Разпределение на паметта при генериране на код

Разпределение на паметта

Дефиниции

При генериране на код се извършва разпределение на паметта и създаване на структури за управление на стек (stack) и heap.

Област за данни – последователност от клетки в паметта, заделени за логически свързани данни.

Понякога (но не винаги) една област от данни съответства на един блок от програмата.

По време на компилирането клетките, заделени за някои променливи могат да бъдат представени чрез двойки от числа (напр. (3,0), (3,1)):

- Първото число означава брой на данните;
- Второто число означава отместване.

При генериране на инструкция тази двойка се конвертира в действителен адрес на променливата. Адресът се получава чрез задаване на базов адрес и регистър на отместване. Пример:

(брой данни, отместване) => (базов адрес, отместване)

Областта за данни може да бъде:

- статична – паметта се разпределя за цялото време на изпълнение на програмата. Достъпът става чрез абсолютни адреси, а не с двойка `<base, displacement>`.
- динамична – паметта се разпределя динамично по време на изпълнение. Областта от данни може да се създава и изтрива. При изтриване на област всички данни в нея се губят.

Пример:

Процедури за създаване и изтриване на блок от паметта:

- `GETSTORAGE(Base, SIZE)` – създава памет с размер SIZE със стартов адрес Base. Следващото извикване на GETSTORAGE ще резервира друга област от данни (с друг базов адрес). Базовият адрес на създадената памет може да бъде записан в клетка от паметта или в регистър.
- `FREESTORAGE(ADDRESS, SIZE)` – унищожава памет със стартов адрес ADDRESS и размер SIZE.

Стекова памет (стек)

- Използва се за съхраняване на параметри на функции и на техните локални променливи
- Стековата памет се разширява и свива при запис и четене на локални променливи
- Няма необходимост от управление на паметта от страна на програмиста, защото стойностите на променливите се записват в стека и се изтриват от него автоматично
- Управлява се от CPU – бързо четене и запис
- Стековата памет има ограничен размер
- Променливите съществуват в стека само докато се изпълнява функцията, която ги е създала

Heap памет

- Управлява се от програмиста (запис и изтриване на променливи) – напр. чрез malloc(), calloc(), realloc(), free()
- Данните са достъпни от всички подпрограми (функции и процедури) в програмата
- Няма ограничения за размера на паметта
- По-бавен достъп за четене и запис
- Може да се получи фрагментиране на области от паметта

Представяне на данни

Прости променливи: обикновено се представят в памет с фиксиран размер:

- символи: 1 или 2 байта
- цели числа: 2, 4 или 8 байта
- реални числа: от 4 до 16 bytes
- булеви данни: 1 бит (най-често се използва 1 пълен байт)

Указатели: обикновено се представят като цели числа без знак. Размерът на указателя зависи от адресното пространство на използваната архитектура. Най-често се използват 4 байта за съхраняване на адрес, като по такъв начин се адресира пространство от 4GB.

Представяне на данни

Едномерни масиви: представят се като съседни блокове от елементи. Размерът на масива е равен най-малко на броя на елементите, умножен по размера памет за представяне на един елемент. Елементите се разполагат последователно, започвайки от първия елемент, от малките към по-големите адреси в паметта. Ако базовият адрес на масива е известен, компилаторът може да генерира код за изчисляване на адреса на всеки елемент чрез отместване на базовия адрес:

&arr[i] = arr + 4 * i (4-byte integers; arr = base address).

Многомерни масиви: за представяне на многомерни масиви се използва един от следните методи*:* действителен многомерен масив или масив от едномерни масиви.

Действителен многомерен масив се разполага последователно. Може да се разположи по редове (напр. в C, Pascal), където редовете се разполагат един след друг. Може да се разположи и по колони (напр. Fortran), където колоните се разполагат една след друга. За масив от цели числа, съставен от m реда и n колони, адресът на всеки елемент се изчислява както следва:

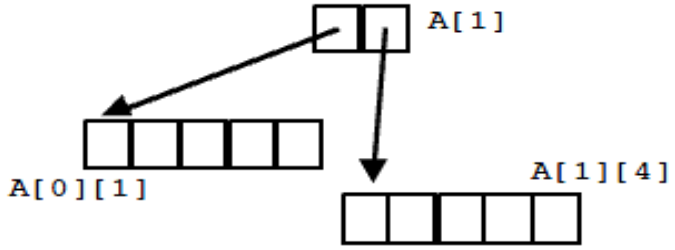
*&arr[i,j] = arr + 4 * (n * i + j)*

Или в по-общ вид за произволен масив:

array[L1..U1, L2..U2] (в Pascal)

*&arr[i, j] = arr + (sizeof T) * ((i - L1) * (U2 – L2 + 1) + (j - L2))*

Алтернативният метод е *масив от масиви*, където елементите на масива са указатели към всеки ред на масива. Например така изглежда масив A[2][5] (първият елемент е с индекс 0):



За прочитане на стойността на елемент A[i][j] е необходимо прочитане на два указателя за разлика от многомерния масив, който изисква прочитане само на един указател. Адресът на ред *i* се получава чрез следното изчисляване:

*A + i * sizeof(pointer).*

След прочитане на адреса към него се добавя

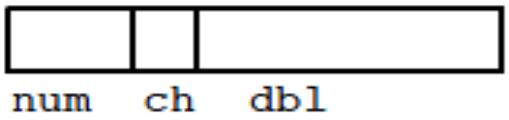
*j * sizeof(arrayElem)*

, за да се получи адрес на елемента, от който след това се получава стойността на елемента.

Масив от масиви изисква повече памет поради използване на допълнителни указатели, докато при многомерните масиви е необходимо пространство само за съхранение на елементите.

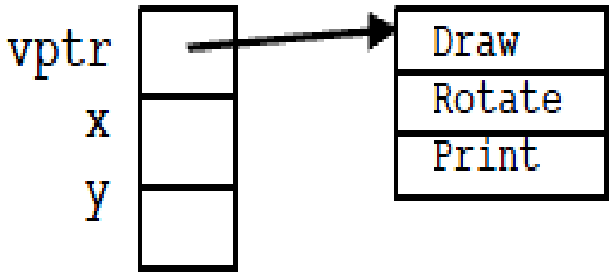
Структури: полетата на структурата се разполагат последователно в непрекъснат блок от малките към по-големите адреси в паметта. Размерът на един запис от структурата е равен най-малко на сумата от размерите на полетата. В дадения пример, ако целите числа заемат памет от 4 байта, символите 1 байт, а реалните числа с двойна точност (тип double) 8 байта, то цялата структура заема 13 байта. Отделните полета са разположени с отмествания съответно 0, 4 и 5 байта в адресното пространство. В много архитектури, обаче, за структурата ще бъдат необходими 16 байта, защото отместванията ще бъдат формирани така, че всяко поле да започва с изравнена граница.

```
struct example {
int num;
char ch;
double dbl;
};
```



Обекти: разполагат се в паметта подобно на структурите, като полетата са инстанции на променливите на класа. Методите не се съхраняват в самия обект. За динамично разпределяне (dispatch) във всеки обект се използва допълнителен скрит указател, който сочи към памет със споделена информация за методите на класа (често наричана *vtable*). Ако езикът не поддържа динамично разпределяне, ако в класа не са дефинирани методи, които го изискват или ако в класа има само private и final методи, тогава не е необходима таблица и скрит указател в представянето на обекта.

```
public class Rectangle {
private int x, y;
public void Draw() {}
public void Rotate(int deg) {}
public void Print() {}
}
```



Инструкции: кодират се като битови последователности, обикновено с размер дума. Различните битове в инструкцията кодират информация за типа на инструкцията (четене, запис, умножение и др.), за използваните регистри и др.

При много компилируеми езици не се записва никаква информация за типа на съхраняваните променливи. Така например при прочитане на съдържанието на адрес 1000 не е ясно дали то трябва да се интерпретира като тип данни, указател, инструкция или др. Езици като LISP и Smalltalk поддържат информация за типа данни на променливите по време на изпълнение. При тях всяка променлива се маркира с таг за типа на променливата.

Област на видимост и време на живот

Област на видимост на променлива (scope) определя части от програмата, в които променливата може да бъде достъпвана. *Времето на живот на променливата (lifetime)* определя периода от време, в който може да се използва променливата.

глобални: Времето на живот съвпада с времето на живот за цялата програма и областта на видимост също е в цялата програма.

статични: Времето на живот съвпада с времето на живот за цялата програма, но областта на видимост е само във функцията (или модула), в който е декларирана променливата.

локални: (наричани също автоматични) Съществуват само за времето на извикване на подпрограмата, в която са декларирани; нова променлива се създава при влизане в процедура и се унищожава при излизане от подпрограмата. Достъп до локални променливи може да се извършва само в подпрограмата, в която са декларирани.

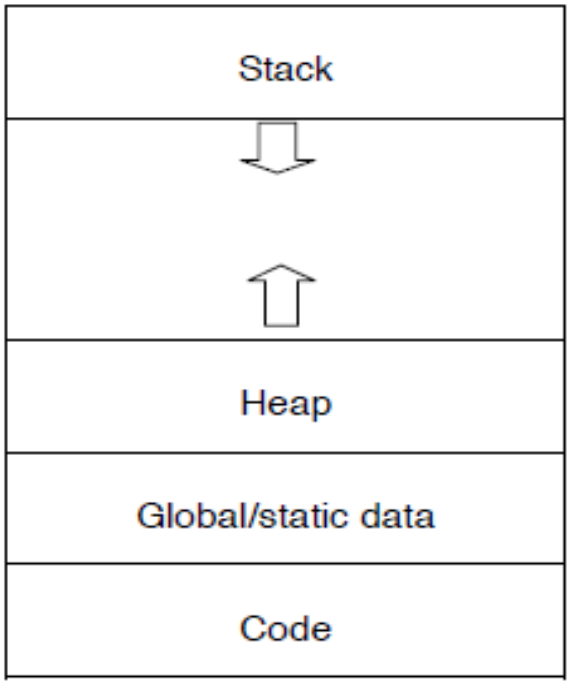
динамични: променливи, които се създават по време на изпълнение на програмата; обикновено се представят чрез адрес, съхраняван в променлива от един от изброените по-горе класове. Времето на живот съвпада с това на програмата, а областта на видимост съвпада с областта на променливата, съхраняваща адреса.

Глобални и *статични* променливи имат единствена инстанция, която съществува по време на живота на програмата; различават се само в областта на видимост. Обикновено тези два класа променливи се групират заедно в сегмент с глобални/статични данни в изпълнимата програма.

Локалните променливи се създават само при влизане в подпрограма и съществуват до нейното приключване. За тяхното обработване по време на изпълнение се използва стекова памет, в която се съхраняват стойностите на локалните променливи. Областта от паметта, използвана за съхраняване на всички локални променливи в подпрограма, се нар. *стеков фрагмент*. Стековият фрагмент за текущо изпълняваната подпрограма се намира във върха на стека. Адресът на текущия стеков фрагмент обикновено е записан в регистър на микропроцесора и се нар. базов указател.

Динамичното заемане на памет по време на изпълнение обикновено се реализира чрез извикване на библиотечни функции (напр. функция malloc). Това адресно пространство се заделя в сегмент от паметта, различен от програмния код, глобални/статични данни и стек. Тази памет се нар. *heap*.

Адресното пространство на изпълняваща се програма се разделя по следния начин:



Стеков фрагмент

При всяко извикване на подпрограма (функция или процедура) за нея се създава стеков фрагмент. В един стеков фрагмент (нар. activation record) се съдържа следната информация:

1) *Стеков указател* (*frame pointer)*: указател към предишния стеков фрагмент. Използва се за връщане към извикваща подпрограма при завършване на текущо изпълняваната подпрограма. Понякога се нарича още динамична връзка.
2) *Статична връзка* (*static link)*: съдържа указател към стековия фрагмент на подпрограмата, в която е декларирана текущата подпрограма.
3) *Адрес на връщане*: указател към точката на връщане в кода след приключване на текущата подпрограма.
4) *Стойности на параметрите*, с които е извикана подпрограмата и локални променливи.

При извикване на подпрограма (напр. на функция) се изпълняват следните основни стъпки:

Преди извикване на функция извикващата функция прави следното:

1) Съхранява стойностите на всички необходими регистри
2) Записва в стека стойностите на параметрите на извикваната функция
3) Задава статична връзка (ако такава съществува)
4) Записва в стека адреса на връщане
5) Прави преход към извикваната функция

По време на извикването на функция извикващата функция изпълнява следните действия:

1) Съхранява стойностите на необходимите регистри
2) Установява новия стеков указател
3) Заделя пространство за всички локални променливи
4) Изпълнява операторите в извиканата функция
6) Възстановява стойностите на всички съхранени регистри
7) Прави преход към съхранения адрес на връщане

След извикване на функция извикващата функция изпълнява следните действия:

1) Изтрива от стека адреса на връщане и параметрите
2) Възстановява стойностите на всички съхранени регистри
3) Продължава изпълнението си

Предаване на параметри

Общите методи за предаване на параметри в програмните езици са:

Предаване по стойност: В извиканата подпрограма се копират стойностите на параметрите. По този начин подпрограмата работи с копия на параметрите, направените промени не се отразяват в извикващата подпрограма.

Предаване по референция (адрес): Вместо копие на параметър извикваната подпрограма получава референция към параметъра. Обикновено референцията се реализира чрез указател към параметъра. В следствие извиканата подпрограма може да променя стойностите на параметрите. Освен това този механизъм позволява ефективно предаване на множества от стойности (напр. структури). Каква е разликата при явно предаване на указател в C?

В някои масови програмни езици съществуват и други средства за предаване на параметри:

Само за четене (read-only) property, напр. за константи в C++, позволява по-ефективно предаване чрез референция.

Параметри по подразбиране (default parameters), което позволява извикваната подпрограма да определя какви параметри да използва, когато при извикването не са зададени параметри или са зададени по-малко на брой параметри.

Позиционно зависими параметри (position-independent parameters), напр. в Ada и LISP, позволява параметрите да бъдат свързвани с имена вместо да бъдат изброявани в строго определен ред.

Управление на паметта

Обикновено програмата използва памет от операционната система, която разделя паметта на страници. Страницата е област от паметта с фиксиран размер, напр. 4-8 KB или повече. Програмата разделя тази памет по собствен начин.

```
a = malloc(12); // Резервиране на 12 байта
a = b;          // Загуба
```

Garbage collection

Някои езици като ML и Java използват т.нар. *събиране на боклука (garbage collection)*, при което програмистът е освободен от задължението да освобождава динамична памет. Изпълнителната среда открива, че дадена област от паметта не може да бъде адресирана от програмата, при което областта се освобождава автоматично. Съществуват няколко метода за реализация на garbage collection. Двата най-често използвани методи са *reference counting* и *mark and sweep*.

Автоматичното управление на паметта е много удобно, но понякога причинява проблеми.