write more than 1 bytes in one memory cycle
  - The memory is arranged as group of M bytes



Memory

- Each group of M cells is called a (memory) **word**
  - Each read/write operation manipulates an entire word
  - Usually M is equal to 4 or 8 (words of 32 or 64 bits)
- The number of bits N to address them is usually a power of 2 (32 or 64)



Variables

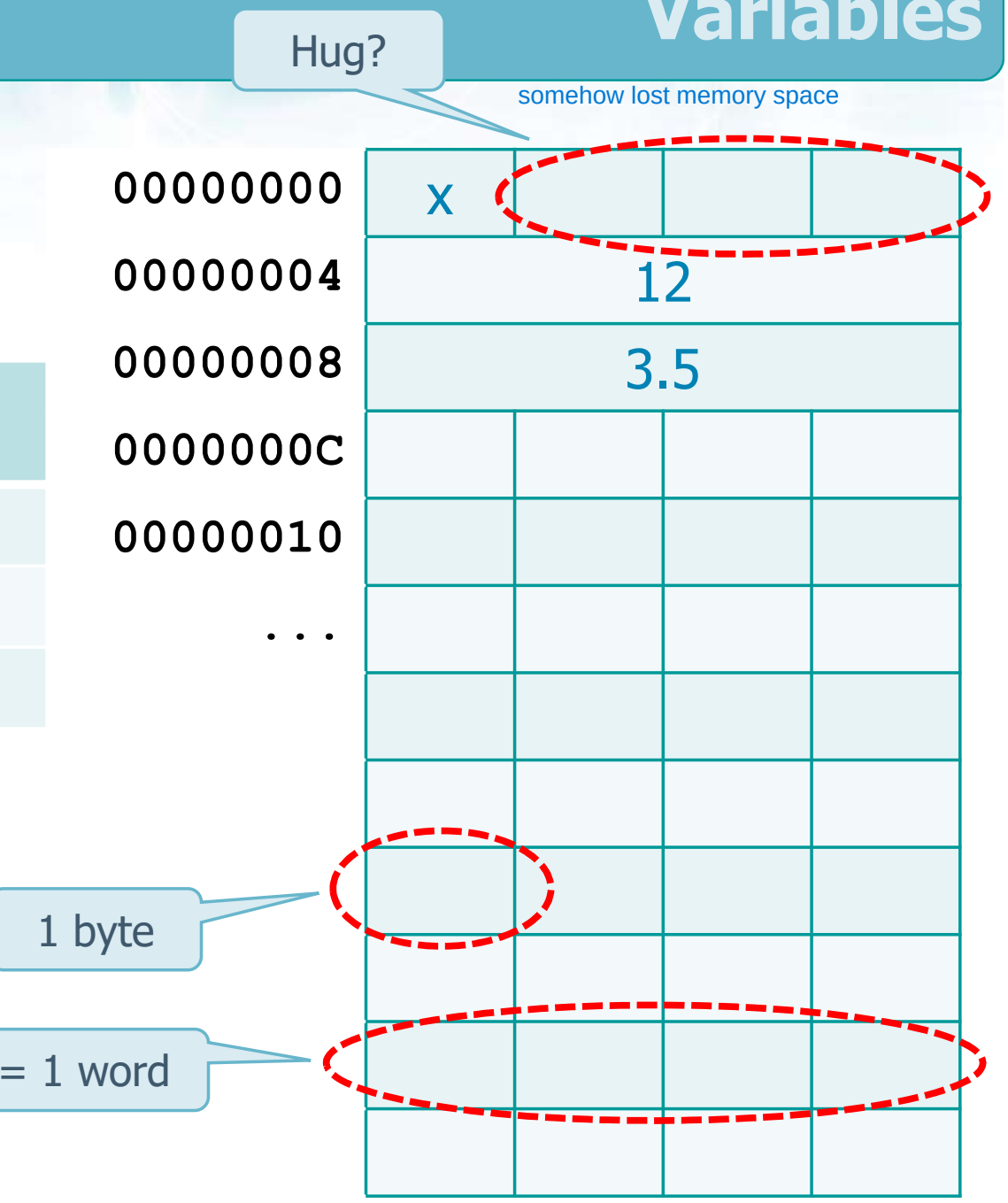
- Variable names correspond to locations in the computer's memory
  - A variable is a name of a memory location
    - Variables are human-readable identifiers
    - Addresses are computer-readable identifiers
  - When we declare a variable we define its **name** and **type**
    - For each type the compiler reserves a specific space for its content
      - The compiler does know the total amount of memory required by the program
    - The compiler maintains a **table** to specify the correspondence name-type and memory address

Variables

```
char c = 'x';
int i = 12;
float f = 3.5;
```

Variable Name	Type	Address
c	Char	00000000
i	Int	00000004
f	Float	00000008

Compiler table



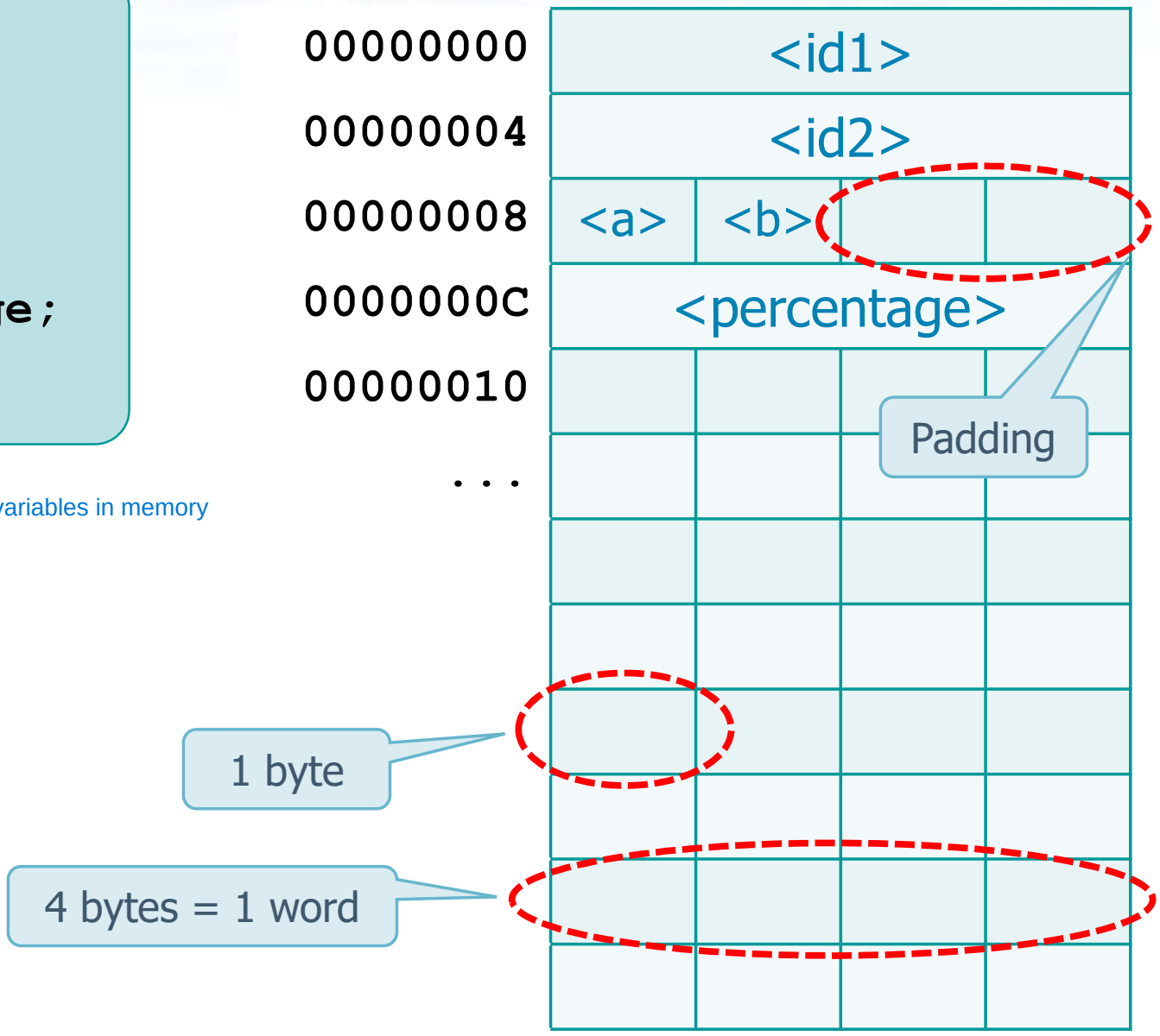
Variables

- Variables span a certain number of bytes starting at a particular memory location (address)
  - Variables often need more than one byte
  - Standard words are usually longer than one byte
  - Variables can be stored on only a part of a single word, on an entire word, or on more than one word depending on the variable and the cell length
- Data type in C have alignment requirements
  - **Padding** aligns variables to **natural** address boundaries

Variables

```
struct student1 {
   int id1;
   int id2;
   char a;
   char b;
   float percentage;
};
```

the compiler chooses how to allocate the variables in memory



Pointers

- Pointers are variables whose values are memory addresses
- They are often used in C to manipulate objects
  - A variable directly contains a specific value
  - A pointer contains an address of a variable that contains a specific value
  - Referencing a value through a pointer is called **indirection**
- In C, pointers are an alternative way (to names) to manipulate objects

Pointers

- Pointers, like all variables, must be defined before they can be used

```
<type> *<pointer>;
<type>* <pointer>;
<type> * <pointer>;
```

<pointer> is a pointer to an object of type <type>

*<pointer> is of type <type>

- As for variables, we can defined more than pointer on the same line

```
<type> *<p1>, *<p2>, *<p3>;
```

you can also define an array of pointers

Pointers

- Pointers can be used through 2 operators
  - The **address** operator, *
    - Returns the value of the object to which its operand (i.e., a pointer) points
  - The **indirection** operator, &
    - Given a variable, it takes its address

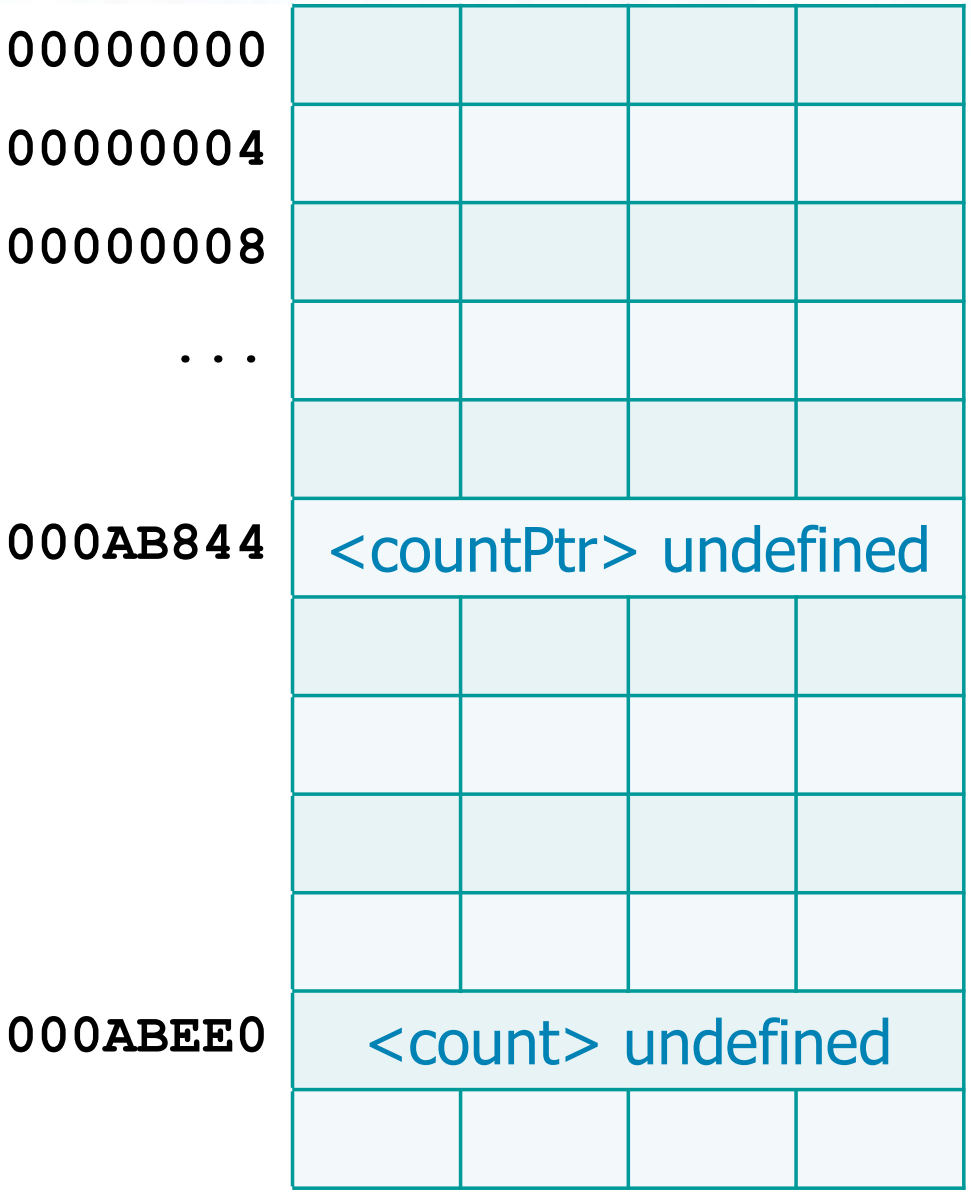
Example

countPtr is a pointer to an integer

```
int *countPtr, count;
countPtr = &count;
```

Variable Name	Type	Address
countPtr	Int *	000AB844
count	Int	000ABEE0

Pointers are addresses to **specific** types.
The type specifies the length of the area references

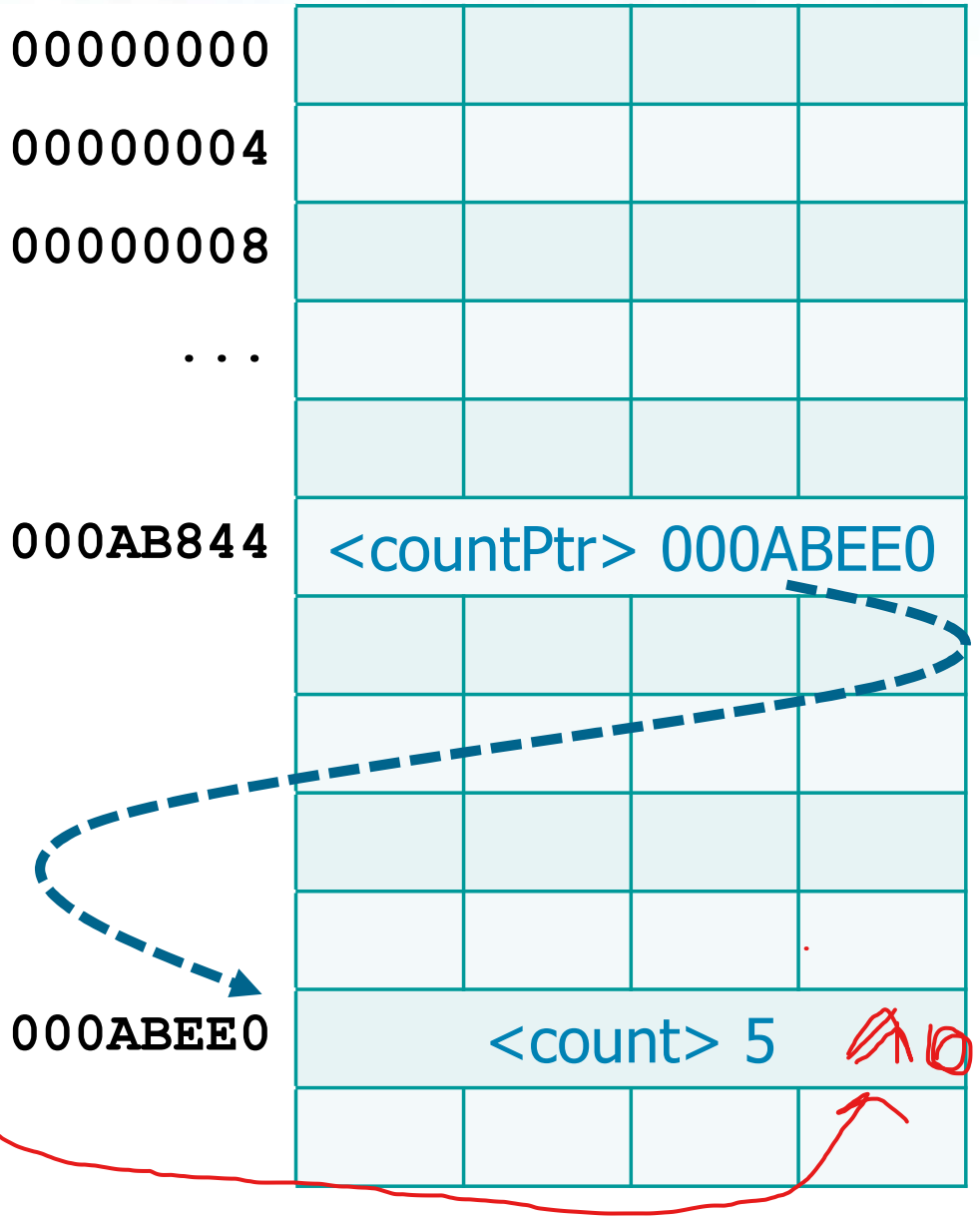


Example

```
int *countPtr, count;
countPtr = &count;
count = 5;
*countPtr = 10;
```

Variable Name	Type	Address
countPtr	Int *	000AB844
count	Int	000ABEE0

Pointers are addresses to **specific** types.
The type specifies the length of the area references



NULL pointers

- Pointers should be initialized (they should be assigned a value) during or after their definition
  - If they are **not** initialized, they are undefined, i.e., they can point to anything
  - If they are initialized, they can be assigned to NULL, to 0, or to an address
  - A pointer with the value NULL points to nothing
  - NULL is defined in <stdio.h>

Void pointers

- A generic pointer type is indicated with void *
  - It is not associated to any type of data
    - It is a generic pointer which can be converted to any type

implicit cast

```
int i, j;
void *pv;
...
pv = &i;
...
j = *pv;
```

explicit cast

```
int i, j;
void *pv;
...
pv = (void *) &i;
...
j = (int) *pv;
```

  - Many functions return **void** pointers to be generic

Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions, assignments, and comparisons
- The set of operations is limited
  - A pointer may be incremented or decremented
  - An integer may be added to or subtracted from a pointer
    - When an integer is added to or subtracted from a pointer, the pointer is **not** incremented or decremented simply by that integer, but by that integer times the size of the object to which the pointer refers
  - The relational operators can be used in specific sircustances

Example

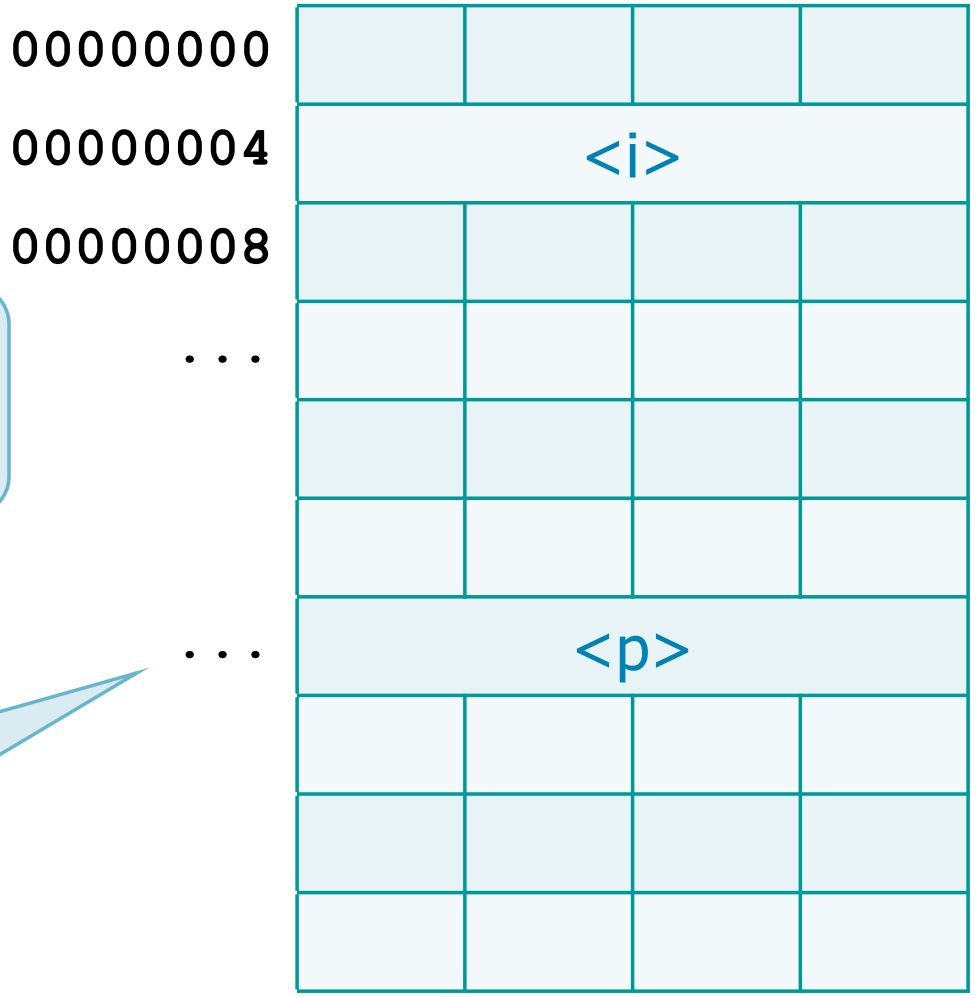
```
int i = 10;
int *p1;
int *p2;
```

Definitions and initializations

```
p1 = &i;
p1++;
```

After p1++, p1 points to the integer placed after the integer i

After int *p1, p1 is undefined
After p1=&i, p1 is 00000004
After p1=&i, p1 is 00000008



Example

```
int i = 10;
int j = 20;
int *p1;
int *p2;
p1 = &i;
p2 = &j;
```

Definitions and initializations

```
if (*p1 == *p2) { ... }
```

Check whether the referenced values are the same even if they are placed in different position within the system memory

```
if (p1 == p2) { ... }
```

Check whether the two pointers refer to the same object, i.e., they store the same memory address

```
if (p1 > p2) { ... }
```

Check whether the address p1 comes after the address p2 into the system memory; this is often meaningless
If p1==p2 also *p1==*p2

Pointers and parameters

- Pointers are obviously used to pass parameters by reference

Variable exchange

```
int i; j;
...
swap (i, j);
```

```
int i; j;
...
swap (&i, &j);
```

to manipulate the original you need to refer to the pointer

```
void swap (int x, int y) {
  int tmp;

  tmp = x;
  x = y;
  y = tmp;

  return;
}
```

Wrong

```
void swap (int *x, int *y) {
  int tmp;

  tmp = *x;
  *x = *y;
  *y = tmp;

  return;
}
```

Correct

Pointers and arrays

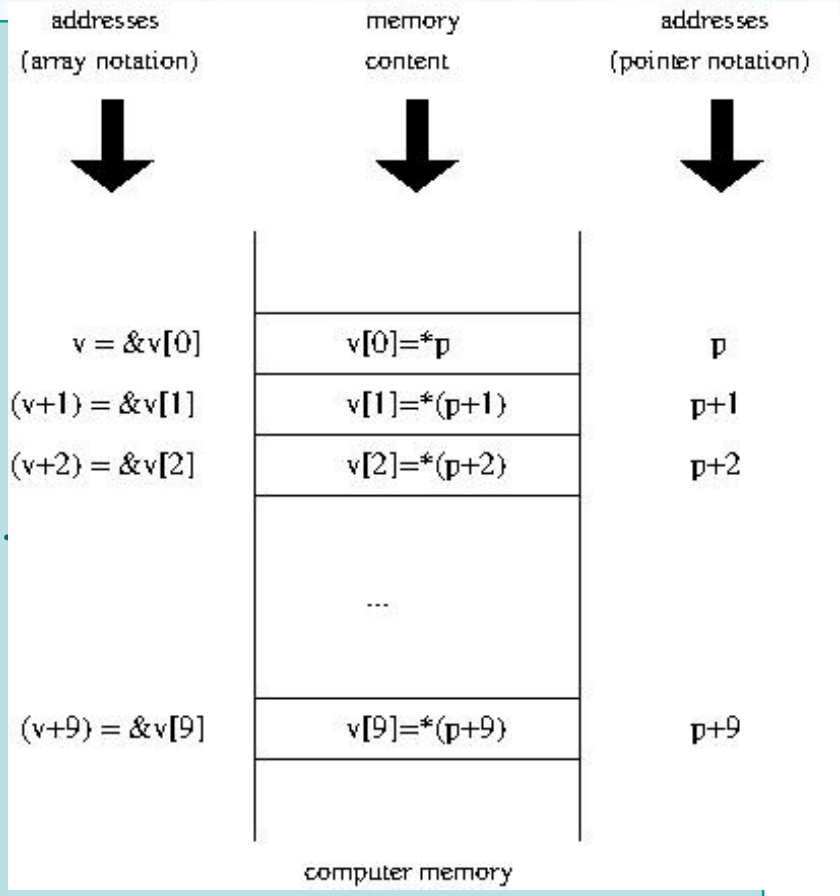
- Arrays and pointers are intimately related
- An array name can be thought of as a constant pointer

```
                    int v[N];
                        ...
v ⇔ &v[0]                       *v ⇔ v[0]
v+i ⇔ &v[i]                     *(v+i) ⇔ v[i]
```

- Pointers can be used to do any operation involving array sub-scripting
  - Moreover, the array name (without a subscript) is a pointer to the first element of the array

Examples

```
#define L ...
int v[L];
int *p;
for (i=0; i<L; i++) {
  scanf ("%d", &v[i]);
  printf ("%d", v[i]);
}
for (i=0, p=v; i<L; i++, p++) {
  scanf ("%d", p);
  printf ("%d", *p);
}
p = &v[0];
for (i=0; i<L; i++) {
  scanf ("%d", (p+i));
  printf ("%d", *(p+i));
}
```



Pointers and strings

- In C strings are null terminated arrays of characters
  - The last element is '\0'
- Thus, the relationship between pointers and strings is similar to the one between pointers and arrays
  - Strings can be manipulated using the array notation, the pointer notation and their arithmetic, and the string library

Example

```
int strlen (char str[]) {
  int cnt;
  cnt =0;
  while (str[cnt] != '\0')
    cnt++;
  return cnt;
}
```

```
int strlen (char *str) {
  int cnt;
  cnt = 0;
  while (*str != '\0') {
    cnt++;
    str++;
  }
  return cnt;
}
```

```
int strlen (char str[]) {
  int cnt;
  char *p;
  cnt =0;
  p = &s[0];
  while (*p != '\0') {
    cnt++;
    p++;
  }
  return cnt;
}
```

```
int strlen (char *str) {
  char *p;
  p = str;
  while (*p != '\0') {
     p++;
  }
  return (p - str);
}
```

Pointers and structures

- To access the members of a structur via a pointer C provides the member access operator **->**
- If a pointer variable is assigned the address of a structure, then a member of the structure can be accessed by a construct such as

```
pointer_to_structure->member_name
(*pointer_to_structure).member_name
```

The brackets are necessary the operators . and -> have the highest precedence and associate from left to right. Thus, the previous construct without brackets would be equivalent to
*(pointer_to_structure.member_name)
This is an error because only a structure can be used with the . operator, not a pointer to a structure

Example

```
struct student {
  char s1[L], s2[L];
  int i;
  float f;
};
```

```
struct student v;
...
my_read (&v);
...
```

```
void my_read (struct student *v) {
  char s1[DIM], s2[DIM];
  int i; float f;
  fprintf (stdout, "...: ");
  scanf ("%s%s%d%d", s1, s2, &i, &f);
  strcpy (v->s1, s1);
  strcpy (v->s2, s2);
  v->i = i;
  v->f = f;
  return;
}
```

Example

```
struct student {
  char s1[L], s2[L];
  int i;
  float f;
};
```

```
struct student v;
...
read (&v);
...
```

```
void my_read (struct student *v) {
  fprintf (stdout, "...: ");
  scanf ("%s%s%d%d", v->s1, v->s2, &v->i, &v->f);
  return;
}
```