Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

09 октября 2014

Пространства имен

Служат для группировки программных понятий (переменные, функции и т.д).

```
namespace имя_пространства_имен
{
переменные;
функции;
};
```

Для обращения к переменной в пространстве имен используется оператор ::

Пространства имен

```
namespace my
{ int x; };
namespace your
{ int x; };
int x;

int main()
{
my::x = 5;
x = 7;
your::x = 8;
}
```

Директива using

Делает все члены пространства имен видимыми.

```
namespace my
{ int x; };
namespace your
{ int x; };
int x;

x = 10; //???
using namespace my;
x = 11; //???
my::x = 12; //??
```

Директива using

```
namespace my
{ int x; int y; };
namespace your
{ int x; int y; };
int x;

x = 10; //???
y = 11; //???
using my::x;
x = 11; //???
y = 11; //???
my::x = 12; //??

using namespace my;
using namespace your;
x = 20;
y = 20;
```

Директивы препроцессора

- Директива должна начинаться с символа «#».
- Директивы препроцессора пишутся на каждой строчке отдельно, для продолженния директивы на другой строке можно использовать символ \.
- Большинство директив модно использовать в любом месте файла.

Список директив препроцессора

- #include
- #define, #undef
- #if, #elif, #else, #endif
- #pragma
- #line
- #error

#define

- Позволяет программисту задавать макроопределения (макросы).
- Компилятор производит подстановку текста макроса в коде программы.
- Задать подстановку можно 2 способами:
  - #define идентификатор тело_макроса.
  - В опциях командной строки компилятора.
- Тело макроса указывать не обязательно

```
#define PI 3.14
double radius = 5.0;
double result = 2 * PI * r;
```

```
#define minutes /60
#define hours /60
int interval = 3600;
std::cout << (interval hours) << std::endl;
std::cout << (interval minutes) << std::endl;
```

Пример

```
#define X 2
const double Y=2;
double res1 = X/5; //???
double res2 = Y/5; //???
```

Вложенные макросы

```
#define RED 1
#define GREEN 2
#define YELLOW RED | GREEN

int color = YELLOW;
```

Макросы с параметрами

#define имя_макроса(параметры) тело_макроса

```
#define SQUARE(x) x * x

double z = 1.2;
double y1 = SQUARE(z); //???
double y1 = SQUARE(z+1); //???
double y1 = SQUARE(z++); //???
```

Конкатенация макросов

Используя сочетание ##, можно склеить несколько лексем в одну, т.е. препроцессор соединит две строки и подставит результат.

```
#define NICK(name,number) name##number

int NICK(vasia,1) = 1; //int vasia_1 = 1;
double NICK(petia,1) = 1.2; //double petia_1 = 1.2;
```

Предопределенные макросы

- __DATE__
- __FILE__
- __LINE__
- __TIME__

```
std::cout << "Current file is: " << __FILE__ << " "
    << __LINE__;
```

Диагностический макрос assert

Работает только в DEBUG версии.

- assert(x $<$ 0);
- Если значение переданное в макрос равно 0, то выводит диагностику с указанием файла и номера строки.
- Вызывает функцию аварийного завершения программы.

Условная компиляция

```
#ifdef _DEBUG
std::cout << debug_variable;
#endif

#ifdef _UNICODE
typedef wchar_t TCHAR;
#else
typedef char TCHAR;
#endif

#if defined RUS
std::cout << "По-русски";
#elif defined ENG
std::cout << "In english";
#else
std::cout << "Unknown language";
#endif
```

Указатели

Указатель — это переменная, содержащая адрес другого программного понятия C/C++-программы. Указатель может указывать на:

- На данные
  - на переменную базового типа;
  - на указатель;
  - на массив (строку);
  - на переменную (объект) пользовательского типа;
- На функции

Свойства указателя

- указатель всегда содержит адрес;
- то каким образом компилятор должен интерпретировать этот адрес, сообщает ему программист, задавая тип «указываемого объекта»;
- количество байтов, выделяемых компилятором под переменную (объект), зависит от типа объекта;
- количество байтов выделяемых под указатель для хранения адреса, не зависит от типа объекта, а определяется только аппаратными особенностями используемого процессора и разрядностью регистров предназначенных для хранения адресов и фиксировано для каждой архитектуры.

Объявление переменной указателя

```
int *p1;
int * p2;
int*p3;
int* p4;
int  *  p5;
double *p6;
```

Выделяется память для переменной типа указатель, но сам указатель пока никуда не указывает. Корректный существующий адрес должен быть присвоен ДО его ПЕРВОГО ИСПОЛЬЗОВАНИЯ!

Объявление переменной указателя

Память под указатель может быть выделена:

- в статической области;
- в стеке;
- в куче.

В не зависимости от того где располагается сам объект на который указывает указатель.

```
int *p; //???

int main()
{
  int *p1; //???
  static int* p2; //???
}
```

После объявления указателя на один тип, попытка его использования для ссылки на другой тип, без явного приведения типа, приведет к ошибке компиляции.

Примеры объявления указателей

```
char *p1, p2, p3; //???

typedef char* PCHAR;
PCHAR p1, p2, p3; //???

#define PCHAR char*
PCHAR p1, p2;
```

Явная инициализация указателя

При явной инициализации указателя, требуется получить адрес переменной располагающейся справа от знака равенства. Для этого используется оператор получаения адреса &;

```
int a;
int *p_a = &a;

double b;
int *p_b = &b;

char c;
char *p_c = &c;

char *p_s = "ABC";
```

Указатели инициализируются по тем же правилам, что и обычные переменные.

Оператор получения адреса &

Оператор & можно применить только к lvalue переменной.

```
int *p1 = &(x+y); //???
int *p2 = &1000; //???

int *p3 = 0; //или nullptr //???
int *p4 = 1000; //???

int *p5 = &((x > y) ? x : y); //???
int *p6 = &(++x); //???
int *p7 = &(x++); //???
```

Оператор разыменования *

Позволяет перейти по адресу и получить значение хранимой в выделенной области памяти переменной.

```
int i = 10;
int *i_p = &i;

double d = 1.2;
double *d_p = &d;


std::cout << "i= " << *i_p << "d= " << *d_p << std
    ::endl;
*i_p = 70;
d = 70.0;
std::cout << "i= " << *i_p << "d= " << *d_p << std
    ::endl;
```

Адресная арифметика

- При изменении значения по адресу, который хранится в указателе происходит изменее переменной такого типа на которую ссылается указатель.
- При выполнении действий с указателями, компилятор всегда интерпретирует то, на что указывает компилятор как совокупность элементов располагаемых линейно (подряд) в памяти. Только программист знает на сколько реально элементов ссылается указатель.
- К указателю можно прибавить или вычисть значение, но только целое.

Примеры

[fragile]

```
double d = 1.1;
double *p = &d;

/* Что произойдет в каждом из фрагментов, если в
   каждый момент времени существует только одна
   переменная res указана? */
double res = *(p++); //1
double res = (*p)++; //2
double res = ++(*p); //3
double res = *(++p); //4

int size1 = sizeof(p);
int size2 = sizeof(*p);
```

Присваивание и сравнение указателей

```
double d = 1.1;
double *p = &d;
double *p2;

p2 = p;

std::cout << *p << "␣" << *p2 << "␣" << (p2 == p)
    << std::endl;
```

Сложения, умножения и деления указателей запрещены.

Тип void*

Может содержать указатель указывающий на любой тип.

```
void *p;
int n = 1;
char c = 'a';
double d = 1.1;

pn = &n;
pn = &c;
pn = &d;

double *p_d;
p_d = p; //???
p_d = (double*) p; //???
p_d = static_cast<double*>(p); //???

double d = *static_cast<double*>(p);
d = *p_d;

std::cout << sizeof(p) << " " << sizeof(*p); //???
```

Указатель на указатель

```
char c = 'A';

char *cP = &a;
char *cPP = &cP;
char *cPPP = &cPP;

std::cout << c;
std::cout << *cP;
std::cout << **cPP;
std::cout << ***cPPP;
```

const и указатели

Ключевое слово const может располагаться, слева или справа от * при объявлении типа. В зависимости от этого, компилятор выбирает:

- const относится к самому указателю, т.е. к значению адреса (адрес не изменен);
- const относится к значению на которое указывает указатель (значение не изменно).

const и указатели

Указатель на константу:

```
const char *pc;
char const *pc1;
pc = "ABC";

*pc = 'Q'; //???

std:: cout << pc;
pc++;
std:: cout << pc;
```

const и указатели

Константный указатель:

```
char * const pc = "ABC";
char * const pc2; //???

*pc = 'Q'; //???

std:: cout << pc;
pc++;
std:: cout << pc;
```