Занятие 9

Раздельная компиляция программ. Продолжительность хранения переменных в памяти, их область видимости и связывание. Пространства имен.

Пример 1. Пример программы, состоящей из трех файлов.
Как правило, большие программы состоят из многих (например, десятки или сотни) файлов. Разбивка по файлам позволяет размещать в одном файле функции в соответствии с какими-либо общими признаками (например, по области приложения) и затем использовать их в различных программах. Таким образом, файлы играют роль библиотек функций. В примере используется файл `coordin.cpp`, где собраны функции для работы с координатами. Эти функции декларированы в заголовочном файле `coordin.h`. В этом же файле описан также новый тип данных – структура `polar`, которая используется этими функциями. Заголовочный файл используется для декларации функций в файле `example_8.1.cpp`, где эти функции непосредственно вызываются.
Файл `coordin.h`:

```
struct polar {
  double distance; // расстояние  от исходной точки
  double angle;    // направление от исходной точки
};

struct rect {
  double x; // расстояние по горизонтали от исходной точки
  double y; // расстояние по вертикали от исходной точки
};

void rect_to_polar(const rect *xy, polar *rphi);
void show_polar(const polar *p);
```

Файл `coordin.cpp`:

```
#include <iostream>
#include <cmath>
#include "coordin.h" // шаблоны структур, прототипы функций

static double square(double x){ return x*x; };

void rect_to_polar(const rect *xy, polar *rphi){
  // Преобразование прямоугольных координат в полярные
  using namespace std;
  rphi->distance = sqrt(square(xy->x) + square(xy->y));
  rphi->angle = atan2(xy->y, xy->x);
}

void show_polar(const polar *p) {
  // Отображение полярных координат с преобразованием радиан в градусы
  using namespace std;
  const double Rad_to_deg = 57.29577951;
  cout << "distance = " << p->distance;
  cout << ", angle = " << p->angle * Rad_to_deg << " degrees\n";
}
```

Файл `example_8.1.cpp`:

```
#include <iostream>
#include "coordin.h"              // шаблоны структур, прототипы функций

using namespace std;
int main(void) {
  rect xy1;
  polar rphi1;
  cout << "Enter the x and y values: "; // ввод значений x и y
  while(cin >> xy1.x >> xy1.y){          // ловкое использование cin
    rect_to_polar(&xy1, &rphi1);
    show_polar(&rphi1);
    cout <<"Next two numbers (q to quit) : ";  // ввод двух чисел
                                               // (q для завершения)
  }
  cout << "Done.\n";
  return 0;
}
```

Создание исполняемого файла `a.out`:
Для создания исполняемого файла надо откомпилировать все файлы, содержашие функции, используемые в программе. Это можно сделать двумя способами. Первый способ заключается в перечислении всех файлов после команды компиляции `g++`:

```
g++  example_8.1.cpp coordin.cpp
```

Второй способ заключается в отдельной компиляции каждого файла в объектный файл, а затем соединения всех объектных файлов:

```
g++ -c coordin.cpp
g++ -c example_8.1.cpp
g++    example_8.1.o  coordin.o
```

Давайте рассмотрим глубже процесс компиляции. Процесс компиляции любой программы (из одного или многих файлов) состоит из двух шагов. На первом шаге из исходного файла (например, с расширением `.cpp`) создается бинарный объектный файл (с расширением `.o`). При этом не все функции, вызываемые в исходном файле, могут в нем находиться. Если каких-то функций там нет, то в объектном файле записывается информация о требуемых функциях. На втором шаге происходит соединининение (линкование, от английсково linking) всех объектных файлов в один исполняемый. При этом все используемые функции должны быть в объектных файлах. Команда компиляции без опции `-c` и с перечислением исходных `.cpp` файлов осуществляет оба шага автоматически. Опция компилятора `-c` при компиляции исходных файлов `.cpp` позволяет создать объектный файл, но не создавать исполняемый. При вызове компилятора с перечислением объектных файлов происходит их объединение в исполняемый файл `a.out`. Имя исполняемого файла можно контролировать опцией компилятора `-o`. Если в программе из нескольки файлов поменялся один исходный файл, то достаточно из него создать новый объектный файл, а затем слинковать его с уже существующими объектными файлами и получить новый исполняемый файл. Таким образом, не надо перекомпилировать все файлы заново.

Пример 2. Классы памяти (storage class), диапазон доступа (scope) и связывание (linkage).

Переменные классифицируются по классу памяти (продолжительности хранения), диапазону доступа (области видимости) и связыванию.

По классу памяти переменные делятся на автоматические, статические и динамические.

- Все переменные, декларируемые внутри функций или блоков и не имеющие перед ними слово `static`, будут иметь автоматическое хранение. Память будет выделяться автоматически при определении переменной и освобождаться при выходе из блока или функции.

- Если переменная задана вне функции, то она приобретает статическое хранение. Такая переменная создается (то есть под ее хранение выделяется память) вначале выполнения программы и существует все время, пока программа выполняется.

- Если переменная задана внутри функции, но перед ней используется ключевое слово `static`, то она приобретает статическое хранение.

- Если память выделяется с помощью операций `new` или функции `malloc()`, то такое хранение называется динамическим. Выделенная память существует пока не освобождается операцией `delete` или вызовом `free()`.

Ограничение области видимости позволяет использовать одинаковые имена для переменных в разных функциях без возникновения конфликтов. По диапазону доступа переменные делятся на глобальные и локальные.

- Глобальные переменные – переменные, которые могут использоваться в различных функциях и файлах. Глобальные переменные – статические, то есть определены вне функций.

- Локальные переменные – переменные, диапазон доступа которых ограничен блоком или функцией. Автоматические переменные будут локальными. Если локальная переменная определена внутри блока (часть кода, ограниченная `{}`), то область ее видимости ограничена этим блоком. Если какая-либо переменная определена внутри блока, то все другие переменные с такими же именами, но определенные вне блока (локальные или глобальные), будут невидимы внутри этого блока, то есть происходит переопределение переменной. Локальные переменные внутри функции делают невидимыми внешние переменные с такими же именами.

- Также локальной переменной будет статическая переменная, объявленная внутри какой-либо функции, то есть со словом `static`. Такая переменная, в отличие от автоматической переменной, сохраняет свое значение между последовательными вызовами функции. Она инициализируется в начале работы программы. Последующие вызовы не инициализируют такую переменную заново.

По типу связывания переменные делятся на переменные без связывания, со внутренним связыванием и со внешним связыванием.

- Локальные переменные никакого связывания не имеют.

- Переменная со внешним связыванием – глобальная переменная, которая может быть использована в нескольких файлах. Такая переменная определяется только в одном файле. Для использования в других файлах необходимо ее декларировать (то есть объявить ее наличие) с использованием ключевого слова `extern`. Глобальные переменные с одинаковыми именами не могут существовать, если их область видимости не ограничена.

- Переменные со внутренним связыванием – глобальные переменные, область видимости которых ограничена одним файлом, то есть могут быть использованы разными функциями внутри одного файла. Такие переменные определяются с ключевым словом `static`.

Пример ниже иллюстрирует работу с переменными в программе из двух файлов и двух функций. Разберитесь, какие значения и адреса имеют переменные с именами `tom`, `dick`, `harry` в зависимости от того, откуда к ним обращаются.

Файл `example_8.2a.cpp`:

```
#include <iostream>

int  tom = 3;  // определение глобальной переменной, может
               // использоватья в других файлах

int dick = 30; // определение глобальной переменной,
               // может использоваться в других файлах

static int harry = 300; // глобальная со внутренним связыванием,
                        // то есть видна только в данном файле

void remote_access();   // прототип функции из другого файла

int main() {
  using namespace std;

  // вывод значений и адресов переменных
  cout << "main() reports the following values and addresses:\n";
  cout << "  tom  =" << tom   << "   at " << &tom   << endl;
  cout << "  dick =" << dick  << "  at "  << &dick  << endl;
  cout << "  harry=" << harry << " at "   << &harry << endl << endl;

  remote_access();  // вызываем функцию

  int dick=23;   // локальная переменная, переопределит глобальную
  cout << " dick=" << dick   << " at " << &dick << endl; // и ее значение

  return 0;
}
```

Файл `example_8.2b.cpp`:

```
#include <iostream>

extern int tom; // Декларация переменной tom, которая определена
                // в другом файле, но может использоваться в этом файле.
                // tom - глобальная со внешним связыванием

```

```
static int dick = 10;    // глобальная со внутренним связыванием,
                         // то есть видна только в данном файле

int harry = 200;       // определение глобальной переменной.
                       // Может использоваться в других файлах.
                       // конфликт с harry из example_8.2a отсутствует

void remote_access() {
  using namespace std;
  cout << "\nremote_access() reports the following values and addresses:\n";

  cout << "  tom  ="  << tom   << "   at " << &tom   << endl;
  cout << "  dick ="  << dick  << "  at "  << &dick  << endl;
  cout << "  harry="  << harry << " at "   << &harry << endl << endl;
}
```

Результат выполнения:

```
main() reports the following values and addresses:
  tom  =3   at 0x55caf06a4058
  dick =30  at 0x55caf06a405c
  harry=300 at 0x55caf06a4060


remote_access() reports the following values and addresses:
  tom  =3   at 0x55caf06a4058
  dick =10  at 0x55caf06a4064
  harry=200 at 0x55caf06a4068

 dick=23 at 0x7fffc08daffc
```

Пример 3. Пространства имен.
Пространство имен (namespace) – это некоторый список имен переменных, функций, структур и т.д. Объединение имен в такие списки позволяет избежать конфликты имен путем ограничения диапазона видимости. Такие конфликты могут возникать при наличии в программе глобальных переменных с одинаковыми именами или функций с одинаковыми именами и списками аргументов.

В примере ниже есть две функции `func()`. В нашем случае для простоты они определены в одном файле, но могут быть определены и в разных. Для избежания конфликта мы помещаем эти функции в разные пространства имен `NM1` и `NM2`. Затем используем эти функции в `main()` в файле `namespace_example.cpp`. Обратите внимание на то, как вызываются функции.
Файл `my_namespace.hpp`:

```
// файл с декларациями пространств имен
// и их содержания.

namespace NM1{
  double func(double x);
}
```

```
7
8  namespace NM2{
9    double func(double x);
10 }
```

Файл my_namespace.cpp:

```
1  // файл с определениями функций
2  namespace NM1{
3    double func(double x){
4      return 2*x;
5    }
6  }
7
8  namespace NM2{
9    double func(double x){
10     return 3*x;
11   }
12 }
```

Файл namespace_example.cpp:

```
1  #include<iostream>
2
3  #include"my_namespace.hpp" // декларируем все функции
4
5  using namespace NM1; // используем по умолчанию
6  using namespace std; // используем по умолчанию
7
8  int main(){
9
10   double x=5;
11
12   double y1 = func(x);       // используется func из NM1 по умолчанию
13   double y2 = NM2::func(x); // используется func из NM2
14
15   cout << "y1 = " << y1 << endl ;
16   cout << "y2 = " << y2 << endl ;
17
18   return 0;
19 }
```

Результат выполнения:

```
y1 = 10
y2 = 15
```

Домашнее задание

Задание 9.1. Предположим, что мы разрабатываем программу для работы с клиентами банков. Нам надо хранить информацию о клиентах (ФИО, остаток средств на счете) и иметь возможность эту информацию изменять (создавать учетную запись, проводить операции со счетом, выводить информацию на экран). Для этого был разработан следующий заголовочный файл bank_account.h:

```
const int Len = 40;
struct bank_account{
  char fullname[Len]; // имя владельца
  double balance;     // деньги на счету
};

void set_bank_account(bank_account &ba, const char *name, double bal);
// Неинтерактивная версия: функция присваивает структуре типа
// bank_account имя владельца и размер счета, используя передаваемые
// ей аргументы

int set_bank_account(bank_account &ba);
// Интерактивная версия: функция предлагает пользователю ввести имя
// и размер счета, присваивает элементам структуры введенные значения;
// возвращает 1, если введено имя, и 0, если введена пустая строка.
// Обратите внимание, что функция set_bank_account() перегружена.

void change_balance(bank_account &ba, double new_balance);
// Функция устанавливает новое значение счета

void show_bank_account(const bank_account &ba);
// Функция отображает содержимое структуры типа bank_account
```

Для создания учётной записи клиента в заголовочном файле заданы прототипы двух функций `set_bank_account()`. Вызов первой функции имеет следующий вид:

```
bank_account ann;
set_bank_account(ann, "Ann Birdfree", 1250);
```

Функция предоставляет информациию для заполнения структуры `ann`. Вызов второй функции имеет следующий вид:

```
bank_account andy;
set_bank_account(andy);
```

Функция предлагает пользователю ввести имя и размер счета, а затем сохраняет эти данные в структуре `andy`.

Постройте многофайловую программу на основе этого заголовочного файла. Один файл по имени `bank_account.cpp` должен содержать определения функций, которые соответствуют прототипам заголовочного файла. Второй файл должен содержать функцию `main()` и демонстрировать работу всех разработанных функций. Например, цикл должен запрашивать ввод массива структур типа `bank_account` и прекращать ввод после заполнения массива, либо когда вместо имени пользователь вводит пустую строку. Чтобы получить доступ к структурам типа `bank_account`, функция `main()` должна использовать только прототипированные функции.