Основы программирования на языке C++

Преподаватель:
Маслов Алексей Владимирович,
доцент кафедры общей физики

Содержание

Функции и структура программы, шаблоны функций

Обзор функций

Программа начинается с вызова функции main().

Все остальные функции вызываются из main() или из других функций.

Чтобы использовать функцию (подпрограмму) в C++, вы должны выполнить следующие шаги:

- Определить (написать) функцию;
- Предоставить прототип функции;
- Вызвать функцию.

Использование функции simple() в main():

```
#include <iostream>       // заголовочный (header) файл

void simple();            // прототип функции,
// void simple(void);     // можно и так

int main(void) {
      using namespace std;
      cout << "main() will call the function simple(): \n";
      simple();                  // вызов функции
      cout << "main() is finished with the function simple().\n";
      return 0;
}

void simple () {  // Определение функции
      using namespace std;
      cout << "function simple()\n";
}
```

Функции, не возвращающие значений

Функции типа void

```
void function_name(parameter_list) {
      statement(s);
      return; // не обязательно, бывает удобно для прерывания циклов
}

void cheers(int n) { // возвращаемое значение отсутствует
    for (int i = 0;  i < n;  i++)
         cout <<  "Hello!  ";  // печатаем в цикле
    cout << endl;
}
```

Функции с возвращаемым значением

```
type_name function_name(parameter_list){
    statement(s);
    return value; // приводится к типу type_name
}
```

Возвращаемое значение не может быть массивом, но может быть указателем.

```
int bigger (int a, int b) {
     if (a > b )
        return a;       // если a > b, функция завершается здесь
     else               // можно и без else
        return b;       // в противном случае функция завершается здесь
}
```

Зачем нужны прототипы?

Прототип сообщает компилятору, каков тип возвращаемого значения, если оно есть у функции, а также количество и типы аргументов данной функции.

В прототипе можно указывать или не указывать имена переменных в списке аргументов.

```
double cube(double x) ;  // имя x значения не имеет
void cheers(int);        // только тип
```

Во время компиляции файла проверяется соответствие вызова функции и ее прототипа, который должен быть указан до этого. Такая проверка называется статическим контролем типов. Если определение функции дано до ее вызова, то этого достаточно для проверки и прототип не нужен.

В многофайловых программах прототипы помещаются в заголовочные файлы и затем вставляются с помощью команды препроцессора #include.

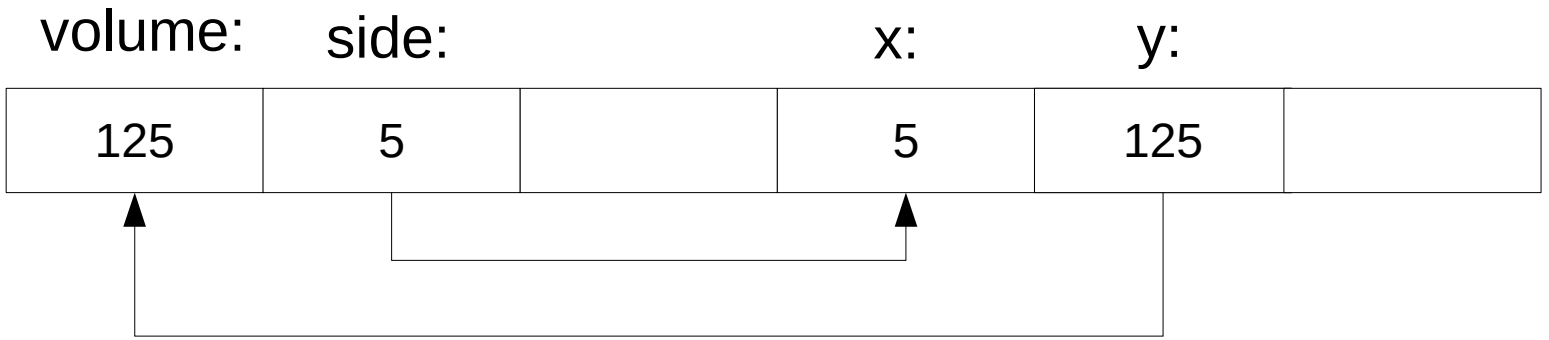
Аргументы функций

В C и C++ аргументы обычно передаются по значению, т.е. числовое значение аргумента передается в функцию, где присваивается новой переменной.

```
#include <iostream>
double cube(double x);  // прототип
int main(void) {
    int side = 5;
    double volume = cube(side); //вызов
    …
}
double cube(double x) {
    double y= x * x * x;
    return y;
}
```

Переменные, включая аргументы, объявленные в функции, являются локальными (local) переменными, так как они локализованы по отношению к этой функции, то есть их область видимости ограничена данной функцией.

Когда функция вызывается, процессор выделяет память, необходимую для этих переменных. Когда функция завершается, процессор освобождает память, которая была использована этими переменными.

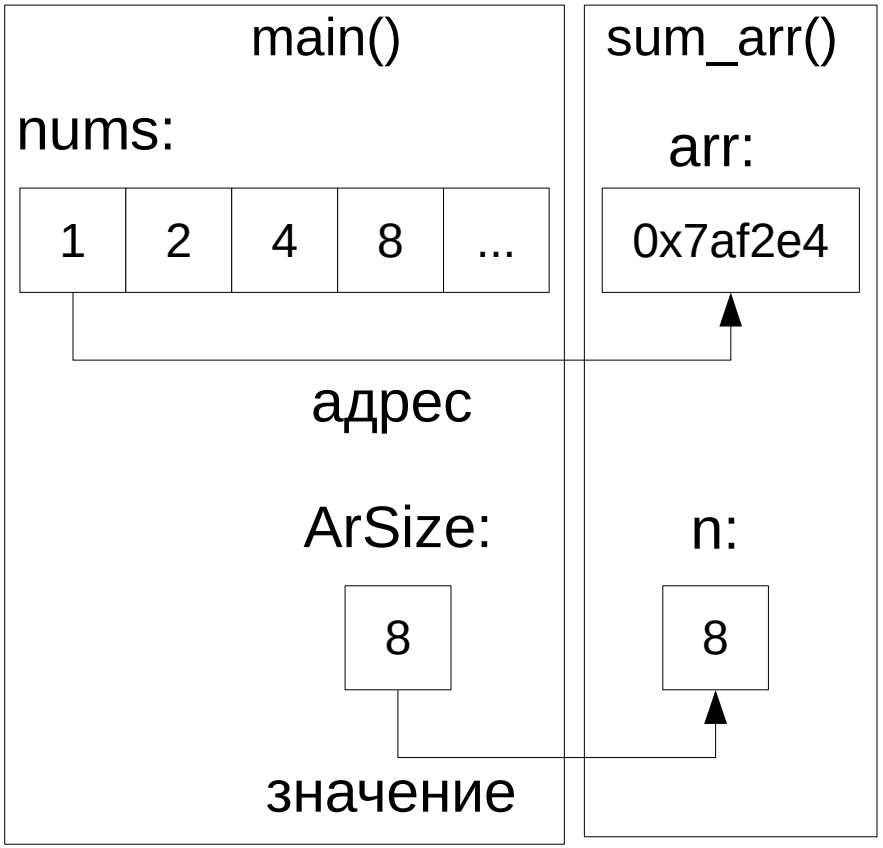


Функции и массивы

C++/C в большинстве контекстов трактует имя массива как указатель.

```
#include <iostream>
const int ArSize = 6;
int sum_arr(int *arr, int n) ; // прототип

int main(void) {
    using namespace std;
    int nums[ArSize] = {1, 2, 4, 8, 16, 32};
    int sum = sum_arr(nums, ArSize);
    cout << "Sum of all elements = "
         << sum << "\n";
    return 0;
}
```




```
int sum_arr(int *arr, int n) { // можно int arr[], нужно передать размер
    int total = 0;  // нужно инициализировать
    for (int i = 0; i < n; i++)
        total +=  arr[i];  // total += *arr++;
    return total;
}
```

nums эквивалентно &nums[0], так как имя массива – адрес его первого элемента

Правильный заголовок функции должен быть таким:

```
int sum_arr(int *arr, int n) // arr = имя массива, n = размер
```

При вызове функции происходит копирование адреса массива в локальную переменную-указатель arr.

В C и C++ нотации int *arr и int arr[ ] имеют идентичный смысл, когда (и только когда) применяются в заголовке или прототипе функции.

```
arr[i]     эквивалентно *(arr + i) // значения
&arr[i]  эквивалентно    arr+i    // адреса
```

Доступ к элементам массива

Поскольку функция с аргументом – именем массива получает доступ к исходному массиву, а не к копии, можно использовать вызов функции для изменения значений его элементов.

Если назначение функции не предусматривает внесения изменений в переданные ей данные, нужно предохранить её от этого, применяя ключевое слово const:

```
void sum_array(const double arr[], int n) ;
void sum_array(const double *arr, int n) ;

arr[0] += 1.0; // компилятор выдаст ошибку, данные нельзя менять
arr++;         // сам указатель можно менять
```

Указатели и const

```
int age = 21;
const int *pt = &age; // нельзя будет менять значение через указатель
```

Это объявление устанавливает, что pt указывает на const int (т.е., 21).

Другими словами, значение *pt является константным и не может быть изменено:

```
*pt += 1;      // неправильно, потому что pt указывает на const int
cin >> *pt;   // неправильно по той же причине
pt++;          // можно, но не нужно, так как age – одно число
```

Однако можно изменить значение age непосредственно

```
*pt = 20;    // неправильно, потому что pt указывает на const int
age = 20;   // правильно, потому что age не объявлено как const
```

```
int * const pt2 = &age; // постоянный указатель, требует инициализации
*pt2 = 18;  // можно
pt2++;      // нельзя менять значение указателя
```

Функции и структуры

```
/***** файл time.hpp *****/
struct time_hm{int hours, mins; }; // структура из двух элементов: часы, минуты

extern const int Mins_per_hour;                    // внешняя глобальная переменная
time_hm add_time1(time_hm t1, time_hm t2); // нахождение суммы
void show_time(time_hm t) ;                        // вывод на экран
```

```
/***** файл time.cpp *****/
#include <iostream>// для вывода на экран
#include "time.hpp" // все декларации
const int Mins_per_hour = 60; // глобальная переменная

time_hm add_time1(time_hm t1, time_hm t2) { // нахождение суммы
    int tot_mins = t1.mins  + t2.mins; // полное кол-во минут
    int hours3   = t1.hours + t2.hours + tot_mins/Mins_per_hour;
    int mins3    = tot_mins % Mins_per_hour;
    time_hm  t3 = {.hours=hours3, .mins=mins3};  // или {hours3, mins3}
    return t3;
}
void show_time(time_hm t) {// вывод на экран
  std::cout << t.hours << ":" << t.mins  << "\n";
}
```

Функции и структуры (2)

Передача структур по значению, то есть копирование всей структуры в локальную переменную-структуру.

```
/****** файл run_time.cpp ******/
#include <iostream> // для вывода на экран
#include "time.hpp" // все декларации

int main (void) {
      using namespace std;
      time_hm t1 = {5, 45}; // создание структуры
      time_hm t2 = {4, 55}; // создание структуры
      time_hm r1 = add_time1(t1, t2); // аргументы копируются !
      cout << "r1= ";
      show_time (r1) ;
…
}
```

Передача адресов структур

Для экономии времени и памяти можно передавать адрес структуры (или ссылки, см. дальше тему «Ссылочные переменные») вместо самой структуры.

Аргументы с указателями на структуры: при вызове копируется адрес структуры в локальную переменную-указатель.

```
/****** файл time.cpp ******/
.....
void add_time2(const time_hm *pt1, const time_hm *pt2, time_hm *pt3) {
  int  tot_mins  = pt1->mins + pt2->mins;
  pt3->hours = pt1->hours + pt2->hours + tot_mins / Mins_per_hr;
  pt3->mins =  tot_mins % Mins_per_hr;
}
time_hm *add_time3(const time_hm *pt1, const time_hm *pt2) {
  int  tot_mins  = pt1->mins + pt2->mins;

  time_hm  *pt3 =  new  time_hm; //  выделили память для структуры
  pt3->hours = pt1->hours + pt2->hours + tot_mins / Mins_per_hr;
  pt3->mins =  tot_mins % Mins_per_hr;
  return pt3; // вернули указатель на выделенную память
}
void show_time2(const time_hm *pt) { std::cout << pt->hours << ":" << pt->mins  << "\n";}
 .....
```

Передача адресов структур (2)

```
/****** файл run_time.cpp *****/
#include <iostream> // для вывода на экран
#include "time.hpp" // все декларации

int main (void) {
      using namespace std;
      time_hm t1 = {5, 45}; // создание структуры
      time_hm t2 = {4, 55}; // создание структуры
      time_hm r1 = add_time1(t1, t2); // аргументы копируются !

      time_hm r2;
      add_time2(&t1, &t2, &r2); // передаются адреса

      time_hm *r3=add_time3(&t1, &t2); // передаются адреса
      cout << "*r3=: ";  show_time (*r3) ; // передаем значение
      cout << "*r3=: ";  show_time2 (r3) ; // передаем адрес
      ......
      delete r3; // обязательно освободить память !
      .....
}
```

Рекурсия

Функции C и C++ могут вызывать сами себя.

```
void recurs(списокАргументов) {
    операторы1;
    if (проверка)
       recurs(аргументы);
    операторы2;
}
```

$$n! = 1 \cdot 2 \cdot 3 \ldots \cdot n$$

$$n! = n \cdot (n-1)!$$

$$0! = \frac{1!}{1} = 1$$

Вычисление факториала:

```
#include<stdio.h>
int fact(int n){
    if (n == 0) {
      return 1;
    }
  return n * fact(n-1);
}

int main(void) {
  int n=5;
  int ff=fact(n);
  printf("ff=%d\n", ff);
  return 0;
}
```

```
ff=120
```

Указатели на функции

Функции, как и данные, имеют адреса. Эти адреса хранят в указателях.

Зачем нужны указатели на функции?

1) Пусть есть две (или больше) взаимозаменяемые функции, которые можно использовать в большой программе. Можно в программе использовать указатель, который инициализируется нужной функцией.

2) Пусть имеется алгоритм, который работает с различными функциями (интегрирование, дифференцирование, нахождение экстремума и т.д.). Тогда функция, реализующая этот алгоритм, должна принимать указатель на функцию.

Декларирование и использование указателей на функции

```
double func1(int); // прототип функции
```

Объявление соответствующего типа указателя:

```
double (*pf)(int);  // pf -- указатель на функцию, которая принимает один
                    // аргумент типа int и возвращает тип double
double *pf2(int);    // pf2() – функция, возвращающая указатель на double

pf = func1;          // pf теперь указывает на функцию func1()
pf = &func1;       //  так тоже можно
double x = func1(4); // вызвать func1(), используя ее имя
double y = (*pf)(5); // вызвать func1(), используя указатель pf
double z = pf(5); // также вызывает func1(),  используя указатель pf

double (*my_sqrt) (double)=sqrt; //указатель на библиотечную функцию sqrt()
cout << my_sqrt(4.0) << endl;
```

Указатели на функции в аргументах

Если есть функция func1(...), то ее адрес записывается как func1.

Чтобы передать функцию в качестве аргумента, нужно передать ее имя, как и для массивов.

```
double func1(double x);    // прототип функции


// прототип фукции, у которой аргумент – указатель на функцию, которая возвращает double и берет double. Имя указателя -- fname
double integrate(double a, double b, double (*fname)(double x));


// внутри integrate() используем указатель fname
double y=fname(x); // получаем значение функции в некоторой точке x


// вызов фунциии integrate()
double a=0.0, b=1.0;
integrate(a, b, func1);
```

Встроенные функции C/C++

Встроенные функции (inline functions) являются усовершенствованием языка C++, предназначенным для ускорения работы программ.

Обычные функции: Когда вызывается обычная функция, то программа сохраняет адрес операции, которая записана сразу после вызова функции, загружает вызываемую функцию в память, копирует аргументы, переходит по адресу нахождения функции, выполняет функцию, сохраняет возвращаемое значение и переходит к адресу следующей операции. Это требует времени.

Встроенные функции: Компилятор вставляет (по своему усмотрению) определение функции в программу во время компилирования. Это ведет к увеличению размера выполняемого файла, но дает выигрыш в скорости.

Чтобы воспользоваться встроенной функцией, нужно выполнить хотя бы одно из следующих действий.

- Предварить объявление функции ключевым словом inline.
- Предварить определение функции ключевым словом inline.

Встроенные функции C/C++ (2)

Определение функции должно находится в том же файле, где происходит вызов функции. Общепринято опускать прототип и помещать полное описание (заголовок и весь код функции) туда, где обычно находится прототип.

Использование слова inline не ограничивает видимость этой функции из других файлов (если нет static). Функция остается глобальной.

Если определение не помещается в одной или двух строках, то такая функция, скорее всего, является плохим кандидатом на то, чтобы быть встроенной.

Опция компилятора -Winline может дать сообщение, если компилятор не сделал встраивание функции, объявленной inline.

```
inline double square (double x) { return x * x; } // определение
int main (void) {
      double a, b;
      a = square(5.0);
      b = square(2.0+3.5+9.0);
…..
}
```

Ссылочные переменные (reference variables)

Ссылка (reference) представляет собой имя, которое является псевдонимом (alias), или альтернативным именем (синонимом), для ранее объявленной переменной. Ссылка не является независимой переменной, то есть новой памяти не выделяется. Ссылка и изначальное имя переменной обеспечивают доступ к одной и той же ячейке памяти.

```
int alpha=3;  // обычная переменная

int &beta = alpha; // задали ссылку с именем beta. Имена beta и alpha

// становятся эквивалентными и относятся к одной и той же ячейке памяти.

// Ссылки должны инициализироваться при декларировании!

int gamma=4;

beta = gamma; // обычное присваивание

cout << alpha; // выведет 4, так как alpha и бета эквивалентны
```

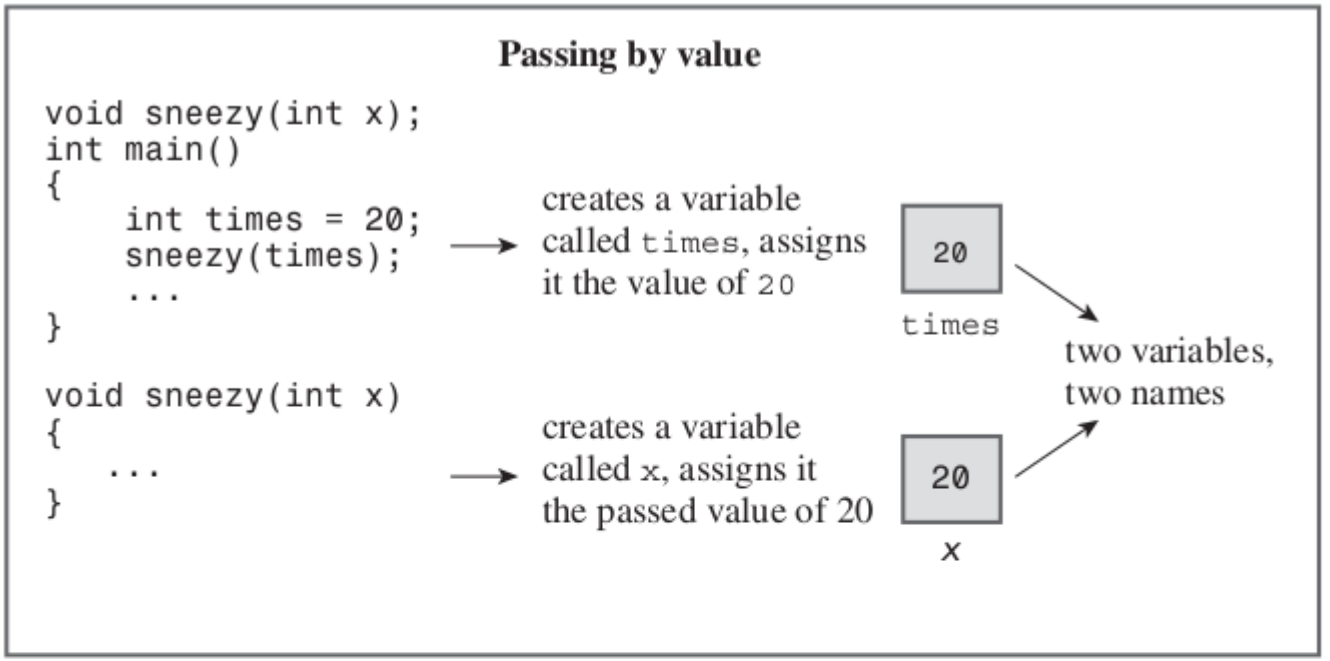
alpha, beta:

Не путать & с операцией получения адреса:

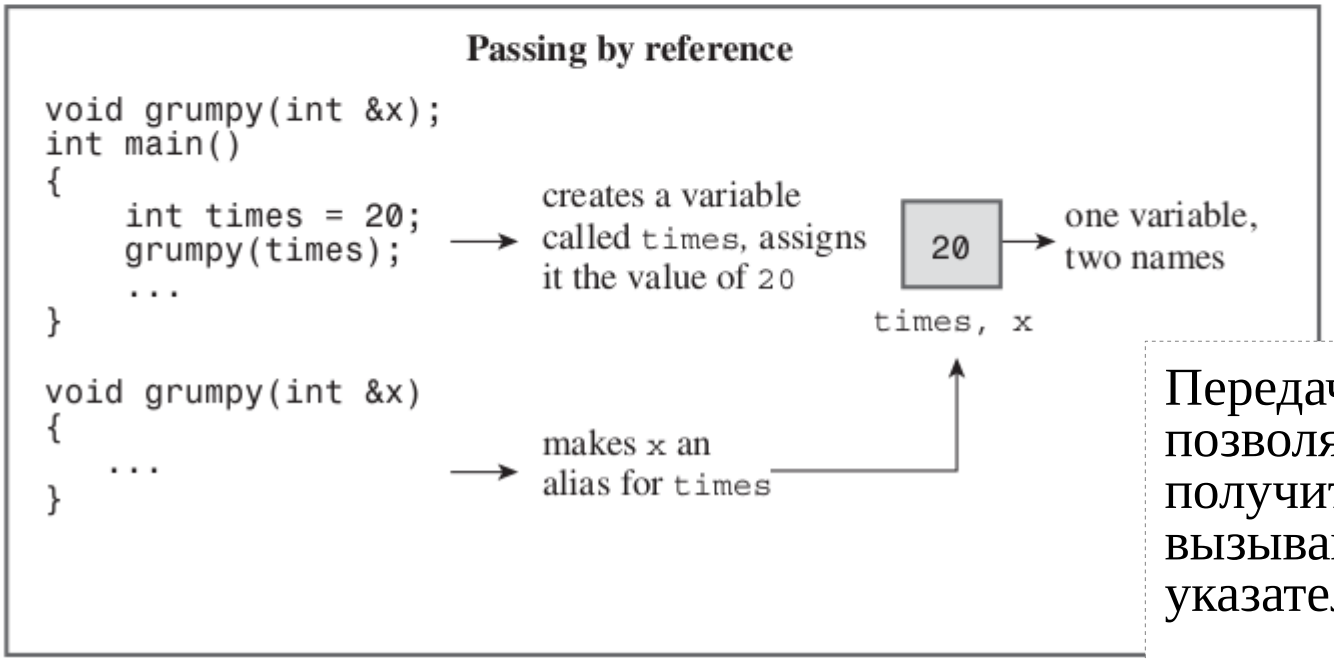
```
int *p = &alpha;  // p - указатель
```

Ссылки в основном используются в аргументах функций и в качестве возвращаемых значений. По функциональности они похожи на указатели.

Ссылки как параметры функций



Передача
по значению



Передача по ссылке.

Здесь x в grumpy() — ссылка на переменную times из main().

Передача параметров по ссылке позволяет вызываемой функции получить доступ к переменным в вызывающей функции (как и с указателями).

Использование ссылок при работе со структурами

```
/****** файл time.cpp ******/
....
void add_time4(const time_hm &t1, const time_hm &t2, time_hm &t3) {
  // передаем ссылки  в аргументе
  int  tot_mins  = t1.mins + t2.mins;
  t3.hours = t1.hours + t2.hours + tot_mins / Mins_per_hr;
  t3.mins =  tot_mins % Mins_per_hr;
}

time_hm &add_time5(const time_hm &t1, const time_hm &t2) {
  // пытаемся вернуть ссылку
  int  tot_mins  = t1.mins + t2.mins;
  time_hm  t3; //  объявили структуру
  t3.hours = t1.hours + t2.hours + tot_mins / Mins_per_hr;
  t3.mins =  tot_mins % Mins_per_hr;
  return t3; // нельзя возвращать ссылку на локальную переменную !!!
}
```

Использование ссылок при работе со структурами (2)

```
/****** файл run_time.cpp *****/
#include <iostream> // для вывода на экран
#include "time.hpp" // все декларации

int main (void) {
      using namespace std;
      time_hm t1 = {5, 45}; // создание структуры
      time_hm t2 = {4, 55}; // создание структуры

      time_hm r4;
      add_time4(t1, t2, r4); // передаются ссылки
      cout << "r4= "; show_time (r4) ;

      time_hm t5=add_time5(t1, t2); // Segmentation fault


      .....
}
```

Зачем возвращать ссылку?

```
double m = sqrt(16.0) ;
cout <<  sqrt (25.0) ;
```

В первом операторе возвращаемое значение 4 .0 копируется во временную ячейку, после чего из этой ячейки копируется в m. Во втором операторе значение 5.0 копируется во временную ячейку и затем из этой ячейки передаётся в cout.

```
p3 = add(p1, p2);
```

Если add() будет возвращать структуру вместо ссылки на структуру, это может повлечь за собой копирование целой структуры во временную ячейку и последующее копирование этой копии в p3. Но благодаря ссылочному возвращаемому значению, p1 копируется напрямую в p3, что является более эффективным подходом.

Выбор объекта, на который указывает возвращаемая ссылка:

Объект, на который возвращается ссылка, должен существовать.

То есть, это не должен быть объект, область существования которого ограничена функцией. Использование таких объектов приводит к ошибке.

Проще всего избежать такой ошибки за счет

1) возврата ссылки, которая была передана функции в качестве аргумента.

2) использовании операции new для создания нового хранилища.

Использование ссылок при работе со структурами (3)

```
/***** файл time.cpp ******/
.....
time_hm &add_time6(time_hm &t1, const time_hm &t2) {// добавляет t2 к t1
  int tot_mins = t1.mins + t2.mins;

  t1.mins =  tot_mins % Mins_per_hr;
  t1.hours += t2.hours + tot_mins / Mins_per_hr;

  return t1; // возвращаем ссылку на существующую переменную
}

time_hm &add_time7(const time_hm &t1, const time_hm &t2) {
  int tot_mins  = t1.mins + t2.mins;

  time_hm *pt3  = new time_hm; // выделили память
  pt3->mins =  tot_mins % Mins_per_hr;
  pt3->hours = t1.hours + t2.hours + tot_mins / Mins_per_hr;

  return *pt3; // вернули ссылку на выделенную память
}
```

Использование ссылок при работе со структурами (4)

```
/****** файл run_time.cpp ******/
....
int main (void) {
      using namespace std;
      time_hm t1 = {5, 45}, t2 = {4, 55}; // создание структур

      time_hm &r6=add_time6(t1, t2);    // r6 становится ссылкой на t1
      cout << "r6= "; show_time (r6) ;

      time_hm r6n=add_time6(t1, t2);      // скопировали t1 в r6n

      add_time6(t1, t2);          // добавили t2 к t1

      add_time6(t1, t2) = t2;  // записали значение t2 в t1 !

      time_hm &r7=add_time7(t1, t2); // r7 - ссылка на выделенную память
      cout << "r7= "; show_time (r7) ;
      delete &r7;  // освободили память
      .....
}
```

Использование указателей

```
/***** файл time.cpp *****/

time_hm *add_time8(time_hm *pt1, const time_hm *pt2) {// добавляет t2 к t1
  int tot_mins = pt1->mins + pt2->mins;

  pt1->mins =  tot_mins % Mins_per_hr;
  pt1->hours += pt2->hours + tot_mins / Mins_per_hr;

  return pt1; // возвращаем указатель
}
```

```
/****** файл run_time.cpp ******/
....
int main (void) {
      using namespace std;
      time_hm t1 = {5, 45}, t2 = {4, 55}; // создание структур

      *add_time8(t1, t2) = t2;    // записали t2 по возвращенному адресу
      cout << "t1= "; show_time (t1) ;

      *add_time3(t1, t2) = t2; // потеряли указатель на выделенную память
}
```

Выводы по типам аргументов

- По аналогии с примерами на основе структур можно работать и с другими типами данных: с базовыми (int, double и т.д) и с объектами (например, string).
- Использование указателей и ссылок позволяет значительно расширить возможности функций (например, избегать копирования значений).
- Если возвращается ссылка, то она должна ссылаться на существующую переменную или выделенную память.
- Если возвращается указатель, то он должен указывать на существующую переменную или выделенную память.
- Если в функции выделяется память, но не освобождается, то функция должна каким-либо образом вернуть указатель или ссылку на эту память, чтобы ее можно было освободить в дальнейшем.

Аргументы по умолчанию (default arguments)

Аргумент по умолчанию -- это значение, которое используется, если соответствующий параметр в вызове функции не указан. Значения по умолчанию, как правило, задаются в прототипе. Определение функции при этом остаётся без изменений.

```
void print_str_n(const char *str, int n = 1) ; // прототип виден до вызова
int main(void){
  const char  *str1 = "hello";
  print_str_n ("hello", 2) ;   // печатает 2 символа
  print_str_n (str1, 10) ; // печатает 5 символов
  print_str_n (str1, 1) ;   // печатает 1 символ
  print_str_n (str1) ;       // тоже печатает 1 символ
  print_str_n () ;           // а так нельзя
  return 0;
}
void print_str_n(const char *str, int n) { //определение функции
   // функция печатает n символов из строки, но не больше ее длины
   for(int  i=0; i<n && str[i]!=0; i++) // не должны выходить за пределы строки
      cout << str[i]; // печать символа
   cout << endl;
}
```

Аргументы по умолчанию (2)

Аргументы по умолчанию должны добавляться в конце списка:

```
int func_1(int n, int m = 4, int j = 5) ; // ПРАВИЛЬНО

int func_2(int n, int m = 6, int j);       // НЕПРАВИЛЬНО

int res;
res = func_1(2);      // то же, что и func1 (2, 4, 5)
res = func_1(1, 8) ; // то же, что и func1(1, 8, 5)
```

Перегрузка функций

Аргументы по умолчанию → Вызов функции с различным количеством аргументов

Полиморфизм (перегрузка функций) → Использование нескольких функций с одним именем

Перегрузка функций позволяет разработать семейство функций, которые выполняют похожие действия, но с использованием ***различных списков аргументов***.

Компилятор выбирает нужную функцию из нескольких функций с одним именем ***на основании типа передаваемых аргументов***. Компилятор выдаст ошибку, если никакая функция не подошла или выбор неоднозначен.

Примеры перегрузки функций

```
// пусть имеется несколько функций print
void print(const char *str, int width);   // #1
void print(double d, int width);          // #2
void print(long a, int width);            // #3
void print(int i, int width);             // #4
void print(const char *str);              // #5

print("Pancakes", 15);          // используется #1
print("Syrup");                 // используется #5
print(1999.0, 10);              // используется #2
print(1999, 12);                // используется #4
print(1999L, 15);               // используется #3

unsigned int year =3210;
print(year, 6); // неоднозначный вызов, может быть  #2, #3, #4
                // компиляция не пройдёт
print((int) year, 6); // #4
```

При проверке сигнатур функций (function signature or functions's argument list, то есть список аргументов функции) компилятор считает, что ссылка на тип и сам тип равнозначны.

Такая перегрузка неправильна, так как невозможно будет выбрать:

```
double cube(double  x) ;     // вызывается как y=cube(x);
double cube(double &x) ; // тоже вызывается как y=cube(x);
```

Учитываются различия между переменными с квалификатором const и без него.

```
void dribble(char *bits);           // перегружена
void dribble (const char *cbits); // перегружена

long     gronk(int n, float m); // одинаковые сигнатуры  (аргументы),
double gronk(int n, float m); // объявления не допускаются. Различия типов
                                          // возвращаемых значений не играют роли!

long     gronk(int n,    float m) ; // различные сигнатуры,
double gronk(float n, float m) ; // объявления допустимы
```

Шаблоны функций

Шаблоны функций (function templates)

Шаблон функции – это обобщённое описание функции (алгоритм).

Шаблон определяет функцию с использованием обобщённого типа данных, вместо которого может быть подставлен определённый тип, такой как int или double.

Компилятор создает определение функции (экземпляр шаблона) на основании типа данных, передаваемых шаблону в качестве параметров.

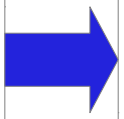
Использование шаблонов иногда называют обобщённым программированием.

Поскольку типы представлены параметрами, шаблоны иногда называют параметризованными типами.

Определение шаблона функции

```
// перегруженные функции
void Swap(int &a, int &b){
      int temp = a;
       a = b;
       b = temp;
}

void Swap(double &a, double &b) {
      double temp = a;
       a = b;
       b = temp;
}
```



```
template <typename AnyType>
void Swap(AnyType &a, AnyType &b) {
      AnyType temp = a;
      a = b;
      b = temp;
}
```

Первая строка указывает, что устанавливается шаблон, а произвольный тип данных получает имя AnyType. Ключевые слова template и typename являются обязательными; при этом вместо typename можно использовать ключевое слово class (старый вариант, как до C++98).

Должны присутствовать угловые скобки. Имя типа может быть любым (в этом примере - AnyType), часто используют простые имена, такие как T.

Использование шаблона функции

```
#include <iostream>

template <typename T>  // или class T, прототип шаблона функции
void Swap(T &a, T &b) ;

int main(void) {
      int i = 10,  j = 20;
      Swap(i, j); // генерирует void Swap(int &, int &)

      double x = 24.5,  y= 81.7;
      Swap(x, y); // генерирует void Swap(double &, double &)

      // Swap(i, y);  - error:  no matching function for call

      return 0;
}
```

Альтернативы шаблонам

```
// Функция Swap для произвольного типа данных. Использует размер данных.
void Swap(void *a, void *b, size_t s){
 // void * - указатель на неопределённый (произвольный) тип данных.
 //  Тип size_t имеет достаточный размер для адреса произвольной ячейки памяти
 // sizeof(size_t) = sizeof(void *),  unsigned long,  8 bytes
    void *p = malloc(s); // можно void *p = operator new(s);
    memcpy(p, a, s);      // копируем *a в *p
    memcpy(a, b, s);      // копируем *b в *a
    memcpy(b, p, s);     // копируем *p в *b
    free(p);                  // освобождаем место, operator delete(p);
}
int main(void) {
      double x = 2.0, y=3.0;
      Swap((void *) &x, (void *) &y, sizeof(x)); // применяем Swap к double

      int a = 2, b=3;
      Swap((void *) &a, (void *) &b, sizeof(a)); // применяем Swap к int

      return 0;
}
```

Ограничения шаблонов

Предположим, что имеется следующий шаблон функции:

```
template <typename T> // или template <class T>
void some_func (T a, T b) {
...
}
```

Часто в коде делаются предположения относительно того, как операции возможны для того или иного типа.

```
a = b;  // Если типом T является  указатель (например, T -> int * ), то их
        // значения будут копироваться.

if (a > b) // не будет справедливо, если T – обычная структура.
           // А если T – массивы, данная операция сравнивает их адреса.

T c = a*b; // умножение не определено для массивов, структур
```

Перегрузка и специализация шаблонов

Можно перегрузить определения шаблонов, точно так же, как перегружаются обычные функции. Как и в случае функций, перегруженные шаблоны должны иметь различные сигнатуры (список параметров). Параметры шаблона могут включать как обобщенные типы данных, так и определённые типы.

```
template <typename T> // исходный шаблон
void Swap(T &a, T &b) ;

template <typename T> // новый шаблон
void Swap(T *a, T *b, int n) ;
```

Специализация шаблона – определение функции для конкретного типа переменных вместо стандартного шаблона.

```
template <> void Swap<char> (char &, char &) ; // специализация шаблона
template<> void Swap(char &, char &); // специализация шаблона
```

Приоритет использования

Приоритет выбора компилятора при наличии одинаковых имен:
1) обычная функция
2) явная специализация шаблона
3) шаблон функции

```
template <typename T> void Swap(T &, T &) ;       // основной шаблон

void Swap (char &, char &) ;  // обычная функция

int main(void) {
      double u, v;
      ..............
      Swap(u, v); // неявное создание экземпляра (implicit instantiation)
      Swap<double>(u, v); // явное создание экземпляра (explicit instantiation)

      char a, b;
      .....
      Swap(a, b); //  обычная функция
}
```