Лекция 5. Функции

Стуктурное программирование

- Три базовых конструкции
  1. Последовательное исполнение
  2. Ветвление
  3. Циклы
- Повторяющийся код -> подпрограммы
- Пошаговая древовидная разработка «сверху вниз».
- Теорема **Бёма-Якопини**: любой алгоритм можно представить в виде описанных базовых конструкций
- Не используйте `GOTO` (Дейкстра бы не одобрил)

Объявление функций

```
1. [спецификатор] <возвр. тип> имя([<аргументы>]){ /*тело*/ }
```

- Формальные аргументы находятся в самой внешней области видимости функции.
- Локальные переменные по умолчанию инициализируется мусором.
- Объявляйте их поближе к месту использования.
- Статические локальные переменные инициализируются при первом «выполнении» определения. Живут до конца программы.

Передача параметров

- Варианты: по значению, указателю/ссылке
- Input:
  - Базовые (int, double) и маленькие (complex, point) типы – по значению
  - Остальные объекты – константные ссылки
  - Нужна внутри копия – хорошо передать по значению
- Ouput (or mix):
  - Необязательный параметр – по указателю (умному). Можно передать 0.
  - Обязательный – лучше по неконстантной ссылке.

Передача параметров. Примеры

- При вызове выполняется проверка и преобразование типов а-ля инициализация (одношаговая).

```
1.  struct X {};
2.  struct Y { Y(){} Y(X){} };
3.  struct Z { Z(){} Z(Y){} };
4.
5.  void apply(Z){}
6.
7.  int main()
8.  {
9.      apply(Y()); // ok, just one-step conversion
10.     apply(X()); // wrong
11. }
```

- Порядок вычисления аргументов не определен.

```
1.  void foo(int a, int& b, int){/*...*/}
2.  foo(++i, arr[i + 1], 0);
```

Передача массивов

- Теряется размер. В случае C-строки не так страшно.
- Массив преобразуется в указатель (`T[] -> T*`)

```
1.  void sort(float* arr, size_t size);
2.  void sort(std::vector<float>& arr);
3.
4.  //...
5.  const size_t size = 256;
6.  float a[size];
7.  sort(a, size);
8.
9.  std::vector<float> b;
10. sort(b);
```

Возврат значения

- Значение возвращается `return`. Вернуть значение можно из любой точки функции.
- Можно вернуть и void. (*Зачем?*)

```
1.  void foo(){}
2.  void bar(){ return foo(); }
```

- При возвращении происходит инициализация временной переменной возвращаемого типа.
- Не возвращайте ссылки/указатели на локальные переменные.

Return Value Optimization (RVO)

```
1.  struct C
2.  {
3.      C()          { cout << "C()";}
4.      C(const C&) { cout << "C(C const&)"; }
5.  };
6.
7.  C f() { return C(); }
8.
9.  int main()
10. {
11.     C obj = f();
12.     return 0;
13. }
```

- В зависимости от компилятора и настроек

```
1.  C()
2.  C(C const&)
3.  C(C const&)
```

```
1.  C()
2.  C(C const&)
3.
```

```
1.  C()
2.
3.
```

R-value reference* (C++11)

```
1.  vector<int> make_indices(vector<int>&& non_opt)
2.  {
3.      vector<int> x = move(non_opt);
4.      vector<int> y = /*...*/;
5.
6.      return (/*...*/) ? x : y;
7.  }
8.
9.  int main()
10. {
11.     vector<int> non_optimized;
12.     vector<int> ind = make_indices(std::move(non_optimized));
13.
14.     return 0;
15. }
16.
17. //...
18. vector& operator=(vector&& other);
```

Перегрузка. Выбор функции

- Порядок выбора перегрузки:
  1. Полное соответствие или тривиальное преобразование (`T -> const T`)
  2. «Продвижение типа» без потерь: `int -> long`, `float -> double`
  3. Стандартные преобразования (возможно, с потерей): `double -> int`
  4. Пользовательское преобразование
  5. Соответствие за счет … (часть синтаксиса)
- Контекстно-независимо: возвращаемое значение не участвует в выборе.
- Много аргументов: один лучше, остальные не хуже.
- **NB** Сложно? А ведь есть еще шаблонные функции!

Перегрузка. Примеры

```
1.  bool do_smth(int);          // (1)
2.  int  do_smth(const char*); // (2)
3.
4.  int main()
5.  {
6.      do_smth(5);        //(1)
7.      do_smth("hello"); //(2)
8.      do_smth(2.71);     //(1)
9.
10.     do_smth(static_cast<const char*>(0)); // (2)
11.     int cond = do_smth(0);                // (1)
12.
13.     return 0;
14. }
```

Аргументы по умолчанию

```
1.  string int_to_str(int value, int base = 10);
2.
3.  int main()
4.  {
5.      string s = int_to_str(15);       // "15"
6.      string h = int_to_str(33, 16);   // "21"
7.
8.      return 0;
9.  }
```

- Описывает в объявлении, нельзя изменить в определении. (*Почему?*)
- По умолчанию могут быть только последние аргументы функции

Произвольное количество аргументов

- Теряется знание о типе

```
1.  printf("name: %s; score: %.2lf\n", name.c_str(), score);
2.
3.  cout << "name: " << name << "; score: "
4.       << std::setprecision(3) << score << endl;
5.
6.  cout << (boost::format("name: %s; score: %.2lf\n") % name % score);
```

```
 1.  void log(size_t level, ...)
 2.  {
 3.      va_list list;
 4.      va_start(list, level);
 5.
 6.      for(const char* str = va_arg(list, const char*);
 7.          str != 0;
 8.          str = va_arg(list, const char*))
 9.      {
10.          cout << str;
11.      }
12.
13.      va_end(list);
14.  }
15.
16.  log(1, "hello ", "world", (const char*)0);
```

Указатель на функцию

```
1.  void foo(int x   ){}
2.  void foo(double y){}
3.
4.  //...
5.  void (*bar)(int)   = &foo;
6.  void (*bad)(float) = &foo; //wrong
7.
8.  bar   (5);
9.  (*bar)(5);
10.
11. //-----------------------------------
12.
13. typedef void(*log_writer)(int, string);
14. void add_writer(log_writer lw);
15.
16. void console_writer(int, string);
17. //...
18. add_writer(console_writer);
```

И снова определения auto

- Можно выводить тип по типу выражения

```
1.  int  x = 5;
2.  auto y = 6;
3.
4.  std::map<int, string> m;
5.
6.  // std::map<int, string>::iterator
7.  auto  it    = m.find(42);
8.  auto& value = m[42];
```

- И упрощать определение member-функций*

```
1.  struct my_class
2.  {
3.      typedef my_class* pointer_t;
4.      pointer_t get() const;
5.  };
6.
7.  // my_class::pointer_t my_class::get() const;
8.  auto my_class::get() const ->pointer_t;
```

Anonymous functions (C++11)

```
1.  // lambda functions
2.  [capture](params) [-> return-type] {body}
3.
4.  // samples
5.  [] (int x) { return x + global_y; }
6.  [] (int x) -> int
7.  { z = x + global_y; return z; }
8.
9.  // capture type
10. [x, &y](){} // capture x by value, y by ref
11. [=, &x](){} // capture everything by value, x by ref
12.
13. // more samples
14. matrix m;
15.
16. auto rot = [&m](point& p){ p *= m; }
17. for_each(points.begin(), points.end(), rot);
```

bind & function*

```
1.  using namespace std;
2.  using namespace std::placeholders;
3.
4.  typedef std::function<void(int, string)> log_writer;
5.  void add_writer(log_writer const& lw);
6.
7.  struct net_writer
8.  {
9.      void write(int level, string host, string app, string msg);
10. };
11.
12. int main()
13. {
14.     net_writer nw;
15.     add_writer(bind(&net_writer::write,
16.                     &nw, _1, "192.168.0.1", "model", _2));
17.
18.     add_writer([&nw](int level, string msg)
19.     {nw.write(level, "192.168.0.1", "model", msg);});
20.
21.     return 0;
22. }
```

Calling conventions*

- Определяет:
  - Как передаются параметры (на стеке/через регистры), как возвращается значение
  - Порядок передачи параметров
  - Кто чистит стек-фрейм (вызывающий/вызываемый)
  - Декорирование имени функции (для линковки)
  - Какие регистры допускается изменять

cdecl, stdcall

- cdecl –конвенция в C++ по умолчанию
  - Передача параметров RTL
  - Очистка стека со стороны вызывающей функции - можно делать printf(…)
  - Возврат через EAX или ST0
  - Именование: _sum
- stdcall – стандартная для WinAPI
  - Очистка стека со стороны вызываемой функции (меньше кода), нельзя делать printf(…)
  - Именование: _sum@8

Stack frame, cdecl Caller*

```
1.
2.  int __cdecl sum(int a, int b)
3.  {
4.      return a + b;
5.  }
6.
7.  int c = sum(2, 3);
```

```
1.  // -- Caller
2.  // push arguments to the stack, from right to left
3.  push    3
4.  push    2
5.
6.  // call the function
7.  call    _sum
8.
9.  // cleanup the stack by adding the size
10. // of the arguments to ESP register
11. add     esp,8
12.
13. // copy the return value from EAX to a local variable (int c)
14. mov     dword ptr [c],eax
```

Stack frame, cdecl Callee*

```
1.      // function prolog
2.      push    ebp
3.      mov     ebp,esp
4.      //...
5.
6.      //    return a + b;
7.      mov     eax,dword ptr [a]
8.      add     eax,dword ptr [b]
9.
10.     // function epilog
11.     //...
12.     mov     esp,ebp
13.     pop     ebp
14.     ret
15.     // ret 8 in case of stdcall
```

Вопросы?