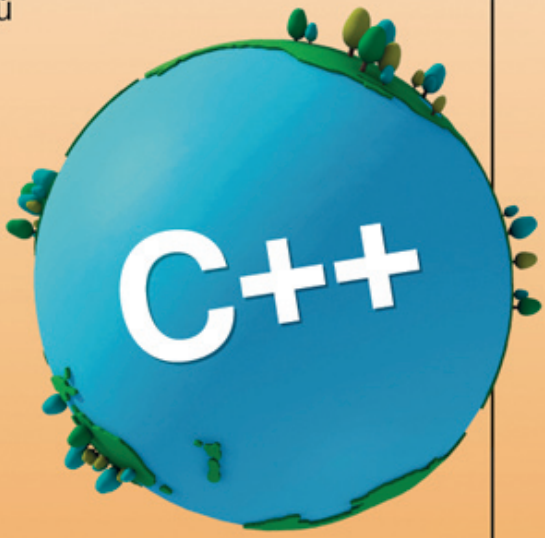
EDICE
myslíme v ...

knihovna programátora

- Programovací jazyky C a C++ podle platných standardů
- Včetně připravovaného standardu C++0x
- Referenční přehled
- Příklady použití popisovaných konstrukcí



Jazyky C a C++

MIROSLAV VIRIUS

kompletní průvodce – 2., aktualizované vydání

GRADA

Upozornění pro čtenáře a uživatele této knihy

Jazyky C a C++

kompletní průvodce – 2., aktualizované vydání

Miroslav Virius

Vydala Grada Publishing, a.s.
U Průhonu 22, Praha 7
jako svou 4475. publikaci

Odpovědný redaktor Pavel Němeček
Sazba Tomáš Brejcha
Počet stran 368
První vydání, Praha 2011

Vytiskla Tiskárna PROTISK, s.r.o., České Budějovice

ISBN 978-80-247-3917-5 (tištěná verze)
ISBN 978-80-247-7417-6 (elektronická verze ve formátu PDF)

Obsah

3. Základní datové typy

4. Výčtové typy, struktury a unie

8. Operátory a výrazy

9. Příkazy

10. Funkce

14. Šablony

15. Dynamická identifikace typů

16. Preprocesor

Předmluva

Otevřeli jste knihu, která vám poskytne referenční příručku programovacích jazyků C a C++ podle platných standardů, a to včetně připravovaného nového standardu jazyka C++ označovaného zatím C++0x.

Co v této knize najdete

V úvodní kapitole najdete příklad jednoduchého programu, základní informace o vytváření zdrojového textu programu a o postupu při překladu. Tato kapitola se poněkud vymyká z rámce celé knihy, neboť spíše než referenční příručku připomíná první kapitolu učebnice. Jejím cílem je poskytnout rámec, který by mu usnadnil pochopení dalších kapitol i čtenářům, kteří s jazyky C a C++ nemají žádnou zkušenost, nebo ji mají jen velmi malou. V závěru první kapitoly najdete také několik slov o historii těchto jazyků, o jejich mezinárodních standardech, a také velice stručný výklad základních pojmů objektově orientovaného programování se zřetelem k C a C++.

Ve druhé kapitole se seznámíte se způsobem popisu jazyků C a C++ a se základními stavebními prvky, jako je množina znaků, identifikátory, klíčová slova atd. Následující kapitoly popisují základní datové typy, uživatelem definované neobjektové typy, výrazy, příkazy, operátory atd. – ostatně to najdete v obsahu. Každý významový celek začíná zpravidla popisem syntaxe, za nímž následuje stručné vysvětlení významu a podstatné informace shrnuté do bodů a doplněné příklady. Tento postup ale nemělo smysl dodržovat vždy.

Převážnou většinu příkladů jsem odzkoušel na současných překladačích jazyků C a C++ na PC. V několika případech jsem převzal příklady přímo ze standardů [1] a [3]; zpravidla se jednalo o rysy jazyka, které běžné současné překladače ještě neimplementují nebo je neimplementují v souladu se standardy. (Může vám připadat podivné, že překladače jazyka C z roku 2010 neimplementují v plném rozsahu novinky standardu [3] tohoto jazyka z roku 1999, ale je to tak; pro tvůrce většiny překladačů je ovšem důležitější kompatibilita s jazykem C++ – a ještě standard [1] jazyka C++ z roku 2003 byl založen na standardu [2] jazyka C z roku 1990. Většiny příkladů týkajících se C++0x jsem převzal z návrhu nového standardu [4]; některé z nich bylo možno vyzkoušet v současných překladačích, ovšem zdaleka ne všechny. Poznamenejme, že dokument [4], z něhož jsem čerpal, je označen jako *konečný návrh standardu*.

Jak tato kniha vznikla

V roce 1999 vydalo nakladatelství Grada Publishing knihu *Programovací jazyky C a C++ podle normy ANSI/ISO – kompletní kapesní průvodce* (D. Louis, P. Mejzlík, M. Virius), která shrnovala oba jazyky i jejich knihovny do jediného svazku. V době, kdy jsme ji psali, tedy v letech 1997–1998, ovšem nebylo ještě k dispozici konečné znění prvního standardu jazyka C++, vydané v září 1998, ani nový standard jazyka C, vydaný v r. 1999. Neobsahovala tedy řadu důležitých informací. Také příklady byly přizpůsobeny nejrozšířenějším překladačům té doby – a ty se od standardu v mnoha ohledech odchylovaly. Ostatně ani struktura knihy nebyla právě šťastná, neboť byla přizpůsobena striktním požadavkům edice (10 kapitol, co nejvíce „postupů" apod.).

Proto jsem se v po dohodě s nakladatelstvím Grada Publishing rozhodl tuto knihu od základu přepracovat. Z původního díla jsem převzal některé příklady a části textu, avšak i ty jsem přizpůsobil novým standardům obou jazyků. Tato verze vyšla v nakladatelství Grada Publishing pod názvem *Jazyky C a C++ – Kompletní kapesní průvodce programátora* (M. Virius, 2005).

Na začátku roku 2011 mne nakladatelství Grada požádalo, abych tuto knihu připravil k novému vydání. Protože koncem roku 2011 má vyjít nový standard C++ s řadou zásadních novinek, rozhodl jsem se k dalšímu přepracování celé knihy, aby obsahovala oba jazyky podle standardů, jež budou platit v době, kdy vyjde. Výsledek držíte v ruce.

Typografické a jiné konvence

Domnívám se, že nemá smysl zdůrazňovat, že *kurzíva* znamená zdůraznění a `neproporcionální písmo` že používám k zápisu ukázek zdrojového textu, identifikátorů, klíčových slov apod. a výstupu počítače. To pozná každý průměrně inteligentní čtenář na první pohled (a programátoři bývají více než průměrně inteligentní).

Způsob a význam popisu *syntaktických konstrukcí*, který v této knize používám, je vysvětlen v kapitole 2.1 na str. 31.

[1] V hranatých závorkách jsou uvedeny odkazy na seznam literatury (str. 359).

C90 Pro jazyk C podle normy ISO 9899:1990 [2] používám zkratku C90.

C99 Pro jazyk C podle normy ISO 9899:1999 [3], která nahradila standard [2], používám zkratku C99.

C++03 Pro jazyk C++ podle standardu ISO 14882:2003 [1] používám zkratku C++03.

C++0x Pro jazyk C++ podle připravovaného standardu ISO, jehož vydání se očekává koncem roku 2011, používám zkratku C++0x.

Poděkování

Chci poděkovat všem, kteří mi při přípravě této knihy pomohli svými radami a připomínkami.

Přes veškerou péči, kterou jsem této knize věnoval, se v ní mohou vyskytnout chyby. Pokud na nějakou přijdete, dejte mi prosím vědět; na webové stránce `http://tjn.fjfi.cvut.cz/~virius/errata.htm` uveřejním opravu.

Miroslav Virius
miroslav.virius@fjfi.cvut.cz

1. Úvod

Je zvykem, že knihy o programovacím jazyce C nebo C++ začínají příkladem, který vypíše nějaký vtipný text a skončí. Pak následuje stručné vysvětlení jeho významu, informace o způsobu překladu apod. Autor si tím vytvoří rámec, který mu umožní předvádět příklady na spustitelných programech. I když v referenční příručce nic takového není nezbytné, přidržíme se této tradice, abychom usnadnili její používání i čtenářům, kteří tyto jazyky znají jen povrchně a potřebují je rychle začít používat.

1.1 První program

V tomto oddílu si ukážeme první programy v jazyce C++ a v C, seznámíme se s pravidly pro vytváření zdrojového textu, s postupem překladu atd. Protože některé z těchto věcí mohou do značné míry záviset na použitém překladači, uvedeme pouze základní informace; podrobnosti musíte hledat v dokumentaci ke svému překladači.
Zdrojový text našeho prvního programu v C++ je jednoduchý:

```
/* Soubor PrvniPlus.CPP
   První program v C++
 */
#include <iostream>                 // 4
using namespace std;                // 5
int main() {                        // 6
   cout << "Hello, World << endl"; // 7
   return 0;                        // 8
}                                   // 9
```

Podobný program v jazyce C může vypadat takto:

```
/* Soubor Prvni.c
   První program v jazyce C
 */
#include <stdio.h>
int main() {
   printf("Hello, World\n");   /* 6 */
   return 0;
}
```

Tento program uložíme do textového souboru. Pro programy v jazyce C++ se typicky používá přípona `.cpp`, pro programy v jazyce C přípona `.c`. Lze se ale setkat i s jinými – např. v některých unixových systémech se pro programy v C++ používají přípony `.C` nebo `.cc`
Podobně jako v jiných programovacích jazycích, i v C a v C++ platí, že tento soubor nesmí obsahovat formátování, tedy např. velikost a řez písma apod. Zpravidla jde o soubory v kódování ASCII (nebo v jiném jednobajtovém kódování, které váš počítač používá), dnešní překladače však už zpravidla akceptují zdrojové soubory v kódování Unicode.
Jestliže tento program přeložíme a spustíme, vypíše

```
Hello, World
```

Než se ale pustíme do překladu a sestavování tohoto programu, vysvětlíme si (bez nároku na úplnost) význam jeho jednotlivých částí.

1.1.1 Co je co

Pokud výslovně nezdůrazním něco jiného, týká se výklad jak C, tak C++. Veškeré pojmy z jazyků C a C++, které si zde naznačíme, budou znovu vysvětleny v referenční části této knihy. Začneme nepochybně nejnápadnější součástí zdrojového textu – komentářem.

Komentář

První tři řádky v obou programech představují *víceřádkový komentář*. Ten začíná dvojicí znaků `/*` (zapsaných bezprostředně zas sebou) a končí dvojicí znaků `*/`, opět zapsaných bezprostředně zas sebou. Překladač ignoruje tyto znaky a vše mezi nimi.
Vedle toho máme v C++ a v C99 k dispozici jednořádkový komentář. Ten začíná dvojicí lomítek, `//`, a končí na konci řádku.
Jednořádkové komentáře jsme v prvním výpisu použili k očíslování některých řádků.

Struktura programu a funkce main()

Programy v jazycích C a C++ jsou tvořeny posloupností deklarací. To zní možná podivně, ale v principu to znamená, že se skládají z deklarací funkcí,[1] typů a proměnných. Zjednodušeně řečeno, právě jedna z těchto funkcí musí mít jméno `main` a program začne prvním příkazem této funkce. Po skončení funkce `main()`[2] program skončí.
Podívejme se na první z výše uvedených výpisů.
Deklarace funkce `main()` začíná v prvním výpisu na 6. řádku. Nejprve je uvedena specifikace typu výsledku (zde je to typ `int`, tedy celé číslo se znaménkem), pak následuje identifikátor funkce a za ním prázdné závorky, které říkají, že tato funkce nemá žádné parametry. Tyto závorky jsou nezbytné.
Pak následuje otevírací složená závorka `{`, jíž začíná tělo funkce. Toto tělo končí uzavírací složenou závorku `}` na 9. řádku. Tělo funkce tvoří příkazy uzavřené mezi těmito závorkami.

---

[1] V C a v C++ se nerozlišuje mezi funkcí a procedurou; jakýkoli podprogram se nazývá „funkce".

[2] Jméno (identifikátor) této funkce je `main`. V textu knihy budeme k identifikátorům funkcí připisovat závorky, aby bylo na první pohled jasné, že se jedná o funkci. Bude-li to potřebné, uvedeme v závorkách i typy parametrů.

Řetězcová konstanta

V 7. řádku souboru `PrvniPlus.CPP` najdeme zápis `"Hello, World"`. To je řetězcová konstanta – posloupnost znaků uzavřená mezi uvozovky.

Výstup v C++

Výstup je jazycích C a C++ je realizován pomocí funkcí nebo objektů, které se nacházejí ve standardní knihovně. Tzv. *standardní výstupní proud* (typicky přesměrovatelný výstup na konzolu) je v C++ představován objektem `cout`. Vystupující hodnoty do tohoto proudu *vkládáme* pomocí operátoru <<. Zápisem

```
cout << "Hello, World" << endl;
```

do tohoto proudu vkládáme postupně dvě věci: zaprvé řetězcovou konstantu představující vypisovaný text a zadruhé tzv. manipulátor `endl`, který způsobí přechod na nový řádek.

Výstup v C

Podívejme se nyní na druhý výpis, tj. na soubor `Prvni.c`. Jazyk C používá pro vstupní a výstupní operace knihovní funkce. Jednou z nich je funkce `prinft()`, která vypíše zadaný znakový řetězec do standardního výstupu. Tuto funkci lze volat i s více parametry.

Hlavičkové soubory

Ani v C, ani v C++ *nesmíme použít nic, co překladač nezná*. To platí i pro součásti standardních knihoven – překladač je nezná, dokud mu o nich neřekneme. Proto je třeba překladači předem oznámit, že budeme používat objekt `cout`, resp. funkci `printf()`, které jsou definovány ve standardní knihovně. Abychom však nemuseli jejich deklarace vypisovat do svých programů ručně, jsou připraveny v tzv. hlavičkových souborech. Tyto hlavičkové soubory vkládáme do svých programů tzv. direktivou preprocesoru `#include`, tedy např. zápisem

```
#include <iostream>
```

Tuto direktivu musíme zapsat na samostatný řádek. Jméno souboru uzavřeme mezi lomené závorky tvořené znaky „menší než" a „větší než". Poznamenejme, že standardní hlavičkové soubory v jazyce C++ nemají žádnou příponu.
Hlavičkový soubor `<iostream>`[3] obsahuje základní nástroje pro vstupní a výstupní operace.
V jazyce C mají standardní hlavičkové soubory typicky příponu `.h`. Soubor `<stdio.h>` obsahuje nástroje jazyka C pro vstupní a výstupní operace.
Poznamenejme, že standardní knihovnu jazyka C můžeme používat i v C++. To znamená, že uvedený program můžeme přeložit i překladačem C++. Překladače C++ mají vlastní verzi hlavičkových souborů z jazyka C. Jejich obsah je prakticky stejný jako v C90, jazyk C++ však umožňuje používat je dvojím způsobem:

[3] Jméno tohoto souboru je `iostream`. Lomené závorky připojujeme, abychom zdůraznili, že jde o hlavičkový soubor. S touto praxí se běžně setkáváme nejen v odborné literatuře, ale i v textu standardu těchto jazyků.

- Stejně jako v jazyce C. Můžeme tedy napsat
  `#include <stdio.h>`
  a pak můžeme používat funkci `printf()`.
- Můžeme pro hlavičkový soubor použít jméno, které vznikne z původního jména v jazyce C připojením znaku `c` před první písmeno názvu a vynecháním přípony `.h` (tedy například `<cstdio>` místo `<stdio.h>`). V takovém případě budou všechny identifikátory, deklarované v tomto souboru, ležet ve jmenném prostoru `std`, takže budeme muset psát např. `std::printf()` místo pouhého `printf()` nebo použít deklaraci nebo direktivu `using` (viz dále).

Blok příkazů

Tělo funkce (nejen funkce `main()`) představuje z hlediska syntaxe jazyků C a C++ *blok*. To je skupina příkazů ohraničená složenými závorkami `{` a `}`. Bloky příkazů obsahují příkazy, které patří k sobě, jako například příkazy tvořící tělo funkce (viz kapitola 10.1.1), tělo cyklu (viz kapitola 9.4) nebo větve příkazu `if` (viz kapitola 9.3.1). Z hlediska syntaxe se blok příkazů chová jako jeden příkaz.

Příkazy, zápis programu

Každý příkaz v programu (kromě bloku) musí být ukončen středníkem. To znamená, že středník je součástí příkazu.
Funkce `main()` v obou výše uvedených příkladech obsahuje dva příkazy. Oba musí být ukončeny středníkem (i poslední příkaz před uzavírací závorkou `}`). Příkaz může být zapsán i na několik řádků a naopak, na jednom řádku může být několik příkazů.
Poznamenejme, že direktiva `#include` se nepovažuje za příkaz. Nepíšeme za ni středník a musí být na samostatném řádku.

Příkaz return

Na 8. řádku funkce `main()` najdeme příkaz

- `return 0;`

Tento příkaz určuje místo, kde provádění funkce končí, a určuje její návratovou hodnotu (co bude výsledkem volání funkce). V případě funkce `main()` to je zpravidla kód ukončení programu, hodnota, kterou lze využít v některých prostředích k testům, zda program skončil úspěšně nebo chybou (a jakou).
Příkaz `return` lze použít na kterémkoli místě v těle funkce.

Jmenné prostory (jen C++)

Jmenné prostory (též *prostory jmen*) jsou jakási „příjmení", které lze připojit k identifikátorům. Je to nástroj, který má pomoci vyhnout se konfliktům jmen v rozsáhlých projektech a při používání několika různých knihoven. Plné jméno funkce, typu, proměnné nebo šablony, deklarované ve jmenném prostoru, se skládá z identifikátoru, ke kterému je prostřednictvím operátoru `::` připojeno jméno jmenného prostoru.
Například všechny identifikátory ze standardní knihovny jazyka C++ leží ve jmenném prostoru `std`. To znamená, že plné jméno objektu představujícího standardní výstupní proud je `std::cout`, plné jméno manipulátoru pro přechod na nový řádek je `std::endl` atd.

Jestliže nehrozí konflikt jmen, můžeme překladači říci, že budeme jméno prostoru jmen vynechávat. K tomu slouží direktiva `using`, která má tvar
using namespace *jméno_jmenného_prostoru* **;**
Pak můžeme psát jen `cout`, `endl` atd.

1.1.2 Překlad a sestavení

Zdrojový program je třeba přeložit do strojového kódu a vytvořit spustitelný program. Přitom lze postupovat několika způsoby:

- Překladač lze spustit z příkazového řádku.
- Překladač lze spustit z integrovaného vývojového prostředí (IDE).
- Překladač lze spustit pomocí utility `make` nebo jiného podobného nástroje (`Ant`, `nmake` apod.).

Zde se podrobněji zastavíme pouze u první možnosti.

Překlad z příkazového řádku: Jeden zdrojový soubor

Pro určitost vezmeme překladač `g++`, který je součástí vývojového nástroje MinGW32. Poznamenejme, že MinGW32 je zdarma a je šířen pod licencí GNU.
Překlad našeho programu, který se skládá z jediného zdrojového souboru, obstará příkaz

```
g++ -o Prvni.exe PrvniPlus.cpp
```

Výsledkem bude spustitelný soubor `Prvni.exe`. Po spuštění příkazem

```
Prvni
```

vypíše

```
Hello, World
```

Přepínač `-o Prvni.exe` určuje jméno výsledného spustitelného souboru. U některých překladačů ho lze vynechat a překladač použije jméno překládaného souboru. (V případě `g++` by však měl výsledný soubor jméno `a.exe`.)

Určujeme jazyk zdrojového souboru

Většina překladačů rozlišuje programovací jazyk podle přípony zdrojového souboru. Přípona `.c` typicky znamená jazyk C, přípona `.cpp` jazyk C++. Podrobnosti najdete v dokumentaci ke svému překladači.
Překladač `g++`, který zde budeme převážně používat, se však chová jinak. Současná verze tohoto překladače zachází se všemi zdrojovými soubory implicitně jako s programem v jazyce C++03. Chceme-li, aby daný soubor překládal jako program v jazyce C90, musíme mu v příkazovém řádku zadat volbu `-x c`. To znamená, že musíme napsat např.

```
g++ -x c -o Prvni.exe Prvni.c
```

Chceme-li, aby tento překladač akceptoval konstrukce, které nejsou v C90, ale jsou v C99, musíme to explicitně povolit dalším přepínačem `-std=c99` v příkazovém řádku.
Chceme-li, aby tento překladač – při překladu programu v C++ – akceptoval vybrané konstrukce zavedené v C++0x, musíme v příkazovém řádku zadat volbu `-std=c++0x`.

Program složený z několika souborů

Jestliže se náš program skládá z několika zdrojových souborů, můžeme je v příkazovém řádku vypsat všechny. Například skládá-li se ze souborů `Prvni.cpp` a `Pomocny.cpp`, napíšeme

```
g++ -o Prvni.exe Prvni.cpp Pomocny.cpp
```

Oddělený překlad

Jazyky C a C++ používají model odděleného překladu. To znamená, že různé zdrojové soubory téhož programu lze překládat samostatně a pak je sestavit.
Chceme-li přeložit pouze soubor `Prvni.cpp`, napíšeme

```
g++ -c -o Prvni.obj Prvni.cpp
```

Přepínač `-c` přikazuje překladači, že má zdrojový soubor přeložit, nikoli však sestavovat. Přepínač `-o` opět určuje jméno výstupního souboru. Tím vznikne tzv. *relativní soubor* (object file) `Prvni.obj`, který obsahuje překlad zdrojového souboru do strojového kódu. Neobsahuje však nic více, tzn., nejsou tam vyřešeny odkazy na funkce a proměnné definované v jiných souborech nebo v knihovně.
Poznamenejme, že u většiny překladačů lze přepínač `-o` vynechat, překladač mu automaticky dá jméno shodné se jménem zdrojového souboru a pod Windows také příponu `.obj`. Překladač `g++` mu dá příponu `.o`.
Relativní soubory je třeba sestavit ve výsledný spustitelný soubor. K tomu slouží specializovaný program, označovaný jako *sestavovací program* nebo *linker*. U mnoha dnešních vývojových nástrojů (včetně MinGW) je však linker součástí překladače. Chceme-li sestavit výsledný program z relativních souborů `Prvni.obj` a `Pomocny.obj`, napíšeme

```
g++ -o Prvni.exe Prvni.obj Pomocny.ob
```

Vývojová prostředí a projekty

Integrovaná vývojová prostředí (IDE) obvykle spojují editor, překladač, ladicí nástroje, nápovědu a další pomocné programy, které usnadňují vytváření aplikací. Zpravidla pracují s tzv. *projektem*.
Na projekt se můžeme dívat jako na seznam souborů, z nichž se má vytvořit výsledná aplikace. V IDE lze pomocí dialogových oken nastavovat volby pro překlad nejen celého projektu, ale někdy i jednotlivých souborů. Podrobnosti o tom, jak definovat projekt a jak ho přeložit, je třeba hledat v dokumentaci k danému vývojovému prostředí.

Utilita MAKE

Jiným nástrojem pro automatizaci překladu je utilita `make`. To je pomocný program volaný z příkazového řádku, který vyhledá soubor se jménem `makefile` obsahující návod pro vytvoření požadovaného spustitelného souboru. V nejjednodušší podobě se skládá z komentářů začínajících znakem # a končících na konci řádku, z popisu závislostí a z pravidel, podle kterých se mají požadované soubory vytvářet. Popis závislostí začíná vždy jménem souboru, za ním následuje dvojtečka a výčet souborů, na nichž daný soubor závisí. (To znamená, že když se změní některý z uvedených souborů, musí se znovu závislý soubor vytvořit. Například zápis

- `prvni.exe: prvni.obj pomocny.obj`

říká, že soubor `prvni.exe` závisí na souborech `prvni.obj` a `pomocny.obj`. Za tímto řádkem musí následovat pravidlo pro vytvoření souboru `prvni.exe`, tedy volání překladače:

- `g++ -o prvni.exe prvni.obj pomocny.obj`

Podobně je třeba definovat i závislosti souborů `prvni.obj` a `pomocny.obj` a pravidla pro jejich vytvoření. Celý soubor `makefile` pro překlad programu složeného ze zdrojových souborů `prvni.cpp` a `pomocny.cpp` může vypadat takto:

```
# Soubor makefile
# Popis vytvoření souboru prvni.exe
# ze souborů prvni.cpp a pomocny.cpp
# pomocí překladače g++  (MinGW32)
prvni.exe: prvni.obj pomocny.obj
  g++ -o prvni.exe prvni.obj pomocny.obj
prvni.obj: prvni.cpp
  g++ -c -o prvni.obj prvni.cpp
pomocny.obj: pomocny.cpp
  g++ -c -o pomocny.obj pomocny.cpp
```

Řádek se závislostmi začíná zpravidla na počátku řádku, řádek s pravidlem je obvykle odsazen o 1 zarážku tabulátoru. Překlad spustíme z příkazového řádku příkazem

- `make`

Utilita `make` vždy nejprve vyhledá soubor jménem `makefile` a pak podle informací v něm kontroluje, zda se změnil některý ze souborů, na nichž jiné soubory závisí. Pokud ano, vytvoří znovu závislý soubor. To znamená, že překládá jen ty zdrojové soubory, které se od posledního překladu změnily.

- Nehodí-li se nám jméno `makefile`, lze použít jakékoli jméno s příponou `.mak` – např. `prvni.mak`. Utilitu `make` pak zavoláme příkazem `make` *jméno_souboru*, tedy např.
  ```
  make prvni.mak
  ```
- V souboru `makefile` lze definovat makra, implicitní pravidla apod. a tak ušetřit v případě rozsáhlých projektů mnoho psaní. To však přesahuje možnosti této knihy; podrobnosti lze najít např. na adrese [9].

1.2 Programovací jazyky C a C++

I když jsou si jazyky C a C++ velmi blízké, je třeba zdůraznit, že C++ není pouhé rozšíření jazyka C, neboť při návrhu C++ byla sice kompatibilita s jazykem C brána v potaz, nikoli však za každou cenu. V jazyce C existují konstrukce, které nejsou v C++ dovoleny; vedle toho narazíme i na konstrukce, které jsou sice správné v obou jazycích, ale v každém z nich mohou mít jiný význam. Je jich ovšem velmi málo a zpravidla jde o konstrukce, které nepatří do dobrého programátorského stylu. V textu na ně samozřejmě upozorním.

Jazyk C navrhl a implementoval na počátku sedmdesátých let 20. století Denis Ritchie a použil ho při implementací jedné z verzí operačního systému Unix. Brzy se ukázalo, že jde

o velice příjemný a přitom i mocný programovací jazyk, a začal být používán k nejrůznějším účelům. S tím ovšem procházel i řadou změn – nejstarší verze tohoto jazyka např. místo operátorů +=, −= apod. používaly operátory =+, =− atd. V roce 1978 vydal D. Ritchie spolu s B. W. Kernighanem proslulou knihu *The C Programming Language* [6], která se stala na dlouhou dobu neoficiálním standardem jazyka C. V roce 1990 byl jazyk C poprvé standardizován na mezinárodní úrovni.
Jedním z prvních kroků na cestě k jazyku C++ bylo tzv. „C s třídami". V roce l985 vznikla verze, kterou již lze považovat za řádný objektově orientovaný programovací jazyk. Známe ji pod názvem C++. Jazyk C++ od té doby dozrál a je stejně úspěšný jako jazyk C.
Jedním z důvodů úspěchu je jistě téměř úplná kompatibilita C++ s C, která programátorům pracujícím v Céčku usnadnila přechod k objektově orientovanému programování a zaručila použití již existujících kódů napsaných v jazyce C.

1.2.1 Standardy

Prvním, neoficiálním standardem jazyka C se stalo první vydání už zmíněné knihy B. W. Kernighana a D. Ritchieho *The C Programming Language* [6].[4] Zde popsaná verze jazyka bývá označována jako „jazyk C podle Kernighana a Ritchieho", případně zkratkou K&R.
Oficiálně byl jazyk C poprvé standardizován na národní úrovni v USA v r. 1989. Tento dokument ANSI měl číslo X3J11/90-013.
O rok později, v r. 1990, byl přijat mezinárodní standard ISO/IEC 9899:1990 [2]. V této knize pro něj budu používat zkratku *C90*. Jazyk podle tohoto standardu je dnes běžně používán, implementují ho prakticky všechny běžně dostupné překladače. Poznamenejme, že zároveň s tím byl v USA stažen standard ANSI a byl nahrazen standardem ISO. Na tom nic nemění skutečnost, že mnozí výrobci překladačů se stále odvolávají na standard ANSI – to, co dodávají, je (nebo by měl být) překladač vyhovující mezinárodnímu standardu ISO.
V roce 1995 byl přijat technický dodatek k tomuto standardu, který především rozšířil jazyk C o nástroje pro práci s tzv. širokými (tj. dvoubajtovými) znaky.
V roce 1999 byla přijata druhá verze standardu jazyka C, označená ISO/IEC 9899:1999 [3], kterou budu označovat zkratkou *C99*. Přinesla řadu významných novinek – především do jazyka zahrnula nejběžnější rozšíření a odstranila některé potenciálně nebezpečné rysy.
Jazyk C++ byl od svého vytvoření v první polovině osmdesátých let spravován B. Stroustrupem pod záštitou firmy AT&T. Program v C++ byl původně překládán nejprve do jazyka C a teprve z něj překladačem jazyka C do strojového kódu. Proto se první verze tohoto jazyka označovaly *Cfront*. Do počátku devadesátých let bylo uvolněno několik verzí, z nichž za nejvýznamnější bývají pokládány verze Cfront 1.0 (objektové rozšíření jazyka C), Cfront 1.2, která přinesla mj. přetěžování operátorů, dále verze Cfront 2, jež přinesla vícenásobné dědění; verze Cfront 2.1 připojila šablony.
V roce 1991 vyšla kniha *The Annotated C++Reference Manual* [8], označovaná často zkratkou ARM, která se stala základem návrhu standardu jazyka. Popisovala vedle šablon výjimky, jmenné prostory a dynamickou identifikaci typů.
V roce 1998 byl přijat mezinárodní standard jazyka C++ ISO/IEC 14882:1998. Tento standard doplnil, upřesnil a v některých detailech i změnil jazyk popisovaný v ARM a přidal standardní

[4] Poznamenejme, že druhé vydání této knihy z roku 1988 popisuje tehdy připravovaný standard ANSI.

šablonovou knihovnu. V roce 2003 byla publikována nová verze standardu [1], která odstranila některé drobné chyby a upřesnila některé drobnosti; nepřinesla však žádnou podstatnou změnu.
Poznamenejme, že jazyk C++ podle ISO/IEC 14882:2003, který zde budu označovat zkratkou C++03, se opírá o jazyk C podle ISO 9899:1990, tedy o C90.
Jazyk C++03 ovšem obsahoval několik problematických míst:

- Především jeho standardní knihovna nebyla úplná, neboť v knihovně kontejnerů chybí mj. implementace hešových tabulek nebo uspořádané *n*-tice hodnot, nástroje pro práci s regulárními výrazy atd.
- Některé syntaktické rysy, které byly do standardu C++03 zahrnuty, se neosvědčily, protože vedly k příliš provázanému kódu (to se týká např. specifikace výjimek v deklaraci funkce).
- Některé syntaktické rysy znemožňovaly nebo ztěžovaly diagnostiku určitého druhu chyb. To se týká např. chyb možných při deklaraci virtuální metody v odvozené třídě.
- Chyběla možnost definovat omezení formálních typů v deklaraci šablony atd.

Proto byl již delší dobu připravován nový standard, který by měl tyto problémy odstranit. Jeho vydání se očekávalo v roce 2008 nebo 2009, proto se pro něj vžilo označení C++0x, a toto označení se používá stále, i když je nyní jasné, že nový standard bude publikován nejspíš koncem roku 2011.
V této knize budeme hovořit převážně o jazyku C++ podle standardu C++03 a C++0x. Přitom ale mějte na paměti, že v době, kdy tento text připravuji k vydání, ještě nebyl nový standard publikován, takže jsem čerpal informace z návrhu standardu publikovaného na počátku roku 2011; ty jsem pak korigoval podle konečného návrhu [4] publikovaného začátkem května 2011. V současné době implementují některé překladače vybrané novinky C++0x, žádný z běžných překladačů však neimplementuje všechny, a ani ty, které implementuje, neodpovídají vždy budoucímu standardu.

1.3 Objektově orientované programování

Objektově orientované programování (dále též OOP) neznamená, že jednoduše využijeme pohodlí, které skýtá C++ oproti jazyku C, nebo že prostě budeme využívat knihovny tříd namísto knihoven napsaných v Céčku. V tomto smyslu nelze například označit náš první program v jazyce C++ za objektově orientovaný, přestože jsme v něm použili instance `cout` třídy `ostream` a přetížený operátor << místo funkce `printf()` z jazyka C.
Objektově orientované programování vlastně začíná tím, že si definujeme třídy a že smysluplně využíváme zapouzdření a polymorfismu. Objektově orientované programování se snaží popsat problém, který program řeší, pomocí vhodně navržených tříd (objektových typů) a jejich instancí (proměnných, konstant atd.). Tím, že nabízí třídy, které lze navíc seskupit do hierarchií příbuzných typů, usnadňuje programátorovi přemýšlení o problému a usnadňuje nejen analýzu řešeného problému, ale i návrh a ladění programu. Snadno přitom vznikají knihovny tříd, které pak lze snadno a bezpečně opakovaně používat.
Všechny podstatné principy objektově orientovaného programování směřují k urychlení procesu vypracování programu. Prostředkem vedoucím k tomuto cíli může být

- opakované užití kódu, který již byl odladěn;

- rychlejší návrh datových typů s využitím skládání, dědění a polymorfismu;
- menší náchylnost k výskytu chyb, a to nejen díky využití zapouzdření;
- lepší čitelnost programu, daná objektově orientovaným způsobem návrhu.

Tam, kde jsou tato kritéria nepodstatná, není důvod neimplementovat program v jazyce C, což sice znamená, že se vzdáme objektově orientovaných principů, avšak někdy tím získáme menší a rychlejší program.

1.3.1 Základní pojmy OOP

Chcete-li využít všech možností, které OOP v C++ nabízí, je důležité osvojit si napřed objektově orientovaný způsob myšlení a porozumět jeho nejdůležitějším principům, a teprve pak se snažit pochopit, jaký je jejich význam ve struktuře jazyka. Zejména pro nováčka není proniknutí do způsobu uvažování používaného v objektově orientovaném programování jednoduchou záležitostí, neboť to znamená zvládnout zcela novou terminologii. Bohužel, definice mnoha pojmů nejsou v literatuře sjednoceny. Další potíže pak má na svědomí používání anglické terminologie v češtině a nejednotnost překladů.
Abych tomuto zmatení pojmů alespoň trochu zabránil, uvedu v následujících odstavcích nejdůležitější pojmy, krátce je vysvětlím a upozorním na synonyma a na odlišné definice.

- **Třída (objektový typ):** Znamená datový typ, který v programu reprezentuje třídu objektů z reálného světa (přesněji z oblasti řešeného problému), je její abstrakcí. (Do programové reprezentace přeneseme pouze ty vlastnosti, které jsou pro nás podstatné, ostatní vynecháme – „abstrahujeme od nich". Proto se o třídách také hovoří jako o *abstraktních datových typech*, i když ve skutečnosti nejsou o nic abstraktnější, než řekněme typ `int` představující celá čísla.)
- **Instance (objekt):** Znamená proměnnou, konstantu nebo parametr objektového typu.
- **Metoda**: Operace s instancí nějaké třídy nebo s třídou jako celkem. Vyjadřuje některou ze součástí chování třídy nebo jednotlivé instance.
- **Členská funkce:** Implementace metody v programu. (V C++ se ale pojmy metoda a členská funkce zpravidla pokládají za synonyma.)
- **Datové složky:** Data, uložená v jednotlivých instancích dané třídy, nebo ve třídě jako celku. Uchovávají stav instance.
- **Abstraktní třída:** Třída, v níž některé z metod nelze implementovat a jejíž instance nemá smysl vytvářet. Typicky jde o třídu, která shrnuje řadu konkrétních pojmů. Představme si např., že implementujeme třídy `Bod`, `Úsečka` a `Kružnice`. Při analýze zjistíme, že mají mnoho společného, a tyto společné vlastnosti shrneme do společného předka těchto tříd, do třídy `GrafickýObjekt`. V programu budeme vytvářet mnoho bodů, úseček i kružnic, ale žádný grafický objekt. Ten ani neumíme nakreslit, zatímco bod, úsečku i kružnice nakreslíme bez problémů. To znamená, že metodu `nakresli()` ve třídě `GrafickýObjekt` nemá smysl implementovat. (Musíme ji tam ale deklarovat, abychom mohli s body, úsečkami i kružnicemi zacházet jednotně, jako s grafickými objekty). `GrafickýObjekt` je tedy abstraktní třída. (V C++ se jako abstraktní označují třídy, které mají alespoň jednu čistě virtuální metodu.)
- **Abstraktní metoda:** Metoda v abstraktní třídě, kterou nemá smysl definovat, neboť je „příliš abstraktní", kterou ale musíme deklarovat, neboť v odvozených třídách ji budeme

používat. Typickým příkladem je metoda `nakresli()` ve třídě `GrafickýObjekt` z předchozího bodu.

- **Zapouzdření:** Vlastně ukrývání implementace. S instancemi třídy i s třídou jako celkem zacházíme jen prostřednictvím jejich veřejně přístupných metod. Tyto metody tvoří uživatelské rozhraní třídy. Využíváme-li důsledně zapouzdření, pak nezměníme-li rozhraní třídy, nezpůsobí změna implementace třídy nutnost měnit implementaci zbytku programu.
- **Dědění:** Mechanizmus, který umožňuje od jedné třídy odvodit jinou, příbuznou třídu. Třídu, od které odvozujeme, označujeme jako *předka*, *rodiče* nebo *bázovou třídu*, *odvozenou třídu* označujeme také jako *potomka* nebo *dceřinnou třídu*. Potomek je vždy zvláštním případem předka. Odvozené třídy přebírají („dědí") vlastnosti bázových tříd (obsahují stejné datové složky i metody.) K nim mohou připojit další, své vlastní datové složky a metody. Mohou také *překrýt* některé z metod předka.
- **Dědická hierarchie:** Skupina tříd, která vznikla odvozováním od společných předků.
- **Polymorfizmus (mnohotvarost):** Obecně znamená, že s instancemi různých tříd můžeme zacházet stejným způsobem. V C++ dosahujeme polymorfizmu pomocí dědění. Jedním ze základních principů OOP je, že potomek – instance odvozené třídy – může vždy zastoupit předka, tj. instanci bázové třídy. To lze říci i jinak: Objekt odvozené třídy je zároveň objektem bázové třídy. (Je-li bázová třída `pes` a odvozená třída `jezevčík`, platí, že `jezevčík` je `pes`.) Potomek je tedy podtypem předka. V C++ je polymorfizmus implementován pomocí tzv. virtuálních metod.
- **Skládání (kompozice):** Zdaleka nejčastější způsob odvozování nových tříd. Spočívá v tom, že nová třída využívá služeb tříd, které již existují – buď tak, že obsahuje datovou složku existující třídy, nebo že obsahuje ukazatel či referenci na instanci jiné třídy.
- **Test je–má:** Umožňuje určit, zda máme novou třídu navrhnout jako potomka existující třídy nebo zda máme použít skládání. Položíme si otázky: *Je nová třída zvláštním případem třídy, kterou chceme použít jako předka? Nebo má (využívá) složku tohoto typu?* Pouze je-li odpověď na první otázku kladná, má smysl uvažovat o dědění. Ovšem ani kladná odpověď na první otázku nezaručuje, že dědění je správnou alternativou.
- **Objektově orientovaný program:** V teorii OOP se zpravidla říká, že objektově orientovaný program se skládá z objektů, které si navzájem posílají zprávy. V C++ to znamená, že objekty volají metody (své i jiných instancí). Řešení problémů v OOP se skládá z určení objektů, jejich implementace ve formě tříd a další práce s těmito třídami. Nejobtížnější přitom zpravidla je právě nalezení a implementace tříd, které musí správně reprezentovat třídy objektů z řešeného problému. Zbytek práce při vývoji programu se tím zjednoduší. Přitom programátor se již nemusí nadále starat o to, jak jsou třídy implementovány (může se na ně dívat jako na černé skříňky, které poskytují určité služby). Tak mohou vzniknout knihovny tříd, které se dají opakovaně použít. Objektově orientovaný přístup se tím ve srovnání s prací s elementárními datovými typy více blíží lidskému způsobu uvažování, které se rovněž vyznačuje tendencí ke klasifikaci (rozřazování pojmů do tříd).
- **Překrytí:** Nová definice zděděné funkce v odvozené třídě. Smyslem je, aby odvozená třída reagovala na odpovídající volání funkce správným způsobem. Má-li být implementace korektní, musíme překryté funkce deklarovat jako virtuální. Překrytí je tedy principem, který umožňuje polymorfismus. (Někdy se hovoří také o *předefinování*.)
- **Přístupová práva:** Třída může označit některé své součásti jako veřejně přístupné a některé jako soukromé. Veřejně přístupné součásti může používat kdokoli (kterákoli část

programu); tyto složky tvoří *rozhraní třídy*. Soukromé složky smí používat jen třída sama (typicky její metody). Představují *implementační detaily*, které třída skrývá před uživatelem (programátorem, který třídu ve svém programu využívá). V některých programovacích jazycích jsou k dispozici i další úrovně řízení přístupu ke složkám mezi těmito dvěma extrémy. Např. v C++ mohou být některé složky označeny jako chráněné; ty smí používat pouze třída sama a třídy od ní odvozené (potomci). Chráněné složky představují *rozšířené rozhraní třídy pro odvozování potomků*.

1.3.2 Některé další pojmy

- **Přetěžování funkcí:** Definice několika funkcí se stejným identifikátorem. Překladač je bude při volání rozlišovat podle počtu a typu parametrů. Příležitostně se tento termín používá i v souvislosti s hierarchií tříd a polymorfismem, přestože v této souvislosti je přesnější a intuitivně srozumitelnější pojem *překrytí* nebo *předefinování*.
- **Přetěžování operátorů:** Rozšíření operátoru na uživatelem definované typy (objektové a výčtové). I zde překladač určuje podle počtu a typu parametrů, kterou operaci chceme provést.
- **Rozhraní (interface):** Též *protokol* – popis služeb, které funkce, modul nebo třída poskytuje, a způsobu, jak ji o tyto služby požádat. Rozhraní funkce je například určeno jejími parametry a vracenou hodnotou; součástí rozhraní mohou být i globální proměnné (i když to je z hlediska bezpečnosti velice nešťastné řešení). Rozhraní třídy pro celý zbytek programu představují především metody nebo datové složky označené jako veřejné (`public`) a funkce specifikované jako spřátelené (`friend`). Složky označené chráněné (`protected`) představují rozšířené rozhraní pro odvozování potomků. Rozhraní by na jedné straně mělo být co možná nejmenší, na druhé straně musí být úplné, tedy obsahovat vše, co je pro smysluplné využití třídy nezbytné.
- **Šablona:** Nástroj, který umožňuje v C++ deklarovat celou množinu objektových typů nebo celou množinu funkcí, které se liší jen v některých typech nebo konstantách („parametrech šablony").
- **Výjimka:** Mechanizmus pro ošetřování chyb, které nastanou za běhu programu. Umožňuje přenést řízení z místa, kde byla chyba detekována, do místa, kde ji lze ošetřit (to může být i v jiné funkci). *Pozor:* Termínem výjimka označujeme jak samotnou chybu, tak i objekt, který nese o této chybě informace z místa vzniku do místa ošetření.
- **Atribut:** Ve starší literatuře se tento termín používal pro složky objektových typů (jak pro datové složky, tak pro metody). V 90. letech se používal (a dodnes občas používá) už pouze pro datové složky. V současné době se však prosazuje jako označení pro dodatečné informace o deklarovaném objektu, které zadáváme v deklaraci. V tomto smyslu se používá také v C++0x.

Termín objekt se také používá ve významu manipulovatelná oblast paměti (tedy vlastně proměnná jakéhokoli typu nebo funkce). V takovém případě nemá samozřejmě s OOP nic společného. I tento význam budu v této knize používat, vždy však na to výslovně upozorním.

2. Základní pojmy

Tato kapitola vysvětluje způsob popisu jazyků C a C++ používaný v této knize, poskytuje přehled klíčových slov, informace o lexikální konvenci, o způsobu překladu programu v C a v C++ a podobné.

Poznamenejme, že v této kapitole používáme termín objekt výhradně ve smyslu *manipulovatelná oblast paměti*.

2.1 Popis jazyků C a C++

Při popisu libovolného programovacího jazyka se rozlišují *terminální* a *neterminální* symboly. Terminální symbol je např. klíčové slovo, operátor apod. – něco, co lze do programu přímo opsat; neterminální symbol představuje pojem, který je třeba dále vysvětlit.
Základní způsob popisu bude vypadat takto:

- V záhlaví popisu bude název vysvětlovaného neterminálního symbolu, následovaný dvojtečkou.
- Pak bude následovat popis vysvětlovaného neterminálního symbolu – vlastně jeho rozklad na posloupnost terminálních a neterminálních symbolů.
- Terminální symboly budeme zapisovat **tučně**, neterminální symboly *kurzivou*.
- Popis jednoho neterminálního symbolu může mít několik alternativ. Jednotlivé možnosti budou uvedeny odrážkou „•".
- Index $_{nep}$ označuje „nepovinné" součásti popisované konstrukce, tj. součásti, které lze vynechat. Význam popisované konstrukce se tím může, ale nemusí změnit.
- Dvě bezprostředně za sebou následující lomítka uvádějí komentář, který končí na konci řádku.

Například popis

příkaz_while:

- **while (** *podmínka* **)** *příkaz*

znamená, že *příkaz_while* začíná terminálním symbolem (zde klíčovým slovem) `while`, za nímž následuje levá kulatá závorka, *podmínka*, pravá kulatá závorka a *příkaz*. Neterminální symboly *podmínka* a *příkaz* musí být vysvětleny jinde.
V některých situacích bude výhodnější nahradit takovýto popis prostým výčtem (pak uvedeme v záhlaví popisu *jeden z*) nebo slovním opisem.

Poznamenejme, že nebude-li hrozit nedorozumění, budeme v neterminálních symbolech vynechávat spojovací podtžítko; např. v záhlaví syntaktického popisu budeme psát *příkaz while* místo *příkaz_while*.

2.2 Množina znaků

Množinu znaků, které lze používat v programech v C++ se skládá ze čtyř základních skupin:
znak:
- *písmeno*
- *číslice*
- *speciální_znak*
- *bílý_znak*

Zde říkáme, že znak je *písmeno* nebo *číslice* nebo *speciální_znak* nebo *bílý_znak*. Tyto neterminální symboly je třeba dále definovat.

písmeno: jedno z
- *univerzální_jméno_znaku*
- **a b c d e f g h i j k l m n o p q r s t u v w x y z**
- **A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Pod pojmem písmeno se tedy skrývá buď *univerzální jméno znaku*, nebo malé nebo velké písmeno anglické abecedy. O univerzálních jménech znaků budeme hovořit dále (viz kapitola 2.2.1).

číslice: jedna z
- **0 1 2 3 4 5 6 7 8 9**

speciální_znak: jeden z
- **_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '**

bílý_znak: jeden z
- *mezera*, *horizontální_tabulátor*, *vertikální_tabulátor*, *konec_řádku*, *konec_stránky*

Definice číslice a speciálního znaku již obsahují výčty terminálních symbolů. To znamená, že uvedené znaky se mohou objevovat v programech a budou mít pro překladač nějaký význam.
- Tzv. základní množina znaků je tvořena písmeny (kromě univerzálních jmen znaků), číslicemi, speciálními znaky a bílými znaky.
- Budeme-li hovořit o „velikosti" písmen, budeme tím myslet, zda se jedná o velká či malá písmena.
- Termín číslice zde znamená číslice desítkové soustavy. Budeme-li potřebovat číslice šestnáctkové (hexadecimální) soustavy, výslovně to zdůrazníme. (Jako číslice s významem 10–15 se používají písmena A–F, příp. a–f. Je jedno, zda použijeme velká nebo malá písmena.)
- Pojmenování, která budeme pro některé znaky používat, shrnuje tab. 2.1.

Tabulka 2.1: Jména některých znaků

( )	okrouhlé (kulaté) závorky	[ ]	hranaté závorky
{ }	složené závorky	_	podtržení (podtržítko)
#	mříž (kriminál)	&	ampersand
^	stříška	|	svislice
/	lomítko	\	obrácené lomítko
~	vlnovka (tilda)	"	uvozovka
'	apostrof	%	procento
< >	lomené závorky		

2.2.1 Univerzální jména znaků

Univerzální jména znaků poskytují způsob, jak zapsat znaky dostupné v kódování Unicode (ISO 10646). Standard [1] v dodatku E přesně vyjmenovává, o jaké znaky jde; zpravidla však naprosto stačí vědět, že jde o malá a velká písmena tak, jak je chápeme my, a to včetně písmen s háčky, čárkami, kroužky, vokáni, přehláskami, akcenty atd. v latince, cyrilici, řeckém písmu, hebrejském písmu, arménském písmu, arabském písmu, sanskrtu, písmu podle japonského průmyslového standardu a v řadě dalších.

univerzální jméno znaku:
- **\u** *hex_čtveřice*
- **\U** *hex_čtveřice hex_čtveřice*

V této definici *hex_čtveřice* znamená čtveřici hexadecimálních číslic.
- Je-li `H` libovolná hexadecimální číslice, znamená `\uHHHH` totéž co `\U0000HHHH`.
- V zápisu `\UHHHHHHHH` představuje `HHHHHHHH` kód znaku podle normy ISO 10646.

Tabulka 2.2 ukazuje univerzální kódy znaků specifických pro češtinu a slovenštinu.

Tabulka 2.2: Univerzální jména českých a slovenských znaků

á	\u00E1	ó	\u00F3	Á	\u00C1	Ó	\u00D3
ä	\u00E4	ô	\u00F4	Ä	\u00C4	Ô	\u00D4
č	\u010D	ŕ	\u0155	Č	\u010C	Ŕ	\u0154
ď	\u010F	ř	\u0159	Ď	\u010E	Ř	\u0158
é	\u00E9	š	\u0161	É	\u00C9	Š	\u0160
ě	\u011B	ť	\u0165	Ě	\u011A	Ť	\u0164
í	\u00ED	ú	\u00FA	Í	\u00CD	Ú	\u00DA
ľ	\u013E	ů	\u016F	Ľ	\u013D	Ů	\u016E
ĺ	\u013A	ý	\u00FD	Ĺ	\u0139	Ý	\u00DD
ň	\u0148	ž	\u017E	Ň	\u0147	Ž	\u017D

Standard jazyka C++ dovoluje používat v identifikátorech (jménech) nejen univerzální jména znaků, ale i odpovídající znaky, pokud je to v daném prostředí možné. Na druhé straně

standard jazyka C++ nedovoluje pomocí univerzálních jmen znaků vyjadřovat malá a velká písmena anglické abecedy.

Digrafy a trigrafy

Standardy jazyků C a C++ umožňují nahradit některé ze speciálních znaků kombinacemi jiných znaků. Tyto kombinace se nazývají *digrafy* a shrnuje je tabulka 2.3.

Tabulka 2.3: Digrafy

znak	náhrada
{	<%
}	%>
[	<:
]	:>
#	%:
##	%:%:

Až na způsob zápisu mají *digrafy* naprosto stejný význam jako znaky nebo znakové kombinace, které nahrazují. Z hlediska překladače představují stejné symboly (token). Alternativní možnost náhrady představují tzv. *trigrafy*, které uvádí tabulka 2.4.

Tabulka 2.4: Trigrafy

znak	náhrada	znak	náhrada	znak	náhrada
#	??=	[	??(	{	??<
\	??/	]	??)	}	??>
??'	^	??!	|	??-	~

Trigrafy zpracovává preprocesor (viz kapitola 16), takže překladač s nimi nepřijde do styku, zatímco s digrafy zachází překladač.

- Některé implementace neumějí zpracovat trigrafy přímo; v tom případě poskytují zvláštní samostatný program, který je nutno zavolat před překladem. Tento program trigrafy odstraní a nahradí je znaky, které představují.

2.3 Identifikátor

Identifikátor je jméno nějaké součásti programu. V C++ můžeme jako identifikátor použít libovolnou posloupnost písmen a číslic a začínající písmenem. Identifikátor může také obsahovat znak podtržení a může jím i začínat. Může obsahovat i univerzální jména znaků (nebo odpovídající znaky) a může jimi začínat. Nesmí obsahovat speciální znaky (např. +, – apod.) a nesmí obsahovat bílé znaky (mezery, tabulátory atd.).
Implementace mohou množinu znaků, považovaných za písmena, ještě rozšířit. Mezi častá rozšíření patří např. přidání znaku $.

- Velká a malá písmena se v identifikátorech považují za různé znaky.
  Např. `bouda` a `Bouda` jsou různé identifikátory.

- Délka identifikátoru v C++ není omezena a všechny znaky v něm jsou významné (signifikantní), tj. překladač bere při rozlišování identifikátorů v úvahu všechny znaky. V identifikátorech se rozlišují malá a velká písmena.
- Identifikátor se nesmí shodovat se žádným z klíčových slov (viz kapitola 2.4). To znamená, že identifikátor nesmí být tvořen stejnou posloupností písmen se stejnou velikostí jako některé z klíčových slov.
- Identifikátory začínající dvěma podtržítky nebo jedním podtržítkem a velkým písmenem jsou vyhrazeny pro vnitřní potřeby překladače, takže bychom se jim měli vyhýbat.
- Starší verze jazyka C omezovaly počet signifikantních znaků v identifikátorech. Např. standard ANSI C požadoval alespoň 31 signifikantních znaků.
- Starší verze jazyků C a C++ nedovolovaly v identifikátorech používání znaků národních abeced.

PŘÍKLAD

Slovo `template` není možné v C++ použít jako identifikátor, neboť se shoduje s jedním z klíčových slov tohoto jazyka. V jazyce C je to však správný identifikátor. Slova `Template`, `TEMPLATE`, `tEmplatE` jsou v obou jazycích správně utvořené a navzájem různé identifikátory, neboť se od klíčového slova `template` liší velikostí písmen.

2.3.1 Oblast platnosti, oblast viditelnosti

Ke každému identifikátoru se váže oblast platnosti (část programu, v níž existuje objekt, který tento identifikátor označuje) a oblast viditelnosti (část programu, v níž je objekt označený identifikátorem dostupný). Tím se budeme podrobně zabývat později (viz kapitola 6.5).

2.4 Klíčová slova

Klíčová slova jsou vlastně vyhrazené identifikátory, které označují datové typy, příkazy a jiné syntaktické konstrukce jazyků C a C++. Tabulka 2.5 obsahuje přehled klíčových slov jazyka C; symbol * označuje klíčová slova, která najdeme jen v C99.

Tabulka 2.5: Klíčová slova jazyka C. Hvězdička označuje klíčová slova zavedená v C99.

`auto`	`double`	`inline *`	`sizeof`	`volatile`
`break`	`else`	`int`	`static`	`while`
`case`	`enum`	`long`	`struct`	`_Bool *`
`char`	`extern`	`register`	`switch`	`_Complex *`
`const`	`float`	`restrict *`	`typedef`	`Imaginary *`
`continue`	`for`	`return`	`union`	
`default`	`goto`	`short`	`unsigned`	
`do`	`if`	`signed`	`void`	

Tabulka 2.6 obsahuje přehled klíčových slov jazyka C++; symbol * označuje klíčová slova, která najdeme jen v C++0x.

Tabulka 2.6: Klíčová slova jazyka C++. Hvězdička označuje klíčová slova zavedená v C++0x.

alignas *	continue	friend	register	true
alignof *	decltype *	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	double	int	signed	typename
break	do	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert *	using
char	enum	new	static_cast	virtual
char16_t *	explicit	noexcept *	struct	void
char32_t *	export	nullptr *	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr *	float	protected	thread_local *	
const_cast	for	public	throw	

Všechna klíčová slova jazyků C90 a C++ obsahují pouze malá písmena anglické abecedy.

- S výjimkou klíčových slov `restrict`, `_Bool`, `_Complex` a `_Imaginary` jsou všechna klíčová slova jazyka C součástí C++.
- Klíčová slova `_Bool`, `_Complex` a `_Imaginary`, s nimiž se setkáme pouze v C99, začínají jedním podtržítkem následovaným velkým písmenem.

Implementace může doplnit další klíčová slova. Měla by začínat dvěma podrtžítky. Tabulka 2.7 ukazuje nejčastější rozšíření, s nimiž se setkáme v překladačích na PC.

Tabulka 2.7: Nejčastější nestandardní klíčová slova v překladačích na PC

__cdecl	__declspec	__export
__except	__far	__fastcall
__finally	__leave	__near
__huge	__pascal	__stdcall
__syscall	__try	__int64

Za vyhrazená slova se také pokládají následující náhrady některých operátorů, uvedené v tabulce 2.8.

Tabulka 2.8: Standardní náhrady některých operátorů

symbol	náhrada	symbol	náhrada	symbol	náhrada
&&	and	|	bitor	!=	not_eq
^	xor	&=	and_eq	~	compl
||	or	^=	xor_eq	&	bitand
!	not	|=	or_eq		

- V jazyce C jsou tyto náhrady k dispozici také, jsou však řešeny prostřednictvím maker #definovaných v hlavičkovém souboru `<iso646.h>`.
- V C++ by tyto náhrady měly být automatické; mnoho překladačů je však zpřístupňuje – podobně jako jazyk C – pomocí hlavičkového souboru `<iso646.h>`.
- V C++0x jsou také vyhrazena slova `final` a `override` (standard o nich hovoří jako o identifikátorech se zvláštním významem).

2.5 Lexikální konvence a zápis programu

Překladač při analýze textu zdrojového programu v jazyce C rozlišuje několik druhů symbolů (token): identifikátory, klíčová slova, literály (přímo zapsané konstanty), operátory, oddělovače atd. V C++ jich rozlišuje více, to však pro nás není podstatné.
Prvním úkolem překladače je rozdělit zdrojový text na tyto symboly. Přitom postupuje tak, že *vyjde od konce posledního nalezeného symbolu a vždy vezme nejdelší řetězec následujících znaků, který může utvořit další symbol*.

PŘÍKLAD

Vezměme příkaz

```
a---b;
```

Ač by se mohlo zdát, že jej lze interpretovat nejméně třemi způsoby,

```
(a--) - b;        a- (--b);        a - (-(- b));
```

postup překladače bude jednoznačný: Za identifikátorem `a` najde tři znaky minus za sebou, a nejdelší symbol, který z nich lze sestavit, je operátor `--`. Proto rozloží uvedený výraz jako

```
(a--) - b;
```

- Jednotlivé symboly se v programu oddělují zpravidla pomocí bílých znaků a komentářů. Jestliže za sebou následují znaky, které nemohou spolu tvořit jeden symbol, nemusí být odděleny bílým znakem (například ve výrazu `1+2` nemusí být mezi `1` a `+` žádný bílý znak).
- Všude, kde může být jedna mezera, můžeme vložit libovolný počet bílých znaků nebo komentářů. Vložením nadbytečných bílých znaků (mezer, přechodů na novou řádku apod.) a komentářů můžeme výrazně zpřehlednit zápis programu, aniž změníme jeho význam.

2.6 Průběh překladu

Většinou stačí vědět, že zpracování zdrojového programu v C++ probíhá ve třech základních fázích.

1. Nejprve preprocesor odstraní komentáře, vloží do překládaného souboru soubory připojované direktivou `#include` a odstraní tyto direktivy ze zdrojového textu, provede ostatní direktivy preprocesoru a odstraní je ze zdrojového textu a rozvine makra. Trigrafy nahradí znaky, které zastupují. Po dokončení této fáze tedy zdrojový text překládaného souboru neobsahuje žádné trigrafy, direktivy ani komentáře.
2. Pak překladač rozloží zdrojový text na symboly (token), získanou posloupnost symbolů analyzuje, a pokud v něm nenarazí na žádné chyby, vytvoří relativní soubor. To je soubor