Программирование с использованием подпрограмм

Препроцессор

Список литературы по курсу

1. Князев А.В. Основы языка C++. Учебное пособие. –М.: Издательство МЭИ, 2013 – 80 с. ISBN 978-5-7046-1425-8.

2. Князев А.В. Работа со сложными структурами данных на языке C++. Учебное пособие. –М.: Издательство МЭИ, 2015 – 48 с. ISBN 978-5-7046-1658-0.

3. Программирование. Сборник задач. Учебное пособие. –Санкт-Петербург: Лань, 2019 – 140 с. ISBN 978-5-8114-3857-0.

URL: https://e.lanbook.com/book/121485

ПОДПРОГРАММЫ

ФУНКЦИИ В C/C++

Встраиваемые функции

- Функцию можно определить со спецификатором *inline*. Такие функции называются *встраиваемыми*.

- Спецификатор *inline* указывает компилятору, что открытая подстановка тела функции предпочтительнее обычной реализации вызова функции и что он должен пытаться каждый раз генерировать в месте вызова код, соответствующий встраиваемой функции, а не создавать отдельно код функции (однократно) и затем вызывать её посредством обычного механизма вызова.

- Спецификатор *inline* не оказывает влияния на смысл вызова функции.

```
inline int max(int x, int y) { return x > y ? x : y; }
```

Открытая подстановка не влияет на результаты вызова функции, чем отличается от макроподстановки.

Встраиваемая функция имеет обычный синтаксис описания функции и подчиняется всем правилам, касающимся области видимости и контроля типов.

Открытая подстановка является просто иной реализацией вызова функции. Вместо генерации кода, передающего управление и параметры единственному экземпляру тела функции, копия тела функции, соответственно модифицированная, помещается на место вызова. Это экономит время для передачи управления.

Для всех функций, кроме самых простейших, время выполнения функций доминирует над издержками времени на обслуживание вызова.

Из этого следует, что для всех, кроме простейших, функций экономия за счёт открытой подстановки минимальна.

Идеальным кандидатом для открытой подстановки является функция, делающая нечто простое, например увеличения или возврата значения.

Параметры функций по умолчанию

- В языке C++ можно задавать так называемые **параметры функции по умолчанию**.
- Если в объявлении формального параметра задано выражение, то оно воспринимается как умолчание этого параметра. Все последующие параметры также должны иметь умолчания. Умолчания параметров подставляются в вызов функции при отсутствии в нём последних по списку параметров.

```
int g(int m = 1, int n);


int h(int m = 1, int n = 2);
int h(int m = 1, int n = 2) { ... }
int h(int m = 0, int n = 0) { ... }


int f(int m = 1, int n = 2);
int f(int m , int n) { ... }
f(5, 6);
f(5);
f();
```

Функции с переменным числом параметров

В языке C++ существует возможность использовать функции с *переменным числом параметров*. Для объявления такой функции надо указать многоточие (,…) в конце списка параметров функции. Для вызова такой функции не требуется никаких специальных действий, просто задается столько параметров, сколько нужно.

Во время интерпретации списка параметров такая функция пользуется информацией, **не доступной** компилятору. Поэтому он не в состоянии гарантировать, что ожидаемые параметры действительно присутствуют или что они имеют правильные типы. Ясно, что если параметр не был объявлен, компилятор не имеет информации, необходимой для выполнения стандартной проверки и преобразований типа.

Функцию только с необъявленными параметрами, в принципе, определить можно, но выбрать параметры будет затруднительно, т.к. макроопределения для работы с необъявленными параметрами используют имя последнего объявленного формального параметра.

Внутри функции программист сам отвечает за выбор из стека дополнительных параметров. Для работы с ними используются макроопределения va_arg, va_start и va_end, определённые в файле stdarg.h.

Пример

Функция с переменным числом параметров, аналогичная функции *printf*

```
#include <cstdio>
 #include <stdarg.h>

 void print(char *format, ...);  //прототип


 int main()
{
 int a = 45, b = 87;
 double f = 2.75;
  print("dfd", a, f, b);
  return 0;
}
```

```
void print(char * format, ...) {
va_list list; // Переменная для работы со списком аргументов
 int n, i; double f;

 va_start(list, format); // Инициализация указателя на список
аргументов

 for (i = 0; format[i]; i++)
    switch(format[i]) {
     case 'd': n = va_arg(list, int); // Выбираем очередной параметр
              printf("%d\n", n); break;
      case 'f': f = va_arg(list, double); // Выбираем очередной параметр
printf("%lf\n", f); break; }

 va_end(list); // Сброс указателя на список аргументов в NULL
 }
```

ПРЕПРОЦЕССОР

ФУНКЦИИ ПРЕПРОЦЕССОРА В C/C++

- Препроцессор – это программа, которая обрабатывает текст вашей
- программы до компилятора.

- Таким образом, на вход компилятора попадает текст, который может
- отличаться от того, который видите Вы.

- Работа препроцессора управляется директивами.

- С помощью препроцессора можно выполнять следующие операции:

- включение в программу текстов из указанных файлов;
- замена идентификаторов последовательностями символов;
- макроподстановка, т.е. замена обозначения параметризованным текстом, формируемым препроцессором с учетом конкретных аргументов;

- исключение из программы отдельных частей текста (условная компиляция).

Включение файлов

Включение файлов производиться с помощью директивы *#include*, которая имеет следующий синтаксис:

#include <*путь*> ИЛИ

#include *"путь"*

Угловые скобки здесь являются элементом синтаксиса.

Директива *#include* включает содержимое файла, путь к которому задан, в компилируемый файл вместо строки с директивой. Если путь заключен в угловые скобки, то поиск файла осуществляется в стандартных директориях. Если путь заключен в кавычки и задан полностью, то поиск файла осуществляется в заданной директории, а если путь полностью не задан – в текущей директории. С помощью этой директивы Вы можете включать в текст программы как стандартные, так и свои файлы.

- Во включаемый файл можно поместить, например, общие для нескольких исходных файлов определения именованных констант и макроопределения. Включаемые файлы используются также для хранения объявлений внешних переменных и абстрактных типов данных, разделяемых несколькими исходными файлами.
- Кроме того, как уже говорилось ранее в курсе, в языке C/C++ ряд функций, такие как функции ввода/вывода, и т.д., не являются элементом языка, а входят в стандартные библиотеки. Для того чтобы пользоваться функциями стандартных библиотек, необходимо в текст программы включать так называемые заголовочные файлы (в описании каждой функции указывается, какой заголовочный файл необходим для неё). Это также делается с помощью директивы препроцессора *#include*.
- Директива *#include* может быть вложенной. Это значит, что она может встретиться в файле, включенном другой директивой *#include*. Допустимый уровень вложенности директив *#include* зависит от реализации компилятора.

Макроподстановки

Макроподстановки реализуются директивой *#define*, которая имеет следующий синтаксис:

#define <*идентификатор*> <*текст*> ИЛИ
#define <*идентификатор*>(<*список параметров*>) <*текст*>

Директива *#define* заменяет все вхождения *идентификатора* в исходном файле на *текст*, следующий в директиве за *идентификатором*. Этот процесс называется макроподстановкой. *Идентификатор* заменяется лишь в том случае, если он представляет собой отдельную лексему. Например, если *идентификатор* является частью строки или более длинного идентификатора, он не заменяется.

Текст представляет собой набор лексем, таких как ключевые слова, константы, идентификаторы или выражение. Один или более пробельных символов должны отделять *текст* от *идентификатора* (или от заключённых в скобки параметров). Если *текст* не умещается на строке, то он может быть продолжен на следующей строке, для этого следует набрать в конце строки символ «обратный слэш» и сразу за ним нажать клавишу «ВВОД».

- *Текст* может быть опущен. В этом случае все экземпляры *идентификатора* будут удалены из исходного текста программы. Тем не менее, сам *идентификатор* рассматривается как определённый.
- *Список параметров*, если он задан, содержит один или более идентификаторов, разделённых запятыми, и должен быть заключён в круглые скобки. Идентификаторы в списке должны отличаться друг от друга. Их область действия ограничена макроопределением, в котором они заданы. Имена формальных параметров в *тексте* отмечают позиции, в которые должны быть подставлены фактические аргументы макровызова.

  В макровызове следом за *идентификатором* записывается в круглых скобах список фактических аргументов, соответствующих формальным параметрам из *списка параметров*. Списки фактически и формальных параметров должны содержать одно и то же количество элементов. Не следует путать подстановку аргументов в макроопределение с передачей аргументов функции. Подстановка в препроцессоре носит чисто текстовый характер. Никаких вычислений или преобразований типа при этом не производится.

- После того как выполнена макроподстановка, полученная строка вновь просматривается для поиска других имен макроопределений. При повторном просмотре не принимается к рассмотрению имя ранее произведенной макроподстановки. Поэтому директива

#define x x

не приведет к зацикливанию препроцессора.

Примеры

- #define N 100
- #define MULT(a, b) ((a) * (b))

  При отсутствии внутренних скобок получилось бы (x + y * z), что неверно.

- Символ #, помещаемый перед аргументом макроопределения, указывает на необходимость преобразования его в символьную строку. При макровызове конструкция

  #<*формальный параметр*> заменяется на "<*фактический параметр*>".
- Замены в тексте можно отменить директивой ***#undef***, которая имеет следующий синтаксис:

•

#undef <*идентификатор*>

- Директива *#undef* отменяет действие текущего определения *#define* для *идентификатора*. Чтобы отменить макроопределение, достаточно задать его *идентификатор*. Задание списка параметров не требуется. Не является ошибкой применение директивы *#undef* к идентификатору, который ранее не был определён или действие которого уже отменено.

Условная компиляция

- Условная компиляция обеспечивается в языке C/C++ набором команд, которые, по существу, управляют не компиляцией, а препроцессорной обработкой. Эти директивы позволяют исключить из процесса компиляции какие-либо части исходного файла посредством проверки условий.

- #if <*ограниченное константное выражение*> [<*текст*>]
  [#elif <*ограниченное константное выражение*> [<*текст*>]] ...
  [#else [<*текст*>]]
  #endif

- Каждой директиве *#if* в том же исходном файле должна соответствовать завершающая её директива *#endif.* Между директивами *#if* и *#endif* допускается произвольное количество директив *#elif* и не более одной директивы *#else*. Если директива *#else* присутствует, то между ней и директивой *#endif* на данном уровне вложенности не должно быть других директив *#elif*.

- Препроцессор выбирает участок текста для обработки на основе вычисления *константного выражения*, следующего за каждой директивой *#if* и *#elif*. Выбирается *текст*, следующий за *константным выражением* со значением «истина». Если ни одно ограниченное константное выражение не истинно, то препроцессор выбирает *текст*, следующий за директивой *#else*. Если же директива *#else* отсутствует, то никакой текст не выбирается.

- *Константное выражение* может содержать препроцессорную операцию defined(<*идентификатор*>). Эта операция возвращает истинное значение, если заданный *идентификатор* в данный момент определён, в противном случае выражение ложно.

```
#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif
```

Многофайловые программы

Пусть в одном файле определена пара функций, а в другом, содержащем функцию main(), осуществляется их вызов.

Файл main.cpp

```
#include <stdio.h>
#define N 5
 int main () {
 char strs[N][10];
 char *p[N];
   int i;
   for(i=0; i<N; i++)
   {
      scanf("%s", strs[i]); p[i] = &strs[i][0];
    }
   l2r(p, N);
 r2l(p, N);
 return 0; }
```

Файл superint.cpp

```
#include <stdio.h>
   void l2r (char **c, int n)
 {
    int i, j;
    for(i=0; i<n; i++, c++) {
     for (j=0; j<i; j++)
        printf("\t");
        printf ("%s\n",*c);
      }
 }

   void r2l (char **c, int n) {
 int j;
   for(; n>0; n--, c++) {
    for (j=1; j<n; j++)
      printf("\t");
      printf ("%s\n",*c);
     }
 }
```

- Каким образом в представленной выше программе функция main() "узнает" о существовании функций l2r() и r2l() ? Ведь в исходном коде файла main.cpp нигде не указано, что мы подключаем файл superprint.cpp, содержащий эти функции. Действительно, если попытаться получить из main.cpp отдельный исполняемый файл, т.е. скомпилировать программу без superprint.cpp:

cl main.cpp, то ничего не получится. Компилятор сообщит об ошибке вызова неопределенных идентификаторов. Получить из файла superprint.c отдельный исполняемый файл вообще невозможно, т.к. там отсутствует функция main().

- А вот получить из этих файлов отдельные объектные файлы можно. Представим, что одни объектные файлы как бы "выставляют наружу" имена определенных в них функций и глобальных переменных, а другие - вызовы этих имен из тел других функций. Дальше объектные файлы "ожидают", что имена будут связаны с их вызовами. Связывание происходит при компиляции исполняемого файла из объектных.

В интегрированных средах разработки, к которым относится Microsoft Visual Studio, все файлы проекта «объединяются». Они компилируются, создаются объектные модули. Затем модули объединяются с помощью линкера (link) и создаётся 1 выполняемый файл (.exe)

Для этого необходимо иметь в составе функций одну единственную функцию с именем main

Особенности

-использование прототипов функций

- Глобальные переменные программы с одинаковым именем не должны объявляться в разных файлах!