System Programming

Sakarya University-Computer Engineering

- http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/CStuff-1/lecture.html
- http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/CStuff-2/lecture.html

Contents

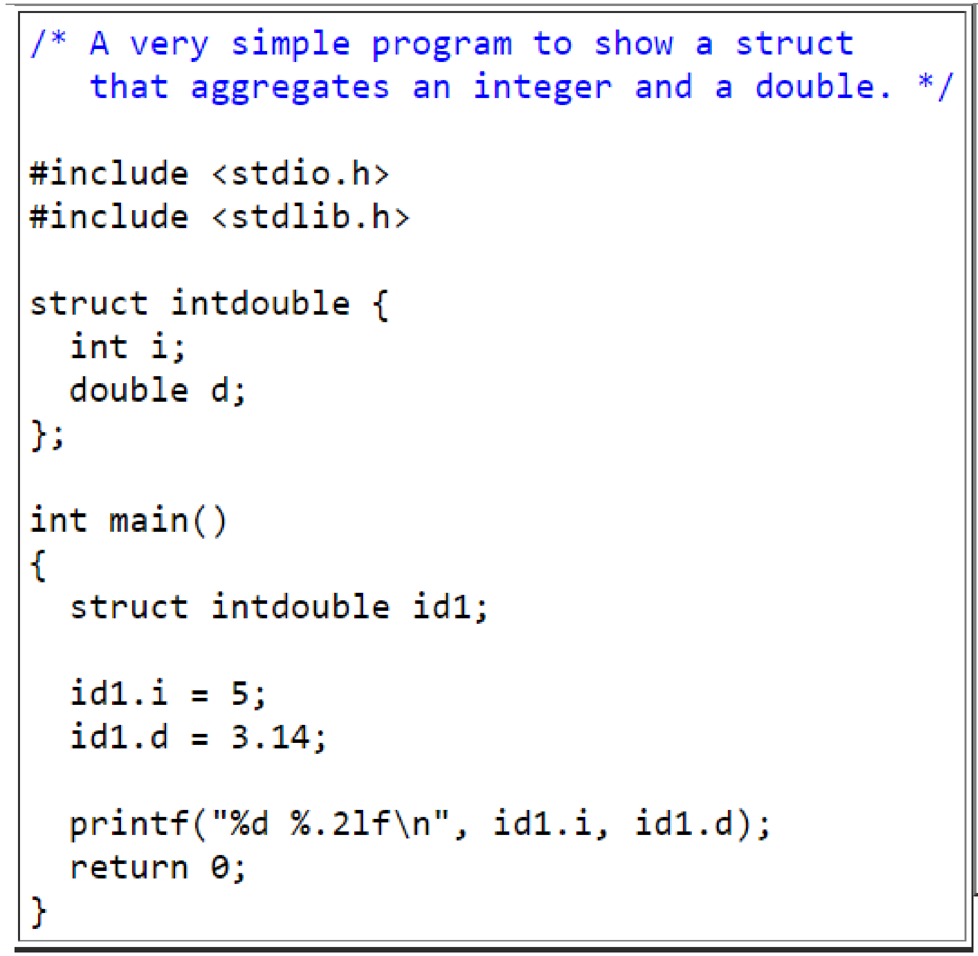
Struct

- In programming languages, the term " struct " refers to a composite data type in which different data types are combined.
- This allows programmers to create a new data type that has components consisting of one or more data types.
- The stack region of memory .
- For example, if we need a lot of data such as a student's name, number and grades, the structure structure allows us to keep this data together.

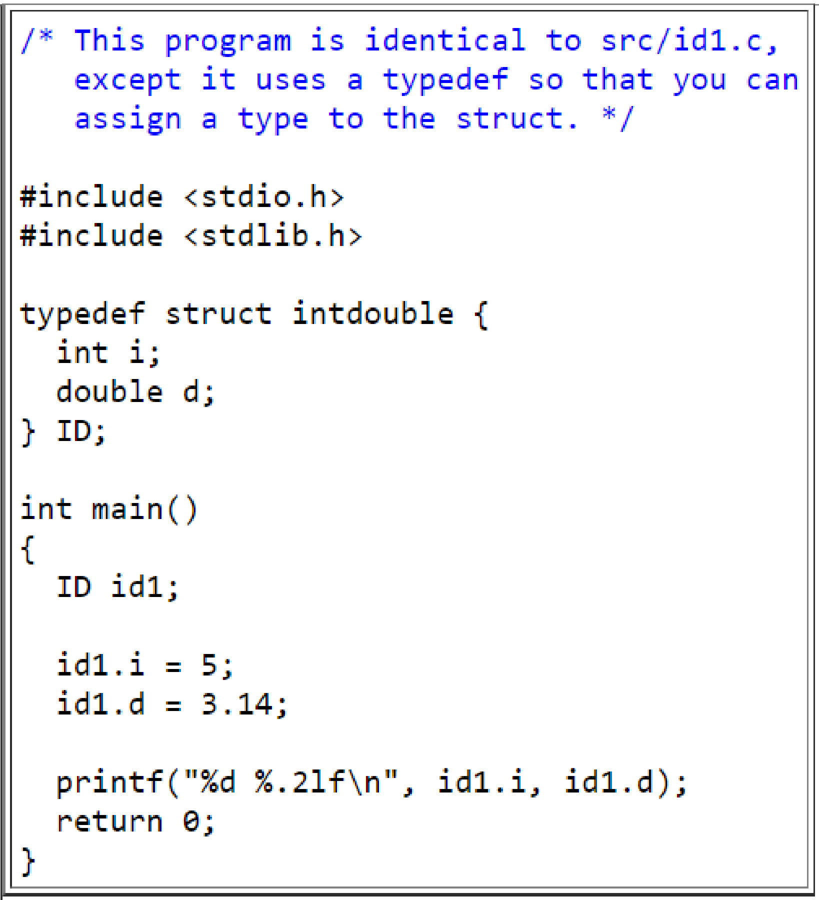
Struct

- Another way to collect data is to use a structure.
- A struct is somewhat similar to a C++ class, with some notable omissions:
  - « public / protected / private ».
  - « constructors / destructors ».
  - There is no default "copy" method.
  - There are no methods.

Struct



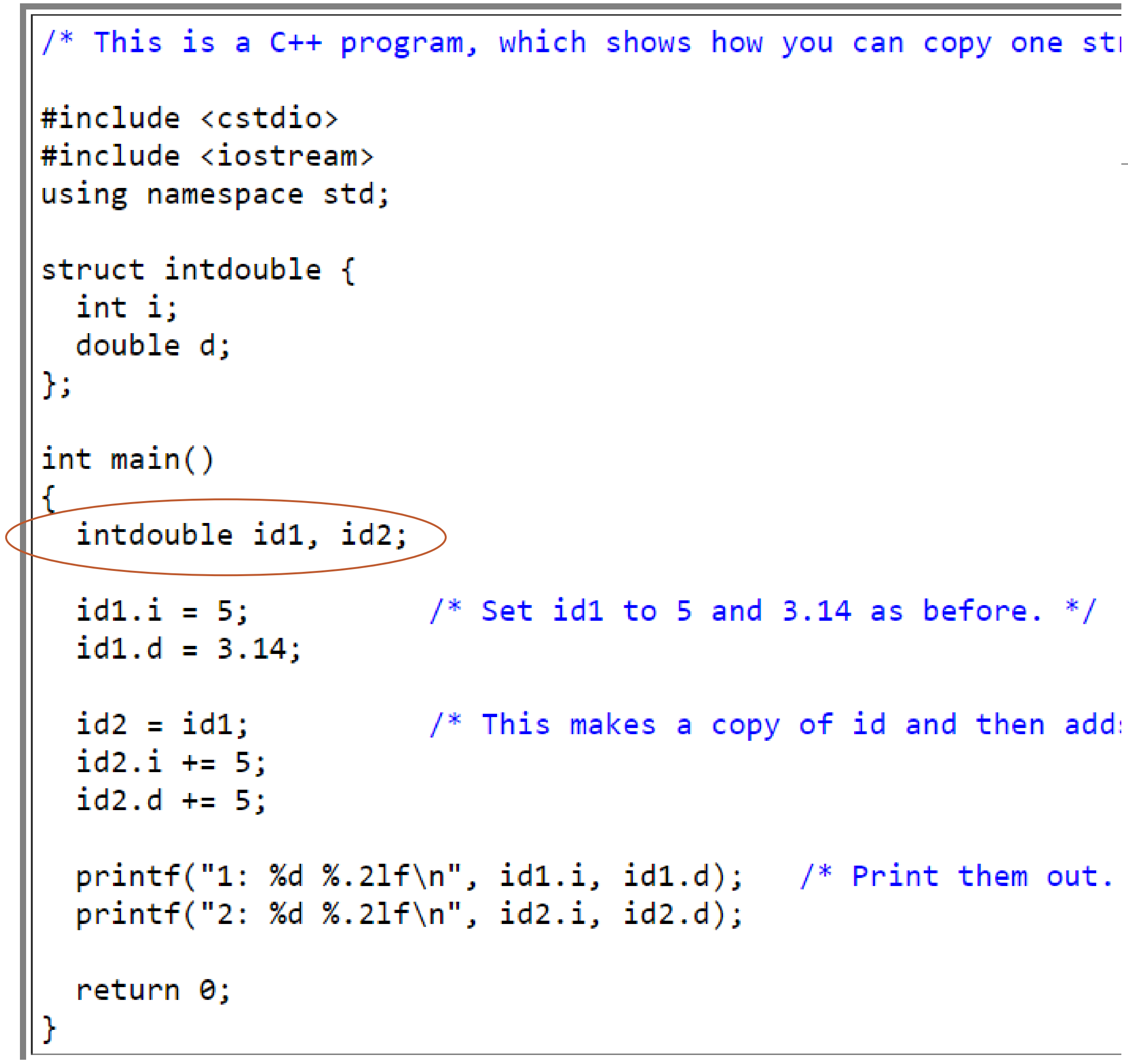
```
/* A very simple program to show a struct
   that aggregates an integer and a double. */

#include <stdio.h>
#include <stdlib.h>

struct intdouble {
  int i;
  double d;
};

int main()
{
  struct intdouble id1;

  id1.i = 5;
  id1.d = 3.14;

  printf("%d %.2lf\n", id1.i, id1.d);
  return 0;
}
```



```
/* This program is identical to src/id1.c,
   except it uses a typedef so that you can
   assign a type to the struct. */

#include <stdio.h>
#include <stdlib.h>

typedef struct intdouble {
  int i;
  double d;
} ID;

int main()
{
  ID id1;

  id1.i = 5;
  id1.d = 3.14;

  printf("%d %.2lf\n", id1.i, id1.d);
  return 0;
}
```

typedef

Struct

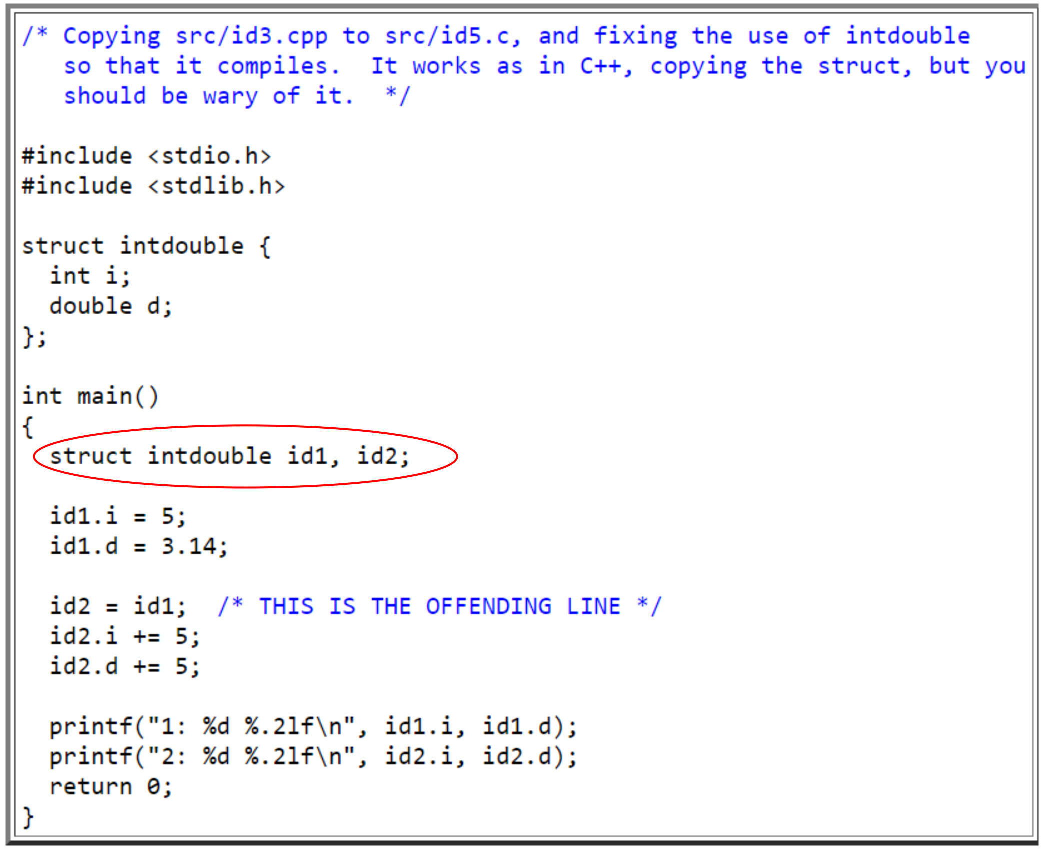


```
/* This is a C++ program, which shows how you can copy one str

#include <cstdio>
#include <iostream>
using namespace std;

struct intdouble {
  int i;
  double d;
};

int main()
{
  intdouble id1, id2;

  id1.i = 5;           /* Set id1 to 5 and 3.14 as before. */
  id1.d = 3.14;

  id2 = id1;           /* This makes a copy of id and then add

  id2.i += 5;
  id2.d += 5;

  printf("1: %d %.2lf\n", id1.i, id1.d);    /* Print them out.
  printf("2: %d %.2lf\n", id2.i, id2.d);

  return 0;
}
```

- Constructs operate with **different semantics in C++,** so it can cause confusion.
- One struct can be copied into another in C and C++

```
UNIX > bin/id3 1:5 3.14
2:10 8.14
UNIX>
```

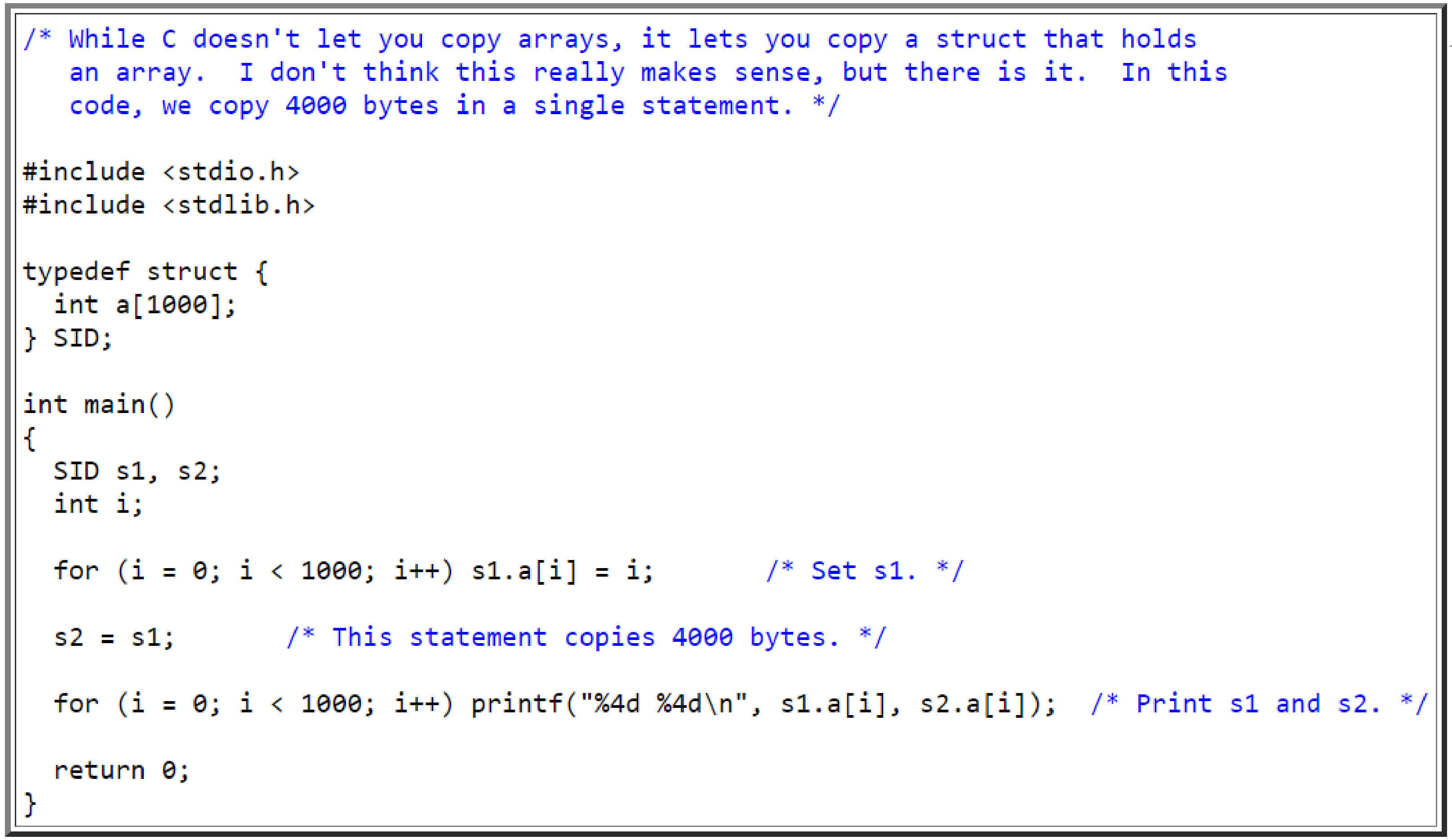
Struct



```
/* Copying src/id3.cpp to src/id5.c, and fixing the use of intdouble
   so that it compiles.  It works as in C++, copying the struct, but you
   should be wary of it.  */

#include <stdio.h>
#include <stdlib.h>

struct intdouble {
  int i;
  double d;
};

int main()
{
  struct intdouble id1, id2;

  id1.i = 5;
  id1.d = 3.14;

  id2 = id1;  /* THIS IS THE OFFENDING LINE */
  id2.i += 5;
  id2.d += 5;

  printf("1: %d %.2lf\n", id1.i, id1.d);
  printf("2: %d %.2lf\n", id2.i, id2.d);
  return 0;
}
```

- (src /id4.c) is incorrect
- " struct " in front of " intdouble" ,

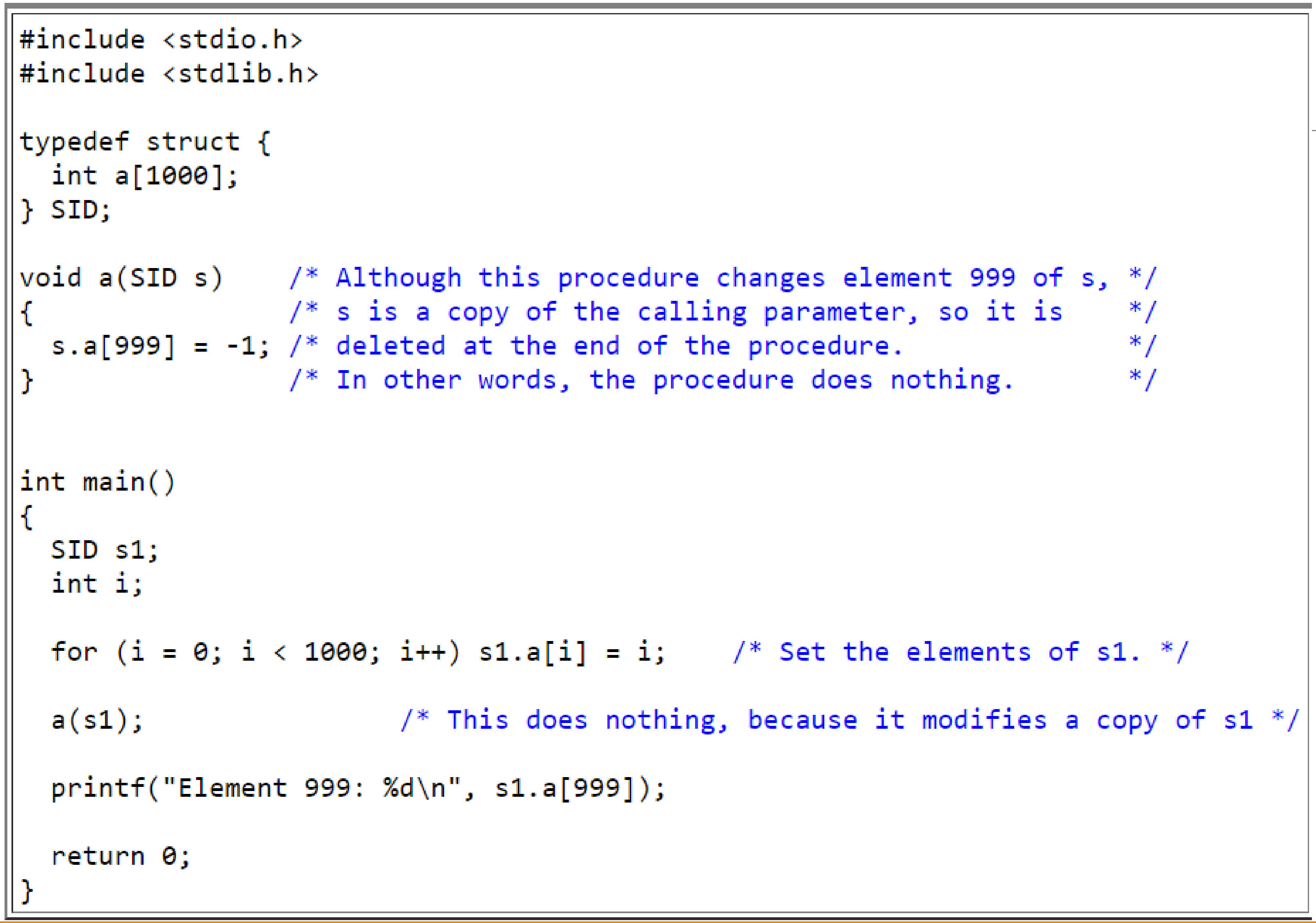
Struct



```
/* While C doesn't let you copy arrays, it lets you copy a struct that holds
   an array.  I don't think this really makes sense, but there is it.  In this
   code, we copy 4000 bytes in a single statement. */

#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int a[1000];
} SID;

int main()
{
  SID s1, s2;
  int i;

  for (i = 0; i < 1000; i++) s1.a[i] = i;       /* Set s1. */

  s2 = s1;       /* This statement copies 4000 bytes. */

  for (i = 0; i < 1000; i++) printf("%4d %4d\n", s1.a[i], s2.a[i]);  /* Print s1 and s2. */

  return 0;
}
```

- ❑It's the only part of C where you can copy an unspecified number of bytes via the assignment statement in structs.
- ❑This can be expressed as a weakness of the language
- ❑The line ``s2 = s1'' copies 4000 bytes.

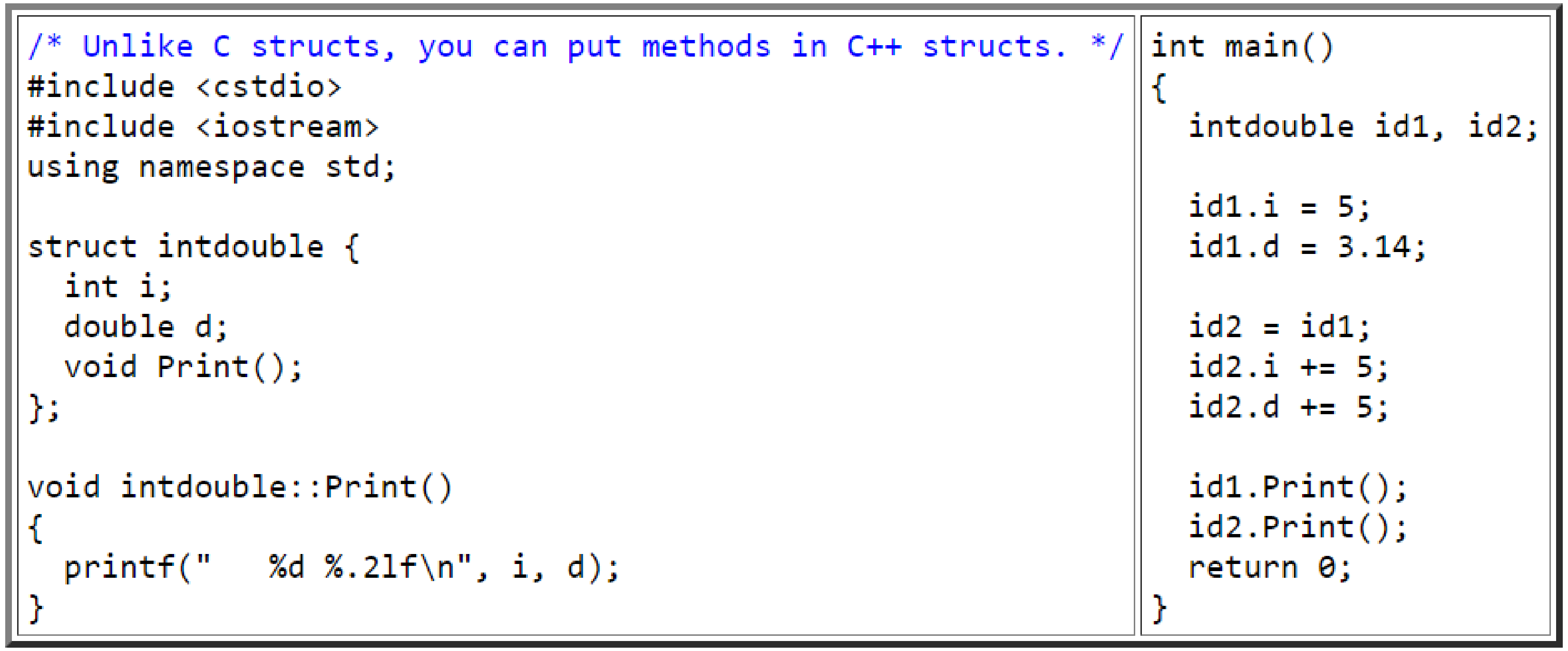
Struct



```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int a[1000];
} SID;

void a(SID s)    /* Although this procedure changes element 999 of s, */
{                /* s is a copy of the calling parameter, so it is    */
  s.a[999] = -1; /* deleted at the end of the procedure.              */
}                /* In other words, the procedure does nothing.       */


int main()
{
  SID s1;
  int i;

  for (i = 0; i < 1000; i++) s1.a[i] = i;    /* Set the elements of s1. */

  a(s1);                 /* This does nothing, because it modifies a copy of s1 */

  printf("Element 999: %d\n", s1.a[999]);

  return 0;
}
```

- We can send it as a parameter to a function,
- However, here a copy is created and the change made to the defined array in the structure disappears after exiting the procedure, that is, there is no change.

Struct



```
/* Unlike C structs, you can put methods in C++ structs. */
#include <cstdio>
#include <iostream>
using namespace std;

struct intdouble {
  int i;
  double d;
  void Print();
};

void intdouble::Print()
{
  printf("   %d %.2lf\n", i, d);
}
```

```
int main()
{
  intdouble id1, id2;

  id1.i = 5;
  id1.d = 3.14;

  id2 = id1;
  id2.i += 5;
  id2.d += 5;

  id1.Print();
  id2.Print();
  return 0;
}
```

- One final note about C++ constructs.
- basically simplified classes - you can put methods in them and then implement the functions using struct variables.

Pointer

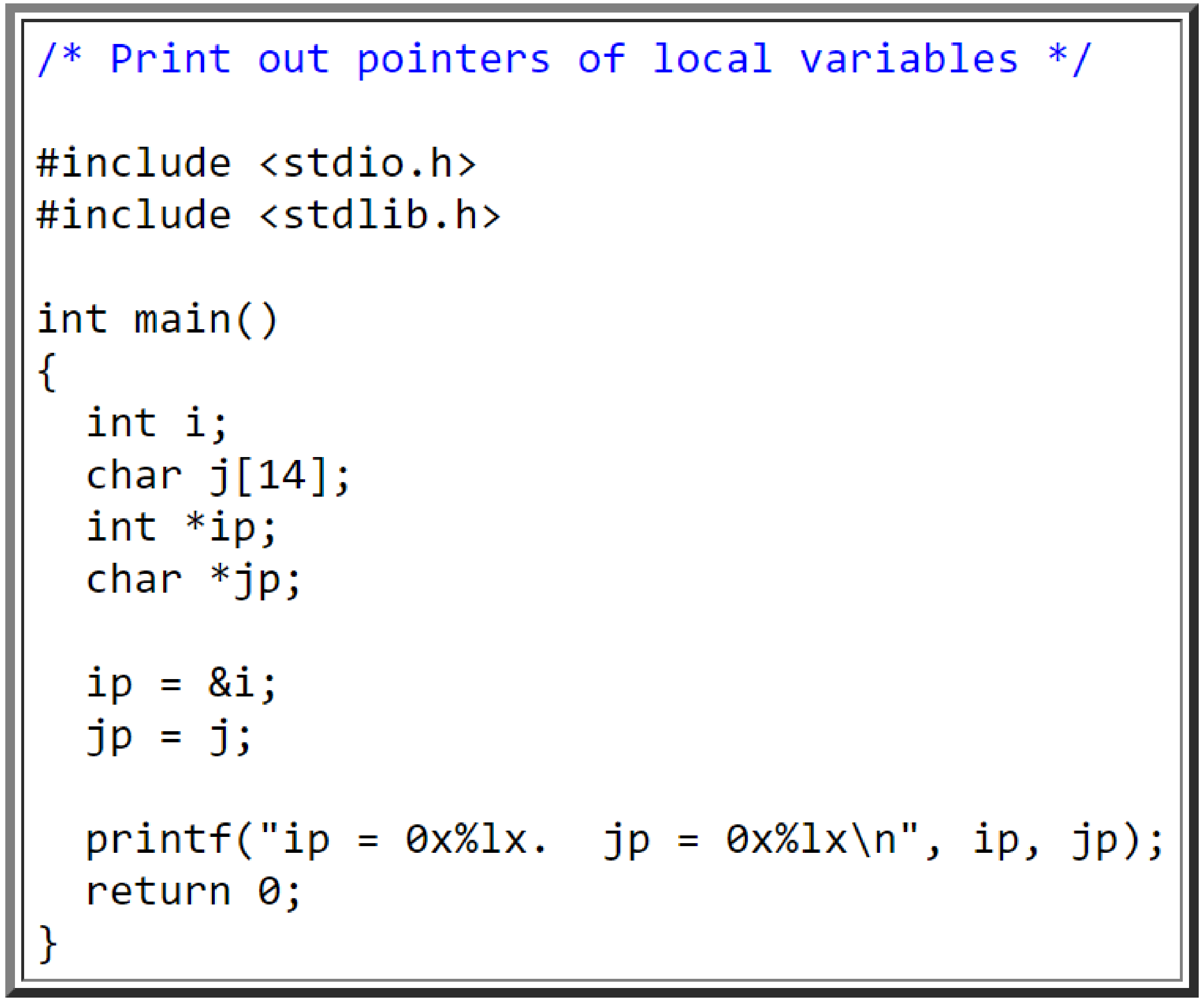
- ❑Pointers are where most people make mistakes in C.
- ❑A pointer is simply an index of memory.
- ❑Memory can be allocated in one of two ways -- by declaring variables or by calling malloc ().
- ❑Once memory is allocated, you can set a pointer to it.
- ❑Whenever we allocate ***x*** bytes of memory, we get ***x from the memory array*** We are <u>**separating adjacent elements**</u> .
- ❑If we set a pointer to these bytes, that pointer will be the index of the first byte allocated in memory.

Pointer

```c
/* Print out pointers of local variables */

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int i;
  char j[14];
  int *ip;
  char *jp;

  ip = &i;
  jp = j;

  printf("ip = 0x%lx.  jp = 0x%lx\n", ip, jp);
  return 0;
}
```

- allocates an integer (**i**), a 14-character string **(j),** and two pointers **(ip and jp) .**
- Then their pointers are set to point to the memory allocated for **i and j .**
- Finally, it prints the values of these pointers - these are the indices of the memory array.
- ' long unsigned While we are compiling we get a ' int ' warning, we will come to this.
- **0x7fff2efcdd9c , 0x7fff2efcdd9d for variable i 0x7fff2efcdd9e** , and **0x7fff2efcdd9f allocated (int** -- 4 bytes)
- **0x7fff2efcdda0 - j array between 0x7fff2efcddad (char --** 1 byte **)**

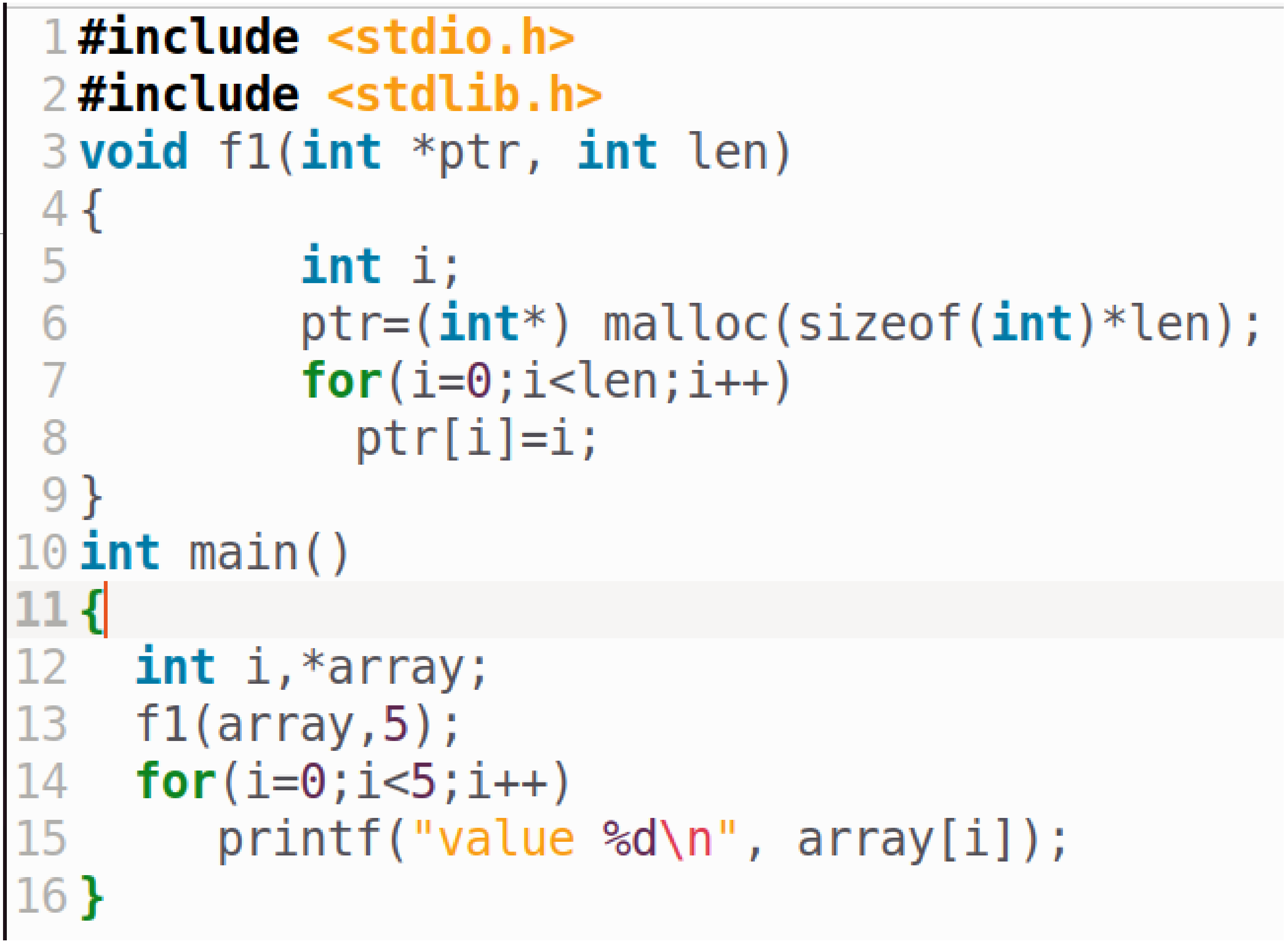
Pointer



```
/* Print out pointers of local variables */

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int i;
  char j[14];
  int *ip;
  char *jp;

  ip = &i;
  jp = j;

  printf("ip = 0x%lx.  jp = 0x%lx\n", ip, jp);
  return 0;
}
```

- `` jp = j'' and ``ip = &i'' attention!
- This is because **an array is equivalent to a pointer** .
- The only difference is that you cannot assign a value to an array variable.
- So you can say `` jp =j'', but you can't say ``j= jp ''
- Also, you cannot get the address of an array variable -- it is illegal to say ``&j''.

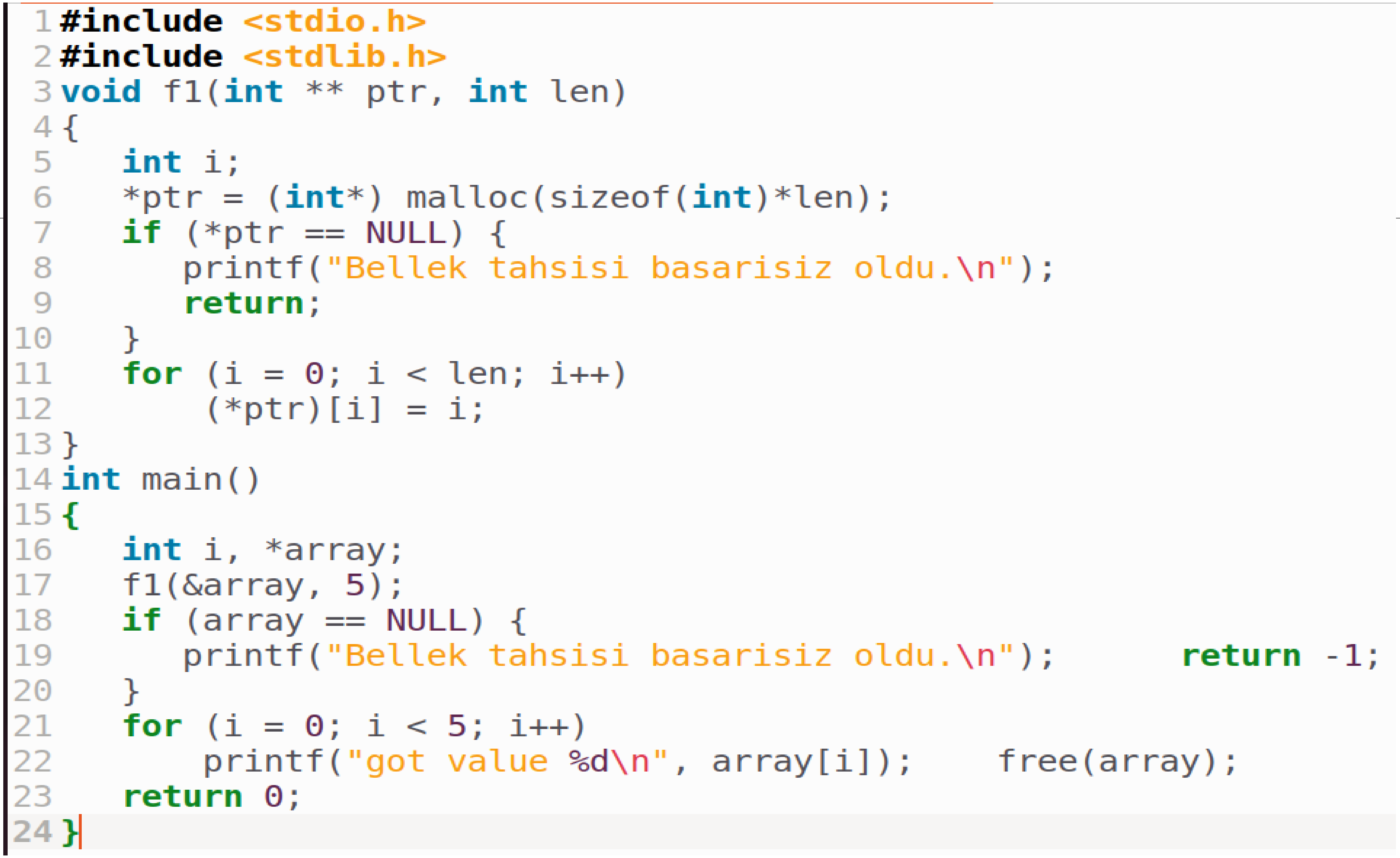
Pointer



```c
#include <stdio.h>
#include <stdlib.h>
void f1(int *ptr, int len)
{
        int i;
        ptr=(int*) malloc(sizeof(int)*len);
        for(i=0;i<len;i++)
          ptr[i]=i;
}
int main()
{
  int i,*array;
  f1(array,5);
  for(i=0;i<5;i++)
     printf("value %d\n", array[i]);
}
```

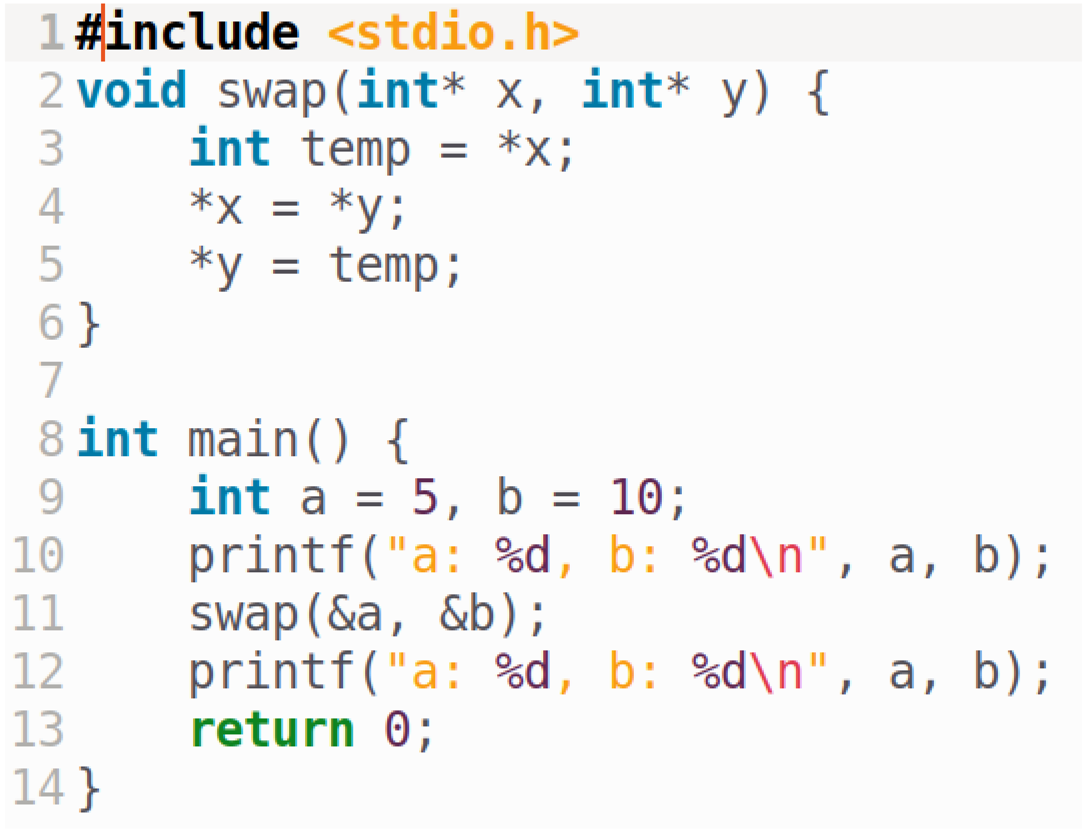
Pointer

- In the above code, function f1 takes a pointer and a length parameter.
- Using the malloc function to create an array from memory .
- This array contains numbers from 0 to **the len variable.** However, there is a major problem with this code.
- In the main function, the pointer variable array is passed to function f1, but since the pointer **itself is reassigned in f1** , the value of the original pointer variable in the main function remains unchanged and the dynamically allocated memory remains unreleased .



```c
#include <stdio.h>
#include <stdlib.h>
void f1(int ** ptr, int len)
{
   int i;
   *ptr = (int*) malloc(sizeof(int)*len);
   if (*ptr == NULL) {
      printf("Bellek tahsisi basarisiz oldu.\n");
      return;
   }
   for (i = 0; i < len; i++)
       (*ptr)[i] = i;
}
int main()
{
   int i, *array;
   f1(&array, 5);
   if (array == NULL) {
      printf("Bellek tahsisi basarisiz oldu.\n");      return -1;
   }
   for (i = 0; i < 5; i++)
       printf("got value %d\n", array[i]);    free(array);
   return 0;
}
```

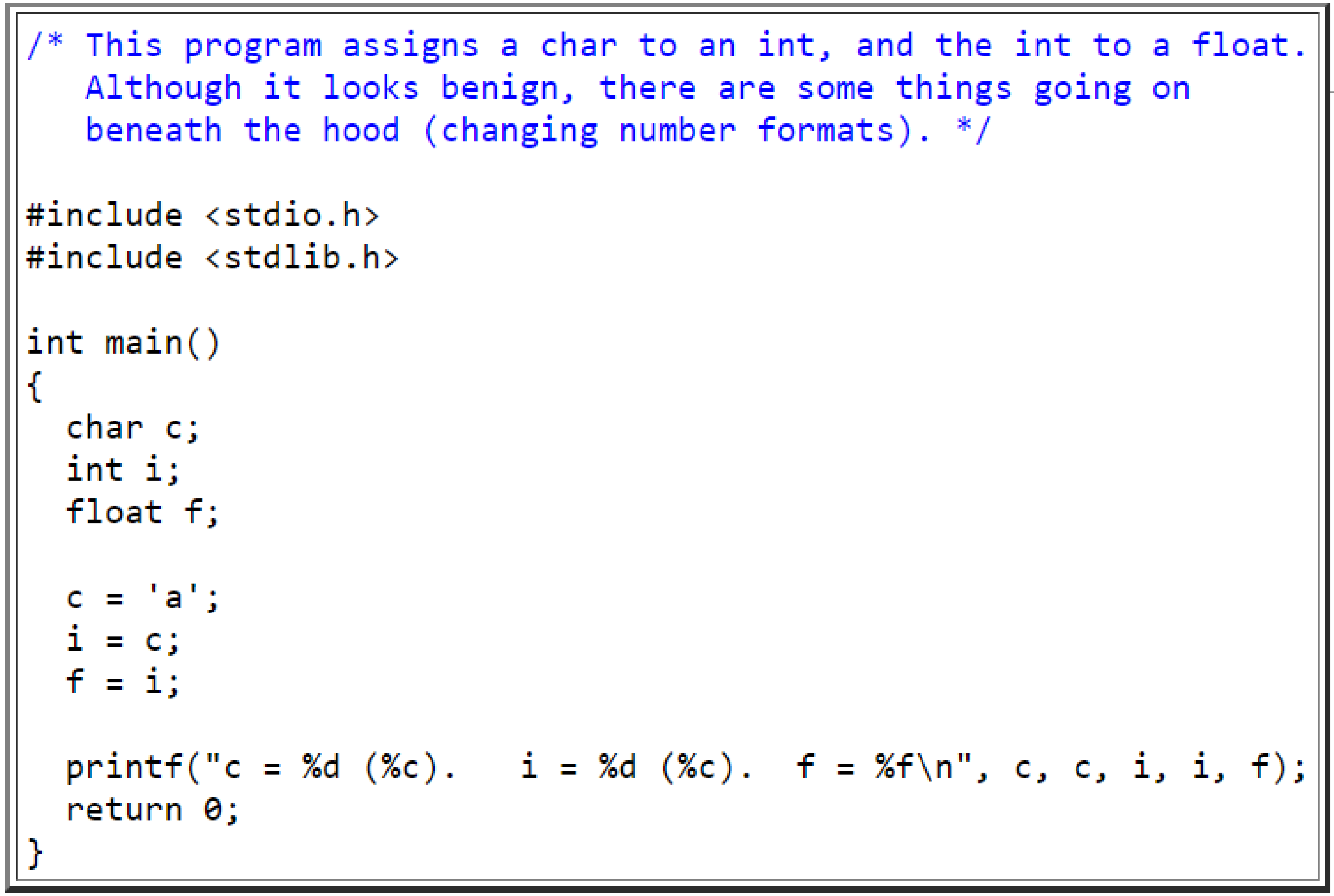
Pointer - Additional Example



```c
#include <stdio.h>
void swap(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 5, b = 10;
    printf("a: %d, b: %d\n", a, b);
    swap(&a, &b);
    printf("a: %d, b: %d\n", a, b);
    return 0;
}
```

- ❑Here, **swap** function takes two integer pointers.
- ❑These pointers are used to change the values of the integers they point to.
- ❑the pointers **themselves that are changed, but the values they point to** .
- ❑Using a temporary variable called **temp** in the function , values are assigned to each other and values are changed.

Type Casting



```c
/* This program assigns a char to an int, and the int to a float.
   Although it looks benign, there are some things going on
   beneath the hood (changing number formats). */

#include <stdio.h>
#include <stdlib.h>

int main()
{
  char c;
  int i;
  float f;

  c = 'a';
  i = c;
  f = i;

  printf("c = %d (%c).   i = %d (%c).   f = %f\n", c, c, i, i, f);
  return 0;
}
```

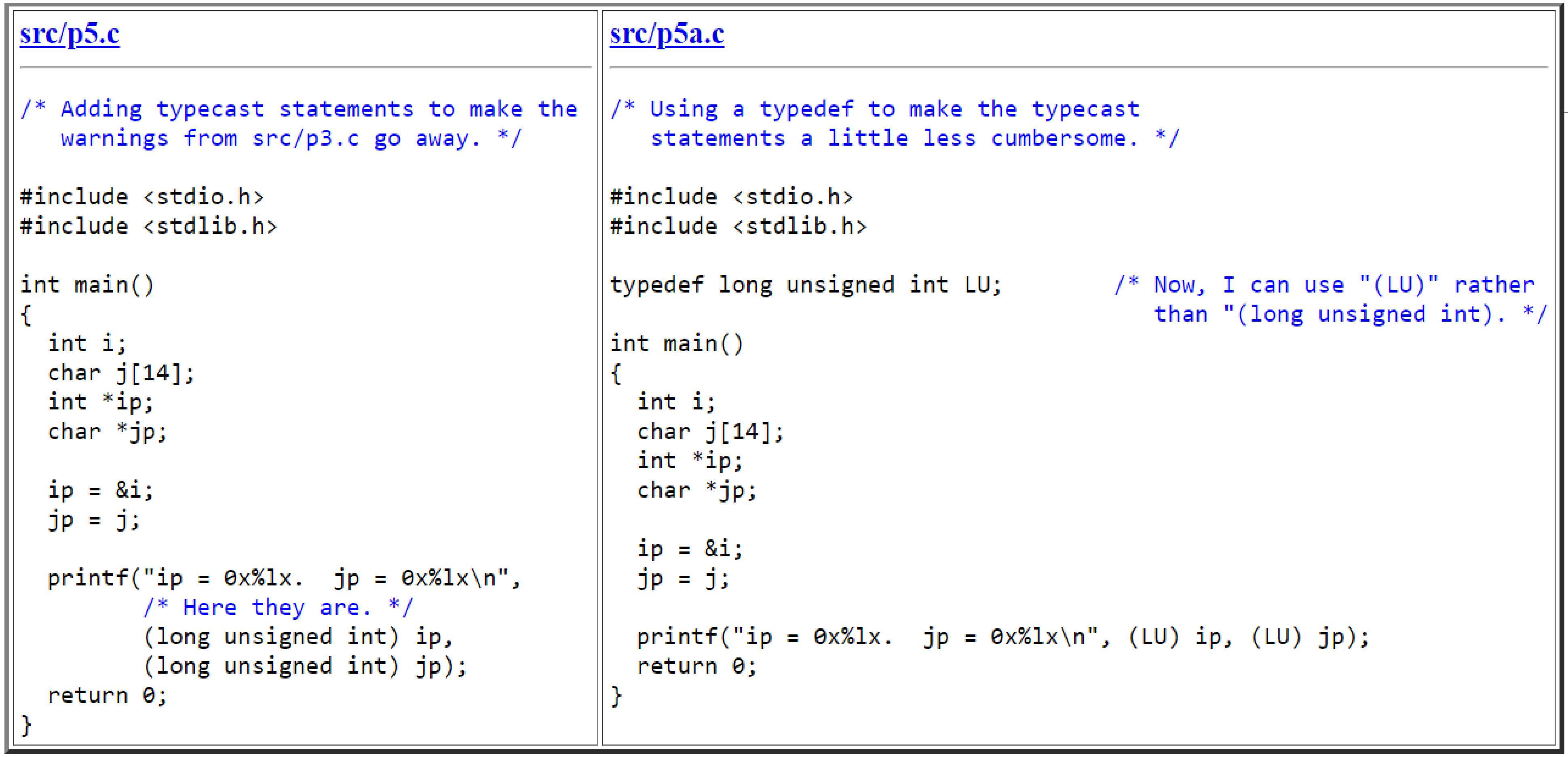
- Sometimes you want to take a variable stored in x bytes and assign it to a variable stored in y bytes.
- This is called ``type conversion''.

•**char** -- 1 byte
•**short** -- 2 bytes
•**int** -- 4 bytes
•**long** -- 4 or 8 bytes, depending on the system and compiler
•**float** -- 4 bytes
•**double** -- 8 bytes
•(pointer -- 4 or 8 bytes, depending on the system and compiler)

Type Casting

- Some type conversions, such as the ones above, are very natural.
- The C compiler will do these things for you without complaint.
- For most others, the C compiler will issue a warning unless you specifically say you're doing a type conversion.
- This is a way of telling the compiler "Yes, I know what I'm doing."

Type Casting



src/p5.c

```c
/* Adding typecast statements to make the
   warnings from src/p3.c go away. */

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int i;
  char j[14];
  int *ip;
  char *jp;

  ip = &i;
  jp = j;

  printf("ip = 0x%lx.  jp = 0x%lx\n",
         /* Here they are. */
         (long unsigned int) ip,
         (long unsigned int) jp);
  return 0;
}
```

src/p5a.c

```c
/* Using a typedef to make the typecast
   statements a little less cumbersome. */

#include <stdio.h>
#include <stdlib.h>

typedef long unsigned int LU;        /* Now, I can use "(LU)" rather
                                         than "(long unsigned int). */

int main()
{
  int i;
  char j[14];
  int *ip;
  char *jp;

  ip = &i;
  jp = j;

  printf("ip = 0x%lx.  jp = 0x%lx\n", (LU) ip, (LU) jp);
  return 0;
}
```

- What happens is that the compiler printf () parses the format string and says "%lx" **long unsigned int It gets** what you want, but an (int *).

  "Yes, it's an (int *), but (long Treat it like an unsigned int) please. I know what I'm doing."

Type Casting

- On some machines (like the Pi, 32 bits-4 bytes), both pointers and int are 4 bytes.
- This has led many people to view pointers and integers as interchangeable.

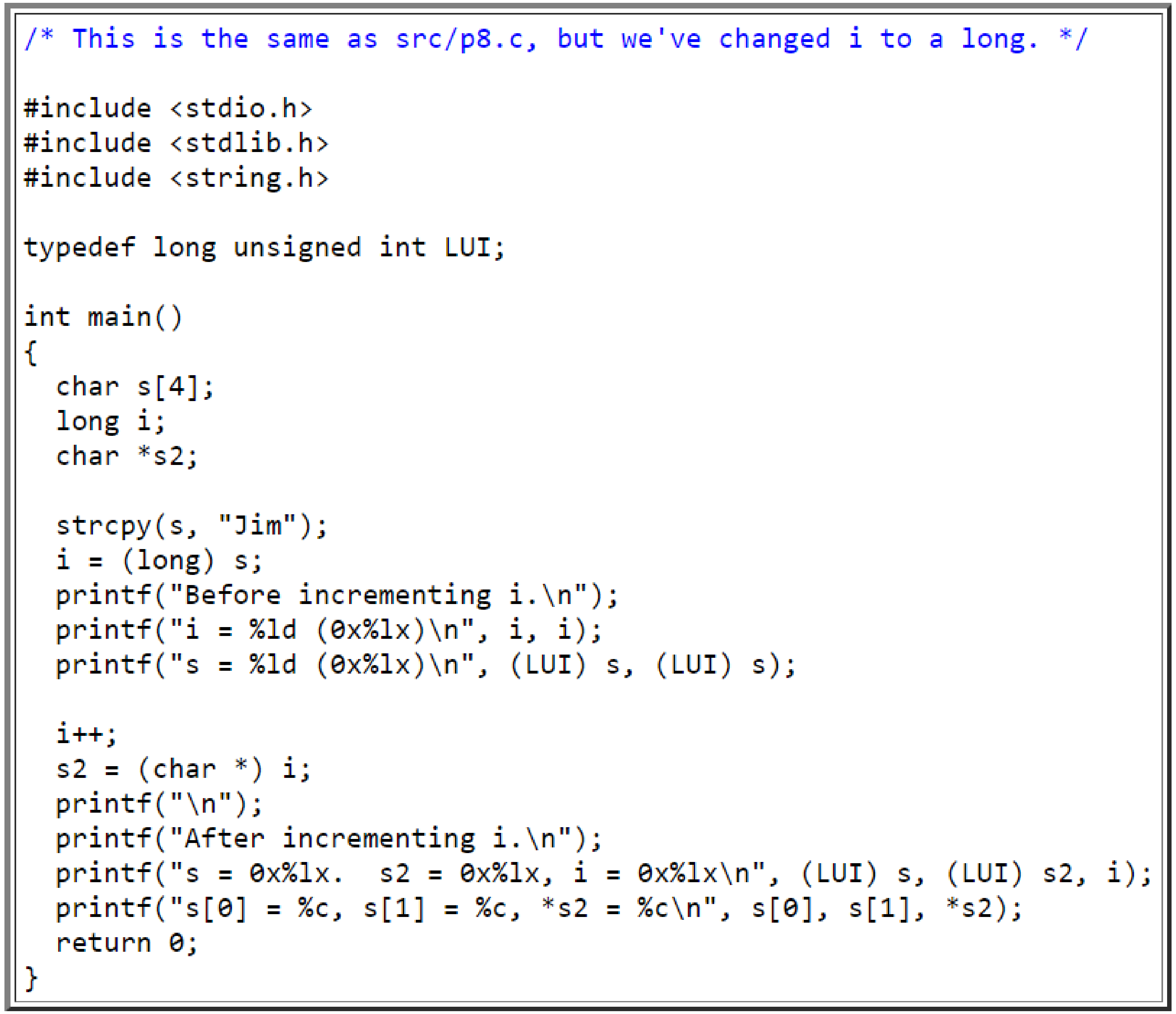
Type Casting

- src /p8.c :
- When you set i equal to s, you lose 4 bytes of information because int is four bytes and pointers are eight bytes.
- When you set s2 back to i, it fills in the four bytes where i is missing, usually with zeros, but sometimes with -1s.
- In both cases, it will be an invalid address and you will get a segmentation violation:
- There is no problem on a 32-bit machine

```
/* A program where we inadvisedly typecast a pointer to an int and back again.
   On machines with 8-byte pointers, this is a buggy activity, because you lose
   data when you typecast from an integer to a pointer. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef long unsigned int LUI;

int main()
{
  char s[4];
  int i;
  char *s2;

  /* Copy the string "Jim" to s, then turn the pointer into an integer i.
     Print out the pointer's value, and i's value. */

  strcpy(s, "Jim");
  i = (int) s;
  printf("Before incrementing i.\n");
  printf("i = %d (0x%x)\n", i, i);
  printf("s = %ld (0x%lx)\n", (LUI) s, (LUI) s);

  /* Now increment i, and turn it back into a pointer.
     Print out the pointers, and then attempt to print out what they point to. */

  i++;
  s2 = (char *) i;
  printf("\n");
  printf("After incrementing i.\n");
  printf("s = 0x%lx.  s2 = 0x%lx, i = 0x%x\n", (LUI) s, (LUI) s2, i);
  printf("s[0] = %c, s[1] = %c, *s2 = %c\n", s[0], s[1], *s2);
  return 0;
}
```

Type Casting

- instead use long for i instead of an int everything works fine,
- because long and pointers are guaranteed to be the same size, whether 4 or 8 bytes.



```c
/* This is the same as src/p8.c, but we've changed i to a long. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef long unsigned int LUI;

int main()
{
  char s[4];
  long i;
  char *s2;

  strcpy(s, "Jim");
  i = (long) s;
  printf("Before incrementing i.\n");
  printf("i = %ld (0x%lx)\n", i, i);
  printf("s = %ld (0x%lx)\n", (LUI) s, (LUI) s);

  i++;
  s2 = (char *) i;
  printf("\n");
  printf("After incrementing i.\n");
  printf("s = 0x%lx.  s2 = 0x%lx, i = 0x%lx\n", (LUI) s, (LUI) s2, i);
  printf("s[0] = %c, s[1] = %c, *s2 = %c\n", s[0], s[1], *s2);
  return 0;
}
```

Malloc and Free

- There is no new or delete in C.
- Their functionality is provided by **malloc () and free ()** library calls.

```
#include < stdlib.h >

void *malloc( size_t size);
void free(void * ptr );
```

- Like new, malloc () **allocates the desired amount of bytes from memory from the operating system.**
- Unlike new, which requires you to provide information about the type of data it allocates, **malloc () simply asks for the number of bytes and, if successful, returns a pointer** to at least that many bytes by the operating system .
- void * meaning it is a pointer , **but malloc () doesn't know what it points to.**

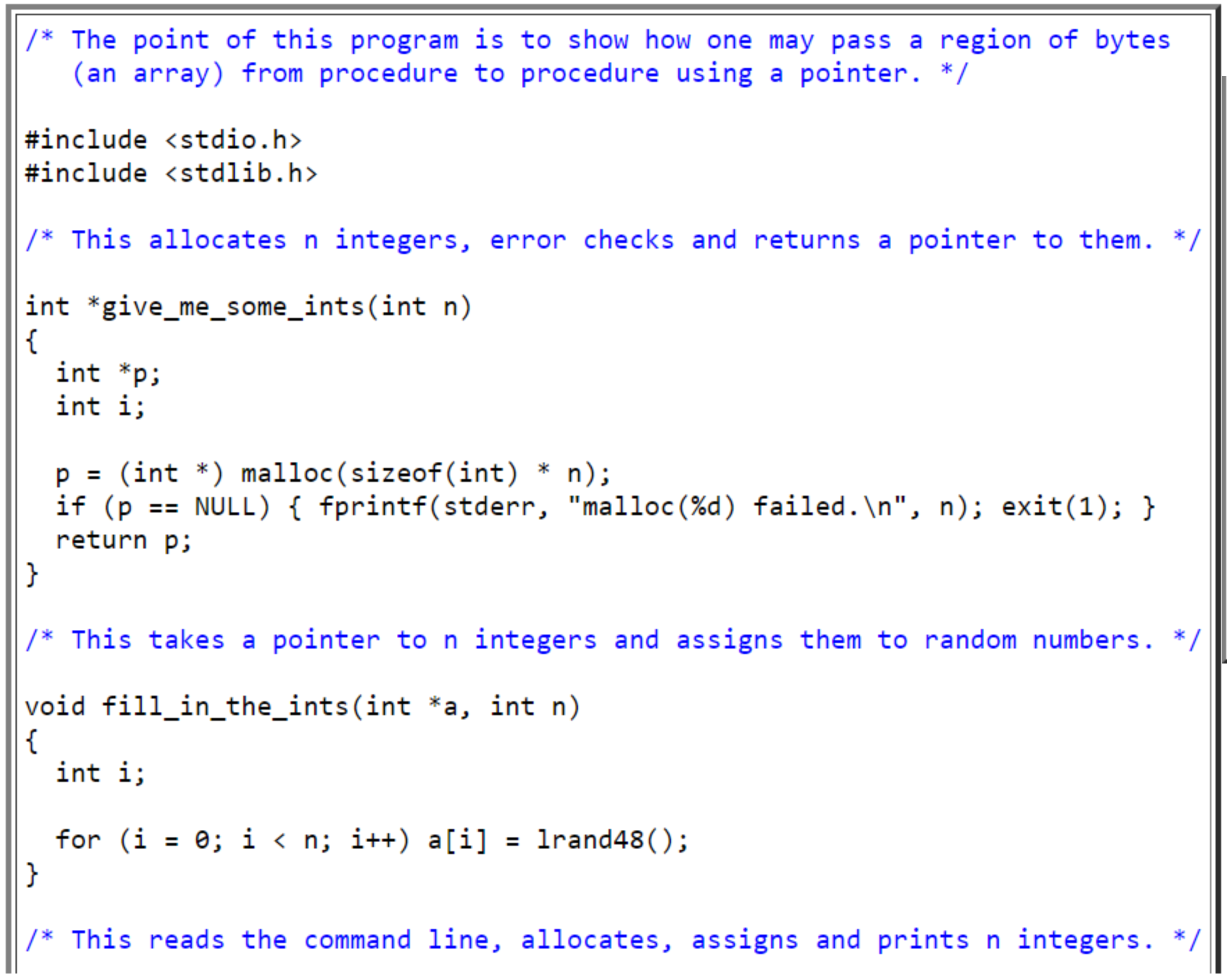
Malloc and Free

- The malloc function allows **programmers to allocate memory at runtime** .
- While programs are running, the memory size required by the program may change for various reasons.
- In this case, functions such as malloc make the needed memory size allocated and available at runtime.
- It is also possible to create non-distributed memory areas to store fixed size data, such as large blocks of memory.
- However**, this process may cause unnecessary complexity for memory management in some cases.** Therefore, the malloc function provides a more suitable solution in such cases.

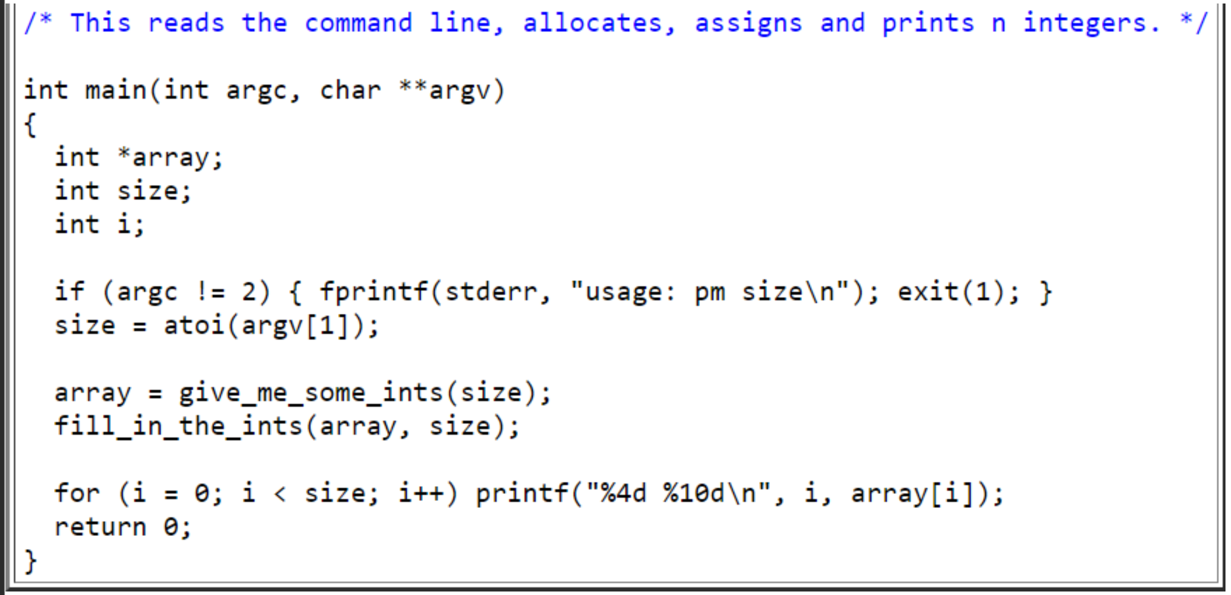
Malloc and Free

- You call **sizeof (type)** to find out how many bytes you need from malloc ().
- You call **malloc (sizeof (int))** to allocate an integer .
- We often want to allocate an array of data types. To do this, you multiply sizeof (type) by the number of elements.
- **Your pointer** will point to the first of these items.
- The next element will be sizeof (type) bytes after the pointer.

Malloc and Free



```c
/* The point of this program is to show how one may pass a region of bytes
   (an array) from procedure to procedure using a pointer. */

#include <stdio.h>
#include <stdlib.h>

/* This allocates n integers, error checks and returns a pointer to them. */

int *give_me_some_ints(int n)
{
  int *p;
  int i;

  p = (int *) malloc(sizeof(int) * n);
  if (p == NULL) { fprintf(stderr, "malloc(%d) failed.\n", n); exit(1); }
  return p;
}

/* This takes a pointer to n integers and assigns them to random numbers. */

void fill_in_the_ints(int *a, int n)
{
  int i;

  for (i = 0; i < n; i++) a[i] = lrand48();
}

/* This reads the command line, allocates, assigns and prints n integers. */
```



```c
/* This reads the command line, allocates, assigns and prints n integers. */

int main(int argc, char **argv)
{
  int *array;
  int size;
  int i;

  if (argc != 2) { fprintf(stderr, "usage: pm size\n"); exit(1); }
  size = atoi(argv[1]);

  array = give_me_some_ints(size);
  fill_in_the_ints(array, size);

  for (i = 0; i < size; i++) printf("%4d %10d\n", i, array[i]);
  return 0;
}
```

Malloc and Free

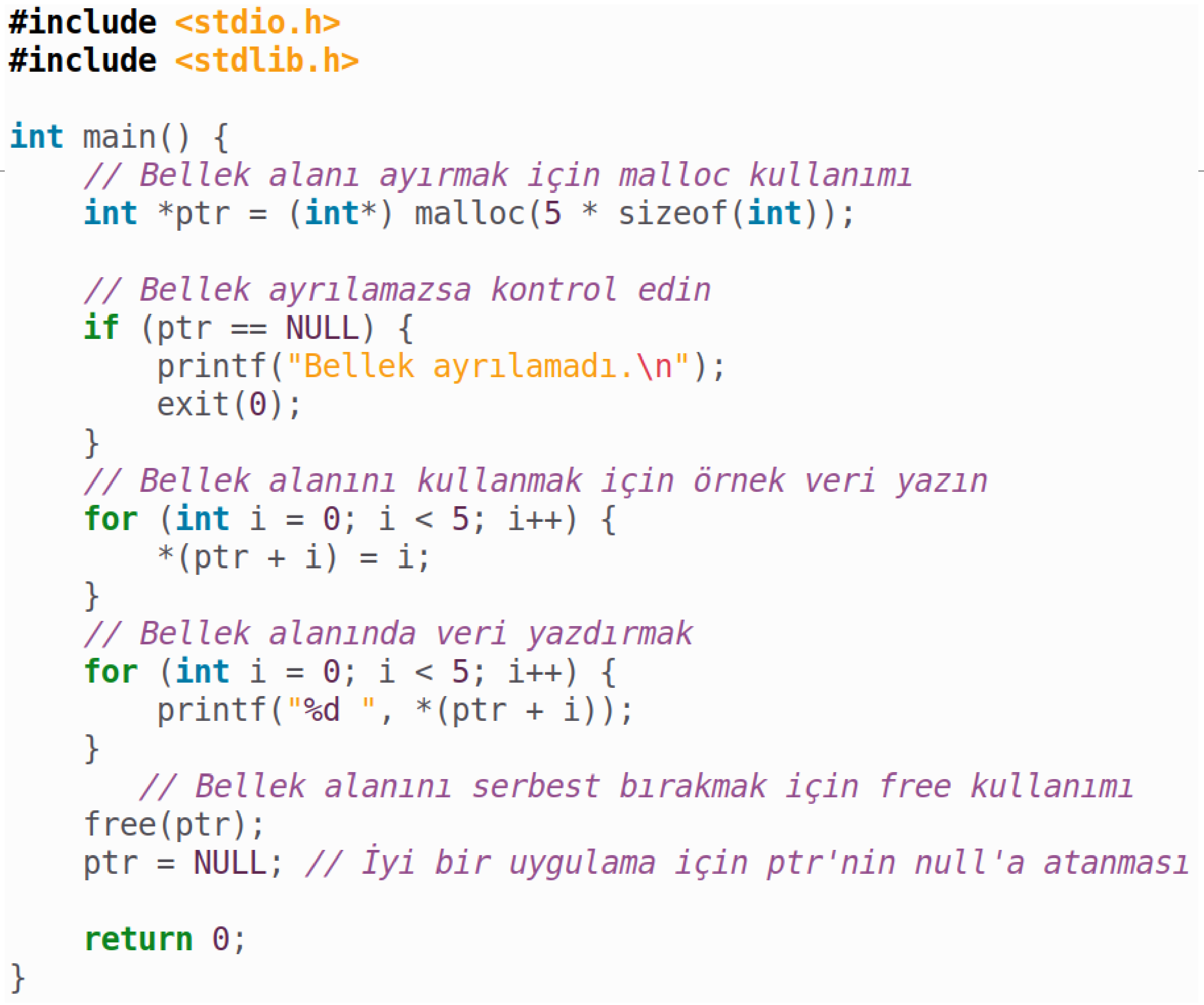
- The give_me_some_ints () procedure allocates an array of n integers and returns a pointer to the array.
- fill_in_the_ints () takes a pointer to the array plus its size and fills it.
- Since we are passing pointers, no copies of the array are made.
- In other words, fill_in_the_ints () fills the array created by the malloc () call.
- Finally, we print the array.

Malloc and Free

- Differences between C and C++;

1. malloc () how many bytes you want.
2. malloc () doesn't know how its memory is used -- it just allocates bytes.
3. You should keep track of the size of the array. This is inconvenient compared to a vector.
4. There are no reference variables in C. Parameters are always copied. Here it is the pointer that is copied, not the data it points to.
5. Therefore, there is only one copy of the array in the above program.

Malloc and Free



```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    // Bellek alanı ayırmak için malloc kullanımı
    int *ptr = (int*) malloc(5 * sizeof(int));

    // Bellek ayrılamazsa kontrol edin
    if (ptr == NULL) {
        printf("Bellek ayrılamadı.\n");
        exit(0);
    }
    // Bellek alanını kullanmak için örnek veri yazın
    for (int i = 0; i < 5; i++) {
        *(ptr + i) = i;
    }
    // Bellek alanında veri yazdırmak
    for (int i = 0; i < 5; i++) {
        printf("%d ", *(ptr + i));
    }
       // Bellek alanını serbest bırakmak için free kullanımı
    free(ptr);
    ptr = NULL; // İyi bir uygulama için ptr'nin null'a atanması

    return 0;
}
```

Q1-Write a C function that copies the contents of one array into another using malloc..

Here's a C function that copies the contents of one array into another using malloc:

```c
#include <stdlib.h>

void copyArray(int *src, int **dest, size_t len) {
    *dest = (int*)malloc(len * sizeof(int));
    if(*dest == NULL) {
        // Handle allocation failure.
        return;
    }
    for(size_t i = 0; i < len; ++i) {
        (*dest)[i] = src[i];
    }
}
```

This function takes as input a source array (`src`), a pointer to the destination array (`dest`), and the length of the arrays (`len`). It allocates memory for `dest` using `malloc`, then checks if the allocation was successful. If it wasn't, it returns without doing anything else. Otherwise, it proceeds to copy each element from `src` to `dest`.

Q2-How would you handle the allocation of 2D arrays with malloc()?

To allocate a 2D array with malloc(), we first allocate memory for the row pointers. Then, for each row pointer, we allocate memory for the columns. This creates an array of pointers to arrays, forming our 2D structure.

Here's an example in C:

```c
int **array;
int rows = 5, cols = 10;

// Allocate memory for row pointers
array = (int **)malloc(rows * sizeof(int *));
if(array == NULL) {
    fprintf(stderr, "Out of memory");
    exit(0);
}

// Allocate memory for each row
for(int i=0; i<rows; i++) {
    array[i] = (int *)malloc(cols * sizeof(int));
    if(array[i] == NULL) {
        fprintf(stderr, "Out of memory");
        exit(0);
    }
}
```

This code allocates a 5×10 integer array. If allocation fails at any point, it prints an error message and exits. Always remember to free your allocated memory when you're done using it to prevent memory leaks.