PALLAVI SINGH
Posted on 22 авг. 2021 г. • Updated on 26 авг. 2021 г.

Structure padding in C- Data Structures

#structurepadding #begginers #concepts

In this article we will learn about the structure padding in C in detail.

Table of contents:

Let’s recall some basics

The memory is assigned/allocated to the members of structure only after the object is declared. Once we declare the object continuous block of memory is allocated to the structure memory.
It will be allocated sequence wise as they are declared.

```
1.  struct student
2.  {
3.      char a;  //1 byte
4.      char b;  //1 byte
```

```
5.     int c;  //4 byte
6.  } stud1;
```

Why structure padding?

Let’s understand with an example:

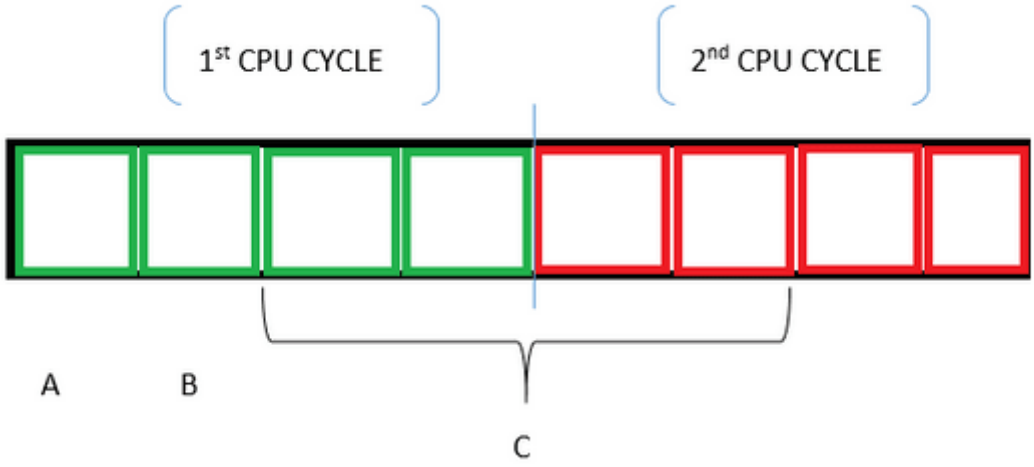
```
1.  struct student
2.  {
3.     char a;  //1 byte
4.     char b;  //1 byte
5.     int c;  //4 byte
6.  } stud1;
```

Normally what we do to calculate the size of the structure is add the size of all the data members present in the structure.
So considering the above given size of every data type ,the size of structure object is 6 bytes acc to our general rule.

But this answer is wrong. Now, we will understand why this answer is wrong? We need to understand the concept of structure padding.

Concept of padding

- The processor reads 1 word at a time not 1 byte at a time.
- A 32-bit processor -> 1 word at a time-> 4 bytes-> 1 CPU cycle (1 word=4 bytes).
- A 64-bit processor -> 1 word at a time-> 8 bytes-> 1 CPU cycle (1 word=8 bytes).
- The number of CPU cycles are inversely proportional to performance .
- It means the more number of cycles CPU takes for performing a specific task the lesser will be it’s performance.
- With increase in the number of CPU cycles the performance is decreasing.



Here if we distribute the no of CPU cycles needed to access each member:

A → 1 CPU cycle.

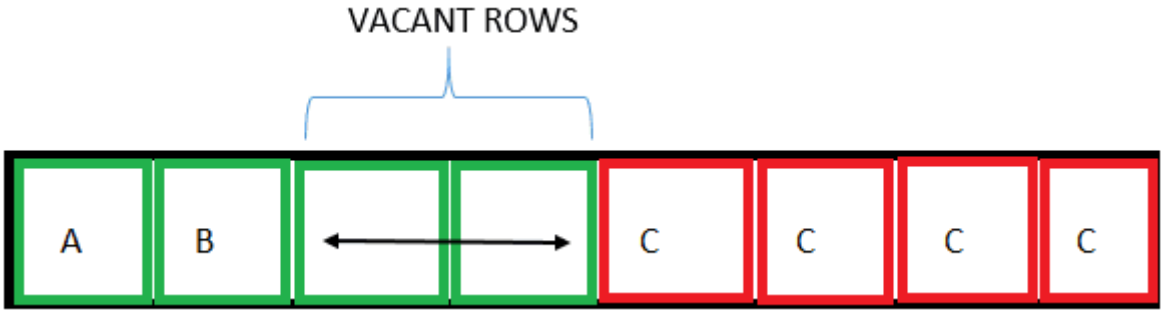
B → 1 CPU cycle.

C → 2 CPU cycles.

- This is evident that it is an unnecessary wastage of CPU cycles. And here the concept of structure padding is introduced.

How it works?

It is done by the compiler automatically and it saves the no of CPU cycles and hence improves the performance.



Here if we distribute the no of CPU cycles needed to access each member:

A → 1 CPU cycle.

B → 1 CPU cycle.

C → 1 CPU cycle.

Now the size of the object is 8 bytes not 6 bytes.
For more clarification let's discuss the distribution of space:

- A occupied=1 byte.//char datatype
- B occupied=1 byte//char datatype
- Vacant rows created occupied=2 bytes C occupied= 4 bytes.//int datatype.

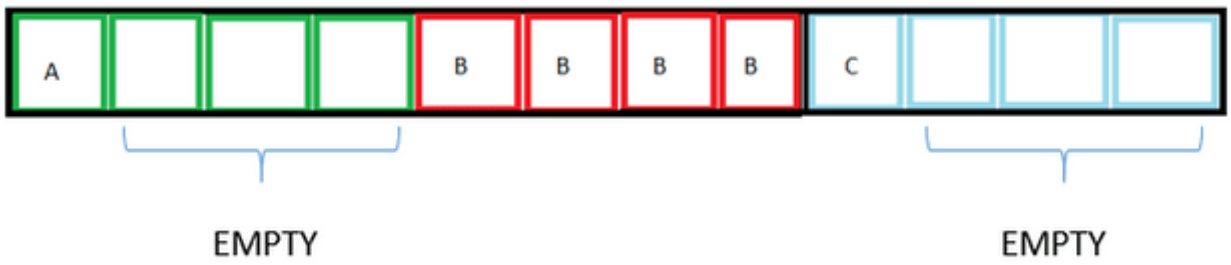
Hence the size of the structure object is 1+1+2+4=8 bytes.
So here we have improved the performance but the memory is wasted due to creation of vacant rows.

More examples to understand better

Let's look at another example :

```
1.  #include<iostream.h>
2.  struct student
3.  {
4.     char a;
5.     int b;
6.    char c;
7.  };
8.  int main()
9.  {
10.    struct student stud1; // variable declaration of the student type..
11.    // Displaying the size of the structure student.
12.    printf("The size of the student structure is %d", sizeof(stud1));
13.    return 0;
14. }
```

Let's look at the allocation of memory:



The total size of the structure object is 4+4+4=12 bytes.

How to avoid the structure padding in C?

The structural padding is an in-built process that is automatically done by the compiler. Sometimes we need to avoid it because it increases the size of the structure from it's actual size.

We can avoid padding using 2 ways:

- Rearranging the attribute.
- Using #pragma pack(1)

Rearrangement of attributes/variables.
There is one way to reduce the memory wastage manually due to padding.
We can align the data variables in such order that the variable containing more size will be declared first and then the variables having small size should be declared.

Understand through example.

```
#include<iostream.h>
struct student
{
   Int a;
   char b;
  char c;
};
int main()
{
   struct student stud1; // variable declaration of the student type..
   // Displaying the size of the structure student.
   printf("The size of the student structure is %d", sizeof(stud1));
   return 0;
}
```

Let’s look at the allocation:

The total size of the struct object is 4+1+1=6 bytes.

Using #pragma pack(1) directive

```
#include<iostream.h>
#pragma pack(1)
struct base
{
    int a;  //4 bytes
    char b;  //1 byte
    double c;//8 bytes
};
int main()
{
  struct base var; // variable declaration of type base
  // Displaying the size of the structure base
  printf("The size of the var is : %d", sizeof(var));
return 0;
}
```

Here if we avoid using pragma here then the size of the object structure will be 4+4+4+4=16 bytes.
But the actual size of the structure members is 13 bytes, so 3 bytes are wasted. To avoid the wastage of memory, we use the #pragma pack(1) directive to provide the 1-byte packaging.

Discussion (0)

PALLAVI SINGH

LOCATION
Delhi

JOINED
7 мая 2021 г.