Тема 3
Указатели и работа с памятью

Указатели

Указатель – специальный тип данных, предназначенный для хранения адресов памяти.

Указатель не является самостоятельным типом данных, он всегда связан с каким-либо другим конкретным типом. В C++ есть три вида указателей:

- типизированные указатели;
- бестиповые (абстрактные) указатели;
- указатели на функции.

Указатели одного вида можно сравнивать друг с другом на равенство/неравенство и присваивать друг другу.

Типизированные указатели

Типизированный указатель содержит адрес области памяти, в которой хранятся данные определенного типа. Простейшее объявление указателя данных имеет вид:

тип *имя [инициализатор];

где тип может быть любым, кроме ссылки и битового поля, причем тип может быть к этому моменту только объявлен, но еще не определен.

Звездочка относится непосредственно к имени, поэтому для того, чтобы объявить несколько указателей, требуется ставить ее перед именем каждого. Например, в операторе

```
int *a, b, *c;
```

описываются два указателя на целое с именами a и c, а также целая переменная b.

Бестиповые указатели

Для описания абстрактных указателей, в качестве базового используется тип **void**:

void *имя [инициализатор]**;**

Абстрактному указателю можно присвоить значение указателя на любой тип, а также сравнивать его с любым указателем. Однако перед выполнением каких-либо действий с областью памяти, адрес которой содержится в абстрактном указателе, необходимо выполнить операцию преобразования типов

Операция разыменования

Операция разадресации (разыменования) имеет вид

*указатель

и позволяет обратиться к содержимому памяти, адрес которой хранится в указателе. Результат выполнения операции – L-value.

```
int *a;
… // здесь указатель должен быть
  // проинициализирован
*a = 10;
```

Получение адреса

Операция получения адреса, имеющая вид

&переменная

возвращает адрес конкретной переменной. Этот адрес можно поместить в указатель соответствующего типа, тем самым, возможно, инициализируя этот указатель.

```
int P = 10, *t;
// разыменование для указателя t делать ещё нельзя!
t = &P;

// теперь обращения *t и P эквивалентны
```

Инициализация указателей

Инициализация указателей, а также присваивание им новых значений может быть выполнена несколькими способами.

1. Присваивание указателю адреса уже существующего объекта, например:

```
char c;
char *pc = &c;
```

2. Присваивание указателю значения другого уже инициализированного указателя, например:

```
char c;
char *pc1 = &c, *pc2 = pc1;
```

Инициализация указателей (продолжение)

3. Присваивание указателю адреса памяти в явном виде:

```
char *pc = (char *)0x000012D4;
```

4. Присваивание указателю специального значения NULL:

```
char *pc = NULL;
```

Значение NULL гарантирует, что объектов по этому адресу не будет, и при попытке разыменования пустого указателя возникнет исключительная ситуация. Следует заметить, что идентификатор NULL необязателен, вместо него можно использовать значение 0. Однако для того, чтобы сделать программу более понятной, рекомендуется использовать именно идентификатор NULL.

5. Выделение динамической памяти

Использование константы nullptr

В стандарте языка C++11 для обнуления указателей появилось специальное ключевое слово nullptr. В более ранних стандартах официально использовалась запись:

```
Foo* foo = 0;
```

либо вариант с макросом NULL. Проблема очевидна: для обнуления указателя используется *целое число*, что может привести к ошибкам.

Константа nullptr имеет свой собственный тип — std::nullptr_t, и компилятор не спутает его ни с чем другим.

Я рекомендую: везде, где это возможно, использовать nullptr вместо NULL или 0

Классы памяти для языка C++

- Статическая память

(выделяется на этапе компиляции для глобальных переменных и переменных, описанных как static. Не может быть переопределена)

- Память стека

(выделяется для локальных переменных при входе в блок, в котором они определены. При выходе из блока память автоматически освобождается)

- Динамическая память

(выделяется и освобождается по запросу. Допускается повторное выделение освобождённой памяти)

Механизмы выделения и освобождения памяти

1. Механизм C

- память выделяется с помощью функции malloc (или подобной ей):

```
void * malloc(объём_выделяемой_памяти);
```

Например,

```
int *pi = (int *) malloc(sizeof(int));
```

- память освобождается с помощью функции free

```
void free(void *);
```

Например,

```
free(pi);
```

Значение указателя не изменяется после выполнения функции free!

Механизмы выделения и освобождения памяти

2. Механизм C++

- память выделяется с помощью операции new:

```
new тип;
```

Например,

```
int *pi = new int;
```

- память освобождается с помощью операции delete

```
delete указатель;
```

Например,

```
delete pi;
```

Значение указателя не изменяется после выполнения операции delete!

Проблемы, возникающие при работе с указателями

- Использование неинициализированных указателей

Неинициализированный локальный указатель содержит какое-то значение, которое трактуется, как адрес памяти. Если случайно окажется, что этот адрес доступен для приложения – последствия непредсказуемы!

- Использование «зависших» указателей

После освобождения динамической памяти её адрес остаётся доступным в программе. Обращение по такому адресу приведёт либо к разрушению динамической памяти, либо к доступу к повторно выделенной памяти!

Проблемы, возникающие при работе с указателями (продолжение)

- «Утечка» памяти

Потеря значения адреса выделенной памяти влечёт невозможность доступа к ней (в том числе и освобождения) до окончания работы программы.

Возможные причины утечки памяти:

- изменение значения указателя, который использовался при выделении памяти;
- выход такого указателя из области своего действия.

Примеры неправильной работы с памятью

Пример 1

```
int *pi1 = new int, *pi2 = new int;
*pi1 = 0;
pi2=pi1;     // хотели написать *pi2 = *pi1  ☹
// получили утечку памяти
…
(*pi2)++;
// тем самым изменилось и *pi1, но мы этого не видим…
…
delete pi1; // пока всё хорошо…
delete pi2; // использование зависшего указателя – крах!
```

Пример 2

```
while (…) {
 int *pi1 = new int;
…
} // освободить память забыли, а указатель исчез!
```

Арифметические операции над указателями

- Поскольку значения указателей, т.е. адреса, по сути являются числами, с ними можно выполнять некоторые арифметические операции: сложение с константой, вычитание, инкремент и декремент. Арифметические операции над абстрактными указателями недопустимы
- При выполнении арифметических операций учитывается длина базового типа, адресуемого указателями. Если у нас было описание **int** *p и мы выполняем операцию p++, то значение p увеличивается не на единицу, а на **sizeof**(**int**) .
- Разность между двумя указателями – это разность их значений, деленная на размер типа в байтах.
- Суммирование двух указателей не допускается.

Пример неправильной работы с памятью (часть 2)

При изменении значений указателя не производится никакого контроля на то, чтобы новый адрес памяти был легально выделен программе. Так, записав последовательность операторов

```
int *pv = new int;
…
pv++;
*pv=1;
```

мы изменим содержимое области памяти, которая либо не была выделена, либо была выделена для совершенно других целей!

Константные указатели

- Константные указатели – указатели, значение которых не изменяется в ходе выполнения программы. Они описываются так:

указательный_тип **const** имя инициализатор

Пример:

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

Изменить значение константного указателя нельзя, а изменить значение памяти, адрес которой находится в указателе – можно:

```
p++;   // нельзя!
(*p)++; // можно!
```

Указатели на константу

- Указатели на константу – указатели, содержащие адрес константы. Они описываются так:

const указательный_тип имя [инициализатор]

Пример:

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

Изменить значение такого указателя можно, а изменить значение памяти, адрес которой находится в указателе – нельзя:

```
p++;   // можно!
(*p)++; // нельзя!
a++; // можно!
```