Распределение памяти

Распределение памяти - это процесс, в результате которого отдельным элементам исходной программы ставятся в соответствие адрес, размер и атрибуты области памяти, необходимой для размещения лексических единиц.

Область памяти - это блок ячеек памяти, выделяемых для данных и каким-то образом объединенных логически.

Распределение памяти выполняется после фазы анализа текста исходной программы **на этапе подготовки к генерации объектного модуля** (перед генерацией кода объектного модуля).

Исходными данными для процесса распределения памяти служат сведения о семантике конструкций ЯП, таблица идентификаторов, построенная лексическим анализатором и информация, полученная синтаксическим анализатором при анализе декларативной чисти программы.

Современные компиляторы, в основном, работают с относительными, а не с абсолютными адресами ячеек памяти.

Распределение памяти

Семантика программ подразумевает, что при их выполнении области памяти будут необходимы для хранения:

- кодов пользовательских программ;
- данных, необходимых для работы этих программ;
- кодов системных программ, обеспечивающих поддержку пользовательских программ в период их выполнения;
- записей о текущем состоянии процесса выполнения программ (например, записей об активации процедур).

По **способу использования** области памяти делятся на ***глобальные*** и ***локальные***, а по **способу распределения** – на ***статические*** и ***динамические***.

Классы памяти

Выделяемую память можно разделить на локальную / глобальную и статическую / динамическую.

Классы памяти

Локальная память - это область памяти, которая выделяется в начале выполнения некоторого фрагмента результирующей программы (блока, функции, оператора…) и может быть освобождена по завершении выполнения данного фрагмента. Доступ к локальной области памяти всегда запрещен за пределами того фрагмента программы, в котором она выделяется.

Глобальная память - это область памяти, которая выделяется один раз при инициализации результирующей программы и действует все время выполнения программы. Как правило, глобальная область памяти доступна из любой части исходной программы.

Статическая память - это область памяти, размер которой известен на этапе компиляции. Для статической памяти компилятор порождает некоторый адрес (как правило, относительный), и дальнейшая работа с ней происходит относительно этого адреса.

Динамическая память - это область памяти, размер которой становится известным только на этапе выполнения результирующей программы. Для динамической памяти компилятор порождает фрагмент кода, который отвечает за распределение памяти (ее выделение и освобождение). Как правило, с динамическими областям памяти связаны многие операции с указателями и с экземплярами объектов (классов) в ООЯП.

Динамические области памяти можно разделить на динамические области памяти, **выделяемые пользователем** и **непосредственно компилятором**.

Общие принципы генерации объектного кода

При генерации объектного кода компилятор переводит текст программы во внутреннем представлении в текст программы на выходном языке (как правило, машинном).

Генерация объектного кода происходит на основе:

- определенной на фазе анализа компиляции синтаксической структуры программы,
- информации, хранящейся в таблице идентификаторов,
- результата распределения памяти.

Характер и сложность отображения промежуточного представления программы в последовательность команд на машинном языке зависит от языка внутреннего представления и архитектуры вычислительной системы, на которую ориентирована результирующая программа.

Часто для построения кода результирующей программы компиляторы используют синтаксически управляемый перевод.

Оптимизация программ

Оптимизация программы - это изменение компилируемой программы (в основном переупорядочивание и замена операций) с целью получения более эффективной объектной программы.

Используются два критерия эффективности результирующей программы :

- **скорость** выполнения программы и
- **объем памяти**, необходимый для выполнения программы.

В общем случае задача построения оптимального кода программы алгоритмически неразрешима. К тому же, компилятор обладает весьма ограниченными средствами анализа семантики входной программы в целом.

Основная оптимизация программы должна производится программистом.

Принципиально различаются два основных вида оптимизирующих преобразований:

- **машинно-независимые преобразования** исходной программы,
- **машинно-зависимые преобразования** результирующей объектной программы.

Оптимизация может привести к изменению смысла программы. Например, в случае исключения из программы вызова функции с "побочным эффектом".

У современных компиляторов существует возможность выбора критерия оптимизации и отдельных методов оптимизации.

Машинно-независимые оптимизирующие преобразования

Машинно-независимые преобразования исходной программы производятся в основном над ее внутренним представлением и основаны на известных математических и логических преобразованиях.

1. Удаление недостижимого кода.

(задача компилятора найти и убрать его).

Пример:

```
if (1)
    S1;          ⇒      S1;
else
    S2;
```

Машинно-независимые оптимизирующие преобразования

2. Оптимизация линейных участков программы.

В современных системах программирования профилировщик на основе результатов запуска программы выдаёт информацию о том, на какие её линейные участки приходится основное время выполнения.

а) Удаление бесполезных присваиваний.

$$a = b * c;\ d = b + c;\ a = d * c;\ \Rightarrow\ d = b + c;\ a = d * c;$$

Однако, в следующем примере эта операция уже не бесполезна:

$$p = \&\ a;\ a = b * c;\ d = *\ p + c;\ a = d * c;$$

б) Исключение избыточных вычислений.

$$d = d + b * c;\ a = d + b * c;\ c = d + b * c; \Rightarrow$$
$$t = b * c;\ d = d + t;\ a = d + t;\ c = a;$$

в) Свертка объектного кода (выполнение во время компиляции тех операций исходной программы, для которых значения операндов уже известны).

$$i = 2 + 1;\ j = 6 * i + i; \Rightarrow i = 3;\ j = 21;$$

Машинно-независимые оптимизирующие преобразования

г) Перестановка операций (для дальнейшей свертки или оптимизации вычислений).

a = 2 * b * 3 * c; $\Rightarrow$ ***a = (2 * 3) * (b * c);***

a = (b + c) + (d + e); $\Rightarrow$ ***a = (b + (c + (d + e)));***

д) Арифметические преобразования (на основе алгебраических и логических тождеств).

a = b * c + b * d; $\Rightarrow$ ***a = b * (c + d);***

a * 1 $\Rightarrow$ ***a,*** ***a * 0*** $\Rightarrow$ ***0,*** ***a + 0*** $\Rightarrow$ ***a .***

е) Оптимизация вычисления логических выражений.

a* || *b* || *c* || *d; $\Rightarrow$ ***a,*** если **a** есть **true**.

Но! ***a* || *f(b)* || *g(c)*** не всегда **a** (при **a = true**), может быть побочный эффект.

Машинно-независимые оптимизирующие преобразования

3. Подстановка кода функции вместо ее вызова в объектный код.

Этот метод, как правило, применим к простым функциям и процедурам, вызываемым непосредственно по адресу, без применения косвенной адресации через таблицы RTTI (Run Time Type Information).

Некоторые компиляторы допускают применять метод только к функциям, содержащим последовательные вычисления без циклов.

Язык C++ позволяет явно указать (inline), для каких функций желательно использовать inline-подстановку.

Машинно-независимые оптимизирующие преобразования

4. Оптимизация циклов.

а) Вынесение инвариантных вычислений из циклов.

```
for (i = 1; i <= 10; i++)
        a [i] = b * c * a [i];                    ⇒
d = b * c;  for (i = 1; i <= 10; i++)
                        a [i] = d * a [i];
```

б) Замена операций с индуктивными (образующими арифметическую прогрессию) переменными (как правило, умножения на сложение).

```
for (i = 1; i <= N; i++)
        a [i] = i * 10;                           ⇒
t = 10; i = 1; while (i <= N) {
                    a [i] = t;    t = t + 10;    i++;
               }

s = 10; for (i = 1; i <= N; i++) {
         r = r + f (s); s = s + 10; }             ⇒
s = 10; m = N * 10; while (s <= m) {
                        r = r + f (s); s = s + 10; }
```

(избавились от одной индуктивной переменной).

Машинно-независимые оптимизирующие преобразования

в) **Слияние циклов**.

```
for (i =1; i <= N; i++)
      for (j = 1; j <= M; j++)
            a [i] [j] = 0;        ⇒

k = N * M;
for (i = 1; i <= k; i++)
            a [i] = 0;
```
(только в объектном коде!)

г) **Развертывание циклов** (можно выполнить для циклов, кратность выполнения которых известна на этапе компиляции).

```
for (i = 1; i <= 3; i++)
            a [i] = i;        ⇒

a [1] = 1;
a [2] = 2;
a [3] = 3;
```

Машинно-зависимые оптимизирующие преобразования

Машинно-зависимые преобразования результирующей объектной программы зависят от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. При этом может учитываться объем кэш-памяти, методы организации работы процессора ….

Эти преобразования, как правило, являются "ноу-хау", и именно они позволяют существенно повысить эффективность результирующего кода.

.

1. Распределение регистров процессора.

Использование регистров общего назначения и специальных регистров (аккумулятор, счетчик цикла, базовый указатель) для хранения значения операндов и результатов вычислений позволяет увеличить быстродействие программы.

Доступных регистров всегда ограниченное количество, поэтому перед компилятором встает вопрос их оптимального распределения и использования при выполнении вычислений.

Машинно-зависимые оптимизирующие преобразования

2. Оптимизация передачи параметров в процедуры и функции.

Обычно параметры процедур и функций передаются через стек. При этом всякий раз при вызове процедуры или функции компилятор создает объектный код для размещения ее фактических параметров в стеке, а при выходе из нее - код для освобождения соответствующей памяти.

Можно уменьшить код и время выполнения результирующей программы за счет оптимизации передачи параметров в процедуру или функцию, передавая их **через регистры** процессора.

Реализация данного оптимизирующего преобразования зависит от количества доступных регистров процессора в целевой вычислительной системе и от используемого компилятором алгоритма распределения регистров.

Недостатки метода:

- оптимизированные таким образом процедуры и функции не могут быть использованы в качестве **библиотечных**, т.к. методы передачи параметров через регистры не стандартизованы и зависят от реализации компилятора.
- этот метод не может быть использован, если где-либо в функции требуется выполнить **операции с адресами** параметров.

Языки Си и C++ позволяют явно указать (**register**), какие параметры и локальные переменные желательно разместить в регистрах.

Машинно-зависимые оптимизирующие преобразования

3. Оптимизация кода для процессоров, допускающих распараллеливание вычислений.

При возможности параллельного выполнения нескольких операций компилятор должен порождать объектный код таким образом, чтобы в нем было максимально возможное количество соседних операций, все операнды которых не зависят друг от друга.

Для этого надо найти оптимальный порядок выполнения операций для каждого оператора (переставить их).

$a + b + c + d + e + f; \Rightarrow$

для одного потока обработки данных: $((((a + b) + c) + d) + e) + f;$

для двух потоков обработки данных: $((a + b) + c) + ((d + e) + f);$

для трех потоков обработки данных: $(a + b) + (c + d) + (e + f);$