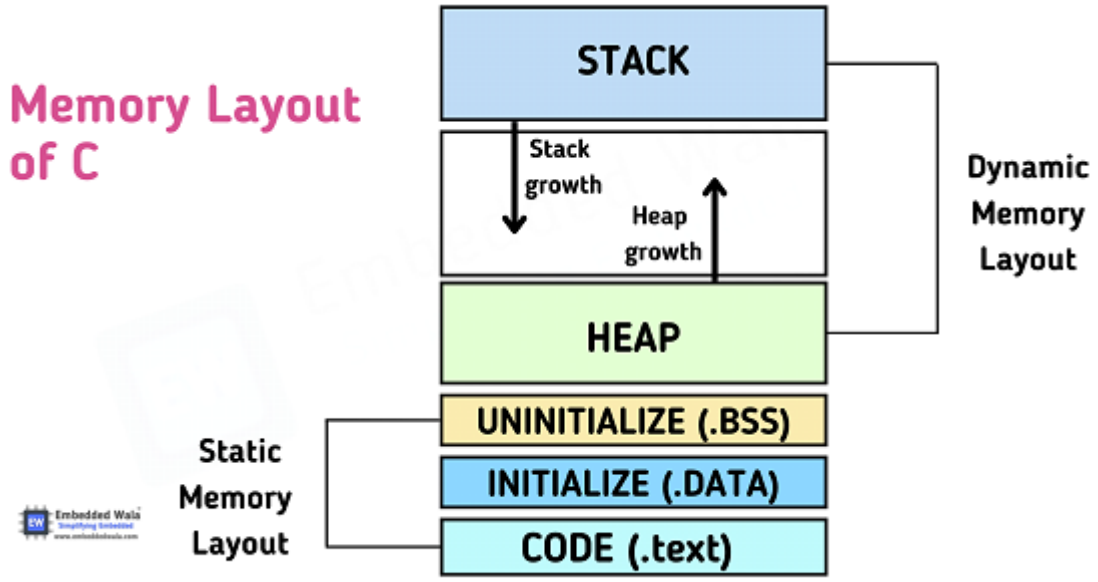
Теста ще включва въпроси от следния материал:

- Целия конспект за мрежи (6-та тема)

Сегментите на програмата



Живот и разположение на променливите

За да можем да обсъждаме живота на една променлива първо трябва да дефинираме понятието обхват(scope). Обхватът е част от кода, която е отделена от останалия код, в смисъл, че нищо извън дадения обхват не може да променя нещата в него. В C обхвата се изразява в къдрави скоби{}. Както се досещате всички цикли, функции и структурни дефиниции имат собствен обхват в които се изпълняват(void func(){ …. }). За да можем да си говорим за разположението на една променлива трябва първо да изясним къде може да бъде разположена тя. Една програма работи главно с 4 вида памет: code, data, stack, heap. В генералния случай тези 4-ри типа работят по следния начин:

- Code – в този сегмент се съдържат поредицата от инструкции, които процесора ще изпълнява.
- Data – в този сегмент се съдържа информация която трябва да е достъпна по всяко време на изпълнение на програмата.
- Stack – в този сегмент се запазват всички локални променливи в даден обхват, които не са достъпни по всяко време.
- Heap – той се използва за динамична памет, която ще бъде разгледана по-натам.

С тези неща изказани могат да се обособят три основни типа променливи:

- Локални;
- Глобални;
- Статични;
-

2.1 Локални променливи

Локалните променливи се създават по време на изпълнение на програмата. Те съществуват само в дадения обхват. При излизане/затваряне на обхвата паметта маркиран за тях се освобождава, в смисъл, че вече може да бъде използвана от други програми. Те се намират в stack частта от паметта на програмата понеже тяхното съществуване е временно.

2.2 Глобални променливи

Една глобална променлива съществува от самото пускане на програмата до нейния край. Една глобална променлива може да бъде достъпвана по всяко време независимо дали в момента сме отделени от обхват или не. Заради тези си характеристики глобалните променливи се намират в data сегмента на паметта. Те трябва да са активни през цялата програма

2.3 Статични променливи

Подобно на глобалните променливи, статичните променливи също съществуват по време на цялата програма. По отношение на живот и местоположение глобалните и статичните променливи са напълно идентични, тоест те също се намират в data сегмента и съществуват докато програмата съществува. Основната разлика при двата типа променливи ,е че статичните могат да бивaт достъпвани само във обхвата в който са дефинирани. Тоест една статична променлива ако е дефинирана във функция тя може да бива прочитана и променяна само в тялото на тази функция, но паметта за нея остава заделена дори след края на изпълнението на функцията. Това ни позволява да запазваме и пренасяме информация между отделните извиквания на една и съща функция, без това да пречи на останалата част от програмата.

Сигнатура на функция (<тип> <име> (<списък с параметри>))

Структура на функция в програмния език C

Всяка функция, която се реализира в C има структура като тази:

```
<тип> <име>(<списък с параметри>)
{
	//Код за изпълнение
	return <резултат от изпълнението>;
}
```

Където:

<тип> - това е типът на резултата, който функцията ще върне. Типът може да е най-разнообразен – могат да се използват елементарните типове от данни в C, разгледани в предходните упражнения, както и потребителски типове, които ще разгледаме в следващи такива. Възможно е някои функции да не се налага да връщат резултат, тогава в полето се записва “void”.

<име> - това е името на функцията, чрез която ще бъде използвана в програмата. Името на функцията е като име на променлива, с тази разлика, че след него се поставят скоби и следват евентуалните параметри

<списък с параметри> - параметри, които ще се използват във функцията, които са записани един след друг, като всеки параметър си има тип и име. Параметрите

записани във вида си трябва да бъдат подадени по абсолютно същият начин и при извикването на функцията.
върнат резултат – всяка функция, която има тип, различен от void трябва задължително да върне резултат от изпълнението си. Този резултат се използва от програмата, където е извикана тази функция. Резултата може да се присвоява на променлива, подава като параметър на друга функция или използва в някой от наличните оператори.

Рекурсия - това една функция да извиква отново себиси докато тя още продължава. Това води до голяма струпване на информация в стека, което води до огромно заделяне на памет, което от своя страна води до терминиране на програмата. И това прави рекурсията метод, който трябва да бива избягван в генералния случай на имплементация на някакъв алгоритъм.

Символни низове (Стрингове)
char str [] = “Hello”; - тука ‘\0’ се слага автоматично и sizeof(str) ще е равно на 6
това е аналогично на:
char str[] = {‘H’,’e’,’l’,’l’,’o’,’\0’};
Винаги заделяйте място за елемента '\0' за край на низ!!!
Разликата между дин обикновен масив от char-ове и символен низ е наличието на терминираща нула (‘\0’) накрая на масива.

Указатели
- **реферирane(&)** - прилага се върху дадена променлива, за да разберем кой е първия от адресите, на които е записана тя(винаги връща адрес, нещо от сорта на 0x56A34DD)
- **дереферирane(*)** - прилага се върху даден адрес, за да взема стойността записана на него. Вече от това какъв е типа зависи колко последователни байта след този адрес ще се прочетат.(Например ако е *int* ще се прочетат 4, при *char* - 1, при *float* - sizeof(float) на брой байта и тн.)

Важно е да се отбележи, че всички указатели са с еднакъв размер, без значение към какъв тип сочат. Това е така защото всичките просто държат адрес, без оглед на това към какво сочи този адрес

Тъй като всеки указател държи адрес, ние спокойно може да си прибавяваме числа към него, за да взимаме друг адрес на базата на указателя. Това действие се нарича аритметика с указатели.

```
{
	int* ptr = &var; //0x0d44
	ptr+1 ------> това ще ни даде 0x0d48, без да променя указателя;
	ptr += 1 ------> това е все едно да напишем (ptr = ptr +1), и така ще ни даде
същата стойност от горе, но също така ще промени и на къде сочи указателя
}
```

Както се забелязва ptr+1 не ни дава директно следващия адрес. Това реално с колко адреса ще се отместим зависи от типа на указателя. Понеже int* сочи към int, който е 4 байта, то като прибавим 1 ще се отиде на следващикя int, а 4рите байта "заети" от сегашния ще се прескочат. Тази методолога се използва за да може по ефикасно да итерираме през поредици елементи(масиви). Тък трябва да вметнем, че името на масива е реално просто указател към първия елемент. Именно това ни позволява да правим аритметиката без проблеми
int arr[3]; int* ptr = a; ---> *(a+2) е същото като a[2]и ще ни върне стойността на променливата еквивалента на втория индекс от масива.

За по-гъвкава работа с указатели може да се възползваме и от кастване(type casting), чието единствено предназаначение е да смени типа на дадения указател временно, без да променя адреса, към който сочи.
int* ptr = &var;   printf("%c", *(char*)ptr);
Тук дереферираме кастнатия към char* указател. Следователно той ще се държи като char* и при дерефериране ще ни върне само 1 байт. В случая първия байт от променливата вар.

Важно е да се отбележи, че освен с кастване този ефект може да се постигне като просто използваме указател към друг тип. Например:

int a = 5;
char* p = &a;
 print(*p) - //псевдо код// - това цели да покаже, че въпреки че гледаме променлива от тип int, все пак я гледаме през char*, тоест ще видим само първия байт от неговите 4; така ако го комбинираме с аритметиката p+1 е втория байт, p+2 - третия и p+3 е байта в който реално е записана 5-тицата. Тоест реда:
printf(%d, *(p+3)) ще принтира 5;

Структури - Декларация
struct <Име>
{
	<тип на поле 1> <име на поле 1>;
	.
	.
	.
	<тип на поле N> <име на поле N>;
}

функцията typedef се използва за ‘преименуване’ на типове
typedef <преименуван тип> <ново име>;

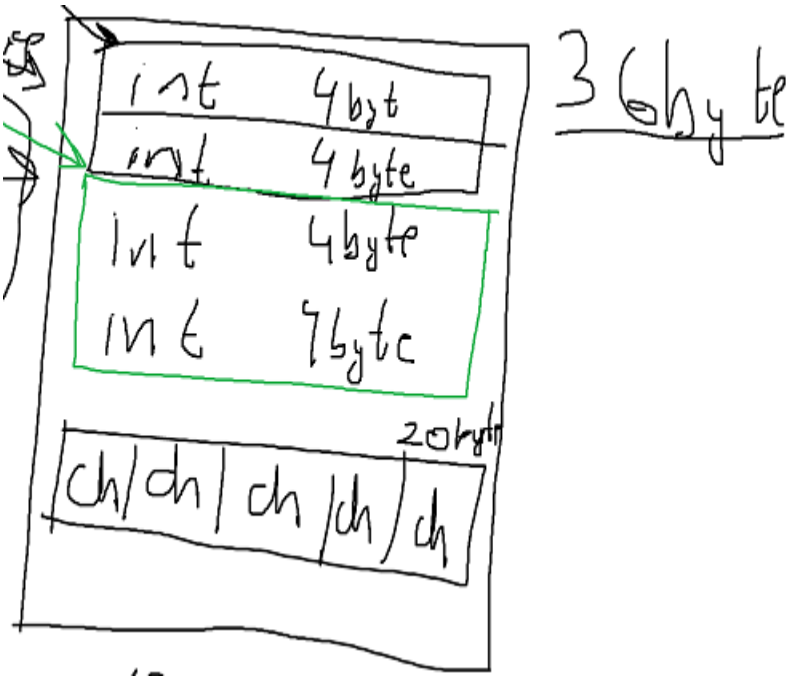
За повече информация и примери погледнете конспекта за структурите (тема 9)

Комбинирано ползване на всичко досега

```
struct Coords
{
	int x;
	int y;
};

typedef struct Coords
Cords;

typedef
struct Plague
{
	Cords center;
	int radius;
	int victims;
	char causer[20];
} Plague;
```




```
Plague b;  // - b е
инстанция на struct Plague
b.radius = 40; // достъпваме полето radius на  b
b.center.x = 160; // достъпваме полето x на center, което е поле на  b

Plague* ptr = &b;
(*ptr).victims = 0; //дереферираме указател, за да вземем реалната променлива и на
нея достъпваме полето victims
ptr->victims = 1; // достъпваме полето victims на  b през указателя ptr
ptr->center.y = 340; // достъпваме полето x на center, което е поле на  b достъпно през
указателя ptr

*тук става малко funky*

Cords* p = &b; //- декларираме указател от тип Cords, който да сочи към b
В случая b не е от тип Cords, но това не е от значение, защото p ще вижда само 8те
байта, за която знае (x и y полетата на Cords)

p->x отговаря на b.center.x
p->y отговаря на b.center.y

(p+1)->x отговаря на b.radius
(p+1)->y отговаря на b.victims

(p+2)->x отговаря на първите 4 байта от масива b.causer (от 0лев до 3ти)
(p+2)->y отговаря на вторите 4 байта от масива b.causer (от 4ти до 7ми)

(p+3)->x отговаря на третите 4 байта от масива b.causer (от 8ти до 11ти)
```

(p+3)->y отговаря на четвъртите 4 байта от масива b.causer (от 12ти до 15ти)

(p+4)->x отговаря на последните 4 байта от масива b.causer (от 16ти до 19ти)
(p+4)->y отговаря на 4 байта, които вече са извън структурата b

тоест нещо от сорта на

(p+2)->x = 0b0100 0001 0000 0000 0000 0000 0000 0000 просто ще направи първия символ от масива b.causer да е равен на 'A'. Това е така, защото първия байт от това двоично число отговаря на 65, което е аски кода на 'A', другите 3 байта са нули. И тъй като (p+2)->x вижда първите 4 байта (защото x е int) на масива b.causer, то този първи байт е първия елемент на масива.

```
typedef
struct
{
        int cifra;
        char array[8];
} Test;
```

Test* tp = &b; //правим указател от тип Test, който да сочи към адреса на b. Подобно на горния пример типовете на указателя и нещото, към което сочи се разминават, но това няма значение, защото указателя просто ще вижда 12-те байта, за които знае, от декларацията на Test

tp->cifra отговаря на b.center.x
tp->array отговаря на 8-те байта след това, тоест b.center.y заедно с b.radius

(tp+1)->cifra отговаря на b.victims
(tp+1)->array отговаря на 8-те байта след това, тоест първите 8 байта от b.causer (от 0лев до 7ми)

(tp+1)->cifra отговаря на следващите 4 байта от b.causer (от 8ми до 11ти)
(tp+2)->array отговаря на 8-те байта след това, тоест последните 8 байта от b.causer(12ти до 19ти)

тоест нещо от сорта

(tp+2)->array[7] = 66 и *((tp+2)->array+7) = 66 са напълно еквиваленти и ще направят последния символ от масива b.causer да е равен на 'B'