# Способы управления памятью

# Основные проблемы связанные с выделением памяти

1. Поиск свободного участка памяти:
   задача поиска свободного участка памяти нужного размера требует просмотра всех свободных блоков памяти

# Основные проблемы связанные с выделением памяти

1. Поиск свободного участка памяти:
   задача поиска свободного участка памяти нужного размера требует просмотра всех свободных блоков памяти
2. Отслеживание перекрытий блоков памяти:
   при ошибочном использовании указателей возможна ситуация, когда одна и та же область памяти одновременно используется для хранения разных структур данных

# Основные проблемы связанные с выделением памяти

1. Поиск свободного участка памяти:
   задача поиска свободного участка памяти нужного размера требует просмотра всех свободных блоков памяти
2. Отслеживание перекрытий блоков памяти:
   при ошибочном использовании указателей возможна ситуация, когда одна и та же область памяти одновременно используется для хранения разных структур данных
3. Отслеживание свободных блоков памяти:
   при ошибочной организации кода освобождения ресурсов возможна ситуация, когда программа утратила указатель на неосвобожденную область памяти, сделав эту область памяти недоступной для себя и для других программ (утечка памяти)

# Основные проблемы связанные с выделением памяти

1. Поиск свободного участка памяти:
   задача поиска свободного участка памяти нужного размера требует просмотра всех свободных блоков памяти
2. Отслеживание перекрытий блоков памяти:
   при ошибочном использовании указателей возможна ситуация, когда одна и та же область памяти одновременно используется для хранения разных структур данных
3. Отслеживание свободных блоков памяти:
   при ошибочной организации кода освобождения ресурсов возможна ситуация, когда программа утратила указатель на неосвобожденную область памяти, сделав эту область памяти недоступной для себя и для других программ (утечка памяти)
4. Фрагментация памяти:
   при интенсивном выделении и освобождении памяти возникает ситуация фрагментации памяти, когда большой объем памяти оказывается распределенным на большое число маленьких блоков, делая невозможным выделение непрерывного блока памяти размера, формально укладывающегося в размер доступной памяти

# Основные способы управления памятью

1. Статическое распределение памяти
2. Размещение в стековой памяти
3. Динамическое распределение памяти в куче (heap)
4. Управляемое динамическое распределение памяти

# Статическое распределение памяти

Статическое распределение памяти выполняется на этапе компиляции программы. Таким способом память выделяется для глобальных переменных в языках C/C++, Pascal и др.

**Достоинства:**
+ поскольку память распределяется разово на этапе компиляции, нет необходимости перераспределять память в процессе работы программы
+ нет возможности для возникновения перекрытий блоков памяти
+ нет возможности для возникновения потерянных блоков, поскольку при завершении программы вся ее память будет возвращена (связным блоком!) операционной системе
+ подход со статическим распределением памяти позволяет создавать очень быстрые программы (драйверы устройств, системы реального времени), фрагментация памяти исключена

**Недостатки:**
- невозможность гибкого управления выделяемой памятью
- требования к памяти должны быть известны при создании программы

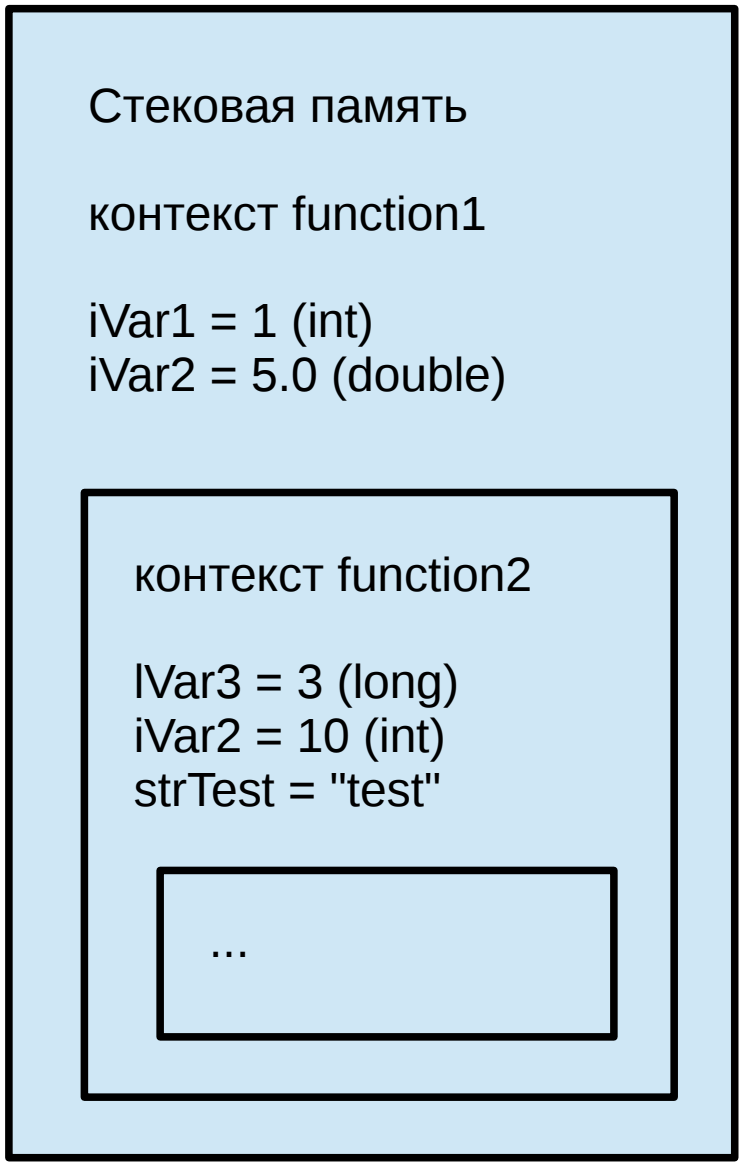
# Размещение в стековой памяти

Размещение в стековой памяти применяется для хранения локальных переменных в подпрограммах.

Это связано с тем, что вызов подпрограммы практически всегда связан с выделением блока в стековой памяти для хранения адреса возврата из подпрограмма.

Удобно в этом же блоке выделить память и для хранения локальных переменных (их набор известен на этапе компиляции, поэтому размер стекового блока известен заранее).

# Размещение в стековой памяти



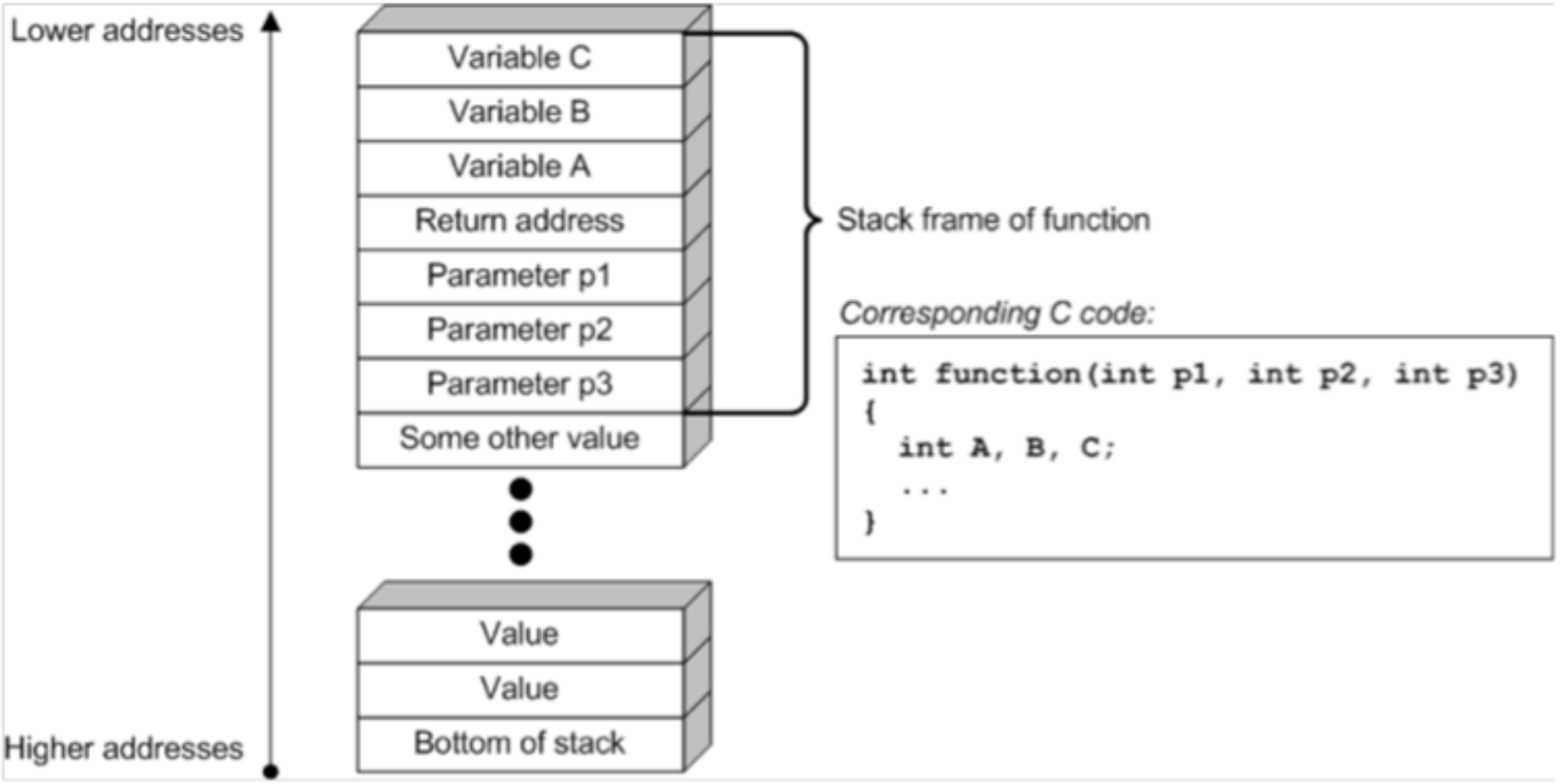

```
void function1(int iVar1)
{
    double iVar2 = 5;
    function2(3);
}


void function2(long lVar3)
{
    int iVar2 = 10;
    string strTest = "test";
}


void main()
{
     function1(1);
}
```

# Размещение в стековой памяти

**Достоинства:**
+ память выделяется быстро, поскольку ее распределение фактически производится на этапе компиляции, в процессе исполнения достаточно лишь зарезервировать блок нужного размера в стековой памяти
+ нет возможности для возникновения потерянных блоков, поскольку при завершении подпрограммы выделенный ей блок автоматически будет возвращен в стековую память
+ никогда не возникает фрагментация памяти

**Недостатки:**
- обычно не следует забирать из стековой памяти большие блоки памяти, поскольку это уменьшает доступную глубину вложенности вызовов в рекурсивных алгоритмах
- некорректное использование указателей на блоки в стековой памяти за пределами подпрограммы, в которой эти блоки были выделены может приводить к перекрытию блоков памяти

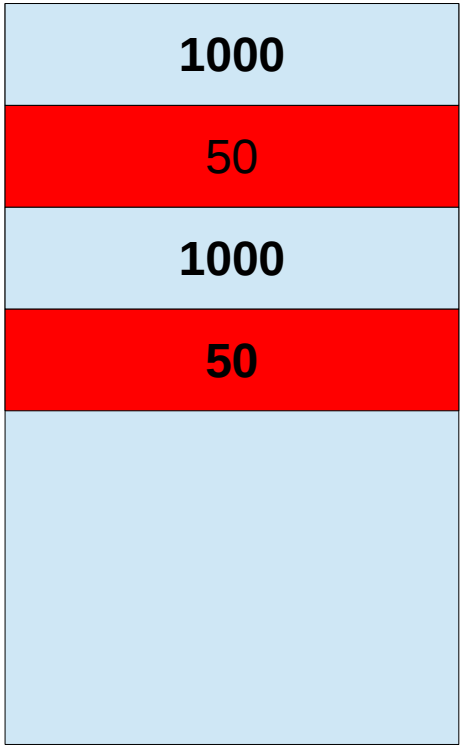
# Размещение в стековой памяти



# Динамическое распределение памяти в куче (heap)

Данный способ управления памятью предполагает выделение блоков памяти произвольного размера из доступного объема памяти

```
// запрашиваем 4 блока памяти
char * var1 = new char[1000];
char * var2 = new char[50];
char * var3 = new char[1000];
char * var4 = new char[50];

// освобождаем 2 блока
delete [] var1;
delete [] var3;

// сейчас гарантированно доступно
// 2000 байтов памяти
// но следующая операция может оказаться
// неуспешной вследствие фрагментации
// памяти
char * var5 = new char[2000];
```

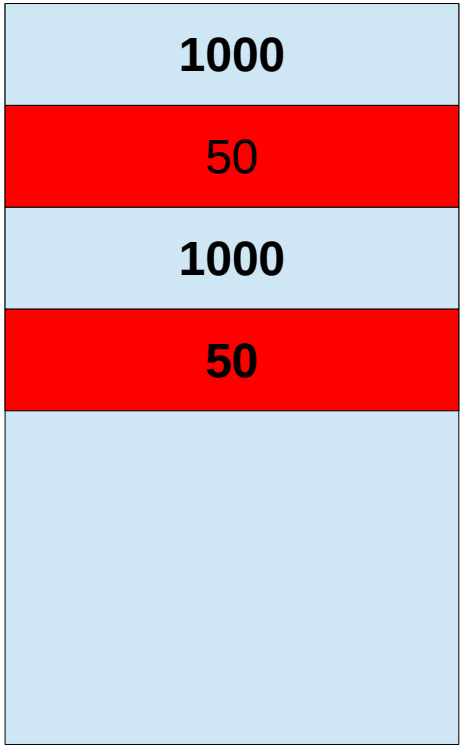



Перераспределить память, чтобы объединить свободные блоки невозможно! Это повлекло бы разрушение указателей (var2, var4).

# Динамическое распределение памяти в куче (heap)

Для эффективной борьбы с фрагментацией алгоритм выделения блока памяти должен учитывать информацию обо всех свободных блоках памяти.

```
// запрашиваем 4 блока памяти
char * var1 = new char[1000];
char * var2 = new char[50];
char * var3 = new char[1000];
char * var4 = new char[50];

// освобождаем 2 блока
delete [] var1;
delete [] var3;

// если память выделять "жадно", то
// вторая операция на выделение 2000 байт
// окажется неуспешной, поскольку в первой
// операции свободный блок в 2000 байт
// был бы порезан на два блока
// по 1000 байт
char * var5 = new char[1000];
char * var6 = new char[2000];
```



# Динамическое распределение памяти в куче (heap)

**Достоинства:**
+ память можно выделять произвольными блоками в произвольный момент времени

**Недостатки:**
- корректное использование указателей (отсутствие перекрытий блоков памяти, отсутствие утечек памяти) перекладывается на программиста
- неизбежна проблема фрагментации памяти
- операция выделения блока памяти является сложной, поскольку требуется анализ всех свободных блоков памяти и выбор оптимального с точки зрения борьбы с фрагментацией памяти

# Динамическое распределение памяти в куче (heap)

Частично проблемы динамического распределения памяти решаются при помощи введения виртуального адресного пространства, разбиения памяти на страницы и отображения страниц виртуального адресного пространства на страницы физической или виртуальной памяти.

Отображением страниц виртуального адресного пространства на страницы физической памяти занимается операционная система. Тем самым решается проблема фрагментации физической памяти (страницы физической памяти можно перемещать в физической памяти, поскольку указатели в виртуальном адресном пространстве при этом не изменяются), но не решается проблема фрагментации в виртуальном адресном пространстве.

С другой стороны, отображение страниц повышает стоимость операции доступа к памяти.

# Управляемое динамическое распределение памяти

Наблюдение: если технологию перемещаемых страниц памяти перенести внутрь программы, то удалось бы избавиться от фрагментации, сохранив возможность выделения произвольных объемов памяти.

Наблюдение: если исключить использование в программе прямой адресации памяти, то удалось бы исключить наложение блоков памяти и частично решить проблему утечки памяти.

В платформах Java и .Net прямой доступ к памяти исключен. Для работы с объектами используются ссылки, которые ссылаются не на физический адрес в памяти, а на некоторые внутренние структуры JVM, позволяющие JVM восстановить положение объекта в физической памяти.

Также действует следующее соглашение: до тех пор, пока блок памяти может быть использован в программе (то есть пока он доступен по цепочке ссылок) он обязан оставаться в памяти.

# Управляемое динамическое распределение памяти

Такой подход позволяет устранить все недостатки присущие динамическому распределению памяти:

+ перекрытие блоков памяти исключается за счет того, что до тех пор, пока какая-либо ссылка ссылается на некоторый блок памяти, этот блок памяти не будет считаться свободным, и, следовательно, не будет повторно выделен под другой объект.

+ утечки памяти невозможны по причине того, что как только какой-то блок памяти становится недоступным из программы, среда выполнения программы получает возможность объявить его свободным.

+ с фрагментацией памяти можно эффективно бороться путем перемещения объектов в физической памяти (структура ссылок при этом сохраняется!)

# Управляемое динамическое распределение памяти

- Однако платой за эти преимущества является небольшое снижение быстродействия программы, поскольку теперь работа по освобождению памяти перенесена в исполняющую среду, и виртуальная машина Java или среда исполнения .Net вынуждены в отдельном потоке заниматься поиском освобождающейся памяти и делать ее доступной для использования. Освобождение памяти - это одна из основных задач, которую нужно уметь решать исполняющей среде программ для платформ Java и .Net.

Подсистема, отвечающая за освобождение памяти называется Garbage collector (**сборщик мусора**).

# Управляемое динамическое распределение памяти

Из устройства подсистемы сборки мусора Java следуют следующие важные факты:

1. Обычно операция создания объекта является очень дешевой, поскольку фактически сводится к сдвигу указателя (от связного блока памяти, заполненного нулями, отрезается кусок нужного размера).

2. Современные сборщики мусора достаточно эффективны, чтобы обеспечивать высокое быстродействие даже при интенсивной работе с памятью.

3. Поскольку сборщик мусора работает в отдельном потоке, это может сказываться на быстродействии программы. В частности, могут возникать паузы в работе программы в моменты, когда сборщик мусора занимается перемещением объектов в памяти.

4. Точный момент, когда объект будет удален из памяти, не может быть предсказан (он зависит от логики работы сборщика мусора). Поэтому нельзя полагаться на то, что финализатор (метод finalize()) будет вызван сразу же, как только объект перестал использоваться.