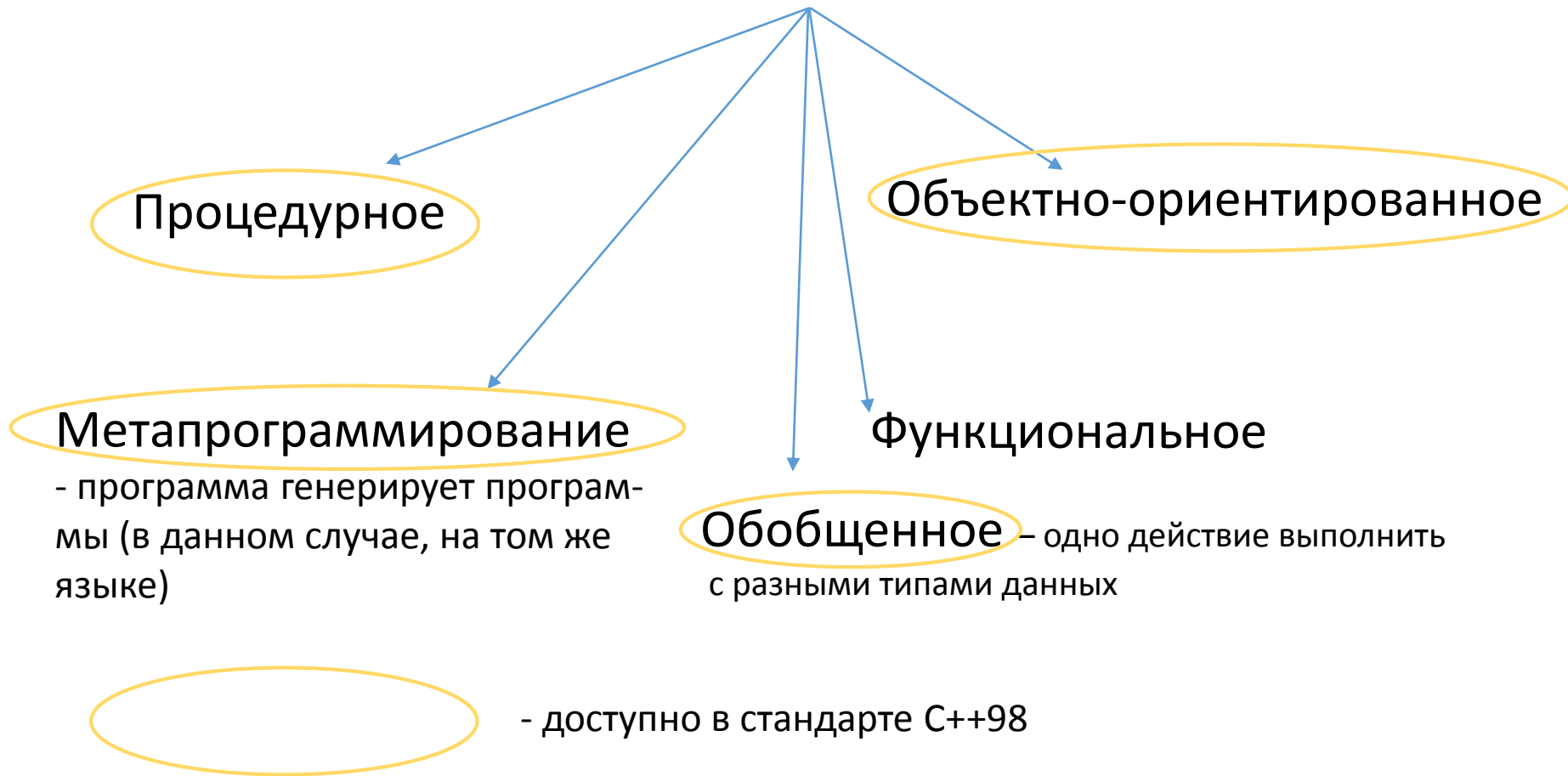


Курс лекций: Язык программирования C++

Лекция 1: Отличия языка C++ от языка C. Типы данных и переменные C++

Преподаватель: Митричев Иван Игоревич,
ассистент кафедры ИКТ, к.т.н.

Программирование



Метапрограммирование и обобщенное – реализуются шаблонами

C++98 -> C++11 (черновик назывался C++0x) -> C++14 -> C++17 ...

Языки программирования



```
graph TD; A[Языки программирования] --> B[Компилируемые]; A --> C[Интерпретируемые];
```

Компилируемые

Специальная программа – компилятор (например, `/usr/bin/gcc`) осуществляет несколько этапов

- 1) препроцессор выполняет преобразование кода: осуществляет подстановку макросов, `#include` и `inline-функций`, оптимизацию кода – раскрутка циклов и т.п.),
- 2) компилятор преобразует код на высокоуровневом языке в низкоуровневый код на языке ассемблера
- 3) ассемблер преобразует код `asm` в объектный код (машинные двоичные инструкции) . Получаем объектный файл
- 4) Линкер (компоновщик) осуществляет сборку (линковку). В каждом объектном файле есть таблица символов – все (сокращенные) названия функций, глобальных переменных. Для каждого вызова функции из объектного файла, которой нет в данном файле, линкер ищет ее во всех других таблицах, а также во всех указанных ему статических и динамических библиотеках. В итоге, собирается один файл, в котором для каждого символа указан адрес, откуда его вызвать.

Интерпретируемые

Построчное выполнение программ специальной программой - интерпретатором (например, `python3.5.exe`). При достижении строки с ошибкой синтаксиса или ошибкой времени исполнения (переполнение памяти, некорректное обращение к памяти и т.п.) происходит остановка работы.

Языки программирования



Низкого уровня

- ориентирован на тип процессора
- высокоэффективные и компактные по размеру программы
- программирование на нем используется для разработки некоторых драйверов, системных приложений, критичных к размеру кода

Язык ассемблера

Высокого уровня

- удобен в использовании для программиста
- легко переносится на любые архитектуры

C, C++, C#, Fortran, Java, Scheme, Pascal, Html...

Основные отличия C от C++

C	C++
Процедурное программирование	Объектно-ориентированное программирование, с C++11 – некоторые возможности функционального программирования
-	Перегрузка операторов и функций
Использование функций для ввода-вывода (scanf, printf)	Использование объектов и перегрузки для ввода-вывода (cin, cout)
-	Пространства имен (namespace)
-	Объявление функций в структурах
-	Ссылочный тип переменных
-	Виртуальные, встроенные и дружественные функции
Функции для работы с памятью - malloc, calloc, realloc	Операторы new, new[], delete, delete[] для работы с памятью
-	Встроенная поддержка обработки исключений (try, catch, throw)

Основные отличия C от C++

C	C++
Многострочные комментарии /* */	Однострочные и многострочные комментарии // /* */
-	Несколько последних параметров функции можно задать по умолчанию: void f(int a, int b=1, int c=2); // f(10), f(10,1,2) – одинаковый результат
-	Функции без аргументов: f() Функции с неизвестным числом аргументов int printf(const char* fmt, ...); // функция с неизвестными аргументами
Объявление переменных только в начале блока {...}	Объявление переменных возможно в любом месте блока с кодом, в т.ч., в объявлении цикла: int h; for (int i=0; i<10; i++) {h=i; int b=h;} Объявление перед первым обращением считается наиболее удобным

Структура программы на C++

Главный файл с программой должен обязательно содержать функцию main

```
#директивы препроцессора
```

```
.....
```

```
#директивы препроцессора
```

```
функция a ( ){
```

```
    операторы;}
```

```
класс b {
```

```
    операторы;}
```

```
void main ( ) //функция, с которой начинается выполнение программы
```

```
{
```

Объявление переменных, вызов функций, операторы (присваивания, сравнения, условные), циклы

```
....
```

```
}
```

В составе программы могут быть не один, а несколько файлов (проект)

Структура проекта на C++

Главный файл с программой `diploma.cpp`:

```
#include a.h  
#include b.h  
....
```

```
a.h:  
#ifndef A_H  
#define A_H  
...  
#endif
```

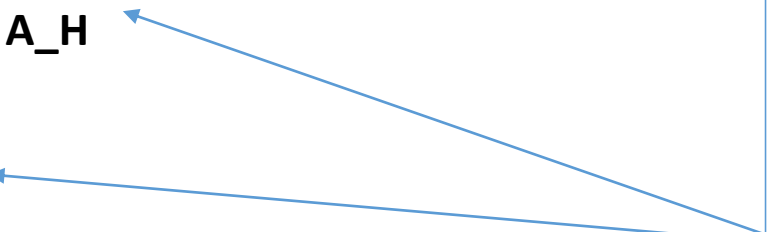
```
b.h:  
#ifndef B_H  
#define B_H  
...  
#endif
```

`a.cpp`:

```
#include a.h  
....
```

`b.cpp`:

```
#include b.h  
....
```



«Стражи включения». Помогают в один `cpp` файл включить код только один раз, а также не зациклиться со включением файлов (предположим, `a.h` внутри себя включает `b.h`, и наоборот)

Пример программы на С и С++

```
/* Example program */
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    char name [20];
```

```
    printf("What is your name? ");
```

```
    scanf("%s", name);
```

```
    printf("Hello, %s!\n", name);
```

```
}
```

```
// Example program
```

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    char name [20];
```

```
    std::cout << "What is your name? ";
```

```
    std::cin >> name;
```

```
    std::cout << "Hello, " << name << "!\n";
```

```
}
```

Объявление и определение

```
int f(int a) {return 10+a;}  
int main()  
{  
    f(5);  
}
```

объявление и определение

```
int f(int);  
int main()  
{  
    f(5);  
}
```

Объявление (declaration)

```
int f(int a) {return 10+a;}
```

Определение (definition)

Объявление и определение в проекте

f.h

```
int f(int);
```

→ объявление

f.cpp

```
#include "f.h"
```

```
int f(int a) {return 10+a;}
```

→ определение

diploma.cpp

```
#include "f.h"
```

```
int main()
```

```
{
```

```
    f(5);
```

```
}
```

Постойте, `#include "f.h"` или `#include <f.h>` ?

Различие между кавычками в include

`#include "f.h"` – искать в директории с текущим файлом (заметьте, в поддиректориях не ищется!), а затем - в директориях, определенных компилятором (так подключают файлы с проектом)

`#include <f.h>` - искать в директориях, определенных компилятором (может зависеть от компилятора. Так подключают обычно системные библиотеки)

Единица трансляции (translation unit)

- «Исходный файл вместе со всеми заголовочными и исходными файлами, включенными с помощью препроцессорной директивы `#include`, кроме строк, пропущенных с помощью специальных директив условного пропуска кода, называется единицей трансляции» (Стандарт языка C++98, ISO/IEC 14882:1998)

f.h

```
int f(int);
```

f.cpp

```
#include "f.h"  
int f(int a) {return 10+a;}
```

Единица трансляции

diploma.cpp

```
#include "f.h"  
int main()  
{  
    f(5);  
}
```

f.h

```
int f(int);
```

Другая единица трансляции

Вред макросов и inline-функции

Код малых функций лучше подставлять в тело функций, их вызывающих, на этапе компиляции (не на этапе разработки – это усложняет понимание кода и превращает все в спагетти-код!)

- `double Sqr(double x) {return x*x;}`

Недостатки: вызов функции – накладные расходы передача параметра через стек/регистры, передача управления функции, возврат из функции, освобождение стека/восстановление регистров

Может быть, макрос? Как в языке C

```
#define Sqr(x) ((x)*(x))
```

```
int main()
```

```
{
```

```
Sqr(5); // сюда до компиляции подставляется не код, а ТЕКСТ, объявленный в define
```

```
...
```

```
}
```

Недостатки: отладчик не отлаживает макросы, поэтому функции в макросах писать неудобно, их трудно отлаживать и тестировать. Также нельзя привести тип при передаче параметра макросу (а функция приводит тип - `double Sqr(double x)`) и многое другое.

Может быть, компилятор сам подставит? Такое бывает, но только если функция описана в той же единице трансляции, что и ее вызывающая.

Вред макросов и inline-функции

Решение:

```
inline double Sqr(double x) {return x*x;}
```

```
int main()
```

```
{
```

Sqr(5); // сюда до компиляции препроцессором подставляется КОД, а не просто текст. В результате – возможность использования удобств функций и отладки отдельной функции.

```
...
```

```
}
```

Типы данных C++

int (целый)

char (символьный) – всегда 1 байт (8 бит). В данных типа signed char можно хранить значения в диапазоне от –128 до 127. При использовании типа unsigned char значения могут находиться в диапазоне от 0 до 255. Для кодировки используется код ASCII.

wchar_t (расширенный символьный) – для символов Unicode. Размер этого типа, как правило, соответствует типу short. Строковые константы такого типа записываются с префиксом L: L“String #1”.

bool (логический)

float(вещественный с плавающей запятой) – обычно, 32 бита

double (вещественный с двойной точностью с плавающей запятой) – обычно, 64 бита

Существует 4 спецификатора типа, уточняющих внутреннее представление и диапазон стандартных типов

short (короткий)

long (длинный)

signed (знаковый)

unsigned (беззнаковый)

Типы данных C++

Размер целочисленных типов зависит от модели представления данных (ОС/разрядность). Для 64-битных систем:

Модель LLP64 или 4/4/8 (int и long — 32 бита, указатель — 64 бита) = Win64 API

Модель LP64 или 4/8/8 (int — 32 бита, long и указатель — 64 бита) = Unix и Unix-подобные системы (Linux, Mac OS X)

(unsigned) int 32 бита: $-2\,147\,483\,648 \dots +2\,147\,483\,647$

- float занимают 4 байта, из которых один разряд отводится под знак мантиссы, 8 разрядов под порядок и 24 — под мантиссу.
- Величины типы double занимают 8 байтов, под порядок и мантиссу отводятся 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а длина порядка - его диапазон.

Переменные в C++

Общий вид оператора описания:

- [класс памяти][const]тип имя [инициализатор];

Класс памяти может принимать значения: auto, extern, static, register. Класс памяти определяет время жизни и область видимости переменной.

Const – показывает, что эту переменную нельзя изменять (именованная константа).

При описании можно присвоить переменной начальное значение (инициализация).

Классы памяти:

auto –автоматическая локальная переменная. Спецификатор auto может быть задан только при определении объектов блока, например, в теле функции. Этим переменным память выделяется при входе в блок и освобождается при выходе из него. Вне блока такие переменные не существуют.

extern – глобальная переменная, она находится в другом месте программы (в другом файле или далее по тексту). Используется для создания переменных, которые доступны во всех файлах программы.

static – статическая переменная, она существует только в пределах того файла, где определена переменная.

register - аналогичны auto, но память под них выделяется в регистрах процессора. Если такой возможности нет, то переменные обрабатываются как auto.

Переменные в C++

Пример

`int a;` //глобальная переменная – описывается вне функций в файле. Видна в других единицах трансляции, если объявлена там как внешняя (`extern`). Хранится до конца работы программы

```
void main(){
```

`int b;` //локальная переменная. Ее область видимости (существования) ограничена концом того блока кода { }, где она объявлена

`extern int x;` //переменная x определена в другом месте. Доступна во всех единицах трансляции, и в хотя бы одной должна быть объявлена без `extern` (как глобальная)

`static int c;` //локальная статическая переменная. Хранится до конца работы программы.

```
    a=1; //присваивание глобальной переменной
```

```
    int a; //локальная переменная a. Перекрывает глобальную переменную
```

```
    a=2; //присваивание локальной переменной
```

```
    ::a=3; //присваивание глобальной переменной, :: - оператор разрешения контекста
```

```
}
```

```
    int x=4; //определение и инициализация x
```

Пространства имен в C++

Мы видели, что две переменные с одинаковым именем в одном файле – это не очень удобно, хотя и возможно, и вероятно возникновение ошибок – багов (забыли, какую переменную используем). Для изоляции переменных можно использовать пространства имен. Так делается в любой крупной библиотеке, например, стандартной библиотеке языка C++ (`#include <iostream>` подразумевается во всех примерах):

```
using namespace std;
```

```
...
```

```
{
```

```
cout<<"Vasya";
```

```
}
```

Можно загружать не все переменные и функции пространства имен, а выборочно:

```
using std::cout;
```

```
...
```

```
{
```

```
cout<<"Vasya";
```

```
}
```

`::` - оператор разрешения контекста, можно получить переменную из другого пространства имен.

// без использования namespace

```
...
```

```
{
```

```
std::cout<<"Vasya";
```

```
}
```

Указатели и ссылки в C++

Указатель – переменная, содержащая адрес памяти.

`const int *foo` или `int const *foo` – указатель на константу, через этот указатель ее нельзя поменять

`int *const foo = &x` - константный указатель на `int`. Нужно инициализировать при объявлении (похож на ссылку). Нельзя переставит указатель на другую переменную. Саму переменную через этот указатель поменять можно.

`const int *const foo = &x` – и то, и другое.

`&` - операция взятия адреса. `*` - операция разыменования указателя (получить значение)

```
int x=10; int *foo=&x; cout << *foo << "\n"; //10
```

Ссылка:

- обязательно при создании инициализировать конкретным значением;
- нельзя изменять ссылку после этого (но значение, на которое ссылается – можно!)

Константная ссылка: не бывает (всегда константная).

Ссылка на константу: нельзя изменять значение, на которая указывает ссылка.

```
const int &foo = x
```

Передача по указателю, ссылке и по значению

```
void f (int& a)
{
    a = 1;
}
int main ()
{
    int e = 5;
    foo (e); // теперь e=?
}
```

```
void f (int* a)
{
    *a = 1;
}
int main ()
{
    int e = 5;
    foo (&e); // теперь e=?
}
```

```
void f (int a)
{
    a = 1;
}
int main ()
{
    int e = 5;
    foo (e); // теперь e=?
}
```

Передача по указателю, ссылке и по значению

```
void f (int& a)
{
    a = 1;
}
int main ()
{
    int e = 5;
    foo (e); // теперь e=1
}
```

```
void f (int* a)
{
    *a = 1;
}
int main ()
{
    int e = 5;
    foo (&e); // теперь e=1
}
```

```
void f (int a)
{
    a = 1;
}
int main ()
{
    int e = 5;
    foo (e); // теперь e=5
}
```