Многофайловые проекты

Создание программы на языке С, состоящей из нескольких файлов

Программа на языке С – это совокупность функций

Запуск любой программы начинается с запуска главной функции, содержащей в себе всю остальную часть программы

Внутри главной функции для реализации заданного алгоритма вызываются все другие необходимые функции

Часть функций создается самим программистом, другая часть – библиотечные функции – поставляется пользователю со средой программирования и используется в процессе разработки программ (например, printf(), sqrt() и др.).

Простейший метод использования нескольких функций требует их размещения в одном и том же файле. Затем выполняется компиляция этого файла, как если бы он содержал единственную функцию

Другие подходы к решению этой проблемы существенно зависят от конкретной операционной системы (Unix-подобные системы, Windows, Macintosh)

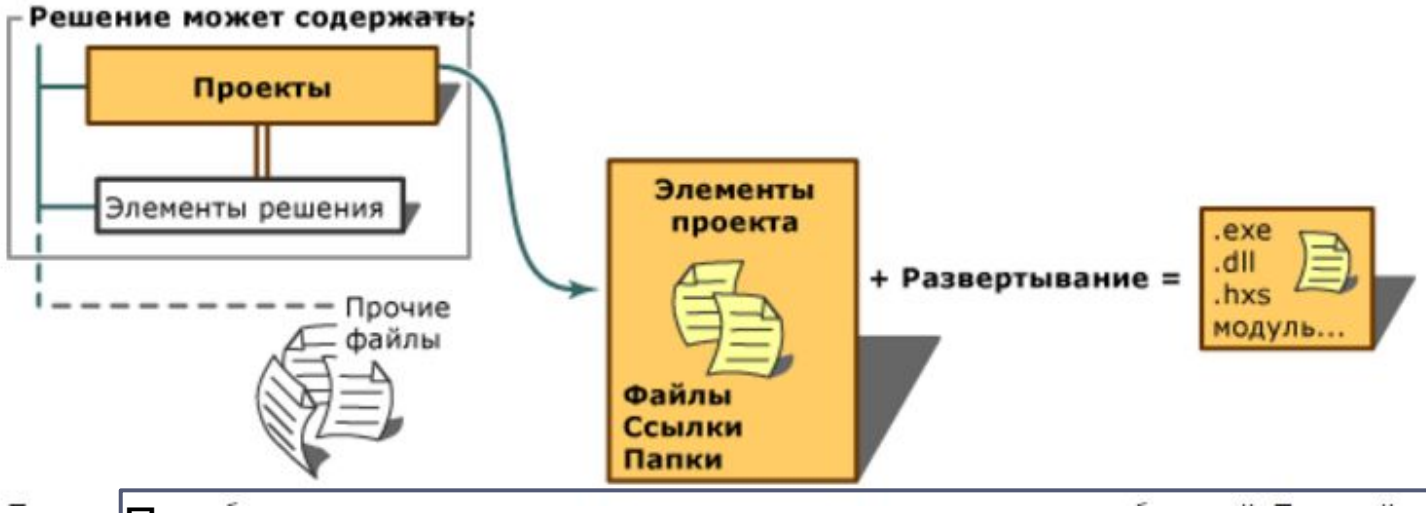
Компиляторы операционных систем Windows и Macintosh представляют собой компиляторы, ориентированные на проекты. Проект описывает ресурсы, используемые программой. Эти ресурсы включают файлы

Создание программы на языке С, состоящей из нескольких файлов

Если поместить главную функцию main() в один файл, а определения собственной функции программиста – во второй файл, то первому файлу нужны прототипы функций. Для этого можно хранить прототипы функций в одном из заголовочных файлов.

Хороший тон в программировании рекомендует размещать прототипы функций и объявлять их константы в заголовочном файле. Назначение отдельных задач отдельным функциям способствует улучшениям программы.

Решения и проекты



Проект

Заголовочный файл – name.h

```
Прототипы функций
Объявление констант
```

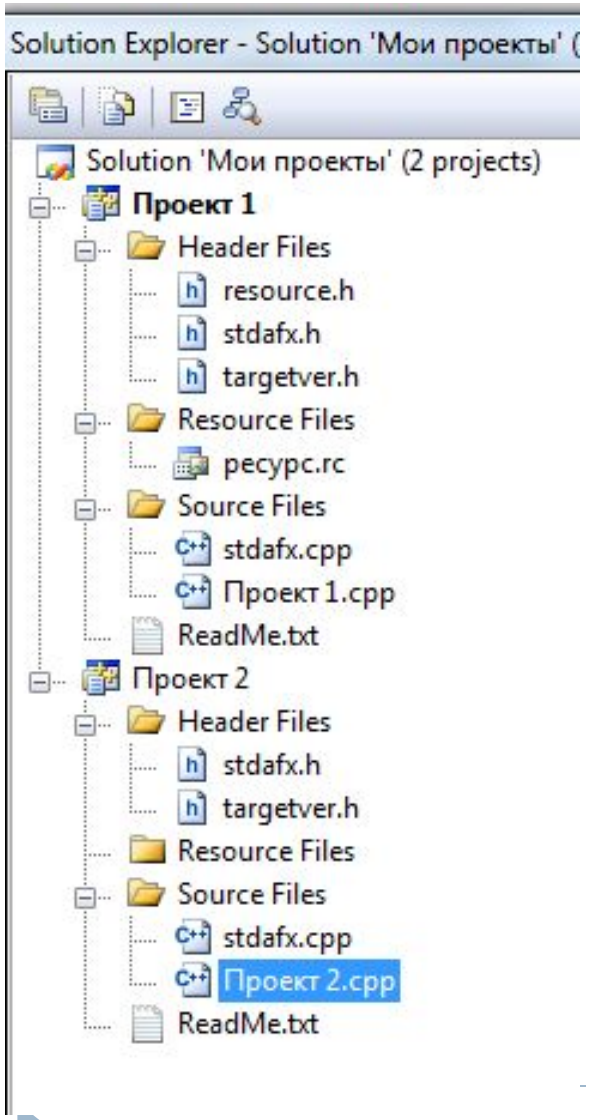
Файл .cpp – name1.cpp

```
int main()
{
...
func1();
func2();
...
}
```

Файл .cpp – name2.cpp

```
void func1()
{
...
}
void func2()
{
...
}
```

Решение, проект, элементы проекта



Решение – контейнер для проектов, чтобы сделать возможным использование в интегрированной среде разработки (IDE) всего диапазона средств, конструкторов, шаблонов и параметров

Папка решения – папка для структурирования связанных проектов по группам и выполнения действий над этими группами проектов

Проект включает набор исходных файлов и связанные метаданные, например ссылки на компонент и инструкции построения:

- ➢ Файлы исходного кода
- ➢ Заголовочные файлы
- ➢ Файлы ресурсов — это элементы интерфейса, предоставляющие информацию пользователю: точечные рисунки, значки, панели инструментов и курсоры

Внешние и статические функции

К внешней функции доступ могут осуществлять функции из других файлов

Статическая функция может использоваться только в файле, в котором она определена.

Например, возможны следующие объявления функций:

```
double gamma(); // внешняя функция по умолчанию
static double beta();
extern double delta();
```

Функция gamma() и delta() могут использоваться функциями из других файлов, которые являются частью программы, тогда как beta() – нет. В силу этого применение функции beta() ограничено одним файлом, поэтому в других файлах можно использовать функции с тем же именем. Одна из причин использования класса статической памяти заключается в необходимости создания функций, приватных для конкретных модулей, благодаря чему во многих случаях удается избежать конфликта имен.

Обычная практика состоит в том, что при объявлении функции, определенной в другом файле, указывается ключевое слово extern. При этом просто достигается большая ясность, поскольку при

Золотое правило для надежного программирования

- Принцип "необходимости знать", или принцип минимально необходимой области видимости
- Рекомендуется держать всю внутреннюю работу каждой функции максимально закрытой по отношению к другим функциям, используя совместно только те переменные, без которых нельзя обойтись по логике программы. Другие классы памяти полезны, и ими можно воспользоваться. Однако всякий раз следует задать вопрос: а есть ли в этом необходимость?

Характеристики памяти, использованной для хранения данных

Продолжительность хранения может быть *статической*, *автоматической* или *распределенной*.

- Если продолжительность хранения статическая, память распределяется в начале выполнения программы и остается занятой на протяжении всего выполнения.
- Если продолжительность хранения автоматическая, то память под переменную выделяется в момент, когда выполнение программы входит в блок, в котором эта переменная определена, и освобождается, когда выполнение программы покидает этот блок.
- Если память выделяется, то она выделяется с помощью функции malloc() (или родственной функции) и освобождается посредством функции free().

Область видимости определяет, какая часть программы может получить доступ к данным. Переменные, определенные вне пределов функции, имеют область видимости в пределах файла и видимы в любой функции, определенной после объявления этой переменной. Переменная, определенная в блоке или как параметр функции, видима только в этом блоке и в любом из блоков, вложенных в этот блок.

Связывание описывает экстент (протяжение, пространство), в пределах которого переменная, определенная в одной части программы, может быть привязана к любой другой части программы. Переменная с областью видимости в пределах блока, будучи локальной, не имеет связывания. Переменная с областью видимости в пределах файла имеет внутреннее или внешнее связывание. Внутреннее связывание означает, что переменная может быть использована в файле, содержащем ее определение.

- Внешнее связывание означает, что переменная может быть использована в других файлах.

Спецификаторы хранения

Стандарт С поддерживает четыре спецификатора класса памяти (хранения):

- extern
- static
- register
- auto

Эти спецификаторы сообщают компилятору, как он должен разместить соответствующие переменные в памяти. Общая форма объявления переменных при этом такова:

спецификатор_класса_памяти тип имя переменой;

Спецификатор класса памяти в объявлении всегда должен стоять первым.

Спецификатор extern

- В языке С при редактировании связей к переменной может применяться одно из трех связываний: *внутреннее*, *внешнее* или *же не относящееся ни к одному из этих типов*.
- В общем случае к именам функций и глобальных переменных применяется внешнее связывание. Это означает, что после компоновки они будут *доступны* во всех файлах, составляющих программу.
- К объектам, объявленным со спецификатором static и видимым на уровне файла, применяется внутреннее связывание, после компоновки они будут доступны только внутри файла, в котором они объявлены. К локальным переменным связывание не применяется и поэтому они доступны только внутри своих блоков.
- Спецификатор extern указывает на то, что к объекту применяется внешнее связывание, именно поэтому они будут доступны во всей программе.
- *Объявление* (декларация) объявляет имя и тип объекта.
- *Определение* (описание, дефиниция) выделяет для объекта участок памяти, где он будет находиться. Один и тот же объект может быть объявлен неоднократно в разных местах, но описан он может быть только один раз.

extern при использовании глобальных переменных:

```
#include <stdio.h>
#include <conio.h>
 // Главная функция
int main (void) {
// объявление глобальных переменных
    extern int a, b;
    printf("\n\t a = %d; b = %d\n", a, b);
printf("\n Press any key: ");
_getch();
return 0;  }
 // инициализация (описание) глобальных переменных
int a = 33, b = 34;
```

Описание глобальных переменных дано за пределами главной функции main(). Если бы их объявление и инициализация встретились перед main(), то в объявлении со спецификатором extern не было бы необходимости.

При компиляции выполняются следующие правила:

- Если компилятор находит переменную, не объявленную внутри блока, он ищет ее объявление во внешних блоках.
- Если не находит ее там, то ищет среди объявлений среди объявлений глобальных переменных.

- Спецификатор extern играет большую роль в программах, состоящих из многих файлов.
- В языке **C** программа может быть записана в нескольких файлах, которые компилируются раздельно, а затем компонуются в одно целое. В этом случае необходимо как-то сообщить всем файлам о глобальных переменных программы. Самый лучший (и наиболее переносимый) способ сделать это – определить (описать) все глобальные переменные в одном файле и объявить их со спецификатором extern в остальных файлах, например, как это сделано в следующей программе:

Первый файл (**main.c**)

```
#include <stdio.h>
#include <conio.h>
#include "D:\second.h”
int x = 99, y = 77; // определение
char ch;
void func1(void);
int main(void){
ch = 'Z';
func1();
printf("\n Press any key: ");_
getch();return 0;}
void func1(void){
func22();
func23();
printf("\n\t x = %d; y = %d;\ ch = %c\n", x, y,
ch);}
```

Второй файл (second.h)

```
extern int x, y; // объявление
extern char ch;

void func22(void){
y = 100;
}

void func23(void)
{
x = y/10;
ch = 'R';
}
```

- В программе первый файл – это основная часть программного проекта.
- Второй файл создан как файл с расширением *.h.
- Список глобальных переменных (**x, y, ch**) копируется из первого файла во второй, а затем добавляется спецификатор extern. Он сообщает компилятору, что имена и типы переменных, следующих далее, объявлены в другом месте. Все ссылки на внешние переменные распознаются в процессе редактирования связей.
- Подключение второго файла выполнено с указанием имени диска (D:), на котором расположен файл **second.h**.
- Для подключения имени файла, созданного пользователем, его заключают в двойные кавычки.

Спецификатор static

- Переменные, объявленные со спецификатором static, хранятся постоянно внутри своей функции или файла. В отличие от глобальных переменных они невидимы за пределами своей функции или файла, но они сохраняют свое значение между вызовами. Эта особенность делает их полезными в общих и библиотечных функциях, которые будут использоваться другими программистами. Спецификатор static воздействует на локальные и глобальные переменные по-разному.

Локальные статические переменные

- Для локальной переменной, описанной со спецификатором static, компилятор выделяет в постоянное пользование участок памяти, точно так же, как и для глобальных переменных. Коренное отличие статических локальных от глобальных переменных заключается в том, что статические локальные переменные видны только внутри блока, в котором они объявлены. Говоря коротко, статические локальные переменные — это локальные переменные, сохраняющие свое значение между вызовами функции.
- Статические локальные переменные очень важны при создании функций, работающих отдельно, так как многие процедуры требуют сохранения некоторых значений между вызовами. Если бы не было статических переменных, вместо них пришлось бы использовать глобальные, подвергая их риску непреднамеренного изменения другими участками программы.

Пример

- Рассмотрим пример функции, в которой особенно уместно применение статической локальной переменной. Это – генератор последовательности чисел, каждое из которых зависит только от предыдущего. Для хранения числа между вызовами можно использовать глобальную переменную. Однако тогда при каждом использовании функции придется объявлять эту переменную и, что особенно неудобно, постоянно следить за тем, чтобы ее объявление не конфликтовало с объявлениями других глобальных переменных.

```
int series(void)
{
  static int series_num;
  series_num = series_num+23;
  return series_num;
}
```

- В этом примере переменная series_num продолжает существовать между вызовами функций, в то время как обычная локальная переменная создается заново при каждом вызове, а затем уничтожается. Поэтому в данном примере каждый вызов series() генерирует новое число, зависящее от предыдущего, причем удается обойтись без глобальных переменных.

Инициализация статической локальной переменной

Статическую локальную переменную можно инициализировать. Это значение присваивается ей только один раз — в начале работы всей программы, но не при каждом входе в блок программы, как обычной локальной переменной. В следующей версии функции series() статическая локальная переменная инициализируется числом 100:

```
int series(void)
{
  static int series_num = 100;
   series_num = series_num+23;
  return series_num;
}
```

Теперь эта функция всегда будет генерировать последовательность, начинающуюся с числа 123. Однако во многих случаях необходимо дать пользователю программы возможность ввести первое число вручную. Для этого переменную series_num можно сделать глобальной и предусмотреть возможность задания начального значения. Если же отказаться от объявления переменной series_num в качестве глобальной, то необходимо ее объявить со спецификатором static.

Глобальные статические переменные

- Спецификатор static в объявлении глобальной переменной заставляет компилятор создать глобальную переменную, видимую только в том файле, в котором она объявлена. Статическая глобальная переменная, таким образом, подвергается внутреннему связыванию. Это значит, что хоть эта переменная и глобальная, тем не менее процедуры в других файлах не увидят ее и не смогут случайно изменить ее значение. Этим снижается риск нежелательных побочных эффектов. А в тех относительно редких случаях, когда для выполнения задачи статическая локальная переменная не подойдет, можно создать небольшой отдельный файл, который содержит только функции, в которых используется эта статическая глобальная переменная. Затем этот файл необходимо откомпилировать отдельно; тогда можно быть уверенным, что побочных эффектов не будет.

Пример

В следующем примере иллюстрируется применение статической глобальной переменной. Здесь генератор последовательности чисел переделан так, что начальное число задается вызовом другой функции, series_start():

```
/* Это должно быть в одном файле  отдельно от всего остального. */
 static int series_num;
void series_start(int seed);
int series(void);
 int series(void) {
  series_num = series_num+23;
  return series_num;
}
 /* инициализирует переменную series_num */
void series_start(int seed) {
  series_num = seed;
}
```

Вызов функции series_start() с некоторым целым числом в качестве параметра инициализирует генератор series(). После этого можно генерировать последовательность чисел путем многократного вызова series().

Спецификатор register

- Первоначально спецификатор класса памяти register применялся только к переменным типа int, char и для указателей. Однако стандарт C расширил использование спецификатора register, теперь он может применяться к переменным любых типов.
- В первых версиях компиляторов C спецификатор register сообщал компилятору, что переменная должна храниться в регистре процессора, а не в оперативной памяти, как все остальные переменные. Это приводит к тому, что операции с переменной register осуществляются намного быстрее, чем с обычными переменными, потому такая переменная уже находится в процессоре и не нужно тратить время на выборку ее значения из оперативной памяти (и на запись в память).
- В настоящее время определение спецификатора register существенно расширено. Стандарты C89 и C99 попросту декларируют "доступ к объекту так быстро, как только возможно". Практически при этом символьные и целые переменные по-прежнему размещаются в регистрах процессора. Конечно, большие объекты (например, массивы) не могут поместиться в регистры процессора, однако компилятор получает указание "позаботиться" о быстродействии операций с ними. В зависимости от конкретной реализации компилятора и операционной системы переменные register обрабатываются по-разному. Иногда спецификатор register попросту игнорируется, а переменная обрабатывается как обычная, однако на практике это бывает редко.

Применение спецификатора

Спецификатор register можно применить только к локальным переменным и формальным параметрам функций. В объявлении глобальных переменных применение спецификатора register не допускается. Ниже приведен пример использования переменной, в объявлении которой применен спецификатор register; эта переменная используется в функции возведения целого числа m в степень.

```
int int_pwr(register int m,  register int e) {
  register int temp;
   temp = 1;
   for(; e; e--) temp = temp * m;
  return temp;
}
```

В этом примере в объявлениях к переменным e, m и temp применен спецификатор register потому, что они используются внутри цикла. Переменные register идеально подходят для оптимизации скорости работы цикла. Как правило, переменные register используются там, где от них больше всего пользы, а именно, когда процесс многократно обращается к одной и той же переменной.

Спецификатор auto

- Спецификатор auto присваивает объявляемым объектам автоматический класс памяти, его можно применять только внутри функции. Объявления со спецификатором auto одновременно являются определениями и резервируют память. Автоматический класс памяти определяет собой автоматический период хранения, который могут иметь только переменные. Локальные переменные функции обычно имеют автоматический период хранения. Ключевое слово auto определяет переменные с автоматическим хранением явным образом.
- Автоматическое хранение способствует экономии памяти, поскольку автоматические переменные существуют только тогда, когда они необходимы. Они создаются при запуске функции, в которой они определены, и уничтожаются, когда происходит выход из нее. Автоматическое хранение является примером реализации принципа минимальных привилегий. Поэтому переменные должны храниться в памяти и быть доступными, когда в данный момент в них нет необходимости. Для переменных со спецификатором auto нет значения по умолчанию.