Списание „Математика, компютърни науки и образование“
том 2, брой 2, 2019

Journal “Mathematics, Computer Science and Education”
vol. 2, No. 2, 2019

Преподаване на C++ като първи език за програмиране

Ивайло Дончев

Teaching C++ as a First Programming Language

Ivaylo Donchev

Abstract: *Nowadays, most C++ introductory courses begin by studying the procedural constructs of C language, following the imperative-first strategy described in CC2001 [7]. While this has been a successful approach in the near past, our experience shows that it does not lead to the development of skills to create a good C++ code. That's why in recent years we've been teaching C++ to beginners in a totally different way – with avoidance of C-style strings and raw arrays; delayed introduction to pointers (just after references); polymorphism with references instead of pointers; smart pointers instead of raw pointers; early use of standard library features, and writing modern C++ from day one.*

Keywords: *teaching, programming languages, C++, polymorphism, smart pointers.*

ВЪВЕДЕНИЕ

Преподаването на програмиране на начинаещи винаги се е считало за трудна задача [5,1]. Предизвикателства са високата степен на абстракция, сложността на синтаксиса и семантиката на езиците, недостатъчно развитото алгоритмично мислене и умения за решаване на проблеми у обучаемите. Изучаването на C++ като първи език беше обичайна практика през 90-те и даваше добри резултати както при студенти, така и при ученици. Той успешно замени процедурния език Pascal, предлагайки ефективна реализация на обектно ориентираните концепции. Въпреки това изучаването му от начинаещи винаги е съпътствано с проблеми най-вече по отношение на управлението на паметта, работата с масиви и *null-terminated* низове – проблеми, в голяма степен свързани и наследени от езика *C*. С широкото навлизане на обектно ориентираната технология в разработката на софтуер в началото на 21 век се утвърдиха нови езици с общо предназначение като Java и C#, предлагащи изчистен синтаксис и нови възможности, които улесняват писането на код и повишават производителността на програмисткия труд. Съвсем естествено тези езици бяха възприети и в обучението, като за подпомагане на учениците и акцентиране на важните концепции бяха създадени специални обучаващи среди за разработка. C++ постепенно беше изместен от Java като най-чест избор за уводен курс [3]. Въпреки това учебните планове на цели 83% от университетските компютърни специалности в България все още залагат точно на C++ като първи език [2], вероятно с идеята, че е подходящ и за последващи дисциплини, в които се изучават алгоритми и абстрактни структури от данни. Има опит дори с обучение на 10-11-годишни деца, макар да става въпрос за подготовка на бъдещи състезатели [8]. Методиката се прилага и днес в много школи. Не е за пренебрегване и фактът, че със стандарта от 2011 година в C++ започнаха промени, които го правят да изглежда като съвсем нов език. Връзката със C е все по-слаба за сметка на развитите абстракции от високо ниво. В този смисъл C++ е по-близо до C# и Java, вместо до C и може да ги замени в обучението.

Много курсове използват обектно ориентиран език, но не по обектно ориентиран начин [4]. Тезата, която защитаваме в тази статия е, че правилният подход за преподаване на C++ на начинаещи е максимално отлагане във времето на конструкциите от ниско ниво, наследени от C; ранно въвеждане на обектно ориентираните концепции, но без употребата на указатели; широко използване на стандартната библиотека и новите езикови конструкции; акцентиране на най-важните C++ концепции – класове и обекти; предефиниране на операции; шаблони; STL; ламбда изрази; обработка на изключения.

КАК НЕ ТРЯБВА ДА СЕ ПРЕПОДАВА C++

По-възрастните разработчици са изучавали програмиране в последователност, следваща историческото му развитие – първо процедурно (C или Pascal), след това обектно ориентирано,

функционално и логическо. За такива обучаеми по-краткият път на изучаване на C++ е разглеждането му като обектно ориентирана надстройка на езика C. Погрешно е обаче да се смята, че изучаването на C е необходимо условие за изучаването на C++. Прилагането на такъв подход непременно ще изисква повече време, а освен това ще се усвоят навици, водещи до проблеми с качеството на C++ кода. Като примери могат да се посочат библиотеките от функции с близки имена (`strlen, strcpy, strncpy, strcmp`), които не са по-добра абстракция от класа. На C не може да се реализира RAII (*resource acquisition is initialization*) идиомът. Няма итератори. Сигурността на типовете (*type safety*) е по-ниска.

Преподавайки C++ по този начин, правим езика да изглежда по-труден, отколкото е в действителност и не насърчаваме начинаещите да пишат съвременен код. Това, че много хора, по стечение на обстоятелствата, са научили C++ като разширение на C не означава, че тези, които сега започват, трябва да научат първо C. Считаме това за погрешно, защото би създало по-зле обучени C++ програмисти, отколкото биха били иначе. В допълнение такъв подход намалява удоволствието от процеса на обучение, скрива истинските предимства на C++, добавя ненужна сложност и замъглява връзките с другите съвременни езици.

В следващите точки ще разгледаме по-конкретно C елементи, които трябва да се избягват или отлагат във времето – символни низове, масиви, извеждане на данни, указатели и адресна аритметика, динамична памет с new и delete.

СИМВОЛНИ НИЗОВЕ

Работата с текстови данни е характерна за почти всяка нетривиална програма. Затова обучаемите трябва да започнат да използват символни низове от първия урок. Ако обаче заложим на *C-style* низове (`char[] è char*`) и вход/изход с функциите от `<stdio.h>`, ето как ще изглежда елементарна Hello World програма, питаща за име и извеждаща поздрав:

```
printf(“Enter your name: “);
char name[20];
scanf(“%19s”, name);
printf(“Hello, %s!\n”, name);
```

Ако в лабораторната работа се използва актуална версия на Visual Studio, ще се наложи добавянето на още код и промяната на съществуващ, за да се потиснат предупрежденията и изобщо да може да се стартира програмата:

```
#pragma warning(disable : 4996)
int res = scanf(“%19s”, name);
```

Скоро ще се наложи да се въведат и понятията масив и указател, за да може да се извършва каквато и да е обработка с функциите от `<string.h>`. Трябва да се изясни връзката между масиви и указатели (името на масива е константен указател към първия му елемент), ролята на *null* терминатора и т.н.

Всичко това изглежда стресиращо дори за студенти. Кодът е труден за четене и изисква запаметяване на имена на функции, ред на аргументите и др. Очаквано се допускат много грешки и обучаемите остават с убеждението, че C++ е труден език. Каква е алтернативата? – низове от хедъра `<string>` на стандартната библиотека:

```
cout << “Enter your name: “;
string name;
cin >> name;
string greeting = “Hello, “ + name + ‘!’;
if (name == “Ivaylo”)
       greeting += “ You are our teacher.”;
cout << greeting << endl;
```

Този код е много по-лесен за четене. Операциите +, == и += са интуитивно ясни, ако преди това са използвани с числови данни. Интуитивно ясно е и че събирането на числа е различно събирането на текст. На този етап не е нужно да се обяснява какво е предефиниране на операции и преобразуване на типове, за да се възприеме поведението на кода. Прави се само пропедевтика на тези механизми, които ще се изясняват подробно в последващи курсове.

Вероятно обучаемите ще възприемат `string` като вграден тип, особено ако имат някакъв опит с друг език, където това наистина е така. Не е необходимо да бъдат разубеждавани в този момент. Правилният подход е да бъдат насочени към фундаменталните понятия „обект", „тип" и „променлива", както прави Stroustrup в учебника за начинаещи [6, стр. 60] – „за да прочетем нещо ни е необходимо място в паметта, където да го разположим. Това място се нарича обект. Обектът е област от паметта с тип, който определя какъв вид информация може да се постави в нея. Именуван обект се нарича променлива. Например символните низове се разполагат в променливи от тип `string`, а целите числа – в `int` променливи. Можете да считате обекта за „кутия", в която можете да поставите стойност от типа на обекта".

Достатъчно е да се покаже как се работи с низове, а реализацията им като стандартен контейнерен клас ще остане за следващ курс. Там е мястото, където може да се коментира и ефективността на реализацията и предимствата в това отношение на *C-style* низовете.

КОНЗОЛЕН ИЗХОД С `printf()`

И в наши дни голяма част от примерите в документацията на Visual C++ използват `printf()` за извеждане на данни към конзолата. Това е така, защото наследената от C функция е икономична на ресурси и работи бързо. Но дори и на старото поколение програмисти ще се наложи да направи справка за форматиращите спецификатори. Когато става въпрос за начинаещи, използването на `printf()` ненужно увеличава възприятието за трудност и е предпоставка те да правят повече грешки. Най-често с `printf()` се извеждат междинни стойности на променливи и резултати от изчисляване на изрази с цел проследяване на работата на програмата. Съвременните програмни среди разполагат с много добри дебъгери, които могат да свършат тази работа. Уменията за работа с дебъгер са задължителни, така че ранното му използване ще бъде само от полза за обучаемите. Друго предимство е, че примерният код ще бъде изчистен от множеството оператори за изход и така ще бъде по-лесен за възприемане, а и няма да се налага постоянно превключване между кода и прозореца с конзолата.

Друга алтернатива е използването на потоци `cout, cin, cerr`. Производителността на входно-изходните потоци не е от значение за начинаещите. Важна е подобрената четимост на кода

```
cout << "Hello, " << name << ". You are " << n << " years old." << endl;
```

Предупреждението, че обработката на големи файлове в условия, при които производителността е критична, тук може да бъде спестено.

Възможност е и курсът да залага на разработката на приложения с графичен интерфейс от самото начало – с подходяща работна рамка и обучаваща среда, която е пригодена за начинаещи. Получаването на изход върху екрана в този случай се свежда до задаването на стойност на свойство на обект, който е част от потребителския интерфейс.

СУРОВИ МАСИВИ

Преподавателската практика показва, че при задачите с масиви (*raw arrays*) се допускат много грешки. Масивът „не знае" своя размер и това изисква допълнителни параметри и постоянни проверки за коректна стойност на индекса. Неизбежно е и замесването на указателите на много ранен етап, а това напълно обърква начинаещите. Изисква се също добро познаване на операторите за цикъл, въпреки че алгоритмите от стандартната библиотека и операторът `for` за диапазон (*range-based for*) поддържат сурови масиви.

Подходяща алтернатива за начинаещи е `std::vector.` Това е първият (и може би единственият) контейнерен клас, който е необходим на начинаещите. Всички стандартни алгоритми от библиотеката го поддържат, синтаксисът е интуитивен, позволява се присвояване между вектори, което работи по очаквания начин:

```
vector<int> v1{ 3,1,2,4,5 };
vector<int> v2;
v2 = v1;
for (auto& x : v2)
       x *= 2;
auto res = count_if(begin(v1), end(v1), [](int x) {return x > 3;});
cout << res << " numbers greater than 3" << endl;
```

Единствената препоръка е да се отложи максимално дискутирането на итератори – евентуално след указателите. Това, че алгоритмите изискват аргументи `begin()` и `end()` не означава, че трябва да се коментира каква стойност връщат те.

ИЗГРАЖДАНЕ НА ОБЕКТНО ОРИЕНТИРАНО МИСЛЕНЕ

Ако се спазят дадените дотук препоръки, а именно:

– замяна на `char*` със `string`;

– работа с низове от първия ден;

– отказ от `printf()` и използване на дебъгера или cout;

– отказ от суровите масиви и използване на `std::vector` от първия ден;

– указатели чак след полиморфизма,

начинаещите ще започнат да мислят на по-високо ниво на абстракция. Те ще възприемат низовете като „неща“, а не като поредица от символи, последният от които има специална стойност. За тях векторът също е „нещо“. Те не знаят как се предефинират операции и дори, че съществува такова понятие, но интуитивно разбират, че + е различен за числа и за низове и не е нужно това да им се казва изрично. Още не трябва да се преподават деструктори и управление на паметта, но е положена основата, върху която трудните неща ще изглеждат по-лесни.

ЛАМБДА ИЗРАЗИ

За да се възприеме по-добре дадено понятие, неговата употреба трябва да се мотивира. Важно е усещането, че езиковата конструкция идва на помощ за решаването на конкретен проблем. Типична употреба на ламбда изразите е като предикати в обръщения към стандартни алгоритми. Естествен подход е елементарните алгоритми да се използват като примерни задачи при изучаването на условните оператори и операторите за цикъл. След като се реализира просто търсене и броене с `for` и `if`, може да се демонстрира как задачата се решава с библиотечните алгоритми `find()` и `count()`, които не изискват предикати. След това задачата може да се усложни – например да се преброят четните числа в масив или да се провери дали има елементи, отговарящи на дадено по-сложно условие. Преминава се към `find_if()` и `count_if()` и тук на помощ идва ламбда изразът (виж примерния код по-горе). Добър пример е сортирането на колекции. След като обучаемите са използвали `sort()` с колекции от числа и низове, тук е мястото да сортират колекция от обекти на потребителски клас – отново с помощта на допълнителен аргумент ламбда израз. Ясно е, че дефинирането на потребителски класове трябва да се изучава на ранен етап в курса (*objects-early* подход). Когато чрез правене у обучаемите е изградена представа за същността на ламбда изразите, може за пълнота да се премине към подробности по синтаксиса им, да се дадат примери за подразбиращи се и експлицитни типове на резултата, различни комбинации от прихващане по стойност и по референция.

ПОЛИМОРФИЗЪМ

Динамичният полиморфизъм е отличителна характеристика на обектно ориентираното програмиране и типичният пример на C++ е общ базов клас с виртуален метод, съответстващ на полиморфичното действие, който се предефинира в няколко производни класа. Полиморфичното действие се активира чрез указател към базовия клас, на който се присвояват адреси на обекти на производните класове. Тоест отново опираме до указателите и всички проблеми, които те носят на начинаещите. Нашият подход е полиморфизмът да се демонстрира само с референции (псевдоними). Това ни спестява нотацията `*, ->, &`. Така обектите се използват по вече познатия начин и единственото ново е да се въведе понятието „независим псевдоним“ и предаване на параметър по референция. Това може да се демонстрира с обикновен интегрален тип като int:

```
int x{ 10 };
int& r = x;
r = 5;
cout << x;    //5
```

Класическият пример с функция `swap()` за размяна на стойностите на две променливи също е подходящ – трябва да се изследва поведението при предаване на параметрите по стойност и по

референция. Няма да отнеме много време да се възприемат синтаксисът и семантиката на този механизъм.

Преходът към полиморфизъм изисква създаването на проста наследствена йерархия с виртуален метод, реализиращ полиморфичното действие:

```
class Base
{
public:
  virtual void Foo() const {
        cout << “Base::Foo()\n”;
  }
};

class Derived : public Base
{
public:
  virtual void Foo() const {
        cout << “Derived::Foo()\n”;
  }
};
```

За активирането е необходима външна функция, която приема параметър от тип референция към базовия клас и чрез нея извиква виртуалния метод:

```
void Proccess(const Base& x){
  x.Foo();
 }
```

Така акцентът е върху самия полиморфизъм, а не върху синтактичните конструкции.

Извикването на функцията става по нормалния начин – с обикновени обекти на базовия и на производния клас:

```
  Base b;
  Derived d;
  Proccess(b);
  Proccess(d);
```

Полезно е да се демонстрира как се изменя поведението, когато вместо по референция, обектите се предават по стойност, а също и когато методът не е виртуален. Тук е моментът да се коментира и ролята на модификатора `const`.

ОБХВАТ И ЖИВОТ НА ОБЕКТИТЕ. RAII

Тази тема може да се преподава по различно време, но най-добре е това да се случи скоро след темата „Класове и обекти“. Тъй като умишлено избягваме динамичната памет, примерите трябва да показват стекова семантика. Управлението на паметта в C++ е грижа на програмиста и е трудно за начинаещи. За да не ги разочароваме, се насочваме първо към детерминистично разрушаване на други ресурси – затваряне на файлове и връзки към бази данни, освобождаване на заключвания (*locks*). Тук могат да се включат деструкторите. Въведението в RAII ще стане по естествен път. Добра идея е да се демонстрира разликата в живота на обектите, предавани по стойност и по референция чрез деструктор, който извежда нещо на екрана.

Усвояването на обхвата и деструкторите подготвя условията за въвеждане на изключенията – друга важна обектно ориентирана концепция.

УКАЗАТЕЛИ

Указателите, разбира се, имат място в учебния курс, но те трябва да се въведат възможно най-късно, защото пунктуацията, свързана с употребата им, е тежка за начинаещи. Често наблюдавана ситуация при работа със студенти е произволното добавяне на * и &, докато компилаторът спре да показва грешки. Ето защо не е нужно да се преподава употребата на указател:

– като итератор в суров масив;

– за предаване на параметри по адрес – за целта можем да използваме псевдоними;

– за реализация на полиморфизъм.

След като обучаемите вече са натрупали известен опит и разбират по-голямата част от пунктуацията, знаят как се предават параметри по стойност и по референция, чувстват се уверени, значи е дошъл моментът, в който ще могат да работят с указателите.

Употребата на указатели е задължителна за работа с динамичната памет (*heap*). Тук трябва да се обяснят операциите `new` и `delete`, да се коментира организацията на *heap* паметта, сравнена със стека и RAII техниката, които вече са преподавани. Удачно е сравнението с други езици, при които инстанция се създава само с `new`. Полезен е пример на RAII клас, ползващ динамична памет.

Обучаемите са свикнали с употребата на `std::vector` и `std::string`. Тук е мястото да им се обясни, че всъщност данните на тези контейнерни класове се разполагат в динамичната памет и контейнерите управляват тази памет вместо тях. Така лесно ще се възприемат умните указатели `std::shared_ptr` и `std::unique_ptr` като обекти, които автоматично извършват `new` и `delete` вместо тях. Важно е в примерите да се избира правилният за конкретния случай умен указател, а не винаги да се предпочита `shared_ptr` като универсален. Насърчава се употребата на `make_shared` и `make_unique` за създаване и инициализиране на умни указатели.

Познавайки полиморфизма чрез референции, за учениците ще е интересно да видят как същото поведение може да се постигне чрез указатели.

ЗАКЛЮЧЕНИЕ

Преструктурираният ни уводен курс извежда на преден план понятията и механизмите, определящи за обектно ориентираното програмиране и C++: класове и обекти, наследяване, полиморфизъм, предефиниране на операции, шаблони, стандартна библиотека, ламбда изрази, изключения. Усвояването на толкова много материал е възможно, защото преподаването акцентира в по-голяма степен на използване на понятията, без да навлиза във всички техни аспекти и подробности по реализацията им – това е оставено за следващите курсове по ООП и АСД. Целта на уводния ни курс не е да подготвя разработчици на библиотеки, а ползватели на библиотеки. Затова изключихме и много неща, които няма да са необходими на повечето хора в професионалната им реализация: извеждането на данни с `printf()`; всички `str` функции за работа с `char*` и `char[]` – как се четат имената им, в какъв ред са параметрите им, кога да се използват `n` и `s` вариантите им; сурови масиви и указатели – с всички свързани с тях проблеми с индексите и операциите `*` и `&`. Разбира се, цената за това е, че студентите няма да развият някои умения, например да четат стар C код и да правят някои трикове с компилатора; ще се затрудняват с приоритета на операциите и размера на типовете, но при завършването на курса те ще пишат по-качествен код. Обучението е по-забавно и по-лесно и в крайна сметка студентите стават по-добри C++ програмисти, защото мисленето им е обектно – използват стандартната библиотека, ламбда изрази, разбират пунктуацията, когато пишат код, спазват RAII идиома.

ЛИТЕРАТУРА

[1] **Bennedsen, J., Caspersen, M. 2007.** Failure rates in introductory programming, ACM SIGCSE Bulletin 39, No. 2, 2007, pp. 32–36.

[2] **Damyanov, I., Borisova, N. 2017.** Programming Languages in Undergraduate Courses and in Software Industry in Bulgaria, International Journal of Pure and Applied Mathematics, Volume 117 No. 2 2017, pp. 271–278.

[3] **Davies, S., Polack-Wahl J., Anewalt, K. 2011.** A snapshot of current practices in teaching the introductory programming sequence. In: Proceedings of the 42nd ACM technical symposium on Computer science education, ACM, pp. 625–630.

[4] **Meyer, B. 2003.** The outside-in method of teaching introductory programming, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2890, pp. 66–78.

[5] **Robins, A., Rountree, J., Rountree, N. 2003.** Learning and Teaching Programming: A Review and Discussion, Computer Science Education, 2003, Vol. 13, No. 2, pp. 137–172.

[6] **Stroustrup, Bj. 2014.** Programming: Principles and Practice Using C++ (2nd Edition), Addison-Wesley Professional; 2 edition, 2014.

[7] **The Joint Task Force on Computing Curricula. 2001.** IEEE Computer Society, Association for Computing Machinery, Computing Curricula 2001. Computer Science. Final Report.

[8] **Yovcheva, B. 2008.** Spiral Teaching of Programming to 10–11 Year-Old Pupils After Passed First Training (Based on the Language C++). In: Mittermeir R.T., Sysio M.M. (eds) Informatics Education – Supporting Computational Thinking. ISSEP 2008. Lecture Notes in Computer Science, vol 5090. Springer, Berlin, Heidelberg, pp. 171–179.

ИНФОРМАЦИЯ ЗА АВТОРА

Ивайло Дончев – доцент, доктор, Факултет „Математика и информатика", Великотърновски университет „Св. св. Кирил и Методий", e-mail: i.donchev@abv.bg

ABOUT THE AUTHOR

Ivaylo Donchev – Associate Professor, Ph.D., Faculty of Mathematics and Informatics,
St. Cyril and St. Methodius University of Veliko Turnovo. E-mail: i.donchev@abv.bg