Операторы управления ходом выполнения программы в C++

Магнитогорск, 2020

Операторы

- Выражение
- Условный
- Выбора
- Циклы
- Передачи управления

В конце оператора всегда ставится ;

Инструкции и блоки

- Выражение (например, x = 0) становится инструкцией, если в конце поставить точку с запятой
  - x = 0;
  - printf(“Hello”);
  - В Си точка с запятой является заключающим символом инструкции, а не разделителем.
- Фигурные скобки **{** и **}** используются для объединения объявлений и инструкций в ***составную инструкцию***, или ***блок***
  - с т.з. синтаксиса языка блок воспринимается как одна инструкция

Блоки и область видимости

- Переменные видимы внутри того блока, где она объявлена
- При покидании своего блока видимости переменная уничтожается, а занимаемая ею область памяти – освобождается
  - (автоматическое управление памятью)

```
int main(int argc, char * argv)
{
    int a = 0;
    if (argc > 1)
    {
        int b = argc - 1;
    }
    return 0;
}
```

Условный оператор

if (выражение) опер.1; [else опер.2;]

Пример. Найти максимум и минимум из двух чисел.

```
int main()
{ int a,b,min, max; cin >> a >> b;
 if (a<b) { min = a; max = b;}
   else {min = b; max = a; }
 cout << min << max;
 return 0;
}
```

Конструкция else-if

- Позволяет осуществлять многоступенчатое решение
  - ```
    if (выражение)
        инструкция
    else if (выражение)
        инструкция
    else if (выражение)
        инструкция
    else if (выражение)
        инструкция
    else
        инструкция
    ```
    ```

Пример, бинарный поиск

```
/* binsearch: найти x в v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, const int v[], int n)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else /* совпадение найдено */
            return mid;
    }
    return -1; /* совпадения нет */
}
```

Условный оператор. Типичные ошибки

- Отсутствие фигурных скобок

```
if (a<b)  min = a; max = b;
```

- Использование = вместо ==

```
if (a=5) cout << a;
```

- Проверка диапазона

```
if (-1 <= x <=1) cout << “есть arcsin”;
Правильно так:
if (-1 <= x && x <=1)
        cout << “есть arcsin”;
```

Оператор выбора

```
switch (выражение) {
  case конст.1: список операторов 1
  case конст.2: список операторов 2
  …
  default: операторы
}
```

Используется для выбора одного из нескольких путей

- Осуществляется проверка на совпадение значения выражения с одной из некоторого набора целых констант, и выполняет соответствующую ветвь программы

- Инструкция break выполняет выход из блока switch

Пример на оператор выбора

По номеру месяца определить время года

```
…
switch (m) {
case 1: case 2: case 12:
      cout<<“Зима”; break;
case 3: case 4: case 5:
      cout<<“Весна”; break;
case 6: case 7: case 8:
      cout<<“Лето”; break;
default: cout<<“Осень”;
}
```

С типом enum

```cpp
#include <string>
#include <iostream>
#include <cassert>

enum class WeekDay
{
  Sunday, Monday, Tuesday, Wednesday,
  Thursday, Friday, Saturday
};

std::string WeekDayToString(const WeekDay & weekDay)
{
  switch (weekDay)
  {
  case WeekDay::Sunday:    return "Sunday";
  case WeekDay::Monday:    return "Monday";
  case WeekDay::Tuesday:   return "Tuesday";
  case WeekDay::Wednesday: return "Wednesday";
  case WeekDay::Thursday:  return "Thursday";
  case WeekDay::Friday:    return "Friday";
  case WeekDay::Saturday:  return "Saturday";
  default:
    assert(!"This is not possible");
    return "";
  }
}

void main()
{
  std::cout << WeekDayToString(WeekDay::Sunday) << std::endl;
}
```

Что такое циклическое выполнение

- **Цикл** – последовательность из нескольких операторов, указываемая в программе один раз, которая выполняется несколько раз подряд
  - Допускается существование **бесконечного цикла**
- **Тело цикла** - последовательность операторов, предназначенная для многократного выполнения в цикле

Виды циклов

- Циклическое выполнение в языке Си осуществляется при использовании следующих операторов цикла:
  - while
  - for
  - do..while
- Внутри циклов могут использоваться операторы управления работой цикла:
  - **break** для досрочного выхода из цикла
  - **continue** для пропуска текущей итерации

Цикл с предусловием

while (*выражение*)
инструкция

Цикл выполняется так:

1. Вычисляется выражение
2. Если оно истинно (не 0) выполняется оператор
3. Снова вычисляется выражение
4. Если оно ложно – выход из цикла.

Пример: вычисление факториала n

```
f = k = 1;
while (k<=n) f *= k++;
```

Цикл с постусловием

do оператор while (выражение)

Цикл выполняется так:

1. Выполняется оператор
2. Вычисляется выражение
3. Если оно истинно (не 0) снова выполняется оператор
4. Если оно ложно – выход из цикла.

Пример: вычисление факториала n

```
f = k = 1;
do f *= k++; while (k<=n);
```

Цикл с параметром

for (инициализация; условие выполнения; модификация) оператор;

Инициализация выполняется перед началом цикла

Выполнение ***оператора*** (тело цикла) продолжается до тех пор, пока ***условие выполнения*** имеет ненулевое значение

Модификация - в конце каждой итерации

Пример: вычисление факториала

```
for (int k = f = 1; k<=n; k++) f *= k;
```

либо `for (int k = f = 1; k<=n; f *= k++);`

Бесконечный цикл for (; ;) { ….}

Range-based for

- Версия цикла for, предназначенная для перебора элементов некоторого диапазона
  - Массивы, строки, контейнеры стандартной библиотеки, пользовательские типы данных
- Синтаксис:
  - for (*тип идентификатор* : *диапазон*)
    инструкция

```
// Обход элементов массива
{
  int numbers[] = { 10, 15, 17, 33,
18 };
  int sum = 0;
  int product = 1;
  cout << "Array items: ";
  for (int number : numbers)
  {
    cout << number << ", ";
    sum += number;
    product *= number;
  }
  cout << endl << "\tSum: " << sum
<< endl << "\tProduct: " << product
<< endl;
}
```

Вложенные циклы

- Один цикл может быть вложен в другой
  - При этом выполнение внутреннего цикла выполняется как часть оператора внешнего цикла

Операторы передачи управления

- `goto метка;` переход на заданную метку внутри текущей функции - использование инструкции goto усложняет структуру программы и без крайней необходимости ею пользоваться не стоит

  `break;` немедленный выход из цикла или оператора выбора

- `continue;` пропуск оставшихся операторов тела цикла и переход к следующей итерации цикла
  - В циклах while и do-while осуществляется переход к проверке условия
  - В цикле for осуществляется переход к приращению переменной цикла

- `return [выражение];` выход из функции с возвратом значения

Указатели

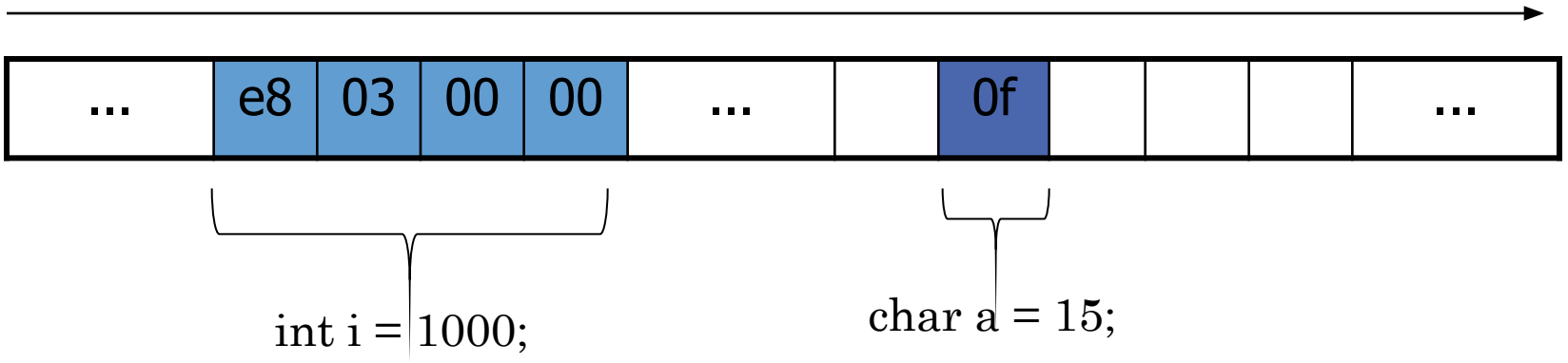
Хранение данных

- В C++ есть три разных способа выделения памяти для объектов
  - ***Статическое***: пространство для объектов создаётся в области хранения данных программы в момент компиляции;
  - ***Автоматическое***: объекты можно временно хранить в стеке; эта память затем автоматически освобождается и может быть использована снова, после того, как программа выходит из блока, использующего её.
  - ***Динамическое***: блоки памяти нужного размера могут запрашиваться во время выполнения программы с помощью оператора new в области памяти, называемой кучей. Эти блоки освобождаются и могут быть использованы снова после вызова для них оператора delete.

Организация памяти в языке C++

- С точки зрения языка C++ память представляет собой массив последовательно пронумерованных ячеек памяти, с которыми можно работать по отдельности или связными кусками
  - Порядковый номер ячейки называется ее **адресом**
  - Эта память используется для хранения значений переменных.
  - Переменные различных типов могут занимать различное количество ячеек памяти, и иметь различные способы представления в памяти

Пример



Что такое указатель?

- **Указатель** – это переменная, которая может хранить адрес другой переменной в памяти заданного типа
  - Указатели – мощное средство языка C++, позволяющее эффективно решать различные задачи
  - Использование указателей открывает доступ к памяти машины, поэтому пользоваться ими следует аккуратно

Объявление указателя

- Указатель на переменную определенного типа объявляется следующим образом:
  <тип> *<идентификатор>;
  - Например:
    int *pointerToInt;
  - Указатель, способный хранить адрес переменной любого типа имеет тип **void***:
    - void * pointerToAnyType;

Константные указатели

- Как и к обычным переменным, к указателям можно применять модификатор **const**:
  - **const int *** pointerToConstInt;
  - **char * const** constPointerToChar = &ch;
  - **const double * const** constPointerToConstDouble = &x;
  - **float * const** constPointerToFloat = &y;
  - **const void *** pointerToConstData;

Модификатор const относится либо к указателю, либо к значению

```
int i;

const int c = 1;

const int *pc = &c; //указатель на константу

int* const pc = &i; //указатель-константа
```

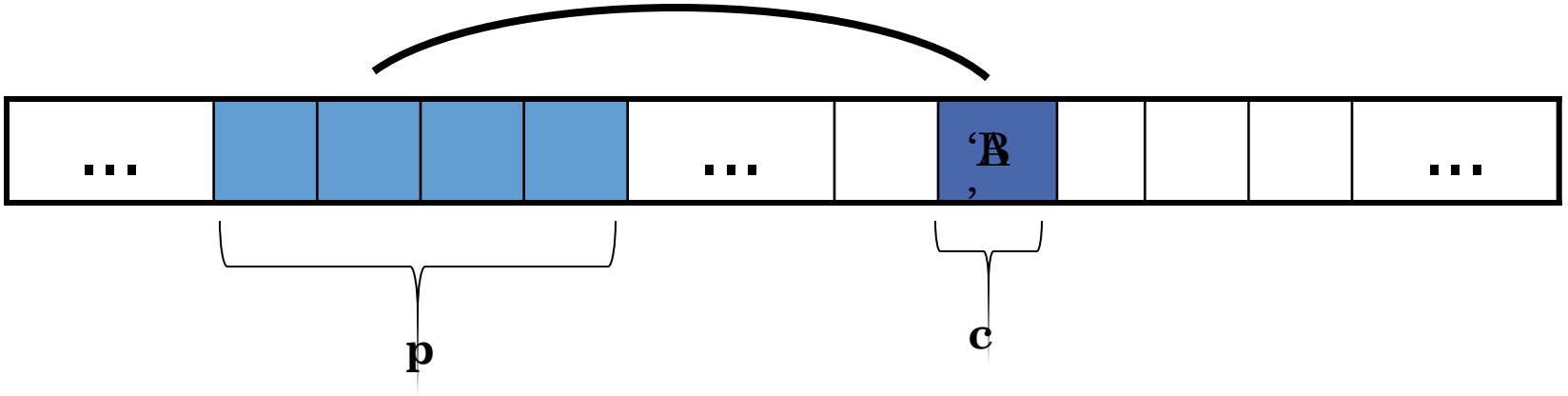
Получение адреса переменной

- Для взятия адреса переменной в памяти служит унарный оператор **&**
  - Этот оператор возвращает адрес переменной, который может быть присвоен указателю совместимого типа
  - Оператор взятия адреса применим только к переменным. Его нельзя применять к числовым константам, литералам, выражениям или регистровым переменным

Оператор косвенного доступа

- Для доступа к значению, на которое ссылается указатель, необходимо его **разыменование** (dereferencing), осуществляемое при помощи **унарного** оператора *****
  - ```
    int * p = &i;
    *p = 5;
    ```
    ```

Пример




```
char c = 'A';
char *p = &c;
*p = 'B';
```

Инициализация указателей

- с помощью операции & (адрес)

```
int a=5;
 int * p = &a;
```

- значением другого указателя

```
int * r = p;
```

- явным адресом памяти

```
char *vp = (char *)0xB8000000;
```

- пустым значением (нулем)

```
int * r = 0;
```

Инициализация указателей(2)

- Значение **неинициализированного** указателя не определено
  - **Разыменование такого указателя приводит к неопределенному поведению**
  - Лучше присвоить указателю нулевое значение (или символическую константу **NULL**), чтобы подчеркнуть, что он не ссылается ни на какую переменную:
    - char * p1 = 0;
      char * p2 = NULL;
  - Разыменование нулевого указателя также приводит к неопределенному поведению, однако появляется возможность проверки значения указателя:
    - if (p != nullptr) // или просто if (p)

```
#include <iostream>

void Print(void * p)
{
    std::cout << "Printing a pointer: " << p << "\n";
}
void Print(int i)
{
    std::cout << "Printing an integer: " << i << "\n";
}
void Print(bool b)
{
    std::cout << "Printing a boolean: " << (b ? "true" : "false") <<
"\n";
}

int main(int argc, char* argv[])
{
    Print(NULL);                                                //
Printing an integer: 0
    Print(nullptr);                                            // Printing a
pointer: 00000000
    bool nullptrAsBool = nullptr;
```

Копирование указателей

- Как и в случае обычных переменных, значение одного указателя может быть присвоено другому при помощи оператора **=**
  - Следует помнить, что в этому случае **копируется адрес переменной, а не ее значение**
  - **Для копирования значения** переменной, на которую ссылается указатель, необходимо применить **оператор разыменования ***
    - ```
      char a = 'A';
      char b = 'B';
      char c = 'C';
      char *pa = &a;
      char *pb = &b;
      char *pc = &c;
      pa = pb; // pa и pb теперь хранят адрес b
      *pa = *pc; // b теперь хранит значение 'C'
      ```
      ```

Динамические переменные

Создание

```
int *n = new int;

int *m = new int (10); // *m=10

int *r = new int [10]; // массив
```

Удаление

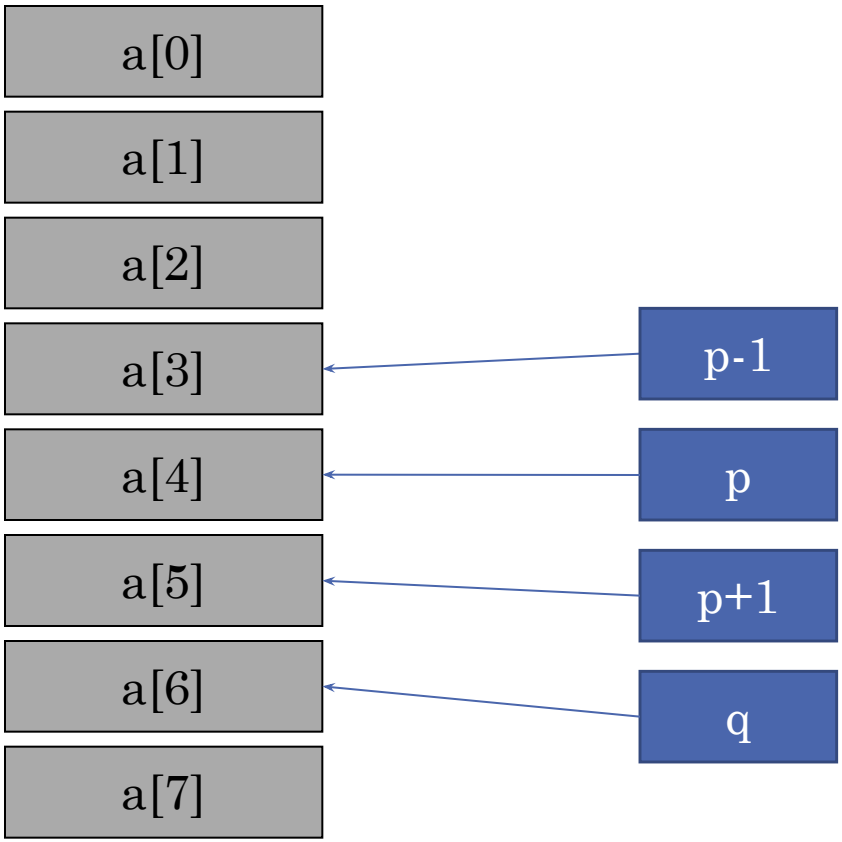
```
delete n;

delete [ ] r;
```

Адресная арифметика

- Если **p** – указатель на некоторый элемент массива, то
  - p+1 – указатель на следующий элемент
  - p-1 – указатель на предыдущий элемент
  - p+j – указатель на j-й элемент после p
  - p[j] разыменовывает j-й элемент относительно p
- Если p и q – указатели на некоторые элементы **одного массива**, то
  - p–q - равно количеству элементов после q, которое необходимо добавить, чтобы получить p
  - p<q принимает значение 1, если p указывает на элемент, предшествующий q, в противном случае - 0
  - p==q, принимает значение 1 если p и q указывают на один и тот же элемент, в противном случае - 0

Адресная арифметика в действии



p = &a[4]

q = &a[6]

q - 2 => p

q – p => 2

p + 2 => q

p < q => true

p + 1 == q - 1

p[3] => a[7]

p[-2] => a[2]

&a[8] - &a[0] => 8

Примеры

```
int arr[10];

// получаем указатель на начальный элемент
массива
int *p = arr; // эквивалентно int *p = &arr[0];

// следующие две строки эквивалентны
*(p + 4) = 5;
arr[4] = 5;

/* несмотря на то, что в массиве всего 10
элементов,
допускается получать указатель на ячейку,
следующую
за последним элементом массива */
p = &a[10];
*(p - 1) = 3; // эквивалентно arr[9] = 3;
```

Указатели на char

- Строковые константы – массивы символов с завершающим нулем
- Передача строковой константы в функцию (напр. printf) осуществляется путем передачи указателя на ее начальный элемент

Особенности

- Присваивание символьных указателей, **не копирует строки**
  - char * p = “Hello”;
    char * p1 = p; // p и p1 указывают на одно и то же место в памяти
- Символьный массив и символьный указатель – различные понятия
  - char msg[] = “Hello”; // массив
    - Символы внутри массива могут изменяться
    - msg всегда указывает на одно и то же место в памяти
  - char *pmsg = “Hello”; // указатель
    - Попытка изменить символы через pmsg приведет к неопределенному поведению
    - pmsg – указатель, можно присвоить ему другое значение в ходе работы программы

Указатели на указатели

- В C и C++ возможны указатели, ссылающиеся на другие указатели
  - char arr[] = “Hello”;
    char *parr = arr;
    char **pparr = &parr; // pparr – хранит адрес указателя parr
    (*pparr)[0] = ‘h’; // arr = “**h**ello”
    pparr[0][1] = ‘E’; // arr = “h**E**llo”;

Инкремент и декремент указателя

- Когда указатель ссылается на определенный элемент массива, имеют смысл операции инкремента и декремента указателя
  - char str[] = “Hello, world!”;
    char *p = str;// p указывает на символ H
    p++; // p указывает на символ e
    *p = ‘E’; // заменяем символ e на E

Указатели и динамическая память

- Часто возможны ситуации, когда размер и количество блоков памяти, необходимых программе, не известны заранее
- В этом случае прибегают к использованию динамически распределяемой памяти
  - Приложение может запрашивать блоки памяти необходимого размера из области, называемой кучей (heap)
  - Как только блок памяти становится не нужен, его освобождают, возвращая память в кучу

Операторы new и delete

- В состав языка C++ вошли операторы **new** и **delete**, осуществляющие работу с динамической памятью на уровне языка
  - Оператор new выделяет память под элемент или массив элементов
    - Тип *p = new Тип()
    - Тип *p = new Тип(инициализатор,...)
    - Тип *p = new Тип[кол-во элементов]
  - Оператор delete освобождает память, выделенную ранее оператором new
    - delete pObject;
    - delete [] pArray;

Прочие средства работы с динамической памятью

- В стандартной библиотеке языка С для работы с динамической памятью служат функции:
  - malloc()
  - calloc()
  - realloc()
  - free()
- Существуют средства работы с динамической памятью, зависящие от используемой ОС или используемых компонентов

Функции memcpy, memset и memmove

- Функция **memcpy()** осуществляет копирование блока памяти из одного адреса в другой
  - void memcpy(void *dst, const void *src, size_t count)
- Функция **memmove()** аналогична memcpy(), но корректно работает, если блоки перекрываются
  - void memmove(void *dst, const void *src, size_t count)
- Функция **memset()** заполняет область памяти определенным значением типа char
  - void memset(void *dst, int c, size_t count)

Пример

```
int n = 30;

// выделяем память под n элементов типа int
int * arr = (int*)malloc(sizeof(int) * n);

memset(arr, 1, sizeof(int) * n);

arr[0] = 5;

free(arr);
arr = NULL;
```

Указатели на структуры и объединения

- Указатели на структуры объявляются аналогично указателям на другие типы
- Для доступа к элементам структуры может применяться оператор ->
  - ```
    struct Point
    {
        int x, y;
    };

    Point p = {10, 20};
    Point *pPoint = &p;
    (*pPoint).x = 1;
    pPoint->y = 2;
    ```
    ```

Правила корректной работы с динамической памятью

- Объекты, выделенные при помощи оператора **new** должны быть удалены при помощи оператора **delete**
  - ```
    MyType * pObj = new MyType;
    delete pObj;
    ```
- Массивы объектов, выделенные при помощи оператора new [] должны быть удалены при помощи оператора delete []
  - ```
    MyType * pArray = new MyType[N];
    delete [] pArray;
    ```
- Блоки памяти, выделенные при помощи функции malloc (realloc, calloc) должны быть освобождены при помощи функции free
  - ```
    void * p = malloc(1000);
    free(p);
    ```
- Использование «непарных» средств освобождения памяти приведет к неопределенному поведению
    ```

Проблемы ручного управления памятью

- «Висячие ссылки» (dangling pointer)
  - После удаления объекта все указатели на него становятся «висячими»
    - Область памяти может быть отдана ОС и стать недоступной, либо использоваться новым объектом
    - Разыменование или попытка повторного удаления приведет либо к аварийной остановке программы, либо к неопределенному поведению
  - Причина возникновения: неправильная оценка времени жизни объекта – команда удаления объекта вызывается до окончания его использования в программе

Проблемы ручного управления памятью (продолжение)

- Утечка памяти (Memory Leak)
  - Причины:
    - Программист не удалил объект после завершения использования
    - Ссылающемуся на объект указателю присвоено новое значение, тогда как на объект нет других ссылок
      - Объект становится недоступен программно, но продолжает занимать память
  - Следствие
    - Программа все больше и больше потребляет памяти, замедляя работу системы, пока не исчерпает доступный объем адресного пространства и не завершится с ошибкой

Примеры **некорректной** работы с динамической памятью

```
int main(int argc, char* argv[])
{
	int * pIntArray = new int[100];
	free(pIntArray);	// Неопределенное поведение: использование free вместо delete []

	int * pAnotherIntArray = new int[10];
	delete pAnotherIntArray; // Неопределенное поведение: использование delete вместо delete []

	// Выделяем в куче один объект float, инициализируя его значением 100
	float * pFloat = new float(100);
	delete [] pFloat;	// Неопределенное поведение: использование delete [] вместо delete

	char * myString = new char[100];
	delete [] myString;
	delete [] myString; // Неопределенное поведение: повторное удаление массива

	char * anotherString = new char[10];
	delete [] anotherString;
	anotherString[0] = 'A'; // Неопределенное поведение: доступ к элементам удаленного массива

	void * pData = malloc(100);
	free(pData);
	free(pData);	// Неопределенное поведение: повторное удаление блока данных
}
```

Еще примеры **некорректной** работы с динамической памятью

```
int main(int argc, char* argv[])
{
	int * someInt = new int(11);
	someInt = new int(12);// Утечка памяти: старое значение указателя потеряно, память не освободить
	delete someInt;

	int someValue = *(new int(35)); // Утечка памяти: выделили в куче, разыменовали, адрес потеряли

	int * p = new int(10);
	if (getchar() == 'A')
	{
		return 0;// Утечка памяти: забыли вызывать delete p перед выходом из функции
	}
	delete p;

	return 0;
}
```

Ссылки

Ссылки

- Ссылку можно рассматривать как еще одно имя объекта
- Синтаксис
  - **<Тип> &** означает ссылку на **<Тип>**
- Применение
  - Задание параметров функций
  - Перегрузка операций

Ссылки в качестве параметров функций

- Функция принимает не копию аргумента, а ссылку на него
  - При сложных типах аргументов (классы, структуры) это может дать прирост в скорости вызова функции
    - Не тратится время на создании копии
    - Простые типы, как правило, эффективнее передавать по значению
      - char, int, float, double
  - Изменение значения формального параметра внутри функции приводит к изменению значения переданного аргумента
    - Альтернативный способ возврата значения из функции
    - Возврат нескольких значений одновременно

Константные ссылки в качестве параметров функций

- Параметр, переданный в функцию по константной ссылке, доступен внутри нее только для чтения
- Если функция не изменяет значение своего аргумента, то имеет смысл передавать его по константной ссылке
  - Простые типы данных следует передавать по значению

Пример 1

```
#include <stdio.h>

void Swap(int & a, int & b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int a = 1, b = 3;
    pritnf("a=%d, b=%d\n", a, b);
    Swap(a, b);
    pritnf("a=%d, b=%d\n", a, b);
}
```

OUTPUT

```
a=1, b=3
a=3, b=1
```

Пример 2

```
struct Point
{
    int x, y;
};

void Print(Point const& pnt)
{
    printf("(x:%d, y:%d)\n", pnt.x, pnt.y);
}

int main()
{
    Point pnt = {10, 20};
    Print(pnt);

    return 0;
}
```

Инициализация ссылки

- Ссылка должна быть обязательно проинициализирована
  - Должен существовать объект на который она ссылается
  - Синтаксис
    - Тип & идентификатор = значение;
- Инициализация ссылки совершенно отличается от операции присваивания
  - Будучи проинициализированной, присваивание ссылке нового значения **изменяет значение ссылаемого объекта, а не значение ссылки**

Пример

```
#include <stdio.h>

int main()
{
    int i = 1;
    int j = 3;

    // инициализация ссылки
    int & ri = i;
    printf("i=%d, j=%d\n", i, j);

    // присваивание значения объекту, на который ссылается ri
    ri = j;
    printf("i=%d, j=%d\n", i, j);
}
```

OUTPUT

```
i=1, j=3
i=3, j=3
```

Ссылки на временные объекты

- При инициализации ссылки объектом другого типа компилятор создает временный объект нужного типа и использует его для инициализации ссылки
  - На данный временный объект может ссылаться только константная ссылка
  - То же самое происходит при инициализации ссылки значением константы
  - Изменение значения объекта в данном случае не отражается на значении временного объекта
- Время жизни временного объекта равно области видимости созданной ссылки

Пример 1

```
int a = 1;
int & refA = a;        // ссылка на a

printf("a = %d\n", a);
++refA;
printf("Now a = %d\n\n", a);

const double & refDoubleA = a;// ссылка на временный объект
printf("refDoubleA = %f\n", refDoubleA);

// изменение a не оказывает влияния на refDoubleA
++a;
printf("Now a = %d, refDoubleA = %f\n", a, refDoubleA);
```

OUTPUT:

```
a = 1
Now a = 2

refDoubleA = 2.00000
Now a = 3, refDoubleA = 2.00000
```

Пример 2

```
#include <iostream>

int Add(int x, int y)
{
    return x + y;
}

int main(int argc, char* argv[])
{
    int & wontCompile = Add(10, 20);// Ошибка компиляции

    const int & result = Add(10, 20);// OK

    std::cout << result << "\n";
    return 0;
}
```