Bitwise operations (1)

Operations like addition, comparison, logical *AND*, etc. operate with bytes. C/C++ has also operations for handling bits. The operands of bitwise operations must be integers (*char*, *int*, *unsigned int*, etc.).

Bitwise negation ~ converts each bit 1 to bit 0 and each bit 0 to bit 1. Example:

```
unsigned char c1 = 0xA5; // bits are 1010 0101
printf("%u\n", (unsigned int) c1); // prints 165
unsigned char c2 = ~c1; // get 0101 1010
printf("%u\n", (unsigned int)c2); // prints 90
```

Remember that there is also negation ! (logical *NOT*) that converts zero (*FALSE*) to 1 (*TRUE*) and any non-zero (*TRUE*) to 0 (*FALSE*).

Bitwise AND & performs bit-by-bit comparison of bits. If the both bits are 1, the resulting bit is also 1, otherwise 0. Example:

```
unsigned char c1 = 0xA5, c2 = 0x20; // bits are 1010 0101 and 0010 0000
printf("%u %u\n", (unsigned int) c1, (unsigned int) c2); // prints 165 32
unsigned char c3 = c1 & c2; // gets 0010 0000
printf("%u\n", (unsigned int)c3); // prints 32
```

Remember that there is also logical AND && in which *TRUE && TRUE = TRUE* and all the other combinations produce *FALSE*.

Bitwise operations (2)

Bitwise OR | performs bit-by-bit comparison of bits. If the both bits are not 0, the resulting bit is 1, otherwise 0. Example:

```
unsigned char c1 = 0xA5, c2 = 0x20; // bits are 1010 0101 and 0010 0000
printf("%u %u\n", (unsigned int) c1, (unsigned int) c2); // prints 165 32
unsigned char c3 = c1 | c2; // gets 1010 0101
printf("%u\n", (unsigned int)c3); // prints 165
```

Remember that there is also logical OR || in which *FALSE* || *FALSE* = *FALSE* and all the other combinations produce *TRUE*.

Bitwise exclusive OR ^ (XOR) performs bit-by-bit comparison of bits. If the both bits are different, the resulting bit is 1, otherwise 0. Example:

```
unsigned char c1 = 0xA5, c2 = 0x20; // bits are 1010 0101 and 0010 0000
printf("%u %u\n", (unsigned int) c1, (unsigned int) c2); // prints 165 32
unsigned char c3 = c1 ^ c2; // gets 1000 0101
printf("%u\n", (unsigned int)c3); // prints 133
```

Bitwise operations (3)

Applying bits instead of bytes we can compress data. Suppose we need to describe properties of a file:

- reading allowed or not alllowed
- writing allowed or not allowed
- on open, if not found, create; if found inform about error
- on open, if found, delete the existing contents or keep it
- …………………………

Suppose there is no more that 8 properties. Then we may pack this information into one byte:

- If bit 7 is 1, reading is allowed; if 0, not allowed
- If bit 6 is 1, writing is allowed; if 0 not, not allowed
- If bit 5 is 1, create the file if not found; if 0 consider that file open operation failed
- If bit 4 is 1, destroy the contents of existing files; if 0 keep it
- ……………………………………………………………

Here bit 7 is the highest (leftmost) bit.

So, the function opening file does not need 9 parameters (filename and properties). 2 is enough – the name of file and properties packed into a variable of type *unsigned char*.

Bitwise operations (4)

Now suppose we have

```
unsigned char properties = 0;
```

and we want to open file both for reading and writing. For that we need to set bits 7 and 6 to 1:

```
properties = properties | 0xC0; // we may write also properties |= 0xC0;
// 0000 0000 | 1100 0000 gives us 1100 0000
```

Next we want to set that if the file exists, its contents must be destroyed:

```
properties |= 0x10; // 1100 0000 | 0001 0000 = 1101 0000
```

So, if we want to set a bit in the target variable to 1, we must bitwise *OR* the target with a constant in which this bit is 1 and all the others are 0. If the bit in the target variable already was 1, it keeps its value. If it was 0, it becomes 1.

The function opening the file must analyse the properties, i.e. to clarify which bits are 0 and which are 1. It can be done with bitwise AND, for example:

```
if (properties & 0x10)
{ // we get 0001 0000 that is TRUE or 0000 0000 that is FALSE
  ………………… // destroy file contents
}
```

So, if we need to know is a bit in the target variable 0 or 1, we must bitwise *AND* the target with a constant in which this bit is 1 and all the others are 0.

Bitwise operations (5)

If we want to set a bit in the target variable to 0, we must bitwise *AND* the target with a constant in which this bit is 0 and all the others are 1. If the bit in the target variable already was 0, it keeps its value. If it was 1, it becomes 0. Example:

```
unsigned char target = 0xD0; // 1101 0000
unsigned char mask = 0xEF; // 1110 1111
target = target & mask; // 1100 0000
```

or

```
target &= mask;
```

Toggling a bit means that if it was 1, it must be converted to 0 and if it was 0, it must be converted to 1. For that we have to bitwise *XOR* the target with a constant in which this bit is 1 and all the others are 0. Example:

```
unsigned char target = 0xD0; // 1101 0000
```

Toggle bit 4:

```
unsigned char mask = 0x10; // 0001 0000
target = target ^ mask; // 1100 0000
```

Toggle once more:

```
target = target ^ mask; // 1101 0000
```

or

```
target ^= mask;
```

Bitwise operations (6)

Binary bitwise shifting left << operation shifts all the bits of the value of left operand to the left by the number of places given by the right operand. The vacated places are filled with zeroes. Example:

```
unsigned char c1 = 0xA5; // bits are 1010 0101
printf("%u\n", (unsigned int) c1); // prints 165
unsigned char c2 = c1 << 5; // gets 1010 0000, the higher bits were lost
printf("%u\n", (unsigned int)c2); // prints 160
unsigned char c3 = c1 << 8; // gets 0000 00000
printf("%u\n", (unsigned int)c3); // prints 0
```

Binary bitwise shifting right >> operation shifts all the bits of the value of left operand to the right by the number of places given by the right operand. The vacated places are filled with zeroes. Example:

```
unsigned char c1 = 0xA5; // bits are 1010 0101
printf("%u\n", (unsigned int) c1); // prints 165
unsigned char c2 = c1 >> 5; // gets 0000 0101, the lower bits were lost
printf("%u\n", (unsigned int)c2); // prints 5
unsigned char c3 = c1 >> 8; // gets 0000 00000
printf("%u\n", (unsigned int)c3); // prints 0
```

Shifting of negative signed values leads to unpredictable results.

Bitwise operations (7)

Example:

```
unsigned int color; // the higher byte is not used, the following bytes present the intensity
                    // of red, green and blue components. For example 0x00FF0000
                    // presents the most intensive red, 0x00FF00FF the most intensive
                    // magenta, 0x00007F00 dark green
unsigned char mask = 0xFF;
unsigned int red = (color >> 16) & mask;
unsigned int green = (color >> 8) & mask;
unsigned int blue = color & mask;
```

If the color is 0x00AA0000 (dark red) or 0000 0000 1010 1010 0000 0000 0000 0000, then shifting right 16 positions gives us 0000 0000 0000 0000 0000 0000 1010 1010. Before bitwise *AND* mask is automatically converted to unsigned int, so we get

```
 0000 0000 0000 0000 0000 0000 1010 1010
&
 0000 0000 0000 0000 0000 0000 1111 1111
  ----------------------------------------
0000 0000 0000 0000 0000 0000 1010 1010 // intensity of red
```

Bitwise operations (8)

If the color is 0x00AA8000 (light brown) or 0000 0000 1010 1010 1000 0000 0000 0000, then shifting right 8 positions gives us 0000 0000 0000 0000 1010 1010 1000 0000. Before bitwise *AND* mask is automatically converted to unsigned int, so we get

```
 0000 0000 0000 0000 1010 1010 1000 0000
&
 0000 0000 0000 0000 0000 0000 1111 1111
  -----------------------------------
 0000 0000 0000 0000 0000 0000 1000 0000 // intensity of green
```

Bit fields (1)

Bit fields is the alternative way to handle separate bits. Suppose we want to store the parameters of font for a section of text. The font may bold, italic, underlined or double underlined or any combination of them. We may use a variable of type *unsigned char* and agree that bit 3 (7 is the highest) is 1 if the text is bold and 0, if not. Similarly bit 2 is 1 if the text is in italic and 0 if not, etc. But it is unpleasant to remember the meaning of each bit. The corresponding bit field may be as follows:

```
struct {
   unsigned char bold: 1;
   unsigned char italics: 1;
   unsigned char single_underlined: 1;
   unsigned char double_underlined: 1;
} font_par;
```

Now we can handle each bit as a member of struct, for example:

```
font_par.bold = 0;
font_par.italics = 1;
font_par. single_underlined = 1;
font_par.double_underlined = 0;
```

Description *unsigned char italics : 1* tells that the type of member *italics* is *unsigned char* but one bit is enough for storing its value.

Bit fields (2)

Number of bits for a bit field member may be greater than 1. For example in a date day cannot exceed 31 (0001 1111), month cannot exceed 12 (0000 1100) and the year cannot exceed 2020 (0111 1110 0100). So to economize the memory usage we can define

```
struct Date {
unsigned char day : 5;
unsigned char month : 4;
unsigned short int year : 11;
};
```

Bit field members can be integers, but not arrays or pointers.

Variable number of arguments (1)

```
#include "stdarg.h"
```

The prototype of such functions should have a parameter list with at least one parameter followed by three points, for example

```
double mean(int n, …);
double mean(int n, double x, …);
int printf(char *, …);
void fun(…); // error, no parameters
void fun(int n, …, double x); // error, points must be at the end of paremeter list
```

The last of the fixed parameters must in some way present the actual number of arguments represented by points. For example:

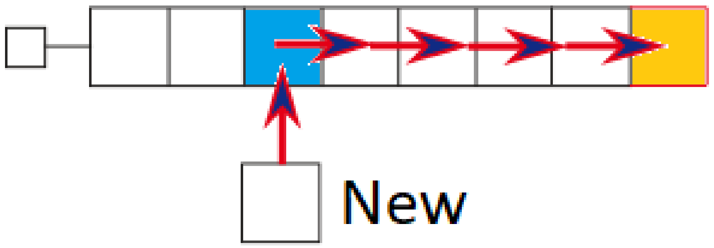
```
double result = mean(5, 1.0, 2.0, 3.0, 4.0, 5.0);
    // here the points are replaced by a list of 5 arguments
printf("%d %d\n", x1, x2);
     // here the format string contains two "%d" format specifyers, so the points are
     // replaced by a list of two integers
```

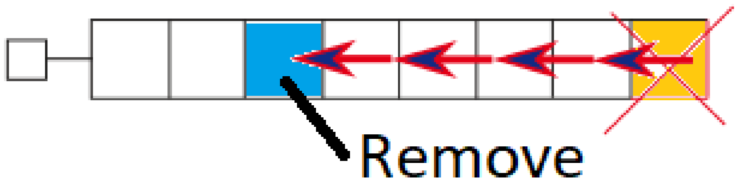
Variable number of arguments (2)

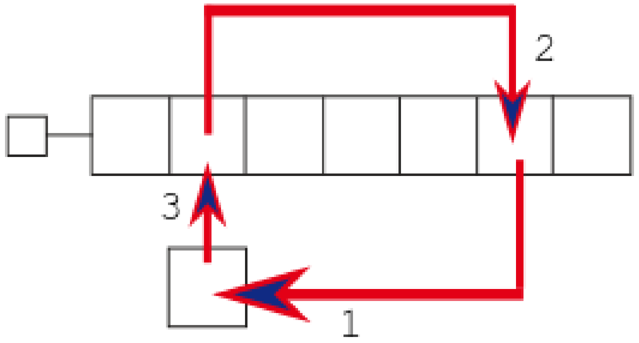
```
double mean(int n, ...)
{
  va_list cursor; // va_list is a typedef from stdarg.h
  va_start(cursor, n); // here we set the cursor to the beginning of variable argument list,
                       // it starts after input parameter n
  double sum = 0;
  for (int i = 0; i < n; i++)
 {
      sum += va_arg(cursor, double);
                      // va_arg returns, one after another, the arguments from list
                      // its second parameter is to specify the type of argument
  }
  va_end(cursor); // closes the list, cleans up everything
  return sum / n;
}
```

Linear data structures (1)

In linear data structures the elements (integers, pointers, structs, etc.) are ordered – i.e. we may say, which member is the first, which is the seconds, etc. The simplest ordered linear data structure is array.







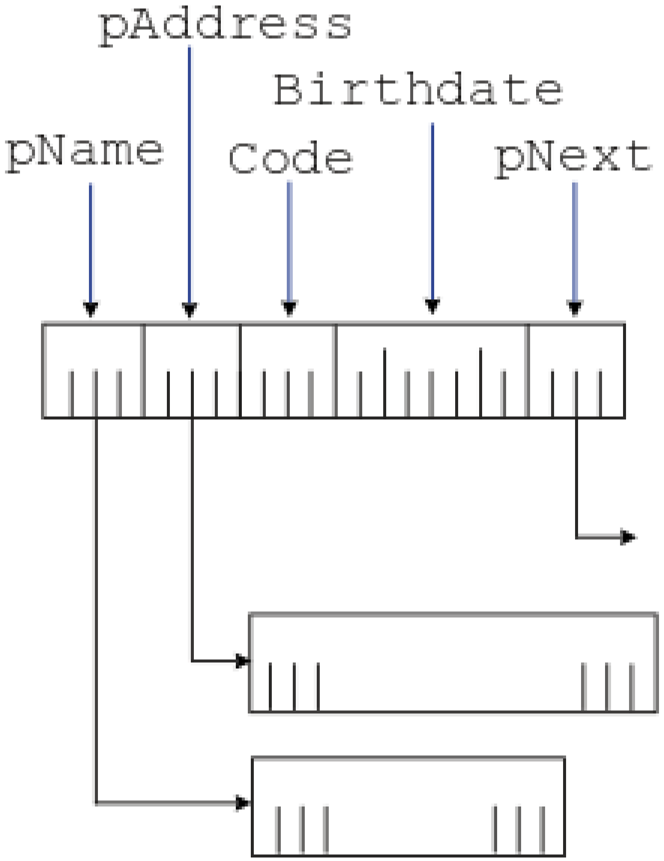
Problems with array:

1. If we need to insert a new element, we must at first check, is there at the end of array some free space. If not, we have to reallocate our memory field. Often it is accompanied with relocating of large amounts of data. After that we need to free the position for the new element: i.e. once more shift data.
2. If we need to remove an element, we must shift data to left to cover the position. One position at the end of array becomes unused. There is an alternative solution: do not shift but somehow mark that the position as empty (for example fill with zeroes).

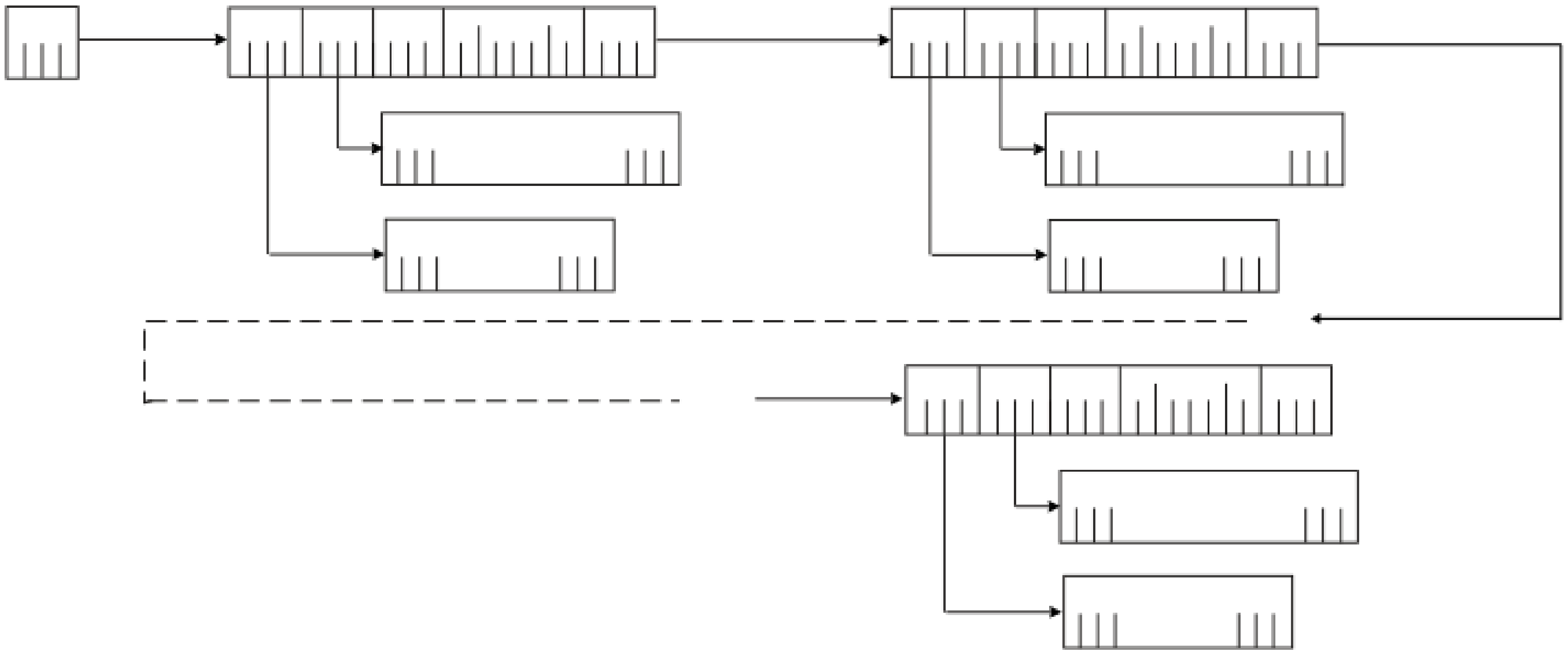
Linear data structures (2)

Let us have:

```
struct date
{
  short int Day;
  char Month[4]; // like "Jan", "Feb", etc.
  short int Year;
}
typedef struct date DATE;
struct person
{
  char *pName,
          *pAddress;
  long int Code;
  DATE Birthdate;
  struct Person *pNext;
}
typedef struct person PERSON;
```



Linear data structures (3)



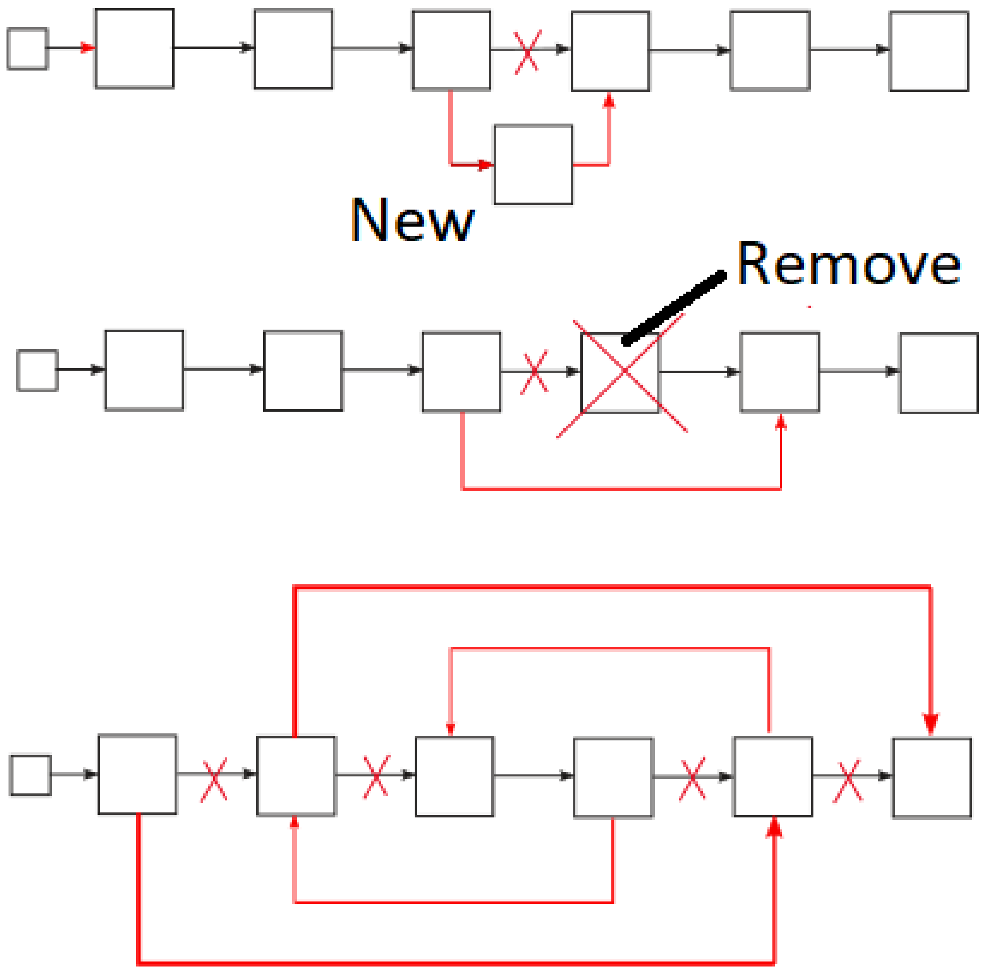
With pointer *pNext* we create linked list:

PERSON *pList; // points to the first element

Pointer *pNext* of the first element points to the second element, pointer *pNext* of the second element points to the third element, etc. Pointer *pNext* of the last element is zero.

The linked list does not need a long compact memory field. The elements may be in the heap higgledy-piggledy, without any order. But due to the pointers the data structure itself is ordered.

Linear data structures (4)

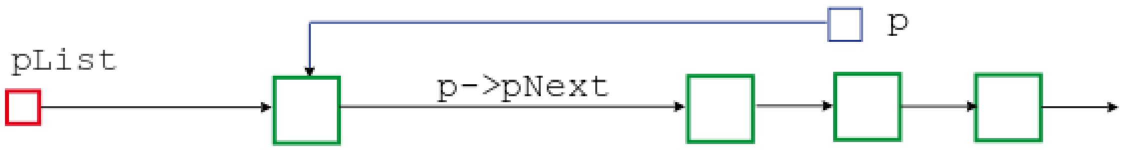


Inserting a new element and removing an existing element is much more effective than those operations with arrays. We do not need to shift large amounts of data and all the elements of list keep their current location. The only task we need to perform is to reset the *pNext* pointers.

The disadvantage of linked list is that we cannot use indeces. To access the i-th element we have to move from the first element (this is the only element we can access directly) to second, from the second to the third, etc. In an array we need just write like _*(pArray +i)_ or *Array[i]*.

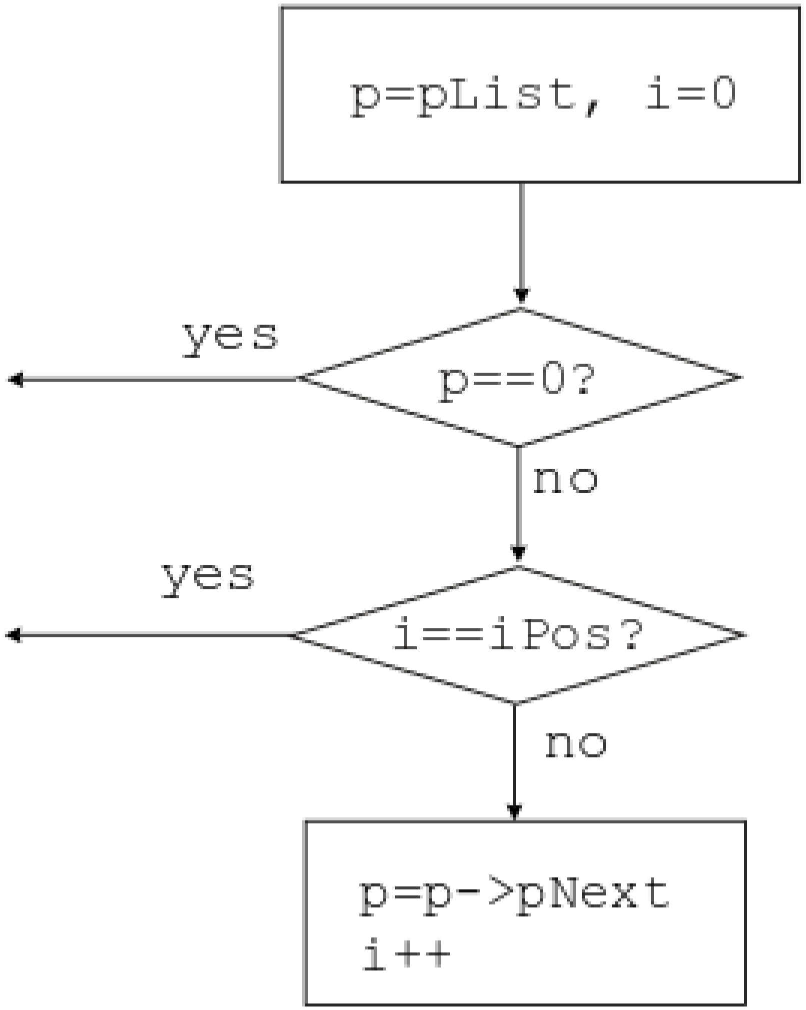
In data processing linked lists are the most used type of linear data structures. Arrays are used only when the amount of data is not large and the expected max number of data is well known. If the number of elements is unpredictable and continually changing, the linked lists have no alternatives.

Linear data structures (5)



Example: iteration through linked list

```
PERSON *GetPerson(PERSON *pList, int iPos)
{ // we want to get the pointer to item on position iPos
  if (!pList || iPos < 0) // check input
     return 0; // errors
  PERSON *p; // auxiliary variable
  int i; // auxiliary variable
  for (i = 0, p = pList;
        p && i < iPos;
        p = p->pNext, i++);
   return p;
}
```

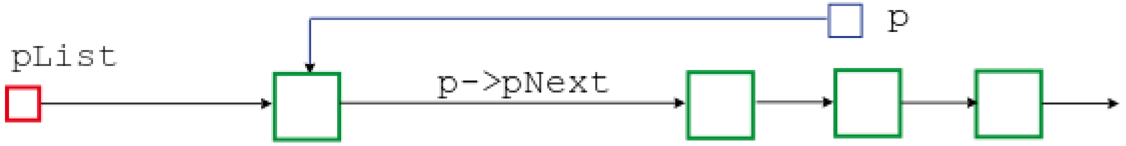


Linear data structures (6)

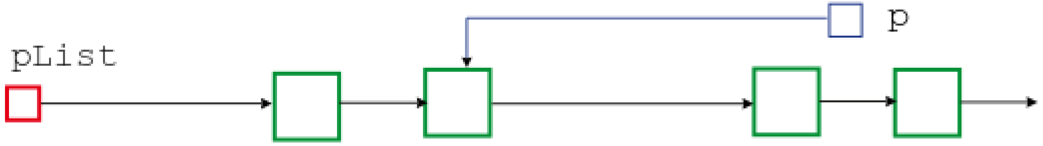
Suppose *iPos* is 2, i.e. we want to get the pointer to third item.

Loop starts: p = pList, i = 0;
p points to the first item.

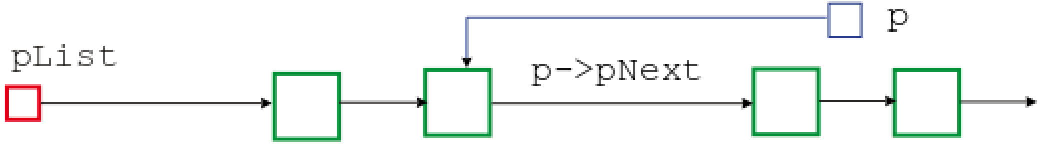
As *p* is not zero and *i < 2*, looping continues. *p->pNext* is the address of second item.



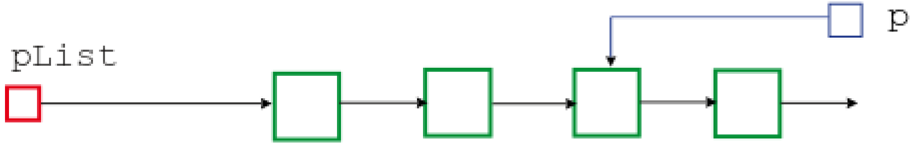
p = p->pNext, i++; After that *p* points to the second item and *i* is 1.



As *p* is not zero and *i < 2*, looping continues. *p->pNext* is the address of third item.

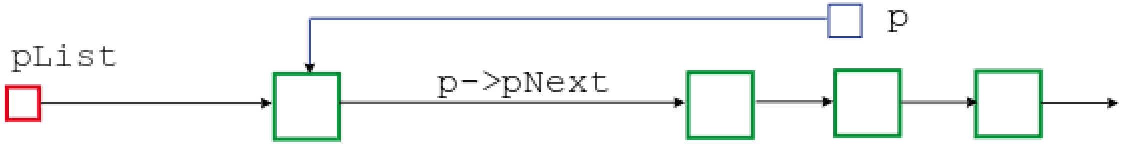


p = p->pNext, i++; After that *p* points to the third item and *i* is 2.



As *i* is now 2, the looping breaks off and we may return *p* as the searching result.

Linear data structures (7)



Example: iteration through linked list

```
PERSON *GetPerson(PERSON *pList, char *pKey)
{ // we want to get the pointer to person with name specified by the key
  if (!pList || !pKey) // check input
      return 0;
  PERSON *p; // auxiliary variable
  for (p = pList;  p && strcmp(pKey, p->pName); p = p->pNext);
  // strcmp() compares two strings. If they are identical, the return value is 0.
  // The iteration stops when p points to item with name identical with key or when
  // p is zero (i.e. the item we need does not exist)
   return p;
}
```

A key is something (string, integer, etc.) that we can directly or after some calculations retrieve from the record. Requirements: there must be algorithms with which we can assert that:

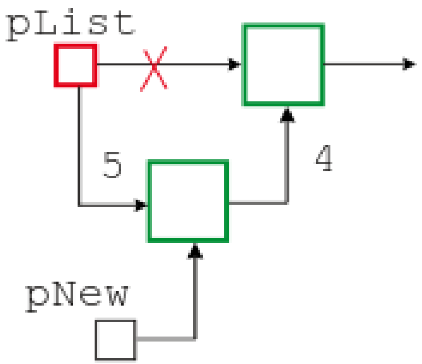
- two keys are equal
- if they are not equal, which of them is less

Linear data structures (8)

Example: insert into linked list

```
PERSON *Insert(PERSON *pList, PERSON *pNew, int iPos)
{ // we want to insert a new item into position iPos.
   // the function returns the pointer to first item
  if (!pNew || iPos < 0) // error in input
    return pList;
  if (!iPos)
 { // insert to the beginning, the new item will be the first one
    pNew->pNext = pList;
    return pNew;
  }
  PERSON *p; // auxiliary variable
  if (p = GetPerson(pList, iPos - 1))
 { // insert into the middle or to the end
    pNew->pNext = p->pNext;
    p->pNext = pNew;
  }
   return pList;
}
```

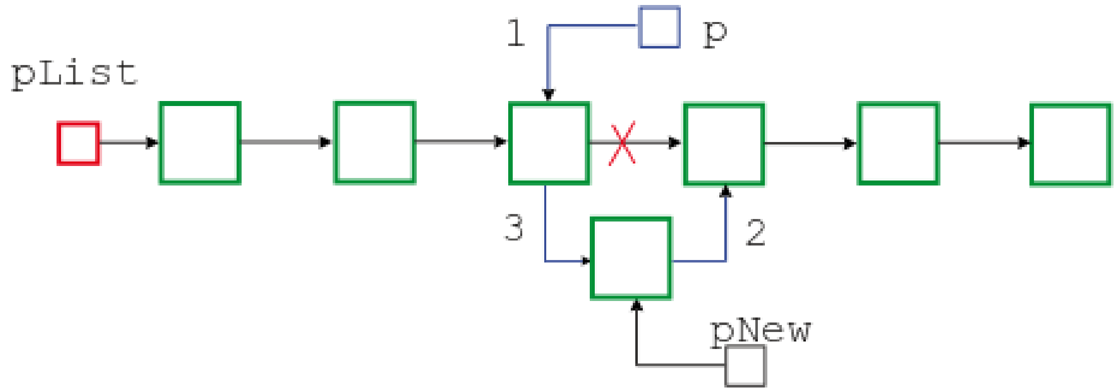
Linear data structures (9)



```
if (!iPos)
{ // insert to the beginning, the new item will be the first one
   pNew->pNext = pList; // 4
   return pNew; // 5
 }
```

On start *pList* points to the first item. As *iPos* is zero, the previous first item must be reduced to the second position. So the *pNext* member of the new item must point to the former first item (operation 4). The return value is the pointer to the new first item (operation 5).

Linear data structures (10)



```
if (p = GetPerson(pList, iPos - 1))// 1
{
    pNew->pNext = p->pNext; // 2
    p->pNext = pNew; // 3
 }
   return pList; // keeps its value
}
```

We want to insert the new item into position *iPos*. Consequently the item on position *iPos – 1* must start to point to the new item. Therefore the first thing to do is to find the pointer to item on position *iPos -1*. For that we may use function *GetPerson()* from slide *Linear data structures (5)* (operation 1). If *iPos* is wrong (negative or too large), *GetPerson()* returns 0 and the inserting will be omitted. If the item on position *iPos - 1* was found, we correct its *pNext* member (operation 3) and set the new item to point to item that was on position *iPos* and now is reduced to position *iPos +1* (operation 2).

Linear data structures (11)

Example: remove from linked list

```
PERSON *Remove(PERSON *pList, int iPos, PERSON **ppResult)
{ // we want to remove the item on position iPos
   // the removed item is not destroyed: the pointer to it is the output value
   // the function returns the pointer to first item
  if (!pList || iPos < 0 || !ppResult)
    return pList; // list is empty or errors in input data
 *ppResult = 0;
  PERSON *p; // auxiliary variable
  if (!iPos)
 {  // remove the first
   *ppResult = pList;
    pList = pList->pNext;
  }
  else if (p = GetPerson(pList, iPos - 1))
 {  // remove from the middle or from the end
     *ppResult = p->pNext;
     p->pNext = p->pNext->pNext;
  }
   return pList;
}
```

Linear data structures (12)

Usage example: we have linked list

```
PERSON *pStudentsGroup;
```

Remove the first and fourth students and print their names.

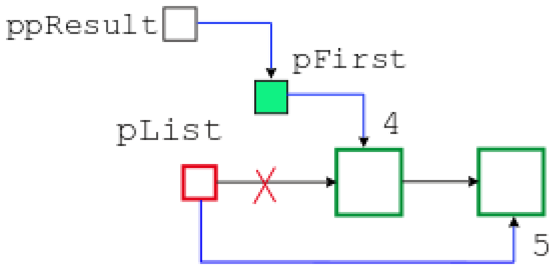
```
PERSON *pFirst, *pFourth;
pStudentGroup = Remove(pStudentGroup, 0, &pFirst);
if (pFirst)
{
  printf("Student %s was removed from list\n", pFirst->pName);
}
pStudentGroup = Remove(pStudentGroup, 4, &pFourth);
if (pFourth)
{
  printf("Student %s was removed from list\n", pFourth->pName);
}
```

On the last call to

```
PERSON *Remove(PERSON *pList, int iPos, PERSON **ppResult) { …… }
```

- the value of *pStudentsGroup* is copied into *pList*
- *iPos* gets value 4
- the pointer to *pFourth* (which itself is also a pointer) is calculated and copied into *ppResult*. In other words, *ppResult* will point to *pFourth*

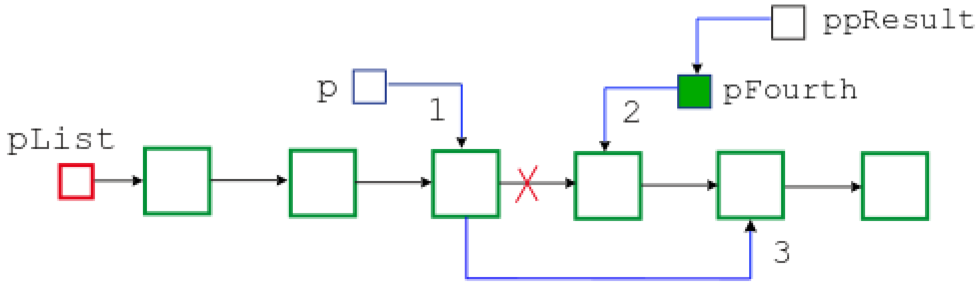
Linear data structures (13)



```
if (!iPos)
{  // remove the first
   *ppResult = pList; // 4
    pList = pList->pNext; // 5
}
```

The second item is now the first and *pList* must point to it (operation 5). To pointer *pFirst* (variable of the calling function and not the variable of *Remove()*) is assigned the pointer to the former first item (operation 4).

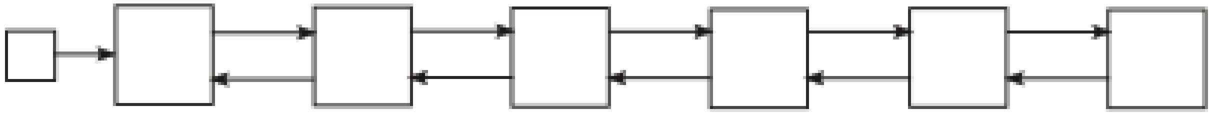
Linear data structures (14)



```
else if (p = GetPerson(pList, iPos - 1)) // 1
{
    *ppResult = p->pNext; // 2
    p->pNext = p->pNext->pNext; // 3
 }
  return pList;
}
```

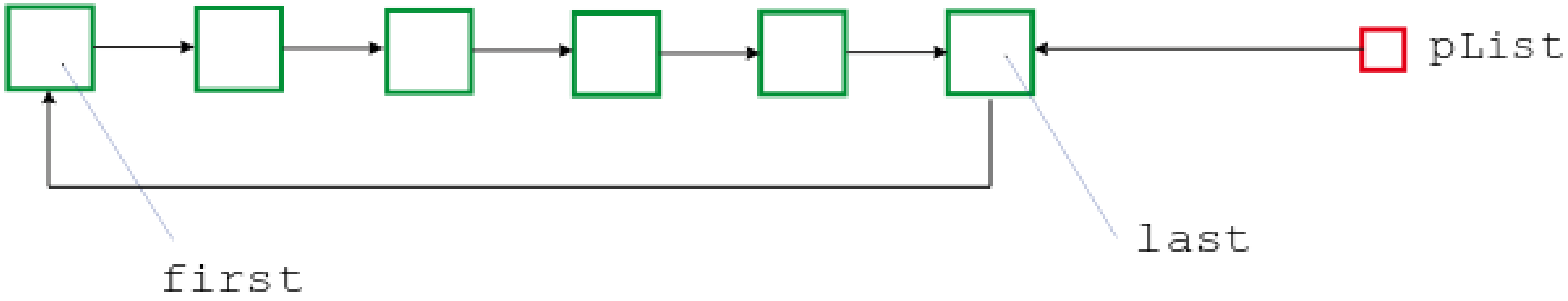
We want to remove the item on position *iPos*. Consequently the item on position *iPos – 1* must start to point to the item that is on position *iPos + 1*. Therefore the first thing to do is to find the pointer to item on position *iPos -1*. For that we may use function *GetPerson()* from slide *Linear data structures (5)* (operation 1). If *iPos* is wrong (negative or too large), *GetPerson()* returns 0 and the removing will be omitted. If the item on position *iPos - 1* was found, we correct its *pNext* member (operation 3). To pointer *pFourth* (variable of the calling function and not the variable of *Remove()*) is assigned the pointer to item that was on position *iPos* (operation 2).

Linear data structures (15)



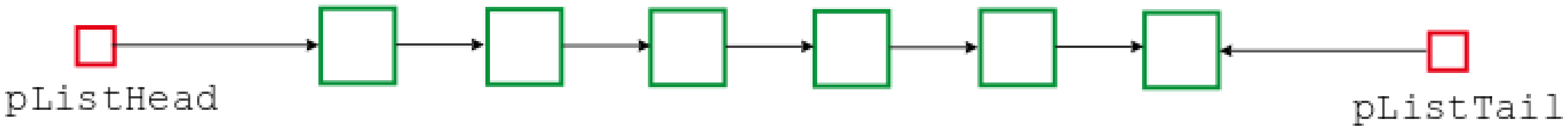
```
struct person
{
  char *pName,
         *pAddress;
  long int Code;
  DATE Birthdate;
  struct Person *pNext,
                      *pPrior;
};
```

In double linked list we can move to both directions. Pointer *pPrior* in the first element is 0.

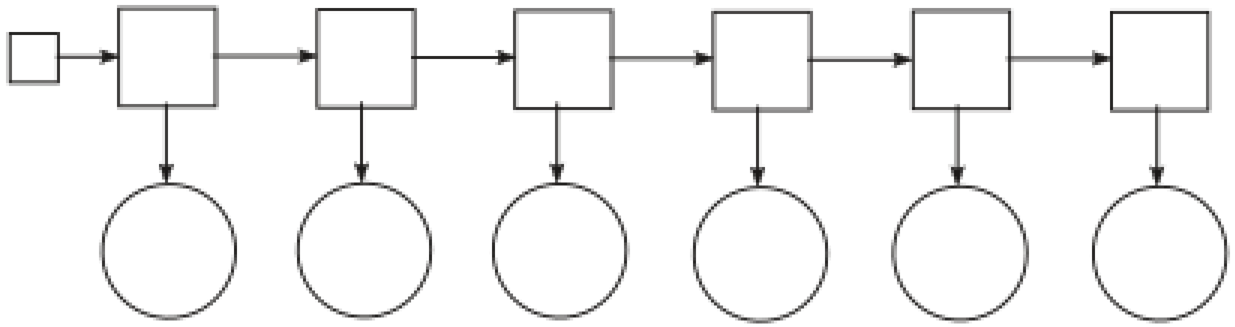


In circularly linked list the "last" element points to the "first" (terms "first" and "last" are conditional here).

Linear data structures (16)



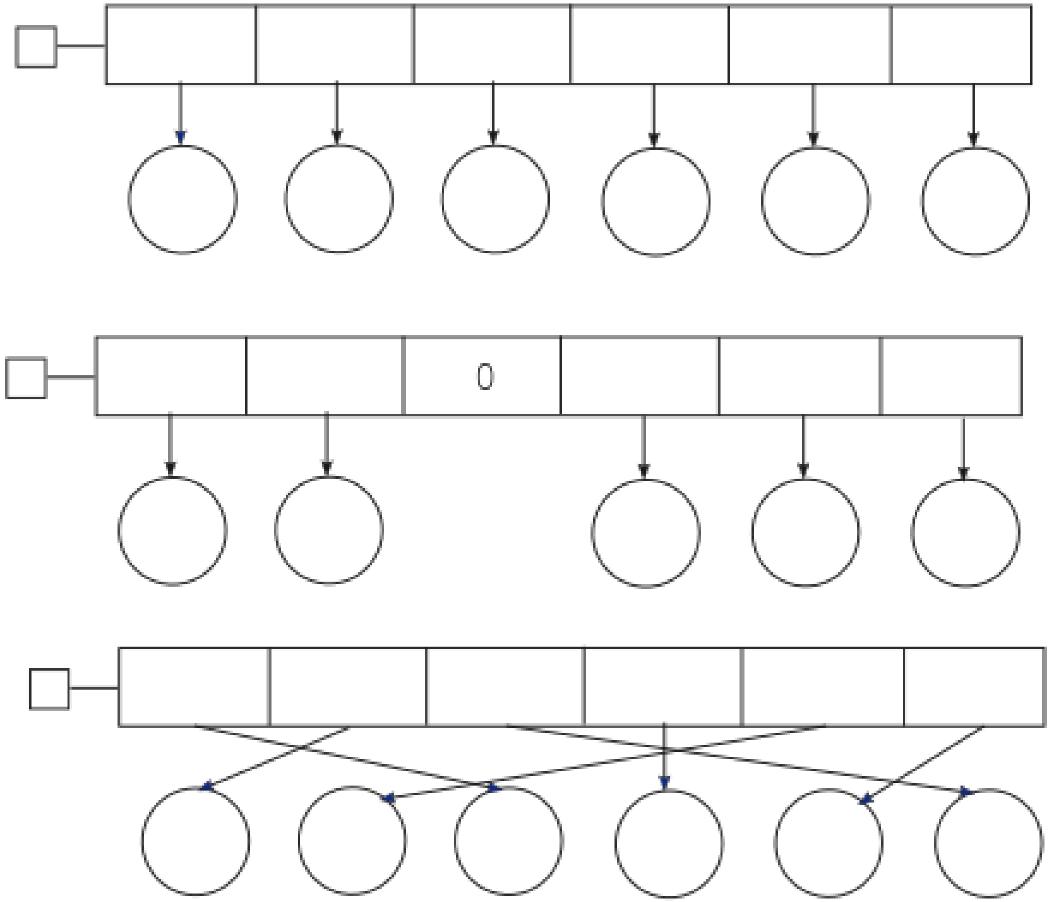
If the new elements must be always appended (and not inserted into the middle of list), it is useful to have 2 outside pointers: one to the first and one to the last element.



```
struct Header
{
  void *pRecord;
   int type;
   struct Header  *pNext;
};
```

The separate headers are needed when the structs in data structure do not have *pNext* pointers or are of different types.

Linear data structures (17)



```
struct Header
{
    void *pRecord;
    int type;
};
```

This solution is very suitable but only if we are able the estimate the number of elements and thus allocate the vector with proper length. When deleting, instead of compressing simply replace the pointer with 0. When sorting, the structs are not moved because we may simply rearrange the pointers.

Unions (1)

The unions enable to store data of different types in the same memory field. Let us have

```
struct sExample {
   int iData;
   double dData;
   char cData;
};
struct sExample struct_example;
```

To store variable *struct_example* we need at least 13 bytes (actually the memory allocation system gives us 16 bytes).

Declaration of an union is very similar:

```
union uExample {
  int iData;
  double dData;
  char cData;
};
union uExample union_example;
```

But to store variable *union_example* we need only 8 bytes, because *dData* is the longest member:

```
union_example.dData = 3.14159; // all the 8 bytes are in use
union_example.iData = 10; // the same 8 bytes, 4 bytes unused
union_example.cData = 'A'; // the same 8 bytes, 7 bytes unused
```

Unions (2)

Example: suppose we want to know how negative integers are stored:

```
union study {
    int number;
    unsigned char uc[4];
};
union study test;
test.number = -10;
for (int i = 0;  i < 4;  i++)
    printf("%02X ", test.uc[i]);
printf("\n");
```

Variable test occupies 4 bytes. Those bytes we handle as an *int*, but after that as an array of *unsigned char*. With this trick we may print out the memory dumpings – the contents of memory field in hex byte-by-byte.

The programmer himself must know which type of data is currently inside union. If, for example, he has stored a double number but handles it as integer, the result is formally OK but actually senseless.

Unions (3)

Example: let us have

```
struct Book {
  const char *pAuthor,
                  *pTitle;
  short int Year;
};
```

```
struct Article { // in a journal
  const char *pAuthor,
                  *pTitle,
                  *pJournal;
  short int Year,
                Number;
};
```

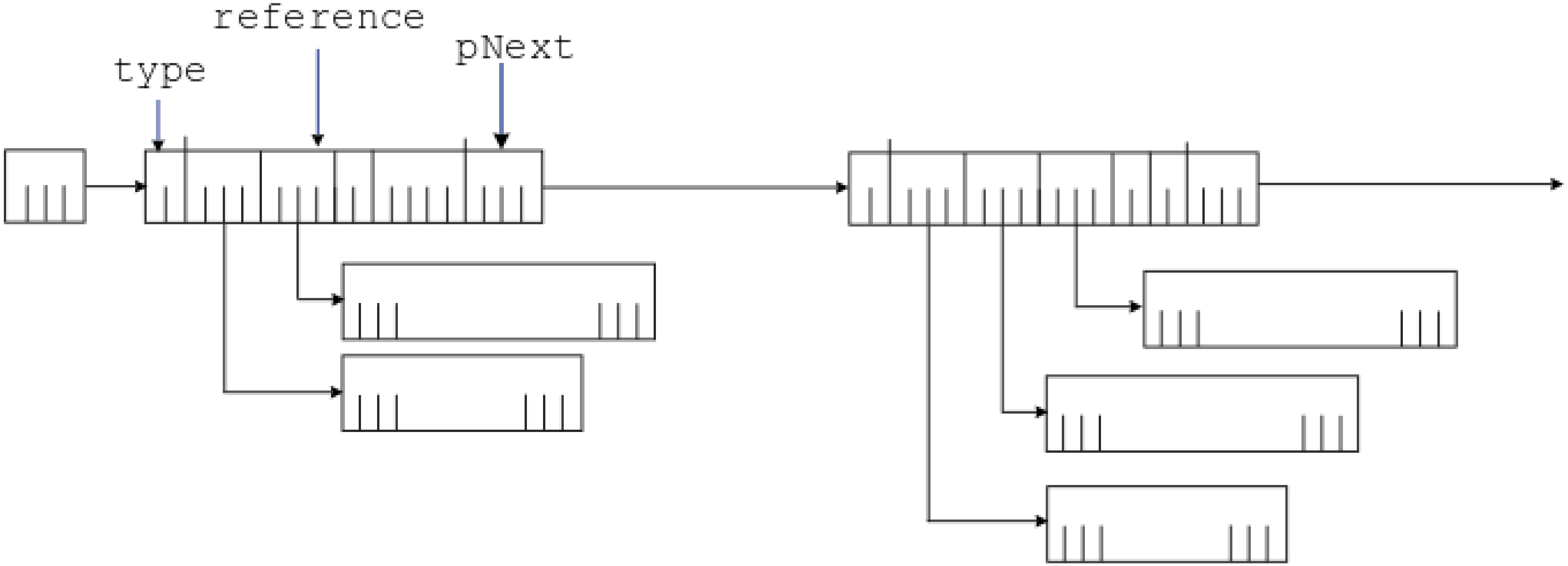
```
union Reference {
  struct Book book;
  struct Article article ;
};
```

```
union Reference Ref1, Ref2;
Ref1.book.pAuthor = "Nicolai M. Josuttis"; // use for storing a book
Ref1.book.pTitle = "The C++ Standard Library";
Ref1.book.year = 2012;
Ref2.article.pAuthor = "Vlodymyr Myrmyy"; // use for storing an article
Ref2.article.pTitle = "A Simple and Efficient FFT Implementation in C++";
Ref2.article.pJournal = "Dr. Dobbs Journal";
Ref2.article.Year = 2007;
Ref2.article.Number = 5;
```

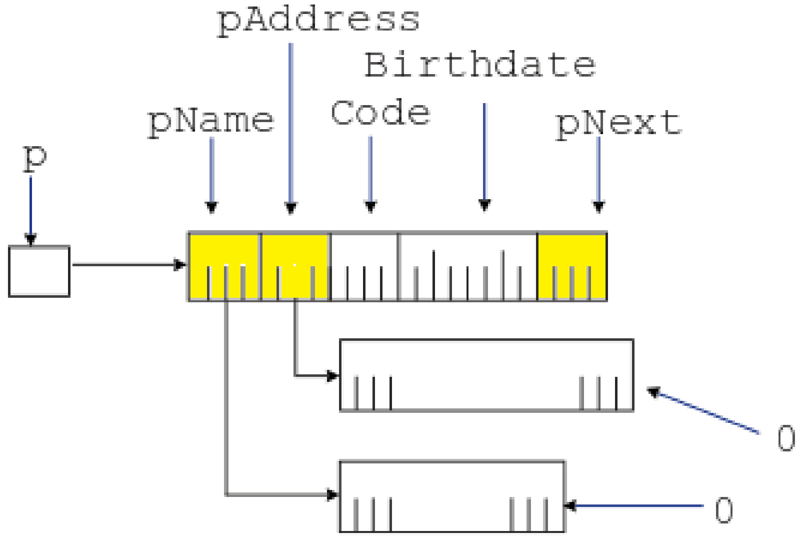
Unions (4)

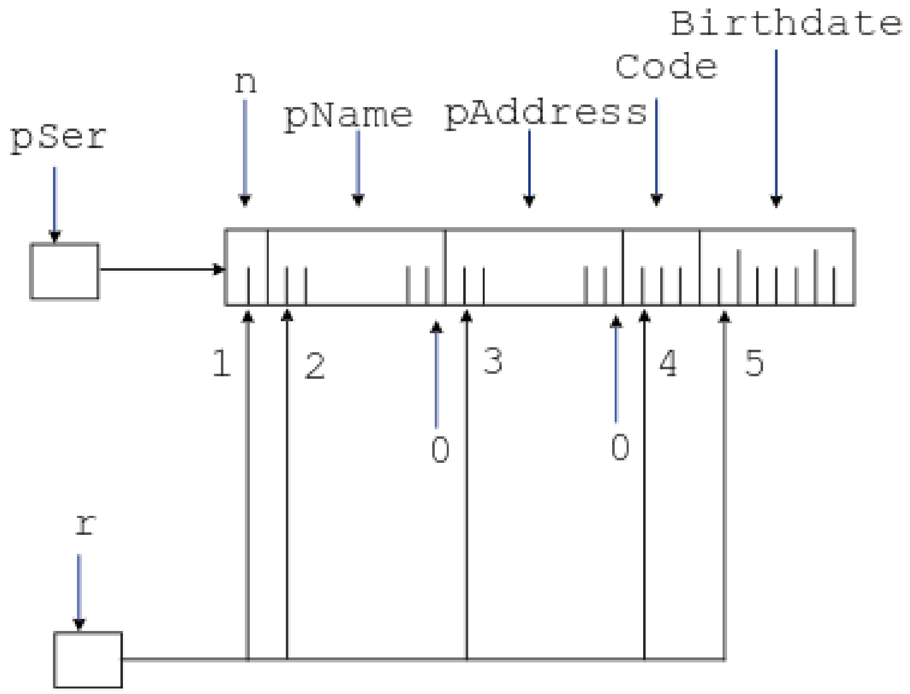
To build a linked list we need one more declaration:

```
#define BOOK 1
#define ARTICLE 2
struct Entry {
   short int type; // BOOK or ARTICLE
   union Reference reference;
   struct Entry *pNext;
};
```



Serialization





```
char *Serialize(PERSON *p) { // on disk memory addresses are senseless
  short int n1 = strlen(p->pName) +1, n2 = strlen(p->pAddress) + 1, n = n1 + n2;
  char *pSer, *r;
  pSer = (char *)malloc(n += sizeof(PERSON) + sizeof(int) – sizeof(PERSON *) -
                         2 * sizeof(char *);
  memcpy(r=pSer,&n,sizeof(int)); //1
  memcpy(r+=sizeof(int),p->pName,n1); //2
  memcpy(r+=n1,p->pAddress,n2); //3
  memcpy(r+=n2,&p->Code,sizeof(long int)); //4
  memcpy(r+sizeof(long int),&p->Birthdate.day, sizeof(DATE)); //5
  return pSer; // serialized compact struct ready for writing to disk
}
```

Long jump (1)

```
#include "setjmp.h" // see http://www.cplusplus.com/reference/csetjmp/longjmp/
```

The long jump mechanism is a part of C. In C++ the long jumps, although allowed, may lead to unpredictable behavor.

```
jmp_buf env; // global variable, stores the current execution environment
switch (setjmp(env)) // return point
{
    case 0: …………………..  // on the first call the setjmp return value is 0
             break;
    case 1: …………………..  // handle abnormal situation 1
             break;
    case 2: …………………… // handle abnormal situation 2
             break;
             ……………………
 }
```

If somewhere an error occurred, call function *longjmp():*

```
longjmp(env, n); // n is the index of abnormal situation
```

Long jump (2)

Example:

```
jmp_buf env;
int main()
{
    switch (setjmp(env)) // return point
   {
     case 0: fun1(); // on the first call the setjmp return value is 0, fun1() is called
             break;
     case 1: printf("Failure\n");
             return 1;
     }
……………………………….
}
void fun2() // called by fun1
{
  ……………………………….
  if (n <= 0)
      longjmp(env, 1); // exists fun2(), jumps back to return point, setjmp returns 1
  ……………………………….
}
```