Imperative programming

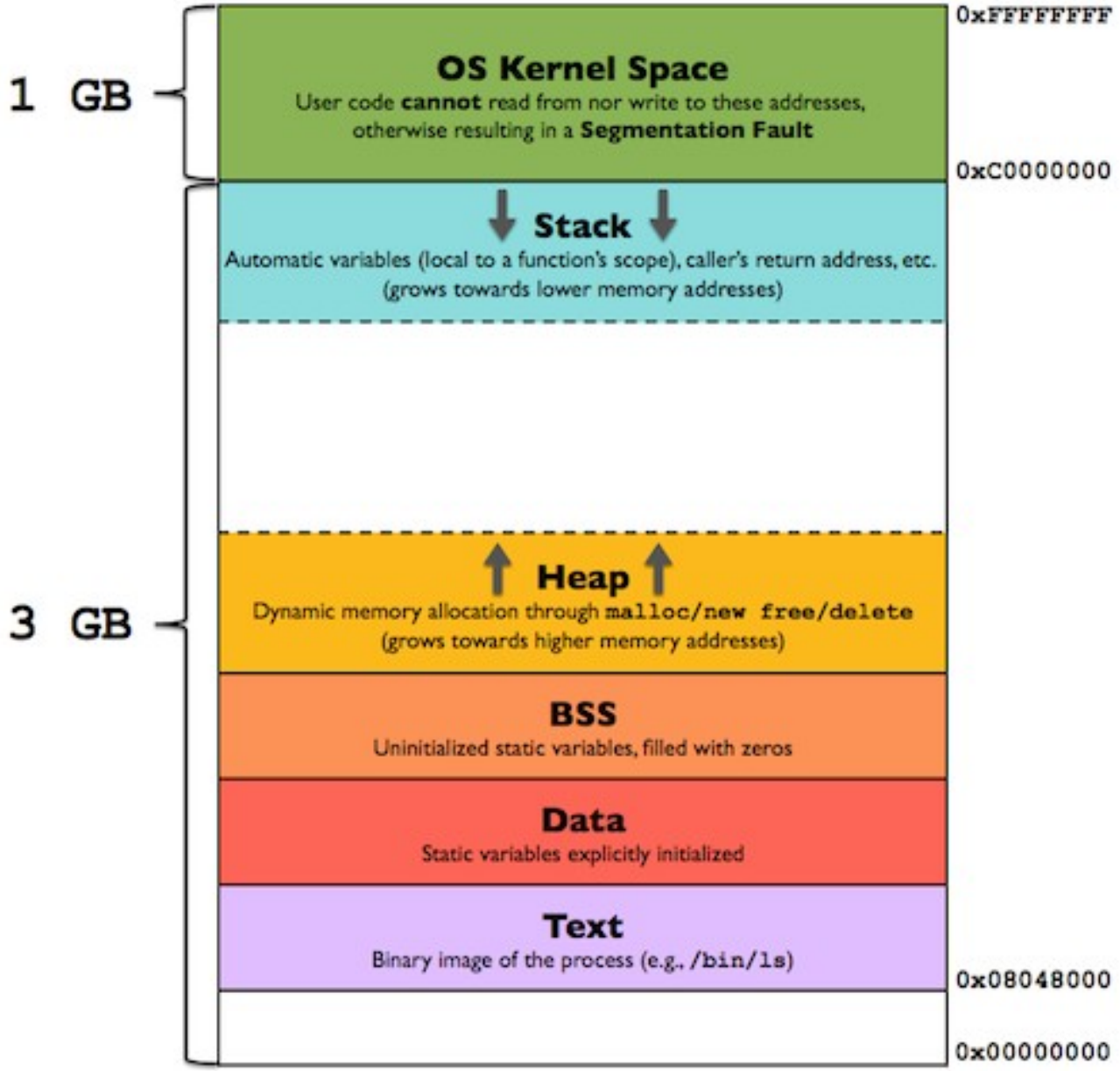
9. Dynamic life

Zoltán Porkoláb

Life

- For memory used during the execution of program
- From now until when is valid to use a memory area
  - Accessing a memory area out of lifetime is a serious error
  - Can cause wrong program behavior or crash
- Lifetime categories in C
  - Static – from the beginning of the program until the end
  - Automatic – during the execution of a block
  - Dynamic – the user request the memory and the user give it back

Typical memory layout



Dynamic lifetime in languages

- Memory is allocated in the **free store** (heap)
- In some languages (Java, C#, etc.) all Class objects are allocated in heap
- Unsuccessful allocation returns NULL pointer (C) or raise exception (C++)
- Heap can be compacted (.NET languages)
  - Heap compacting invalidates pointers
  - Managed languages updates pointers which is costly
  - If no compacting, memory can be **fragmented**
- Heap can garbage collected
  - Many modern languages, like ADA, Java, C#, Python, Golang
  - Effective garbage collection is hard and can “freeze” the program
- Heap is thread-safe
- Heap allocation/deallocation is relatively expensive

Dynamic lifetime in C

- Memory is allocated in the **free store** (heap)
- User requests memory by **malloc( )** or **realloc( )**
- Size parameter is in unit of **bytes**, and should be greater than 0
- If allocation is unsuccessful, NULL pointer returns
- **Always check the return value of malloc() and realloc( ) !**
- User must deallocate the memory by **free( )** to avoid **memory leak**
- Realloc may deallocate old memory and returns new (copying the content) or expanding the existing buffer
- Unreferred heap memory is “lost” and cause **memory leak**
- Heap can be **fragmented** (but allocators has startegy to avoid that)

Dynamic lifetime

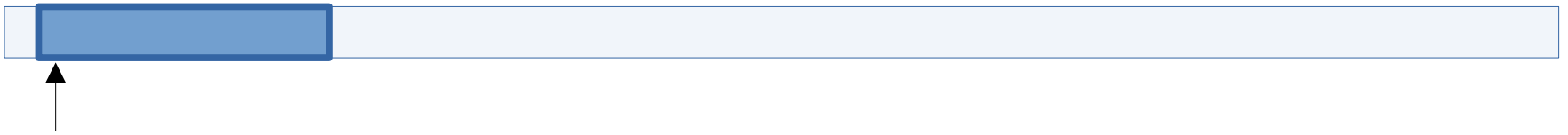
```
// in real life, always check the return value of malloc and realloc
int  f(void)
{




}
```

Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...




}
```

Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...




}
```

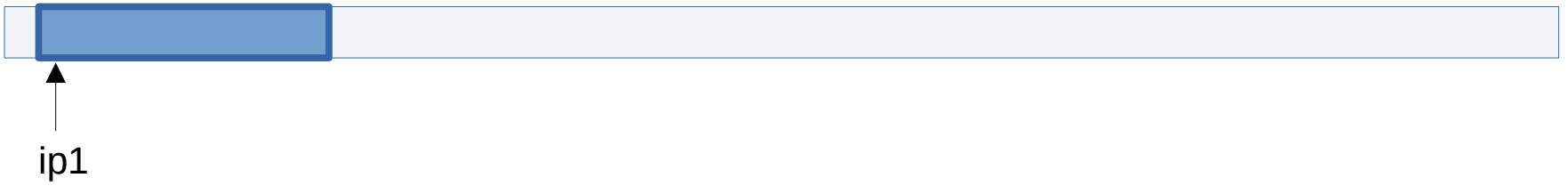


ip1

If allocation is unsuccessful, NULL pointer returns, otherwise new memory is allocated, the value of the area is undefined

Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...




}
```



Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...




}
```

New memory is allocated, the value is undefined

Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) realloc(NULL,6*sizeof(int)); // ...




}
```

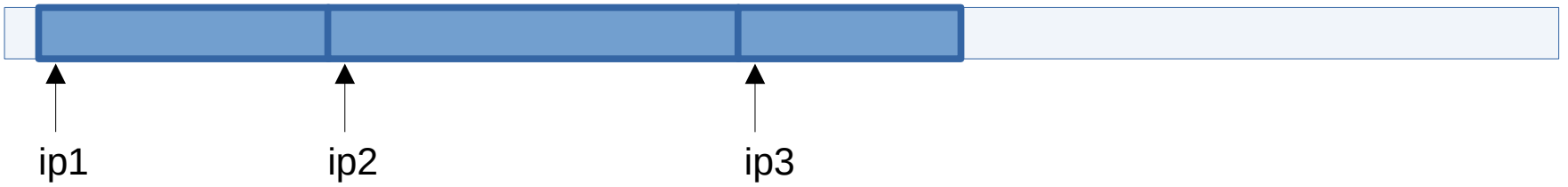
realloc with NULL first parameter is the same as malloc

Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...



}
```

Dynamic lifetime

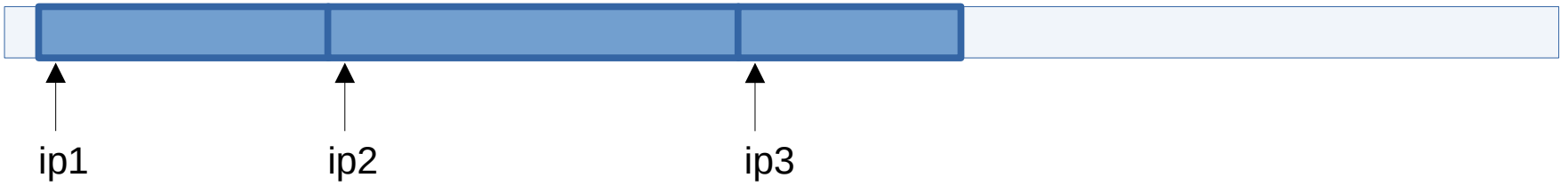
```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...



}
```



New memory is allocated, the value is undefined

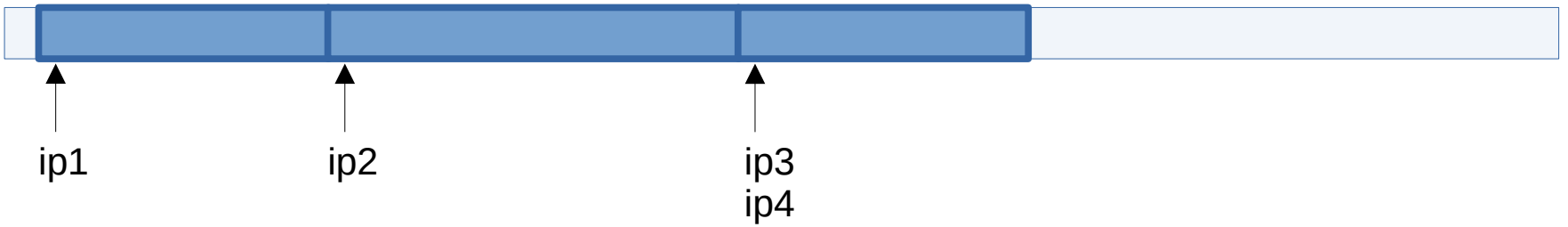
Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...



}
```



Dynamic lifetime

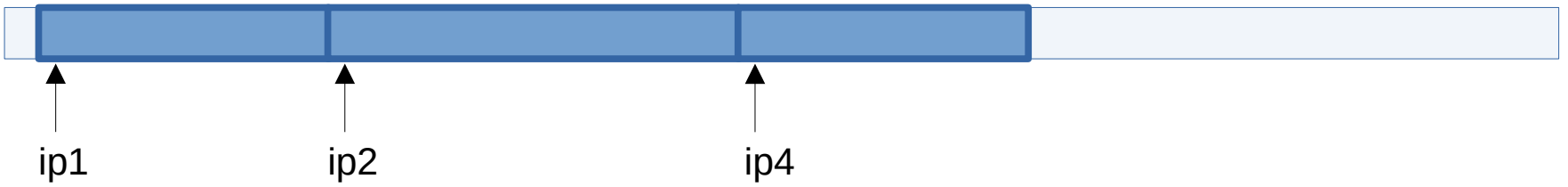
```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...



}
```



If expansion is possible in the original place, extended area has undefined value
Do not use ip3, it may be invalidated

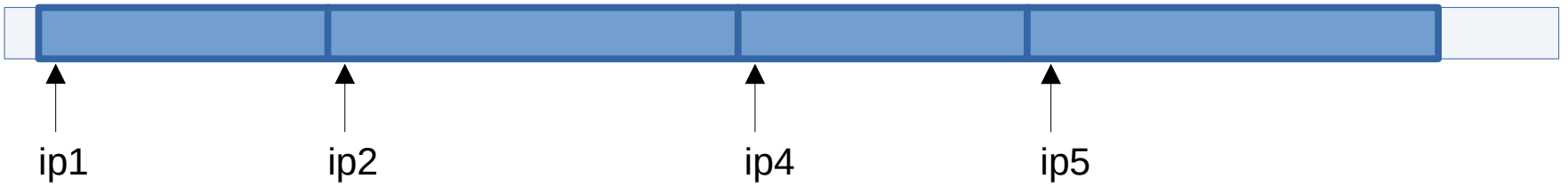
Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...


}
```



Dynamic lifetime

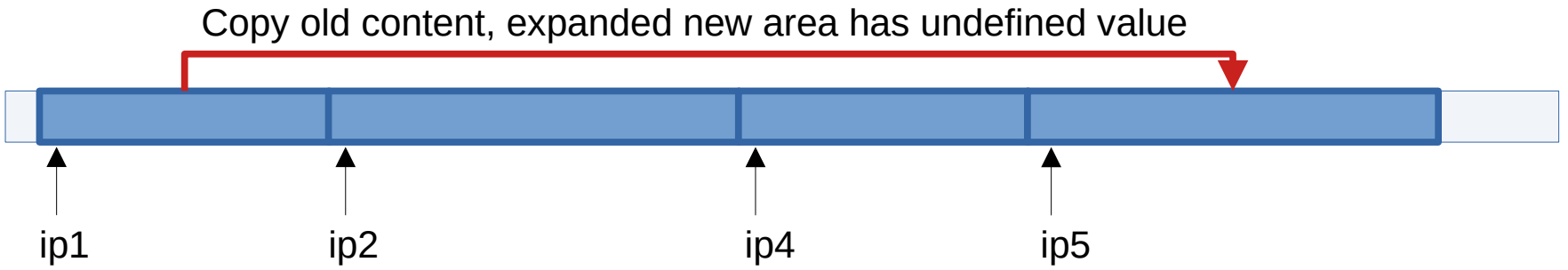
```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...


}
```



If extension is not possible in the original place, a new memory will be allocated

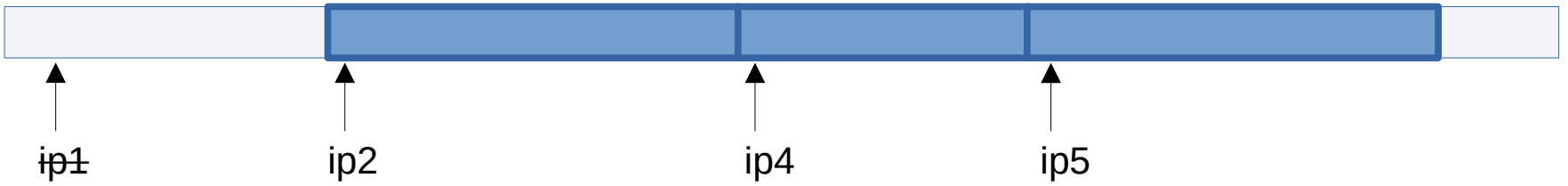
Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...


}
```



Dynamic lifetime

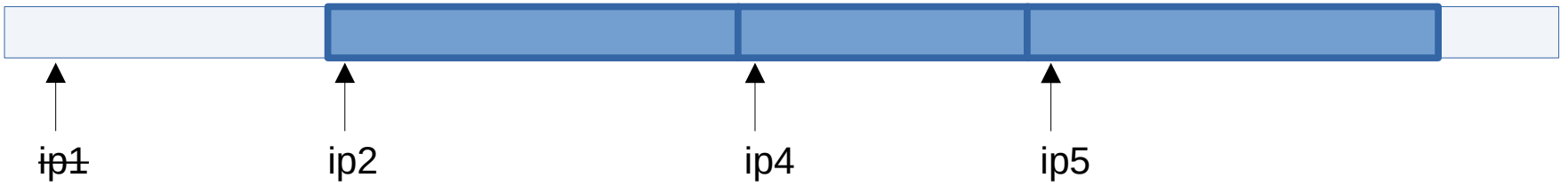
```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...


}
```



Old memory is freed, ip1 is **dangling** pointer, must not use

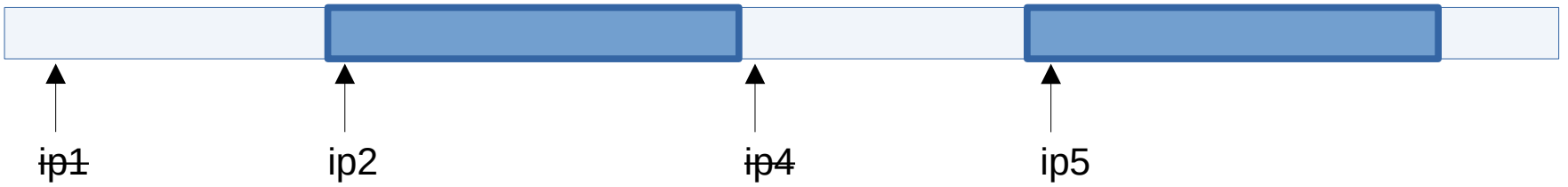
Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...
  free(ip4);

}
```



Dynamic lifetime

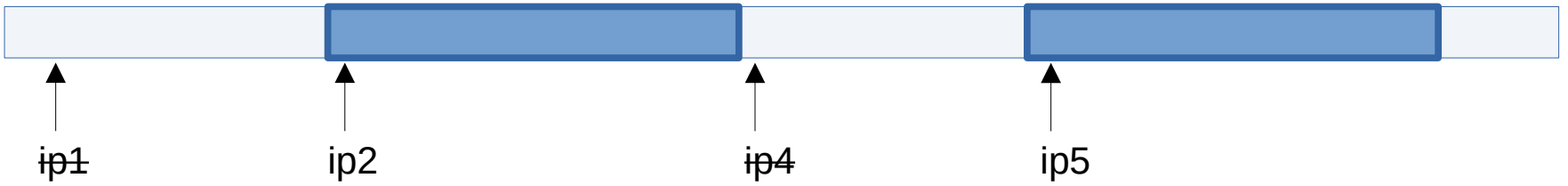
```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...
  free(ip4);
  free(ip5);
}
```



Memory is freed, ip4 is **dangling** pointer, must not use
Memory is a bit **fragmented**

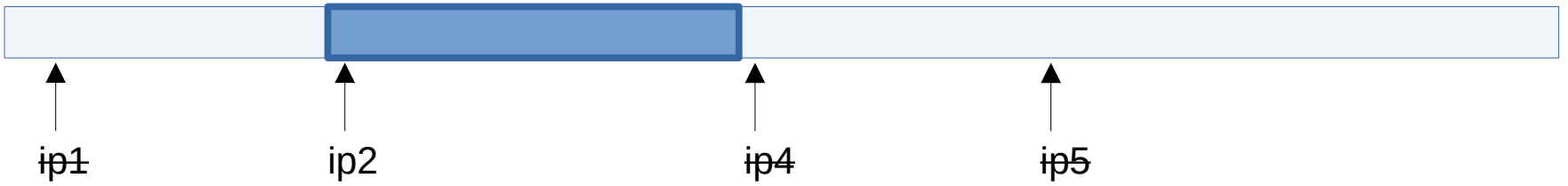
Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...
  free(ip4);
  free(ip5);
}
```



Dynamic lifetime

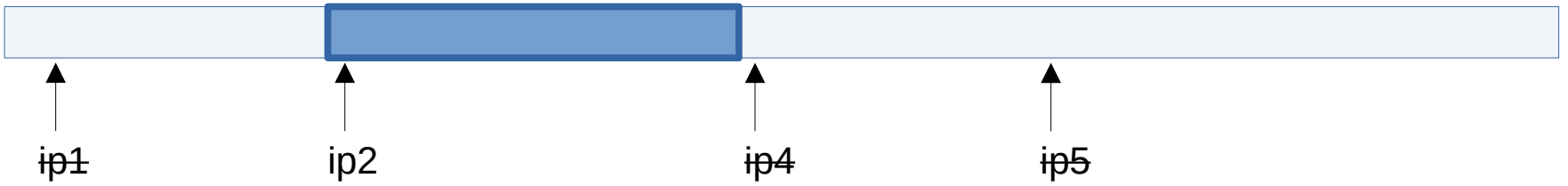
```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...
  free(ip4);
  free(ip5);
}
```



Memory is freed, ip5 is **dangling** pointer, must not use

Dynamic lifetime

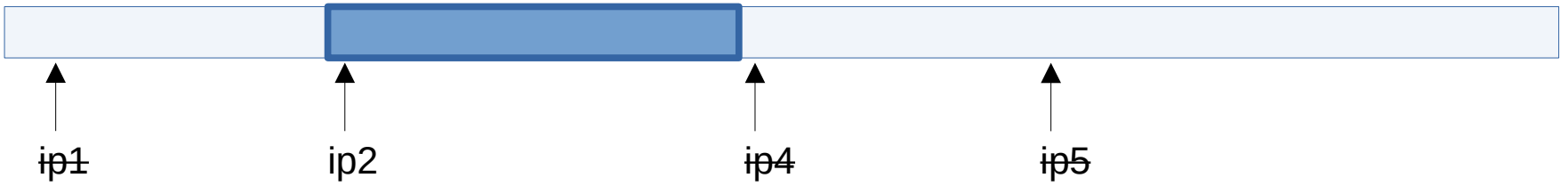
```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...
  free(ip4);
  free(ip5);
}
```



No change of ip1, ip4, ip5.
They will **NOT** be NULL

Dynamic lifetime

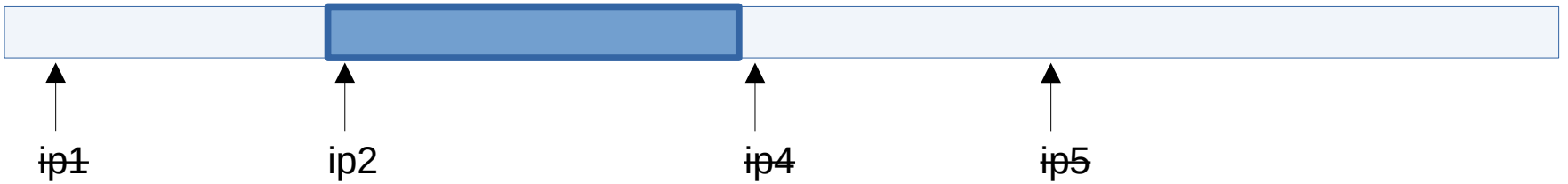
```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...
  free(ip4);
  free(ip5);
}
```



We leave the function,

Dynamic lifetime

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ip2 = (int *) malloc(6*sizeof(int)); // ...
  int *ip3 = (int *) malloc(3*sizeof(int)); // ...
  int *ip4 = (int *) realloc(ip3,4*sizeof(int)); // ...
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...
  free(ip4);
  free(ip5);
}
```



We leave the function,
Local automatic life variable **ip2** goes out of life.
We cannot refer to the allocated 6*sizeof(int) memory anymore
MEMORY LEAK!

Garbage collectors

- Many modern programming languages have garbage collector
- Garbage collector marks those object which are definitely **lost**
- Marked objects are collected regularly in a separate step
- GB has a runtime (and sometimes) memory overhead
- Most common strategy: **mark-and-sweep**
  - Starting from root set (global and stack variables) identifying live objects
  - Traversing recursively on all objects marking accessible ones
  - Collecting (free) unmarked objects
- Alternative: C++ destructors

Why no garbage collector in C?

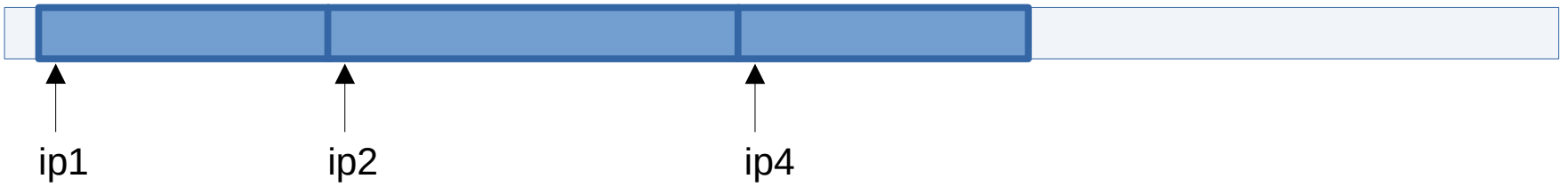
```
long long ugly_hack(int n)
{
  assert( sizeof(long long) >= sizeof(int *) );

  long long hide_the_ptr;
  int *buf = (int *)malloc(n*sizeof(int)); // try to allocate int[n]
  if ( NULL == buf )  // no memory
  {
    return 0LL;  // return "false"
  }
  memcpy(&hide_the_ptr, &buf, sizeof(int *));
  return hide_the_ptr;
}

void use_pointer(long long hide_the_ptr, int n)
{
  assert( sizeof(long long) >= sizeof(int *) );
  if ( 0LL == hide_the_ptr )
    return;

  int *buf;
  memcpy(&buf, &hide_the_ptr, sizeof(int *));
  // use buf
  free(buf);
}
```

Why no garbage collector in C?

```
long long ugly_hack(int n)
{
  assert( sizeof(long long) >= sizeof(int *) );

  volatile long long hide_the_ptr;
  int *buf = (int *)malloc(n*sizeof(int)); // try to allocate int[n]
  if ( NULL == buf )  // no memory
  {
    return 0LL;  // return "false"
  }
  memcpy(&hide_the_ptr, &buf, sizeof(int *));
  return hide_the_ptr;
}

void use_pointer(long long hide_the_ptr, int n)
{
  assert( sizeof(long long) >= sizeof(int *) );
  if ( 0LL == hide_the_ptr )
    return;

  volatile int *buf;
  memcpy(&buf, &hide_the_ptr, sizeof(int *));
  // use buf
  free(buf);
}
```

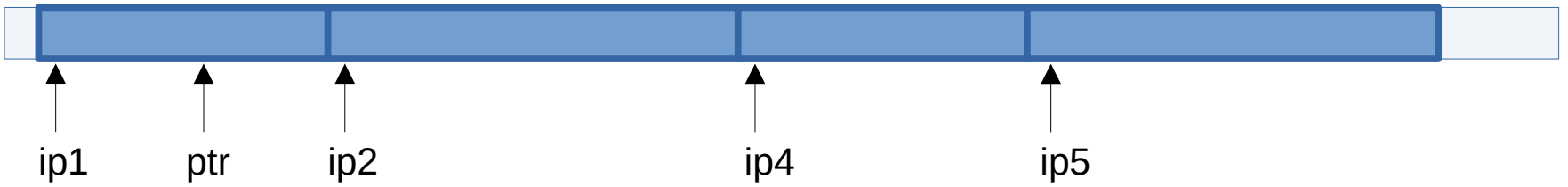
Dynamic lifetime traps

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...



}
```



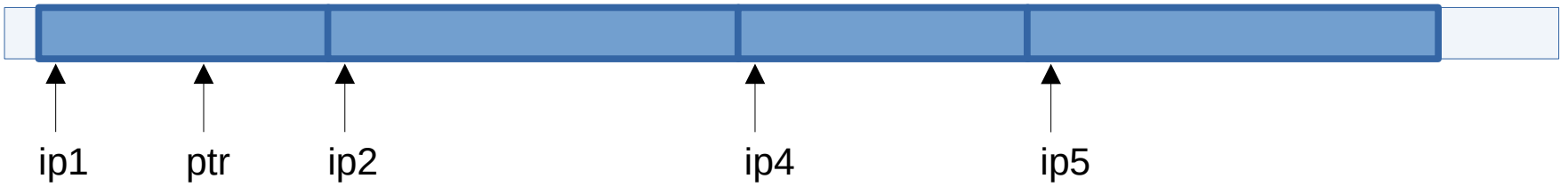
Dynamic lifetime traps

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ptr = &ip1[2];



}
```



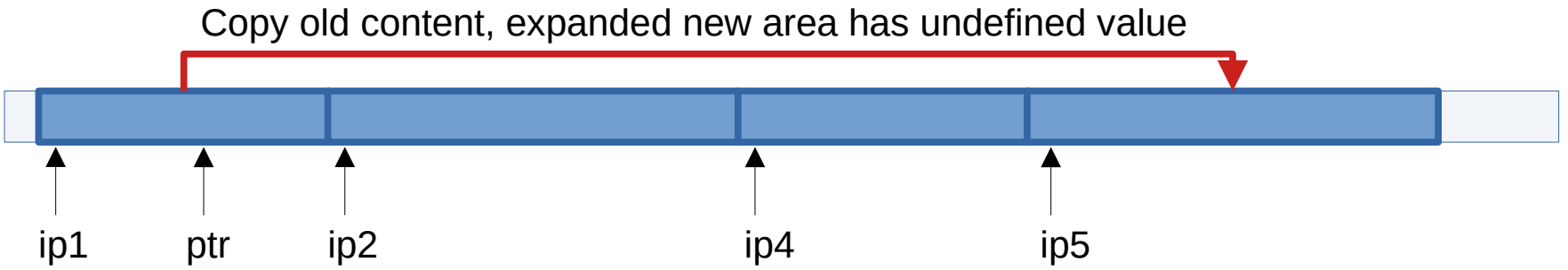
Dynamic lifetime traps

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ptr = &ip1[2];
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...


}
```



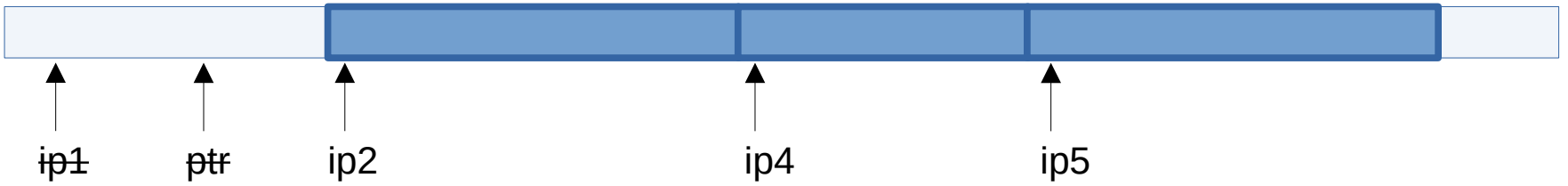
Dynamic lifetime traps

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ptr = &ip1[2];
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...


}
```



Dynamic lifetime traps

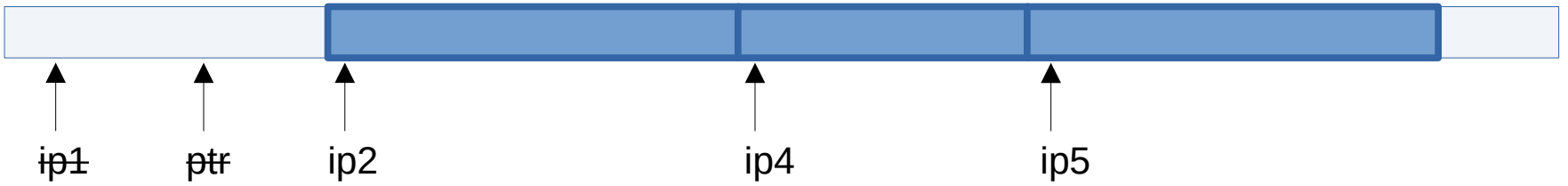
```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ptr = &ip1[2];
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...


}
```



Old memory is freed, ip1 and ptr are **dangling** pointers, must not use

Dynamic lifetime traps

```
// in real life, always check the return value of malloc and realloc
int  f(void)
{
  int *ip1 = (int *) malloc(4*sizeof(int)); // ...
  int *ptr = &ip1[2];
  int *ip5 = (int *) realloc(ip1,6*sizeof(int)); // ...


}
```



Old memory is freed, ip1 and ptr are **dangling** pointers, must not use

Dynamic lifetime traps

```
char *revert_string(char *s)  // suppose s is not NULL
{
  int n = strlen(s);  // number of characters of s

  char *q = (char *)malloc(n); // try to allocate char[n]
  if ( NULL == q )  // no memory
  {
    return NULL;  // unsuccessful
  }

  for (int i = 0; i < n; ++i)
  {
    q[i] = s[n-i-1];  // reverse s
  }


  return q;     // return the result
}
```

Dynamic lifetime traps

```
char *revert_string(char *s)  // suppose s is not NULL
{
  int n = strlen(s);  // number of characters of s without ‘\0’

  char *q = (char *)malloc(n); // try to allocate char[n]
  if ( NULL == q )  // no memory
  {
    return NULL;  // unsuccessful
  }

  for (int i = 0; i < n; ++i)
  {
    q[i] = s[n-i-1];  // reverse s
  }
  q[n] = '\0';  // terminating the result

  return q;     // return the result
}
```

Dynamic lifetime traps

```
char *revert_string(char *s)  // suppose s is not NULL
{
  int n = strlen(s);  // number of characters of s without ‘\0’

  char *q = (char *)malloc(n+1); // try to allocate char[n+1]
  if ( NULL == q )  // no memory
  {
    return NULL;  // unsuccessful
  }

  for (int i = 0; i < n; ++i)
  {
    q[i] = s[n-i-1];  // reverse s
  }
  q[n] = '\0';  // terminating the result

  return q;     // return the result
}
```

Dynamic lifetime traps

```
char *revert_string(char *s)  // suppose s is not NULL
{
  int n = strlen(s);  // number of characters of s without ‘\0’

  char *q = (char *)malloc(n+1); // try to allocate char[n+1]
  if ( NULL == q )  // no memory
  {
    return NULL;  // does the receiver prepared for NULL pointer?
  }

  for (int i = 0; i < n; ++i)
  {
    q[i] = s[n-i-1];  // reverse s
  }
  q[n] = '\0';  // terminating the result

  return q;     // return the result
}
```

Dynamic lifetime traps

```
char *revert_string(char *s)  // suppose s is not NULL
{
  int n = strlen(s);  // number of characters of s without ‘\0’

  char *q = (char *)malloc(n+1); // try to allocate char[n+1]
  if ( NULL == q )  // no memory
  {
    return "";  // safer
  }

  for (int i = 0; i < n; ++i)
  {
    q[i] = s[n-i-1];  // reverse s
  }
  q[n] = '\0';  // terminating the result

  return q;     // return the result
}
```

Off-by-one error (OB1)

```
void f(char *s)  // suppose s is not NULL
{
  char *q = (char *) malloc(strlen(s));


  strcpy(q,s);
  // ...
}
```

Off-by-one error (OB1)

```
void f(char *s)  // suppose s is not NULL
{
  char *q = (char *) malloc(strlen(s));
  char *q = (char *) malloc(strlen(s+1));


  strcpy(q,s);
  // ...
}
```

Off-by-one error (OB1)

```
void f(char *s)  // suppose s is not NULL
{
  char *q = (char *) malloc(strlen(s));
  char *q = (char *) malloc(strlen(s+1));
  char *q = (char *) malloc(strlen(s)+1);

  strcpy(q,s);
  // ...
}
```

Reversing n integers

```
// read n numbers and print them in reverse order
int  reverse(void)
{
  int n;
  scanf("%d", &n);  // read n from stdin

  int *buf = (int *)malloc(n*sizeof(int)); // try to allocate int[n]

  if ( NULL == buf )  // no memory
  {
    fprintf( stdout, "No memory\n");
    return 0;  // return false
  }
  for (int i = 0; i < n; ++i)
  {
    scanf("%d", &buf[i]);   // read elements from stdin
  }
  for (int i = n-1; i >= 0; --i)
  {
    printf("%d, ", buf[i]); // print elements in reverse order
  }
  free(buf); // deallocate heap memory
  return 1;  // return true
}
```

Dynamically extending buffer

```
int  f(void)
{
  int ch;
  int i = 0;
  int capacity = 16;
  char *buf = (char *)malloc(capacity); // try to allocate char[16]

  if ( NULL == buf )  // allocation unsuccessful
  {
    fprintf(stderr,"no memory");
    return -1;
  }
  while ( EOF != (ch = getchar()) ) // reading characters until EOF
  {









    buf[i++] = ch;   // put the character to the next place
  }
  // do something with buf                          and free!!!
}
```

Dynamically extending buffer

```
int  f(void)
{
  int ch;
  int i = 0;
  int capacity = 16;
  char *buf = (char *)malloc(capacity); // try to allocate char[16]

  if ( NULL == buf )  // allocation unsuccessful
  {
    fprintf(stderr,"no memory");
    return -1;
  }
  while ( EOF != (ch = getchar()) ) // reading characters until EOF
  {
    if ( i == capacity ) // if buffer is full, grow buffer
    {









    }
    buf[i++] = ch;   // // put the character to the next place
  }
  // do something with buf                          and free!!!
}
```

Dynamically extending buffer

```
int  f(void)
{
  int ch;
  int i = 0;
  int capacity = 16;
  char *buf = (char *)malloc(capacity); // try to allocate char[16]

  if ( NULL == buf )  // allocation unsuccessful
  {
    fprintf(stderr,"no memory");
    return -1;
  }
  while ( EOF != (ch = getchar()) ) // reading characters until EOF
  {
    if ( i == capacity ) // if buffer is full, grow buffer
    {
      char *newbuf = (char *)realloc(buf,2*capacity); // expand the buffer
      if ( newbuf )  // reallocation was successful
      {


      }
      else  // reallocation is unsuccessful, but the old memory is still valid
      {
         break;   // buf is valid, capacity is valid, i is valid
      }
    }
    buf[i++] = ch;   // // put the character to the next place
  }
  // do something with buf                          and free!!!
}
```

Dynamically extending buffer

```
int  f(void)
{
  int ch;
  int i = 0;
  int capacity = 16;
  char *buf = (char *)malloc(capacity); // try to allocate char[16]

  if ( NULL == buf )  // allocation unsuccessful
  {
    fprintf(stderr,"no memory");
    return -1;
  }
  while ( EOF != (ch = getchar()) ) // reading characters until EOF
  {
    if ( i == capacity ) // if buffer is full, grow buffer
    {
      char *newbuf = (char *)realloc(buf,2*capacity); // expand the buffer
      if ( newbuf )  // reallocation is successful
      {
         capacity *= 2; // capacity duplicated
         buf = newbuf;  // newbuf is not necessary the same as buf
      }
      else  // reallocation is unsuccessful, but the old memory is still valid
      {
         break;   // buf is valid, capacity is valid, i is valid
      }
    }
    buf[i++] = ch;   // // put the character to the next place
  }
  // do something with buf                          and free!!!
}
```

Linked lists

```
typedef struct Elem
{
  double val;
  struct Elem *prev;
  struct Elem *next;
} elem_t;

typedef struct List
{
  elem_t *first;
  elem_t *last;
  int     size;
} list_t;
```

Linked lists

```
typedef struct Elem
{
  double val;
  struct Elem *prev;
  struct Elem *next;
} elem_t;

typedef struct List
{
  elem_t *first;
  elem_t *last;
  int     size;
} list_t;

void list_init(list_t *lp)
{
  lp->first = NULL;
  lp->last  = NULL;
  lp->size  = 0;
}
```

Linked lists

```
typedef struct Elem
{
  double val;
  struct Elem *prev;
  struct Elem *next;
} elem_t;

typedef struct List
{
  elem_t *first;
  elem_t *last;
  int     size;
} list_t;

void list_init(list_t *lp)
{
  lp->first = NULL;
  lp->last  = NULL;
  lp->size  = 0;
}
```

```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

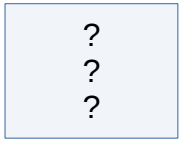
Linked lists

```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```

```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

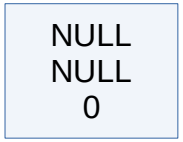
```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```




```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

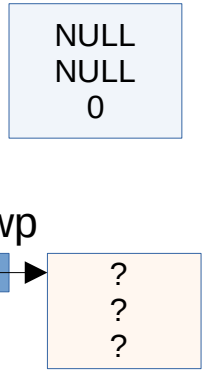
```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```




```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

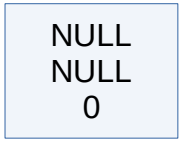
```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```

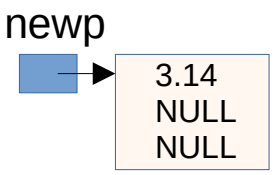



```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```

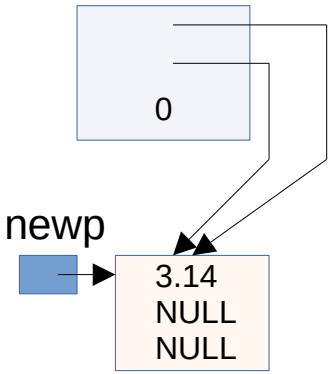





```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

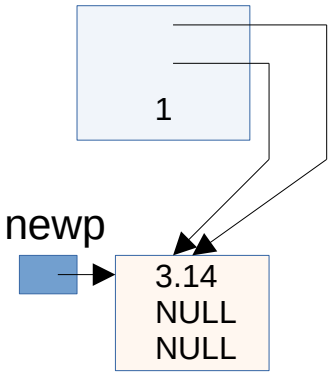
```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```




```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

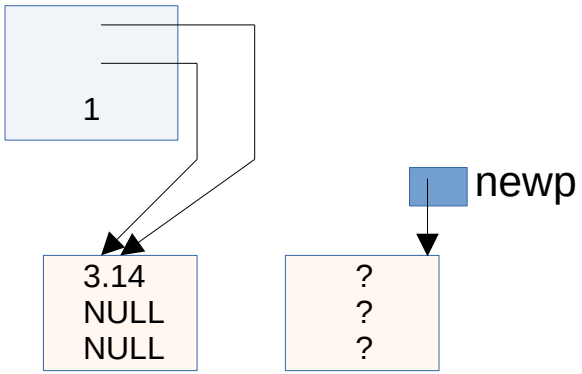
```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```




```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

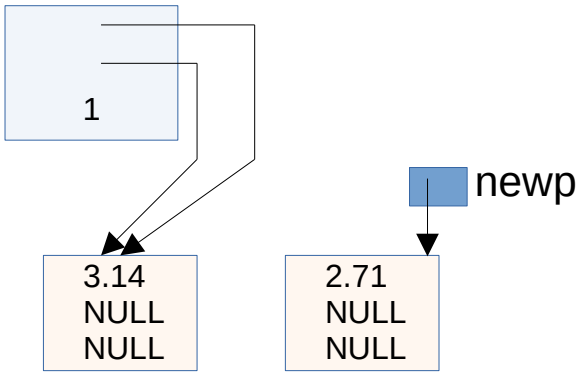
```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```




```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

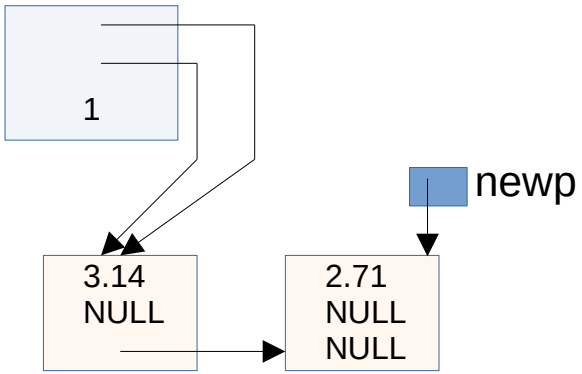
```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```




```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

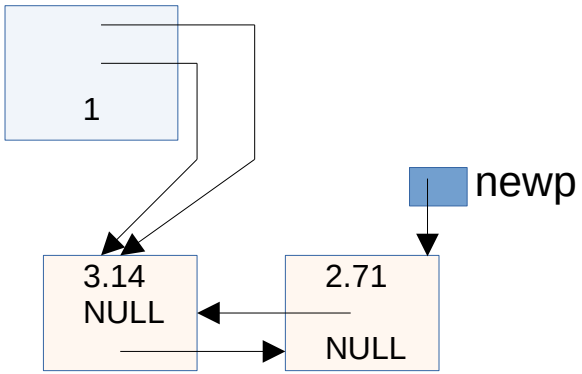
```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```




```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

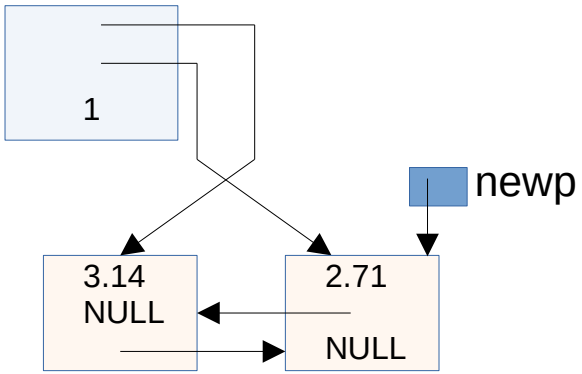
```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```




```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

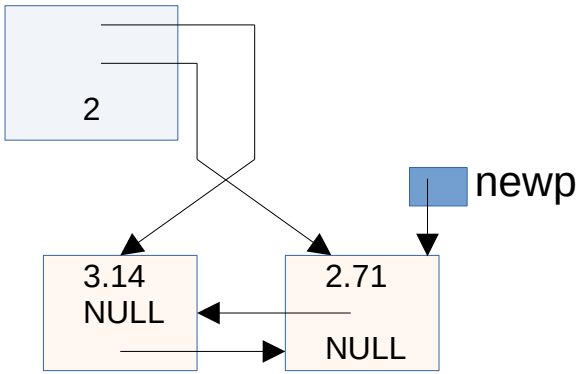
```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```




```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

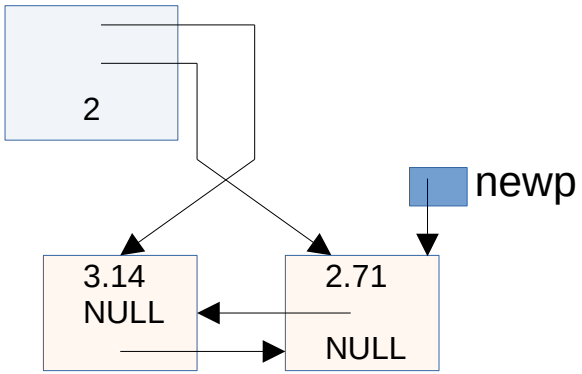
```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);

}
```




```
int list_push_back(list_t *lp, double d)
{
  elem_t *newp =
        (elem_t*) malloc(sizeof(elem_t));
  if ( NULL == newp )
    return 0;  // fail

  newp->val = d;
  newp->prev = newp->next = NULL;

  if ( 0 == size ) {  // list is empty
    lp->first = lp->last = newp;
  }
  else {
    lp->last->next = newp;
    newp->prev = lp->last;
    lp->last = newp;
  }
  ++lp->size;
  return 1; // success
}
```

Linked lists

```
int main()
{
  list_t lst;
  list_init(&lst);

  list_push_back(&lst,3.14);
  list_push_back(&lst,2.71);
  list_clear(&lst);
}
```




```
int list_clear(list_t *lp)
{
  elem_t *p = lp->first;

  while ( p )  // NULL != p
  {
    elem_t *curr = p;
    p = p->next;
    free(curr);
  }
  lp->first = lp->last = NULL;
  lp->size = 0;
}
```