Виды памяти в языке C++

Существуют четыре спецификатора, управляющих размещением программных объектов в памяти и влияющих на область видимости и время жизни объектов:

- register,
- auto,
- static,
- extern.

В языке C++ предусмотрены четыре основных вида памяти, в которых могут размещаться программные объекты:

- регистровая память,
- локальная (стековая) память,
- статическая память,
- динамическая (свободная) память.

Регистровая память. Спецификатор register

Спецификатор register предписывает компилятору попытаться разместить объявляемую переменную в регистрах процессора. Если это компилятору не удаётся, переменная ведёт себя как обычная локальная переменная со спецификатором auto.

Размещение переменных в регистрах ускоряет выполнение программы, так как процессор оперирует с данными в регистрах существенно быстрее, чем с данными в оперативной памяти. Но так как количество регистров процессора ограничено, этим спецификатором следует помечать небольшое число наиболее интенсивно используемых переменных программы.

Например:

```
int main()
{
	register double sum;
	…
}
```

Спецификатор register нельзя применять к глобальным переменным программы.

Компилятор Visual C++ .NET игнорирует спецификатор ***register***.

Локальная память. Спецификатор auto

В локальной памяти компилятором размещаются аргументы функций и локальные переменные, которые видны только в блоке, в котором они были объявлены. Каждый блок, включая функции, получает свою копию локальных переменных. **При выходе из блока (или из функции) его локальные переменные уничтожаются автоматически.**

Локальную память называют **стековой**, так как аргументы функций и локальные переменные компилятором размещаются в стековой памяти, которая выделяется операционной системой каждому параллельно работающему потоку (thread).

Спецификатор auto для локальных переменных используется по умолчанию, поэтому его практически никогда не указывают. Например:

```
int main()
{
	auto int a = 1;  // то же, что int a = 1;
	…
}
```

Статическая память. Спецификатор static

В статическую память компилятор помещает объекты, которые нужны на протяжении всего времени выполнения программы. В этой памяти располагаются все глобальные переменные (переменные, объявленные глобально, т. е. вне любой из функций программы) и статические переменные функций *и статические члены классов, а также переменные из пространств имён*.

При объявлении статических объектов используется спецификатор *static*. Если такой объект объявлен глобально, то он инициализируется при запуске программы, а его область видимости простирается от точки объявления до конца файла.

Если же статический объект объявлен внутри функции или блока, то он инициализируется при входе в соответствующую функцию или блок. Значение сохраняется от одного входа в функцию или блок до другого входа.

Как было сказано ранее, если инициализатор для статического объекта не задан, то объекту присваивается нулевое значение соответствующего типа.

Статические переменные можно использовать для передачи значений от одного вызова функции к другому, либо между разными функциями. Например:

```
int Count(void)
{
   static int  counter;
   return counter++;
}

int main()
{
   int count = 0;
   double result = 0.0;
   while (count < 30)
      result += 1.0 / (count = Count());
   cout << result;
}
```

Здесь в главной функции в цикле 30 раз будет вызвана функция *Count()*, которая, благодаря статической переменной *counter*, последовательно возвратит ряд чисел 1, 2, …, 30. В итоге, в цикле будет вычислена сумма 1/1 + 1/2 + … + 1/30, которая по окончании цикла будет выведена на экран.

Автоматическая сборка мусора

Суть автоматической сборки мусора заключается в следующем: диспетчер памяти находит объекты, к которым программа не может обратиться, так как на них не осталось ссылок, и забирает их память, считая её свободной. Сам такой диспетчер памяти называется сборщиком мусора.

Принцип сборки мусора состоит в том, что объект, на который больше нет ссылок в программе, более не доступен, и поэтому занимаемую им память можно освободить. Например:

```
void f()
{
        int* p = new int;
        p = 0;
        char* q = new char;
}
```

В программе инструкция присваивания p = 0; оставляет без ссылки целую переменную, размещённую в динамической памяти инструкцией int *p = new int;. Поэтому, при наличии в системе автоматической сборки мусора, на её место инструкцией char *q = new char; может быть помещена новая переменная типа char. В этом случая указатель q будет содержать тот же адрес, что и p.

Автоматическая сборка мусора становится всё более популярной.

В настоящее время автоматическая сборка мусора используется в системе программирования Visual C++ .NET при работе программ, написанных на управляемом C++, и является краеугольным камнем этого диалекта C++.

Изначально языки Visual Basic и C#, а также системы программирования для них проектировались в расчёте на наличие в системе автоматической сборки мусора. Поэтому ручное управление распределением памяти в программах на этих языках невозможно и, как следствие, в этих языках отсутствуют указатели и прямая работа с ними.

Фрагментация памяти

В результате многочисленных операций выделения и освобождения памяти может произойти фрагментация: куча будет содержать много свободной памяти, но ни одного блока, размер которого был бы достаточен для удовлетворения текущего запроса.

Независимо от наличия в системе автоматической сборки мусора, **когда участки динамической памяти выделяются и освобождаются для множества объектов разного размера, память фрагментируется. Это происходит из – за того, что новый объект, как правило, размещается в чуть большем участке памяти, чем ему нужно, и остаток памяти этого участка уже трудно использовать для чего – то другого.**

При отсутствии в системе сборщика мусора из – за фрагментации памяти оператор new может выдать ошибку нехватки памяти даже в том случае, когда суммарный объём свободных участков памяти многократно превышает размер участка, требуемый для размещения объекта.

В системах с автоматической сборкой мусора проблема фрагментации памяти решается применением одного из двух видов сборщиков мусора:

- **копирующего сборщика, который копирует «живые» объекты и таким образом сжимает память, устраняя фрагментацию;**
- **консервативного сборщика, который размещает объекты так, чтобы минимизировать фрагментацию памяти.**

С точки зрения языка C++ предпочтительнее консервативные виды сборщиков мусора, так как перемещение объектов и правильная модификация указателей на них копирующим сборщиком может потребовать значительного времени, что чревато большими одновременными перерывами в работе всех потоков приложения.

Тем не менее, в системе .NET, обеспечивающей функционирование программ, написанных на C++, C# и Visual Basic, используются алгоритмы именно копирующего сборщика мусора.

4.9. Типы данных, определяемые пользователем

4.9.1. Переименование типов

Типу можно задавать имя с помощью ключевого слова ***typedef:***

typedef тип имя_типа [размерность];

Пример 93

```
typedef unsigned int UNIT;
typedef char Msg[100];
```

Такое имя можно затем использовать также как и стандартное имя типа:

```
UNIT a,b,c;          //переменные типа unsigned int
Msg str[12];         // массив из 12 строк по 100 символов
```

4.9.2. Перечисления

Если надо определить несколько именованных констант таким образом, чтобы все они имели разные значения, можно воспользоваться перечисляемым типом:

enum [имя_типа] {список констант};

Константы должны быть ***целочисленными*** (логические, символьные, целые типы) и могут инициализироваться обычным образом. Если инициализатор отсутствует, то первая константа обнуляется, а остальным присваиваются значение на единицу большее, чем предыдущее.

Пример 94

```
enum Err{ErrRead, ErrWrite, ErrConvert);
Err error;
. . . .
switch(error)
{
case ErrRead: …..
case ErrWrite: …..
case ErrConvert: …..
}
```

4.9.3. Структуры

Структура – это объединенное в единое целое множество поименованных элементов данных. Элементы структуры (поля) могут быть различного типа, они все должны иметь различные имена.

Форматы определения структурного типа следующие:

1. **struct имя_типа** //способ 1
{
тип 1 элемент1;
тип2 элемент2;
. . .
};

В первом случае описание структур определяет новый тип, имя которого можно использовать наряду со стандартными типами.

Пример 95

```
struct Date              //определение структуры
{
int day;
int month;
int year;
};
Date birthday;           //переменная типа Date
```

2**. struct** //способ 2
{
тип 1 элемент1;
тип2 элемент2;
. . .
} список идентификаторов;

Во втором случае описание структуры служит определением переменных.

Пример 96

```
struct
{
int min;
int sec;
int msec;
}time_beg, time_end;
```

3.Структурный тип можно также задать с помощью ключевого слова ***typedef***:

Пример 97. способ 3

```
typedef struct
{
float re;
float im;
}Complex;
Complex a[100];          /*массив из 100 комплексных чисел.*/
```

Способы 1 и 3 способ идентичны (определяется новый тип). Способ 2 описывает переменные заданной структуры.

Инициализация структур

Для инициализации структур значения ее полей перечисляют в фигурных скобках.

Пример 98

```
1. struct Student
{
char name[20];
int kurs;
float rating;
};
Student s={”Иванов”, 1, 3.5};

2.struct
{
char name[20];
char title[30];
float rate;
} employee={“Петров”, “директор”, 10000};
```

Присваивание структур

Для переменных одного и того же структурного типа определена операция присваивания. При этом происходит ***поэлементное копирование***.

```
Student ss=s;
```

Доступ к элементам структур

Доступ к элементам структур обеспечивается с помощью уточненных имен:
Имя_структуры.имя_элемента

Пример 99 Ввести данные о 15 студентах: фамилия, группа, средний бал. Вывести фамилии студентов, средний бал которых меньше 4.

```
#include <iostream>
using namespace std;
void main()
{
	struct Student
	{
		char name[30];
		char group[12];
		float rating;
	};
	Student mas[15];
	int i;
	//ввод значений массива
	for( i=0; i<15; i++)
	{
		cout<<"\nEnter name:"; cin>>mas[i].name;
		cout<<"\nEnter group:"; cin>>mas[i].group;
```

```
            cout<<"\nEnter rating:"; cin>>mas[i].rating;
        }
        cout<<"Raitng <4:";
        for( i=0;i<15;i++)
        if(mas[i].rating<4)
        cout<<"\n"<<mas[i].name;
}
```

Указатели на структуры

Указатели на структуры определяются также как и указатели на другие типы:

Student*ps;

Пример 100. Ввод указателя для типа *struct*, не имеющего имени (способ 2):

```
Struct
 {
        char *name;
        int age;
  } *person;            //указатель на структуру
```

При определении указатель на структуру может быть сразу же проинициализирован:

Student *ps=&mas[0];

Указатель на структуру обеспечивает доступ к ее элементам двумя способами:

1. **(*указатель).имя_элемента** – *прямой доступ:*

cin>>(*ps).name;

2. **указатель->имя_элемента** – *косвенный доступ:*

cin>>ps->title.

4.9.5. Битовые поля

Битовые поля – это особый вид полей структуры. При описании битового поля указывается его длина в битах (целая положительная константа).

Пример 101.

```
struct {
int a:10;
int b:14;
}   xx,*pxx;
. . . .
   xx.a=1;
   pxx=&xx;
   pxx->b=8;
```

Битовые поля могут быть любого целого типа. Они используются для плотной упаковки данных. Например, с их помощью удобно реализовать флажки типа «да» / «нет».

4.9.6. Объединения

Объединение (union) – это частный случай структуры. Все поля объединения располагаются по одному и тому же адресу. Длина объединения равна наибольшей из длин его полей. В каждый момент времени в такой переменной может храниться только одно значение. Объединения применяют для экономии памяти, если известно, что более одного поля не потребуется. Также объединение обеспечивает доступ к одному участку памяти с помощью переменных разного типа.

Пример 102

```
union {
        char s[10];
        int x;
        } u1;
```

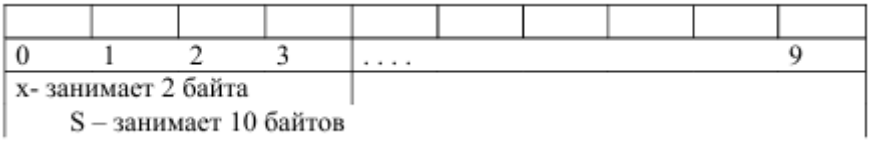

Рис. 24. Расположение объединения в памяти

Переменные s и x располагаются на одном участке памяти. Размер такого объединения будет равен 10 байтам.

Пример 103. Использование объединений

```
#include <iostream>
#include <windows.h>
using namespace std;
void main()
{
        SetConsoleOutputCP(1251);
        enum paytype{CARD,CHECK}       ;//тип оплаты
        struct  {
                paytype ptype;                  /*поле, которое определяет с каким полем
        объединения будет выполняться работа */
                union  {
                        char card[25];
                        long check;
                        };
                }info={CARD, "VISA"};
switch (info.ptype)
{
case CARD: cout<<"\nОплата по карте:"<<info.card;break;
case CHECK:cout<<"\nОплата чеком:"<<info.check;break;}
getch();
}
```

```
Оплата по карте: VISA
```

Если

```
info={CHECK,123};
```

то

```
Оплата чеком:123
```