# Úvod do programování

Petr Horáček

Revize: v0.8

# Obsah

# 1 Úvod

## 1.1 Co je to programování

Programování je proces, při kterém programátor píše zdrojový kód v programovacím jazyce, aby vytvořil instrukce, které počítač vykoná. Zdrojový kód je textový zápis programu, který obsahuje přesné kroky k řešení určitého úkolu. Programovací jazyk je prostředek, kterým programátor komunikuje s počítačem, a překládá své myšlenky do formy, které počítač rozumí. Programátor je člověk, který tyto instrukce tvoří a tím dává počítači schopnost vykonávat různé činnosti.

Programování je tvůrčí činnost, která umožňuje přetvořit nápady na realitu pomocí počítače. Díky tomu můžete automatizovat nudné činnosti, analyzovat velké množství dat, vytvářet užitečné aplikace nebo třeba i hry.

## 1.2 Proč se učit programovat

V dnešním digitálním světě je programování dovedností, která otevírá dveře k mnoha příležitostem, ať už profesním, nebo osobním. Možná si myslíte, že programování je určeno pouze pro IT odborníky, ale ve skutečnosti má široké využití v každodenním životě.

- **Schopnost řešit problémy** - Programování vás naučí logicky myslet a rozdělit složité problémy na menší, snadněji řešitelné části. Tato dovednost se hodí nejen při práci na počítači, ale i v běžném životě.
- **Zábava a kreativita** - Programování může být zábavné! Například vytvoření vlastní webové stránky, jednoduché hry nebo aplikace vám dá pocit uspokojení, když váš nápad ožije na obrazovce a dělá přesně to co chcete.
- **Nový způsob uměleckého vyjádření** - Programování umožňuje proměnit myšlenky v dynamická díla, kde se logika algoritmů snoubí s kreativitou a dává vzniknout umění, které by jinak nebylo možné vytvořit.
- **Příležitosti na pracovním trhu** - Dovednost programování je vysoce ceněná na pracovním trhu. I základní znalosti mohou být velkou výhodou, protože mnoho firem dnes hledá zaměstnance, kteří rozumí technologiím.

## 1.3 Je programování pro každého

Určitě ano! Možná jste slyšeli, že programování je obtížné nebo že k němu potřebujete být „matematický génius“. To není pravda. Moderní nástroje a jazyky jsou navrženy tak, aby byly přístupné každému, kdo má chuť učit se něco nového. Navíc začít programovat je dnes jednodušší než kdy dříve - existuje mnoho interaktivních tutoriálů, kurzů a nástrojů, které vás provedou prvními kroky. Nejdůležitější je však ochota zkoušet nové věci, nebát se chyb a učit se z nich.

## 1.4 Jak se učit programovat

Učení programování je dlouhodobý proces, který vyžaduje trpělivost a pravidelnost. Nejedná se o sprint, kde byste se vše naučili za pár dní, ale spíše o maraton, kde postupně

sbíráte dovednosti krok za krokem. Když se každý den nebo týden zaměřte/naučíte jednu jednoduchou věc - třeba základní příkazy, nové funkce nebo malý projekt, tak i malé krůčky se časem sečtou a v dlouhodobém měřítku se z vás stane schopný programátor. Klíčem je pravidelná praxe a ochota učit se z chyb, protože každá překážka, kterou překonáte, vás posouvá dál.

## 1.5 Programovací jazyk C

Jazyk C je nízkoúrovňový programovací jazyk, který v 70. letech 20. století položil základy moderní informatiky. I přes svůj věk zůstává klíčovým nástrojem, protože definoval způsob, jakým dnes píšeme kód a jak uvažujeme o struktuře programů.

### 1.5.1 Základní vlastnosti

- **Efektivita a výkon** - Programy v C jsou velmi rychlé, protože jsou překládány přímo do instrukcí, kterým procesor rozumí bez nutnosti dalších prostředníků.
- **Přístup k hardwaru** - Jazyk umožňuje přímou práci s pamětí počítače, což je nezbytné pro řízení technických zařízení.
- **Přenositelnost** - Kód napsaný v C lze (s drobnými úpravami) spustit na široké škále zařízení, od mikrokontrolérů po superpočítače.
- **Normovaná syntaxe** - Pravidla psaní kódu (používání středníků, složených závorek apod.) se stala standardem, který převzaly moderní jazyky jako C++, Java, C# nebo JavaScript.

### 1.5.2 Využití v praxi

Jazyk C se používá všude tam, kde je kritická rychlost a spolehlivost:

- **Operační systémy** - Základní části systémů Windows, macOS a Linux jsou napsány v C.
- **Vestavěné (embedded) systémy** - Software v pračkách, automobilech, lékařských přístrojích nebo v chytrých domácnostech.
- **Ovladače a knihovny** - Programy, které zajišťují komunikaci mezi hardwarem (např. grafickou kartou) a zbytkem systému.

### 1.5.3 Proč začít právě s C?

Studium jazyka C poskytuje studentům hlubší vhled do fungování počítače. Zatímco moderní jazyky mnoho věcí skrývají pod povrchem, C nutí programátora pochopit, jak se data ukládají do paměti a jak procesor vykonává instrukce. To vytváří pevný základ pro pochopení jakéhokoliv dalšího programovacího jazyka a studium informatiky.

# 2 Základy algoritmizace

## 2.1 Co je to algoritmus

**Algoritmus** je přesně definovaný postup skládající se z konečného počtu jasně popsaných kroků, které vedou k řešení určitého problému.

Příklady algoritmů můžeme najít všude kolem nás:

- **V běžném životě**: Recept na přípravu jídla je algoritmus - obsahuje přesné kroky, jak dosáhnout požadovaného výsledku (například upečení koláče).
- **V programování**: Počítačový algoritmus může vyhledat informace, seřadit data nebo vypočítat matematický problém.

Algoritmus je základem každého programu. Než začneme programovat, je důležité nejprve vymyslet a navrhnout algoritmus, protože právě on definuje, jakým způsobem se problém vyřeší.

## 2.2 Vlastnosti algoritmu

Algoritmus je složením dat a instrukcí, které říkají co se s těmito daty dělá. Každý algoritmus musí splňovat několik klíčových vlastností, aby byl považován za správný a funkční.

- **Vstupní bod** - Každý algoritmus musí mít jeden přesně definovaný vstupní bod, aby bylo jasné kde algoritmus začíná, tedy jakou instrukcí začít.
- **Výstupní bod** - Algoritmus musí mít minimálně jeden, ale i více výstupních bodů, tedy stavů kdy algoritmus již nepokračuje ve své činnosti. Algoritus totiž může skončit různými způsoby, například úspěšně a nebo neúspěšně.
- **Konečnost** - Algoritmus musí vždy skončit po konečném počtu kroků. Nemůže pokračovat nekonečně, jinak by problém nevyřešil.
- **Vstup** - Algoritmus přijímá vstupní data, která zpracovává. Kdyby do algoritmu nebyly vloženy žádná data, algoritmus by neměl s čím pracovat.
- **Výstup** - Výsledkem algoritmu musí být jednoznačný výstup, který odpovídá zadanému problému. Tento výstup je závislý na vstupu. Kdyby výstupem algoritmu nebyl žádný výsledek, algoritmus by nedělal nic užitečného a byl by zbytečný.

## 2.3 Procesor

Pojem procesor je běžně chápán jako elektronický čip uvnitř počítače. V kontextu algoritmizace má však širší význam: představuje jakoukoli entitu (vykonavatele), která dokáže číst a provádět jednotlivé kroky algoritmu.

Počítač je tedy pouze jedním z mnoha možných typů procesorů. Zatímco v digitálním světě procesor vykonává algoritmus ve formě strojových instrukcí, v běžném životě může být procesorem například kuchař, který postupuje podle receptu, nebo hudebník hrající z not. Tato abstrakce nám pomáhá pochopit, že algoritmus je samostatný logický postup, který existuje nezávisle na tom, zda ho zrovna vykonává stroj nebo člověk.

## 2.4 Kontext/vnitřní stav algoritmu

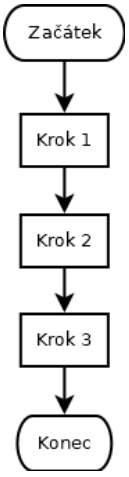
Algoritmus při svém průchodu vytváři tzv. **kontext** nebo také **vnitřní stav**. V reálném světě například v případě kuchařského receptu je kontext algoritmu stav v jakém se nachází vařené jídlo. V případě počítačového programu je kontext algoritmu tvořem hodnotami uloženými v paměti RAM. V průběhu vykonávání algoritmu se tento vnitřní stav algoritmu mění a je na něj možné reagovat pomocí **řídících konstrukcí**.

## 2.5 Řídící konstrukce algoritmu

Každý algoritmus je tvořen kombinací tří základních konstrukcí, které určují, jak jsou jednotlivé kroky algoritmu prováděny. Pro grafický popis se používají takzvané diagramy UML, ve kterých jsou jednotlivé konstrukce značeny speciální značkou a směr toku programu je vyznačen orientovanou přímkou s šipkou.

### 2.5.1 Sekvence

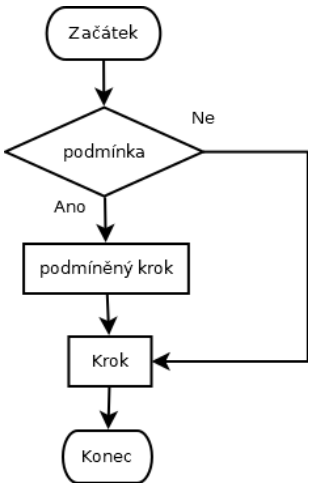
Sekvence je definovaná jako posloupnost kroků, pracující s daty, které v přesně stanovaném pořadí vedou k řešení problému.




**Obrázek 2.5.1** Diagram sekvence

### 2.5.2 Selekce

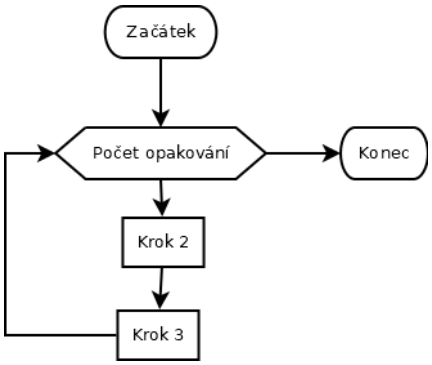
Selekce umožňuje rozhodování na základě podmínky, která část algoritmu se vykoná. Díky tomu lze vytvořit felexibilní chování, které reaguje na aktuální vnitřní stav algoritmu.




**Obrázek 2.5.2** Diagram selekce (větvení)

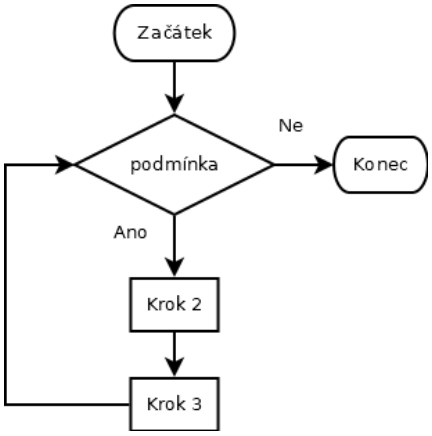
### 2.5.3 Iterace

Iterace, nebo také cyklus nebo smyčka umožňuje opakované vykonávání určité sekvence kódu, na základě platnosti nějaké podmínky. Je možné rozlišit dva případy iterace. Iterace s pevným počtem opakování se používá v případech kdy předem víme kolikrát je třeba vykonat určitou sekvenci kódu.




**Obrázek 2.5.3** Diagram iterace s pevným počtem opakování

Podmíněná iterace je použita ve chvíli kdy je třeba vícekrát vykonat určitou sekvenci kódu, ale není předem možné určit kolikrát to bude třeba.




**Obrázek 2.5.4** Diagram podmíněné iterace

## 2.6 Algoritmizace problému

Algoritmizace je proces vytváření algoritmu - tedy přesného a efektivního postupu, který vede k vyřešení konkrétního problému. Tento proces zahrnuje analýzu problému, návrh a testování algoritmu, který tento problém efektivně vyřeší.

### 2.6.1 Kroky algoritmizace

- **Porozumnění problému** - Prvním krokem je důkladné porozumění tomu, jeho podstatu a co vlastně problém vyžaduje. To zahrnuje definování vstupních a výstupních dat a specifikaci požadavků.
- **Analýza problému** - V tomto kroku se zaměřujeme na rozložení problému na menší části a pochopení, jak jednotlivé části souvisejí. Identifikujeme podproblémy a zjistíme, jaký přístup bude nejvhodnější pro jejich řešení.
- **Návrh algoritmu** - Vytvoříme plán, jakým způsobem problém vyřešit. Zvolíme vhodné kroky a struktury, které pomohou dosáhnout správného výsledku. Tento krok se často realizuje pomocí pseudokódu, diagramů nebo popisu v přirozeném jazyce.
- **Implementace algoritmu** - Jakmile máme dobře navržený algoritmus, přistoupíme k jeho implementaci/realizaci v konkrétním programovacím jazyce. Tento krok zahrnuje přenos algoritmu do kódu, který bude vykonávat počítač.
- **Testování implementace** - Po napsání kódu algoritmus testujeme, abychom ověřili, že implementace funguje podle očekávání, bez chyb a že vykonává algoritmus efektivně.

### 2.6.2 Dekompozice a zobecnění problému

V praxi je složité a nepraktické řešit složité problémy jako celek, jednodušší a přehlednější je použít dekompozici a využít abstrahování informací k zobecnění problému. Rozložení složitějších problémů na jednodušší je jednou z nejdůležitějších technik v algoritmizaci. Tento proces, známý také jako dekompozice, spočívá v rozdělení velkého problému na menší, lépe zvládnutelné podproblémy, které jsou jednodušší k pochopení a řešení. Každý z těchto podproblémů je řešen samostatně, což následně vede k řešení celkového problému.

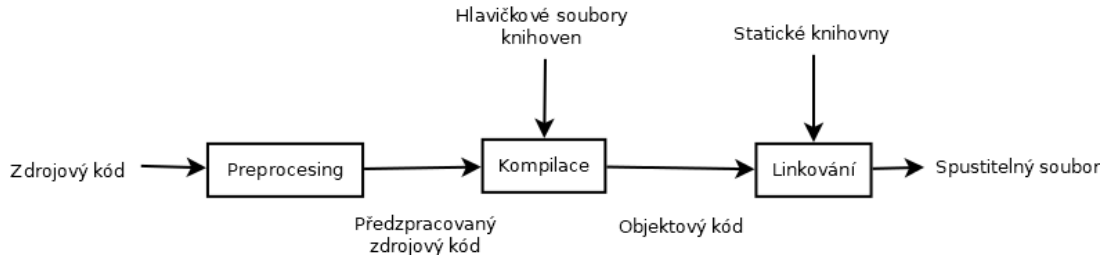
Zobecnění řešení znamená nalezení obecného postupu nebo vzoru, který lze aplikovat na více různých problémů, nejen na jeden konkrétní. Tato technika je velmi užitečná, protože umožňuje vytvářet flexibilní a univerzální algoritmy, které mohou být použity na širokou škálu problémů. Zobecnění často znamená zjednodušení problému tak, že se zaměříme na jeho základní vlastnosti a ignorujeme detaily, které se mohou měnit mezi jednotlivými případy. Díky tomu vytvoříme algoritmy, které nejsou závislé na konkrétním zadání, ale jsou aplikovatelné na širokou paletu problémů.

# 3 Proces překladu zdrojových kódů

Algoritmus je pouze myšlenkový popis jak řešit určitý problém, ale algoritmus je třeba převést do formy programu, který je možné spustit na **procesoru**.

## 3.1 Fáze překladu

Programy napsané v jazyce C (a dalších programovacích jazycích) jsou nejprve vytvořeny jako zdrojové kódy - textové soubory s příponou *.c* obsahující instrukce v srozumitelném formátu pro člověka. Tyto zdrojové kódy ale počítač neumí přímo vykonat. Aby se program stal spustitelným, musí projít procesem překladu (kompilace), který převede zdrojový kód do formátu, kterému procesor rozumí. Tento proces má několik fází.




**Obrázek 3.1.1** Proces překladu zdrojových kódů

Překlad zdrojového kódu do spustitelného programu zahrnuje několik kroků, které zajišťují správnost a efektivitu výsledného programu.

### 3.1.1 Preprocesor (úprava zdrojového kódu)

První fáze překladu je preprocesing. V této fázi se zdrojové kódy předpřipraví pro proces kompilace. To obnáší odstranění komentářů a rozbalelení příkazů pro preprocesor, které začínající znakem # (tzv. preprocesorové direktivy).

### 3.1.2 Překlad (kompilace do strojově nezávislého kódu)

V této fázi překladač analyzuje a převádí zdrojový kód na tzv. **objektový kód**. Objektový kód obsahuje strojový kód procesoru, ale nejedná se ještě o spustitelný kód.

### 3.1.3 Linkování (spojování a vytváření spustitelného souboru)

Linker (spojovací program) je zodpovědný za spojení všech objektových souborů a knihoven do jednoho spustitelného programu.

## 3.2 Základní struktura projektu v jazyce C

Tento Program se nazývá **Hello World**, je to nejjednodušší možný program, který má za úkol pouze vytisknout pozdrav "Hello World" do terminálové konzole a slouží k ověření, že prostředí pro programování funguje správně a zároveň pro prvotní seznámení se základními vlastnostmi programovacího jazyka. V tuto chvíli není potřeba rozumět co který řádek kódu dělá, ale jsou zde obsaženy všechny základní stavební kameny jazyka C a díky tomu je možné si vytvořit představu o tom jak se s tímto jazykem bude pracovat.

```c
#include <stdio.h>
int main(void) {
    printf("Hello World\n");
    return 0;
}
```

# 4 Příprava prostředí pro vývoj v jazyce C

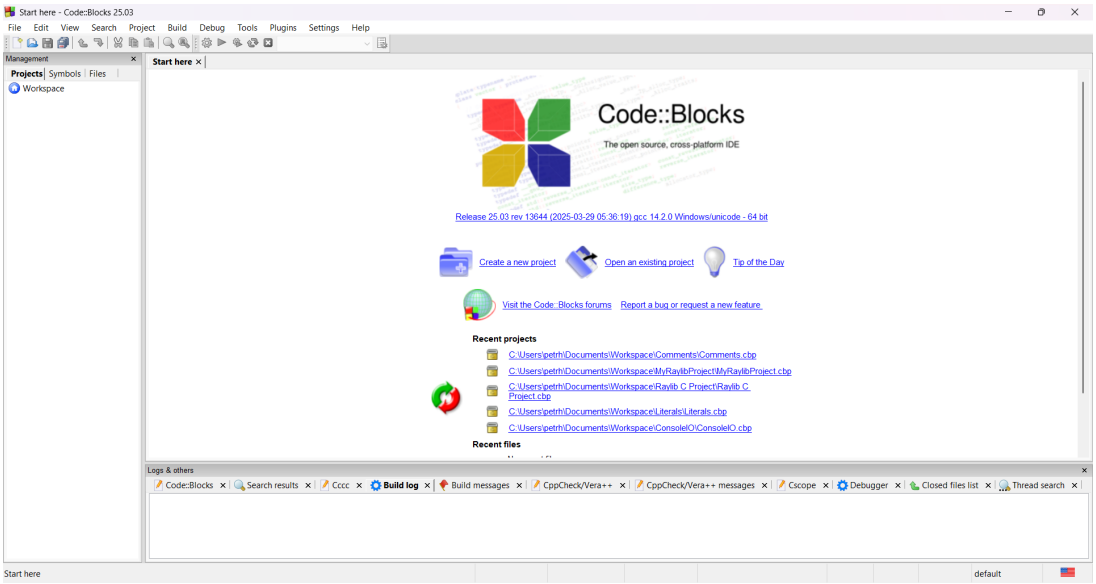
Prostředí pro programování v jazyce C je možné připravit na jakémkoli počítači s pomocí nástrojů, které jsou zdarma ke stažení z internetu. V základu se jedná o kompilátor **gcc** (případně alternativní clang), který má za úkol převést zdrojový kód na spustitelný soubor a vývojové prostředí s pomocí kterého se zapisuje kód programu.

## 4.1 Vývojové prostředí

Vývojové prostředí, anglicky **Integrated Development Environment - IDE** je podpůrný nástroj, který programátorům usnadňuje psaní kódu, spravovat projekty a automatizovat překlad zdrojových kódů do spustitelné podoby. Jedná se o textový editor, který umožňuje barevně zvírazňovat syntaxi použitého programovacího jazyka, doplňovat názvy příkazů nebo identifikátoru a podobně.

## 4.2 Code::Blocks IDE

Nejjednodušší IDE pro jazyk C je **Code::Blocks IDE**, který, je buď ve formě instalovaná do systému, nebo přenositelná forma, kterou je možné si uložit na flashdisk a přenést jednoduše bez instalace na jiný systém. Zároveň je Code::Blocks poskytován ve variantě buď jako samostné IDE, který si automaticky vyhledá kompilátor a další nástroje instalované v systému a nebo ve variantě, která si sebou nese všechny potřebné nástroje, které jsou pro programování v jazyce C potřeba. Taková varianta Code::Blocks je pak okamžitě po stažení připravená k programování.




**Obrázek 4.2.1** Úvodní obrazovka Code::Blocks IDE

### 4.2.1 Instalace a stažení.

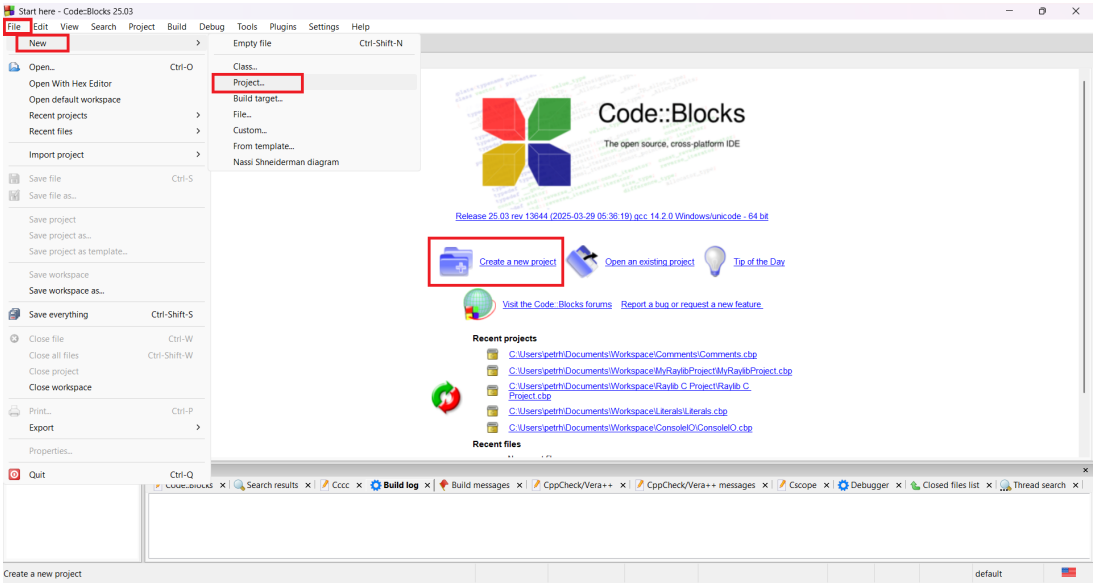
Code::Blocks je možné zdarma stáhnout z webových stránek: `http://www.codeblocks.org/`.

Při prvním spuštění pravděpodobně IDE otevře okno kde uživatele informuje o nalezených vývojových nástrojích a dotáže se zda chceme soubory zdrojových kódů s koncovkou

*.c* (zdrojové soubory) a **.h* (hlavičkové soubory) asociovat s programem Code::Blocks (aby se po dvojkliku myší soubory otevíraly v programu Code::Blocks). Obě tato okna stačí jednoduše odkliknout a dále se již nebudou zobrazovat.

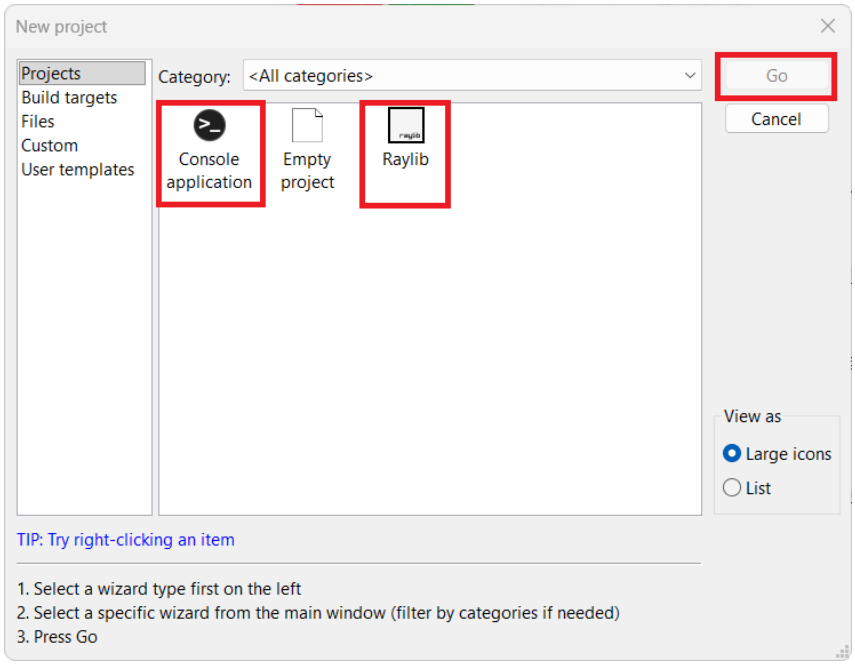
## 4.3 Založení nového projektu

Nový projekt je možné založit jednoduše z úvodní obrazovky kliknutím na tlačítko **Create a new project** a nebo přes menu: **File** → **New** → **Project**



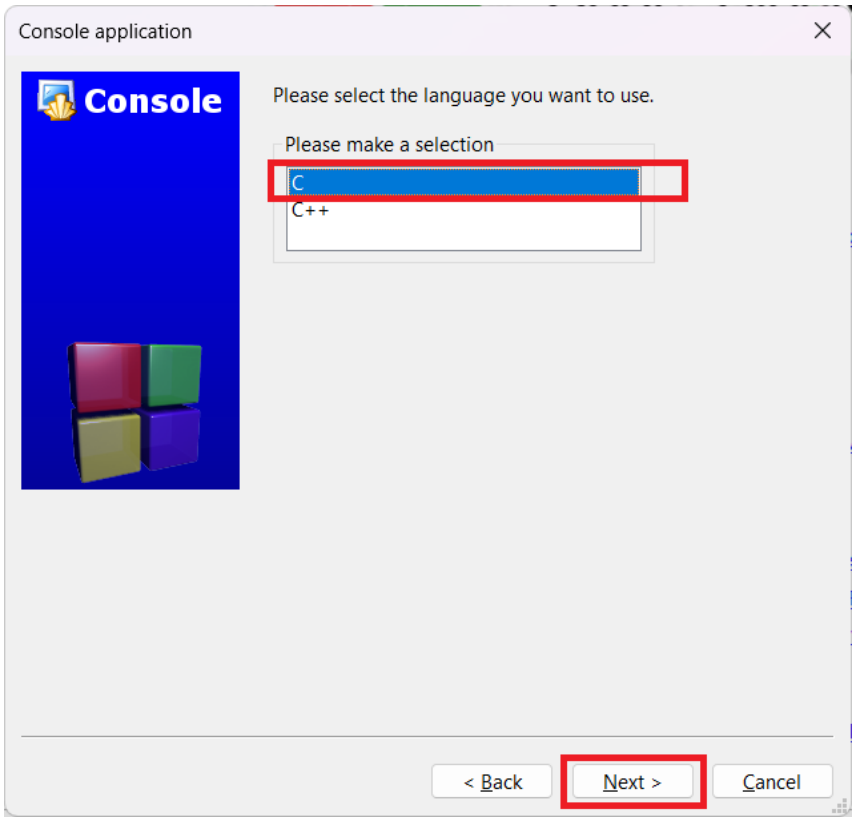

**Obrázek 4.3.1** Založení projektu

Poté je třeba projít jednoduchým formulářem, do kterého se vyplní požadavky na nový projekt. Nejdříve se vybere typ projektu. Code::Blocks má ve své výchozím stavu připravené velké množství šablon projektů, což může být dost nepřehledné. Proto jsem větší přehlednost všechny ostatní šablony schoval a nechal pouze ty které jsou pro začátek relevantní. V podstatě jsou na výběr dvě možnosti. Buď **Console application**, která vytvoří nejjednodušší možný projekt v jazyce C, bez žádných dodatečných doplňků, komunikující s uživatelem pomocí příkazové řádky (vhodné na jednoduché ukázky kódů, nebo systémové nástroje) a nebo šablona **Raylib**, která připraví projekt pro programování grafického rozhraní. Raylib je vhodný pro programování počítačové grafiky, tedy například her, nebo nástrojů, které vykreslují nějaký grafický obsah.




**Obrázek 4.3.2** Šabloba projektu

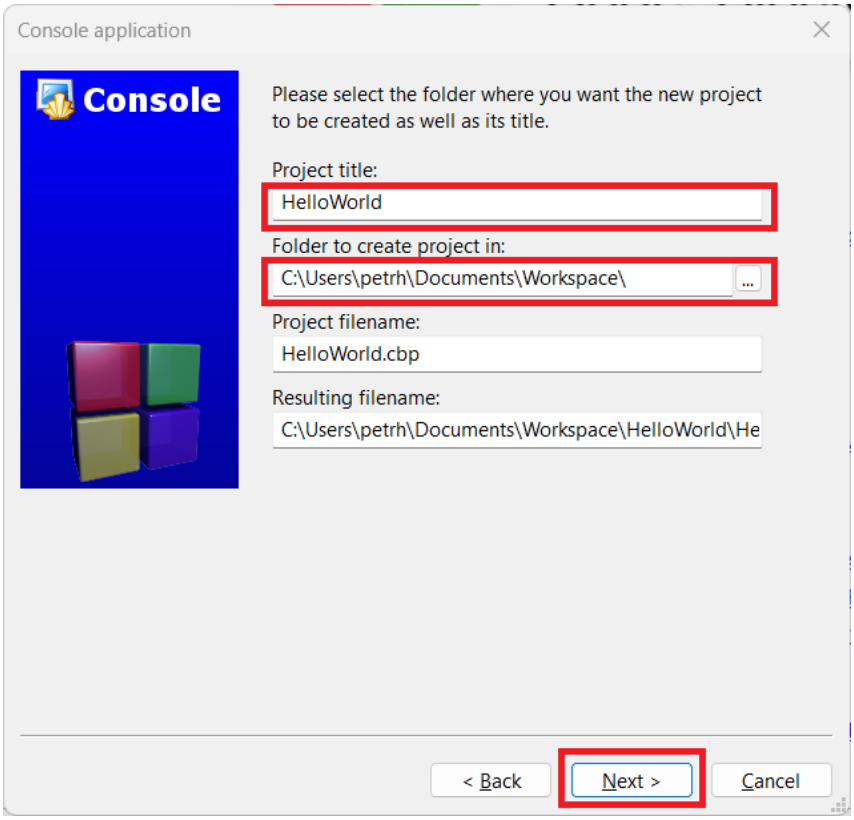
Následně je potřeba vybrat cílový programovací jazyk projektu. Na výběr je buď jazyk C a nebo C++ (jazyk C++ je komplexnější nadstavba jazyka C). **Při vytváření projektu je třeba vždy vybrat jazyk C!**




**Obrázek 4.3.3** Výběr programovacího jazyka

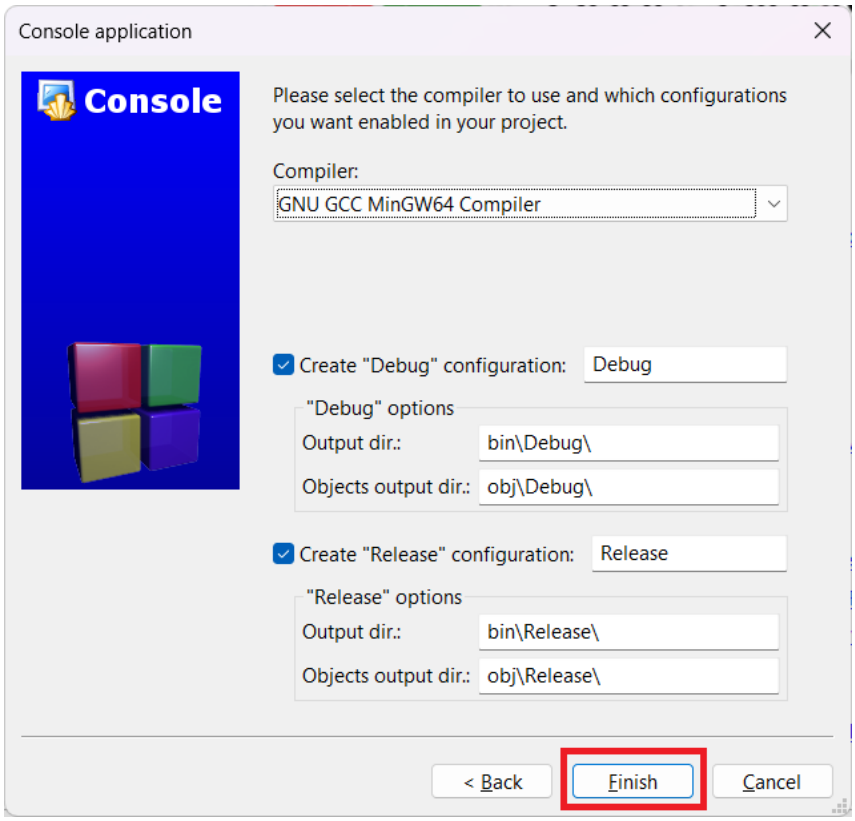
Pak již stačí jen zadat jméno nového projektu (například HelloWorld) a cestu kam se nový projekt uloží. Standardně se pro projekty vytváří pracovní složka, která se běžně nazává **Workspace**, do které se pak budou vytvářet jednotlivé projektové složky ve kterých budou obsaženy jak zdrojvé soubory projektu tak konfigurační soubory Code::Blocks IDE

a výsledné přeložené spustitelné soubory. Cestu pro uložení projektu je potřeba nastavit pouze ve chvíli kdy se na systému vytváří projekt poprvé, Code::Blocks si tuto cestu zapamatuje a bude ji pro následující použití předvyplňovat (bude tedy potřeba vyplnit pouze název projektu).




**Obrázek 4.3.4** Zadání názvu a cesty k uložení projektu

Nakonec se zobrazí okno s přehledem a potvrzením pro vytvoření nového projektu. V tuto chvíli stačí již jen stisknout tlačítko **Finish** a nový projekt je vytvořený.

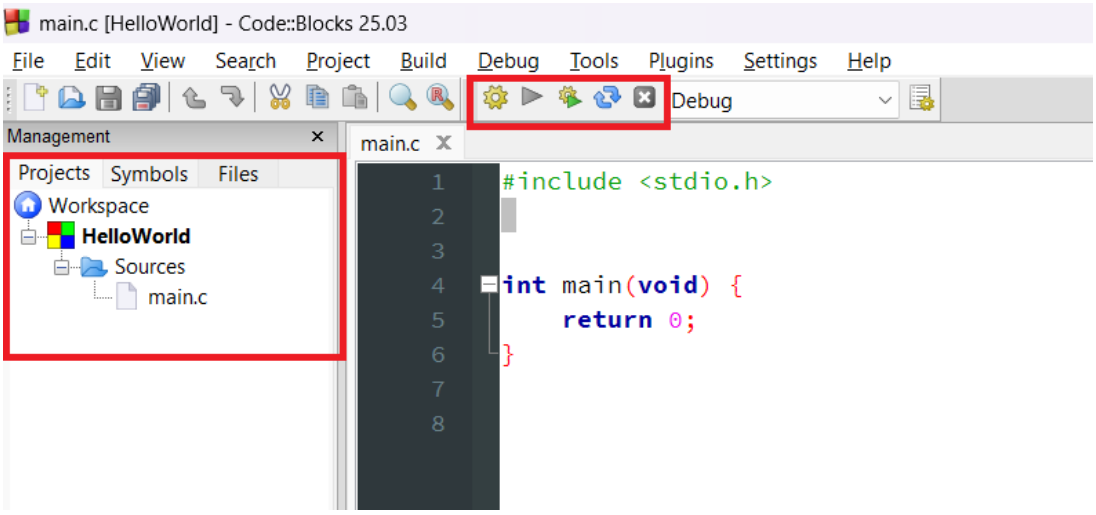



**Obrázek 4.3.5** Dokončení

## 4.4 Přehled a použití Code::Blocks

Prostředí Code::Blocks je relativně intuitivní a přímočaré. V levé části je projektový strom, ve kterém je zobrazen obsah projektu. Po rozbalení (kliknutí na malý symbol plus) je možné zobrazit seznam souborů. Po vytvoření nového projektu se zde nachází pouze soubor **main.c**. Takto se v jazyce C standardně nazývá hlavní zdrojový soubor kterým začíná vykonávání programu. Po dvojkliku je možné jej otevřít v textovém editoru a následně jej upravovat.

Nejdůležitější pro programování jsou zvýrazněná ovládací tlačítka v obrázku. Tlačítko **Build** v podobě žlutého ozubeného kola slouží pro překlad projektu do spustitelné podoby, to je první krok, aby bylo možné program spustit. Poté co je projekt přeložen ze povolí tlačítko **Run**, které má vzhled zelené šipky. Toto tlačítko pak slouží ke spuštění přeloženého programu. Při běžné práci a v 99% případů je nejdůležitější tlačítko **Build and run**, které kobinuje žluté ozubené kolečko a zelenou šipku. Toto tlačítko provede překlad a následně spuštění výsledného spustitelného souboru v jednom kroku. To znamená, že za normálních okolností stačí používat pouze toto tlačítko. Pro snadné a rychlé ukončení běžícího programu slouží tlačítko **Abort**, které vypadá jako červené tlačítko s křížkem.

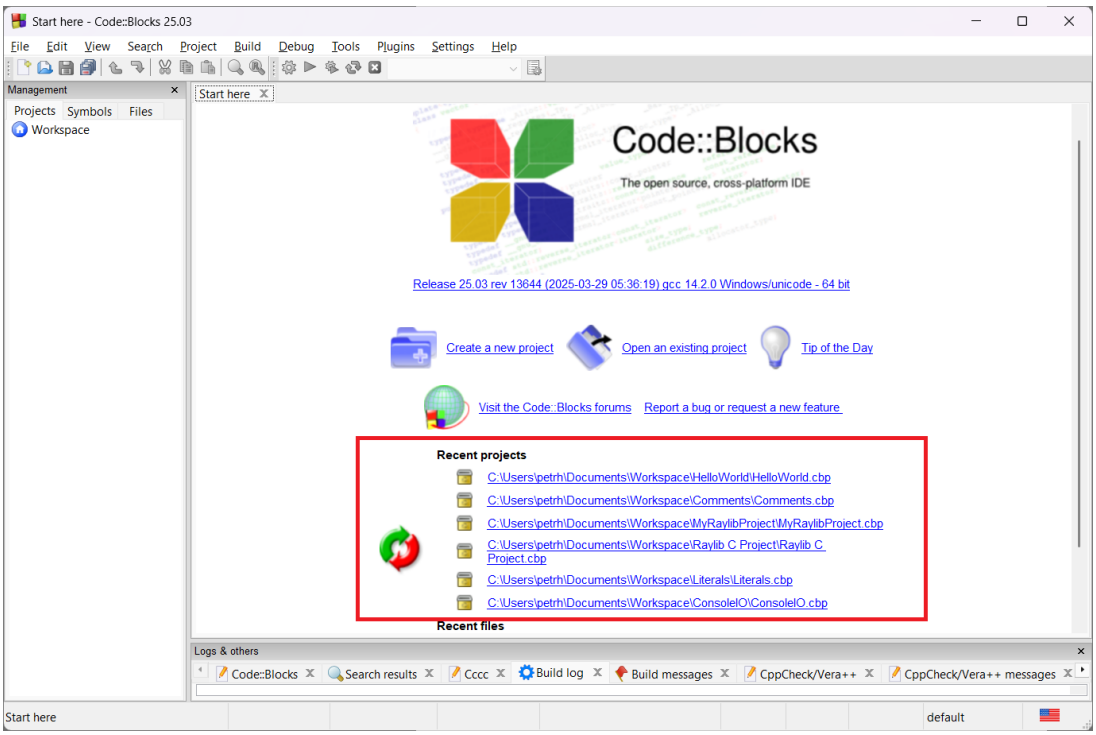



**Obrázek 4.4.1** Přehled prostředí Code::Blocks

## 4.5 Zavření projektu

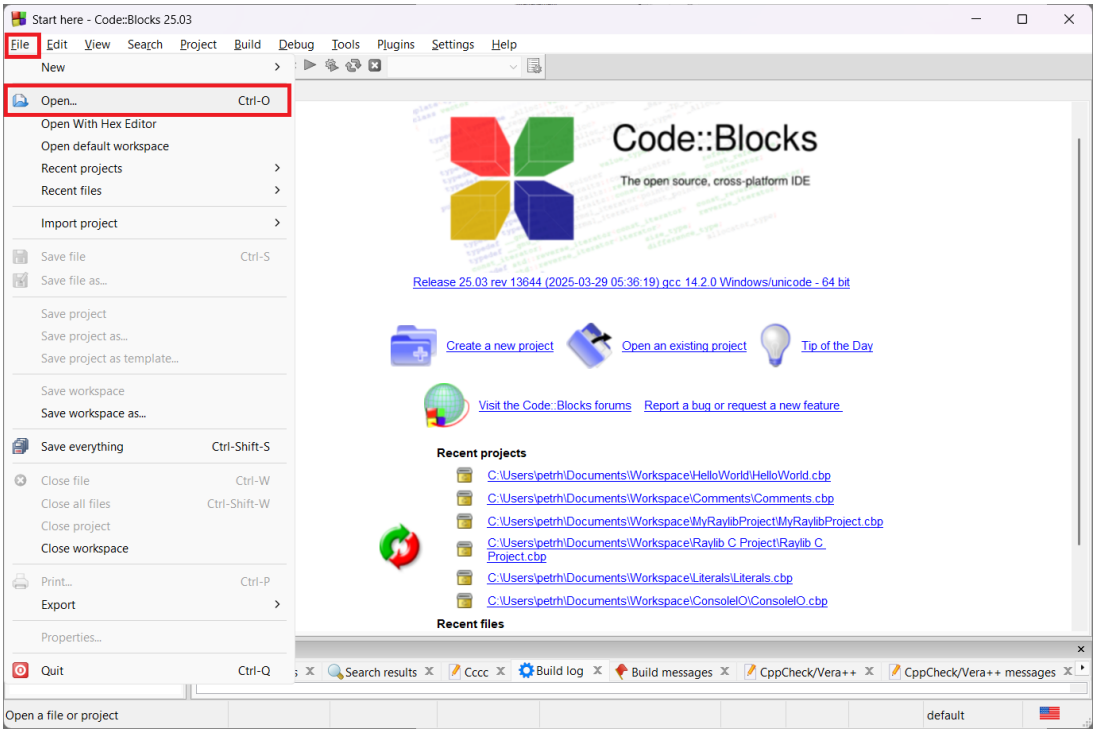
## 4.6 Otevření existujícího projektu

Jakmile je projekt jednou vytvořený je pravděpodobné, že jeho vývoj nebude jednorázová záležitost a bude potřeba se po vypnutí Code::Blocks IDE k danému projektu opět vrátit. V Code::Blocks je možné existující projekt otevřít několika způsoby. První nejjednodušší možnost jak otevřít existující projekt je otevřít projekt z historie naposledy otevřených projektů, která se nachází na hlavní obrazovce Code::Blocks IDE. V této historii jsou vypsány odkazy na otevření posledních několika otevřených projektů. Pro otevření daného projektu stačí kliknout na odkaz vedoucí na požadovaný projekt a daný projekt se okamžitě otevře.



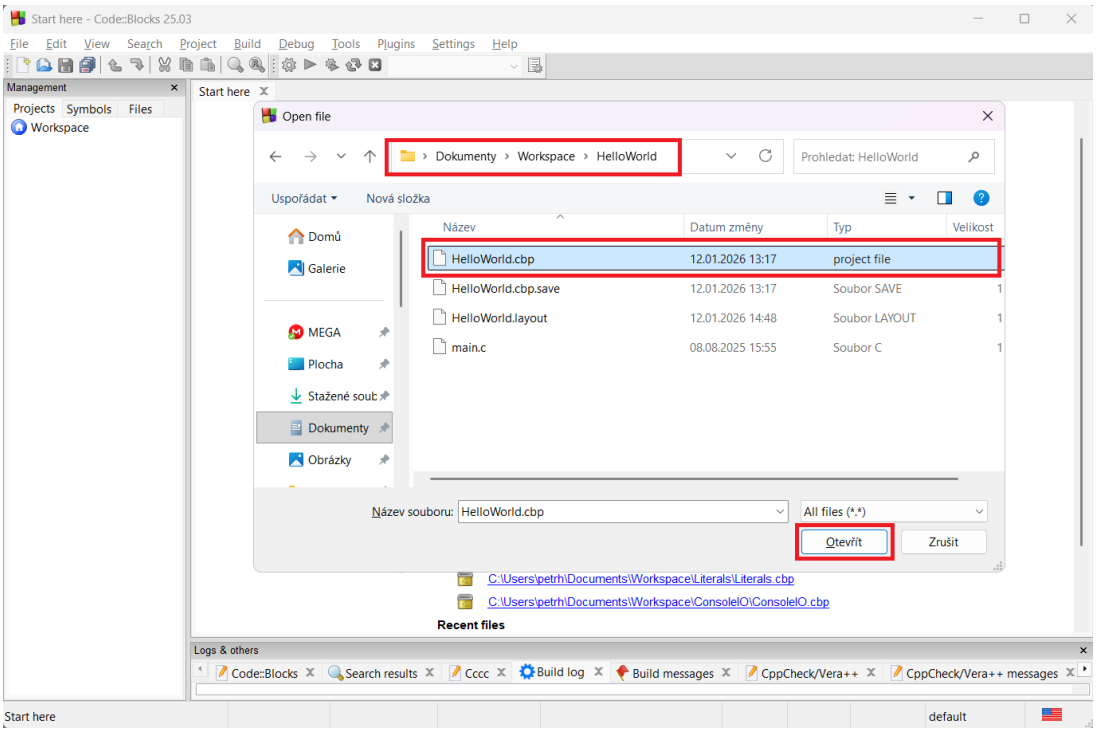

**Obrázek 4.6.1** Historie naposledy otevřených projektů

Druhou možností jak otevřít existující projekt je přes nástrojové menu: **File** → **Open** (nebo pomocí klávesové zkraty *Ctrl + O*):




**Obrázek 4.6.2** Otevření existujícího projektu

Poté se otevře průzkumník souborů kde je třeba najít adresář s projektem v souborovém systému a následně vybrat soubor s koncovkou <**název projektu**>**.cbp**. Tento soubor v sobě nese veškeré nastavení projektu a slouží pro načtení projektu do Code::Blocks IDE:




**Obrázek 4.6.3** Otevření projektu v souborovém systému

# 5 Komentáře

Psaní kódu je kreativní proces, a stejně jako každá tvůrčí práce, může se stát s postupem času složitější. Někdy se i ten nejsrozumitelnější kód stane obtížným na pochopení, zvláště když na něm pracuje více lidí. Z tohoto důvodu každý programovací jazyk obsahuje konstrukci, která umožňuje vkládat informativní poznámky do zdrojového kódu, díky kterým je možné vytvořit srozumitelnější a čitelnější kód.

## 5.1 Dva druhy komentářů

Komentáře jsou části kódu, které překladač zcela ignoruje. Slouží jako poznámky a vysvětlivky pro udržení srozumitelnosti kódu, nebo pro tvorbu komplexní dokumentace. V jazyce C rozlišujeme dva typy: **řádkové komentáře** a **blokové komentáře**

### 5.1.1 Řádkový komentář

Řádkový komentář se vytváří pomocí sekvence dvou lomítek a umožňuje vytvářet komentář v rámci jednoho řádku. Důležité je že řádkový komentář je definovaný od zahajovací sekvence lomítek až do konce řádku:

```
1 // this is line comment
```

### 5.1.2 Blokový komentář

Blokový komentář se vytváří pomocí uvozovací sekvence */** a ukončovací sekvence ***/* a cokoliv definované uvnitř je překladačem považováno za komentář nehledě na odřádkování.

```
1 /*
2 this is
3 block comment
4 defined on multiple lines
5 */
```

## 5.2 Správné použití komentářů

Komentáře jsou mocný nástroj, který by měl každý programátor používat již od začátku. Přispívají k efektivitě a kvalitě kódu, ovšem pokud se používají nesprávně, mohou naopak působit matoucím dojmem a snižovat čitelnost výsledného kódu.

Komentáře by se měly do zdrojových kódů zapisovat ideálně v anglickém jazyce, protože díky tomu jim bude rozumět i někdo kdo nemluví česky. Rozhodně nemíchat české a anglické komentáře!

Nejdůležitější pravidlo zní: *Nekomentuj, co kód dělá, ale proč to dělá. Pokud je kód sám o sobě srozumitelný, komentář není potřeba.*

Vždy je snahou spíš psát takový kód, který je srozumitelný sám o sobě a nevyžaduje komentáře a komentovat pouze to co kód sám nedokáže říct.

## 5.3 Příklad použití komentářů

```
1  /*
2  Overview of the source code:
3  HelloWorld project demonstrating the basic features of a
4  programming language
5  */
6
7  #include <stdio.h>
8
9  int main(void) {
10     printf("Hello World\n"); // printing message to console
11     return 0;
12 }
```

# 6 Identifikátory

Identifikátor je v kontextu programování název, který zastupuje určitou konstrukci (proměnné, funkce nebo datové struktury) a na které se lze ve zdrojovém kódu odkazovat. To umožňuje vytvářet intuitivní a dobře čitelnou strukturu kódu, kterou lze snáze zpětně číst a chápat jeho účel. Dobře zvolená jména identifikátorů tak tvoří jedno z měřítek, které oddělují dobře čitelný kód od "rozsypaného čaje". Při psaní kódu je tak dobré se zaměřit nejen na pravidla jak identifikátory v daném programovacím jazyce zapisovat, ale také na to jak dobře popisují svůj účel.

## 6.1 Pravidla zápisu identifikátorů v jazyce C

Názvy identifikátorů se v jazyce C musí řídit sadou jednoduchých pravidel, protože použití nesprávného formátu způsobí chybu při překladu programu.

- Identifikátor **může** obsahovat pouze velká nebo malá písmena, číslice a nebo znak podtržítko.
- Identifikátor také **musí** vždy začínat písmenem nebo podtržítkem. Nikdy nesmí začínat číslicí.
- Identifikátor **nesmí** být klíčové slovo, které je rezervováno jazykem C. Těmito klíčovými slovy jsou například názvy datových typů, if, for, while a další.

Jazyk C je tzv. **case-sensitive**, to znamená, že záleží na velikosti písmen v identifikátoru. Z pohledu jazyka C jsou tedy názvy **name** a **Name** dva různé identifikátory.

## 6.2 Jak vytvořit dobrý identifikátor

Při vytváření názvu nového identifikátoru je dobré se řídit několika zásadami, které pomohou vytvořit srozumitelný a čitelný kód. Název identifikátoru by měl být **jednoznačný**, **unikátní** a **srozumitelně** popisovat, co dělá nebo co obsahuje. Zároveň je standardem v programátorské komunitě **používat** pro názvy identifikátorů **angličtinu**. Důvod je ten, že se kód stane univerzálním a snadno srozumitelným pro kohokoli, kdo jej v budoucnu uvidí. Například:

```
hit_counter, cursor_position, player_state_update
```

Zkratky jako **tmp** nebo **cnt** mohou sice ušetřit pár znaků, ale často **snižují čitelnost kódu**. Pokud to není naprosto nezbytné, je dobré používat celá slova. Výjimkou mohou být všeobecně známé zkratky jako jsou například **ID**, **min**, **max** a podobně.

## 6.3 Styl zápisu identifikátorů

I když to není vyloženě špatně a občas se tomu nelze vyhnout, je dobré používat v rámci projektu **jednotný styl zápisu identifikátorů**. Nejběžnější styly pro pojmenování jsou **camel case** a **snake case**. Camel case začíná malým písmenem, každé další slovo začíná velkým písmenem.

```
camelCase
```

V případě snake case pak všechna slova jsou malá a oddělují se pomocí znaku podtržítko.

```
snake_case
```

# 7 Proměnné

Každý program, aby vůbec mohl fungovat, potřebuje pracovat s daty. Ať už jde o **čísla**, **text**, nebo třeba **obrázky**, všechna tato data je potřeba někde uložit. K tomuto účelu slouží **operační paměť** počítače - **RAM**. Paměť RAM si lze představit jako dlouhou řadu paměťových buněk (byty), které jsou schopné uchovávat libovolnou číselnou hodnotu od 0 do 255. Na každou paměťovou buňku se lze odkázat pomocí její specifické adresy, která definuje umístění v rámci paměti RAM.

| Adresa buňky | 0x0000 | 0x0001 | 0x0002 | 0x0003 | 0x0004 | ... |
|---|---|---|---|---|---|---|
| Uložená hodnota | 0 | 86 | 11 | 61 | 240 | |

**Obrázek 7.0.1** Schématické zobrazení RAM

Aby nebylo potřeba si v programu pamatovat paměťovou adresu konkrétní paměťové buňky, tedy jaká data jsou na dané paměťové adrese uložena pro pozdější použití, existuje koncept **programových proměnných** (nebo jen proměnné). Ty umožňují pojmenovat konkrétní úsek bytů v paměti, aby k nim bylo možné snadno přistupovat a ukládat do nich informace.

## 7.1 Datové typy

**Datový typ** je klíčový konstrukt, který v programu umožňuje přesně vyjádřit, jaký druh dat - například celé číslo, znak, nebo desetinné číslo - bude v paměti uložen a kolik místa k tomu bude potřeba. Datové typy také zajišťují, že se s daty správně pracuje. Díky tomu je například možné předejít situacím, kdy by se program snažil sečíst číslo a text, protože taková operace mezi tak rozdílnými typy dat prostě není definovaná. Jedná se tedy o mechanismus, který umožňuje programátorovi nejen optimalizovat práci s pamětí tak aby s ní zbytečně neplýtval, ale zároveň mu brání v tvorbě chyb nesprávním použitím operací na daný typ dat.

### 7.1.1 Primitivní datové typy

V jazyce C jsou definované **primitivní datové typy**, tedy datové typy které jsou přímo vestavěnné v kompilátoru jazyka C a nedají se dále rozdělit na jednodušší datové typy (viz. datové struktury). Mezi primitivní datové typy v jazyce C patří:

| Datový typ | Bytová šírka | Účel | Rozsah |
|---|---:|---|---:|
| **char** | 1 byte | znak/celé číslo | -128 až 127 |
| **short** | 2 byty | celé číslo | -32 768 až 32 767 |
| **int** | 4 byty | celé číslo | -2 147 483 648 až 2 147 483 647 |
| **long** | 8 bytů | celé číslo | $-2^{63}$ až $2^{63}-1$ |
| **float** | 4 byty | desetinné číslo | $\pm 1.8 \times 10^{-38}$ až $\pm 3.4 \times 10^{38}$ |
| **double** | 8 bytů | desetinné číslo | $\pm 2.23 \times 10^{-308}$ až $\pm 1.80 \times 10^{308}$ |
| **long double** | 16 bytů | desetinné číslo | $\pm 3.4 \times 10^{-4932}$ až $\pm 1.1 \times 10^{4932}$ |
| **void** | 0 bytů | žádná data | N/A |

**Tabulka 7.1.1** Přehled primitivních datových typů

Při rozhodování, který primitivní datový typ použít v jazyce C, je klíčové zvážit požadavky na paměť, rozsah hodnot a účel proměnné. Jednoduše řečeno, když nevíš jaký datový typ pro celá čísla použít, použij *int*, protože má obvykle vyvážený poměr mezi požadavky na paměť a rozsahem hodnot. Pokud ale pracujete s velkými čísly, použijte long, zatímco pro úsporu paměti u menších hodnot můžete zvolit short nebo dokonce char. Pro čísla s desetinnou částí je vhodné použít float nebo double, přičemž float je úspornější, zatímco double poskytuje vyšší přesnost.

### 7.1.2 Datový typ void

Datový typ **void** představuje prázdný datový typ, který slouží k vyjádření **absence dat**, a proto podle něj nelze vytvořit konkrétní proměnnou. V jazyce C se využívá pro speciální účely, jako je deklarace funkcí bez návratové hodnoty nebo definice univerzálních ukazatelů na paměť.

## 7.2 Definice proměnné

Proces vytvoření proměnné se nazývá **definice proměnné**. V základu proces definice proměnné spočívá v uvedení datového typu (podle druhu uložených dat), popisného identifikátoru a volitelně přiřazení inicializační hodnoty.

```
<datový typ> <identifikátor>;
```

Například:

```
int value;
```

V tuto chvíli je pro danou proměnnou zarezervován úsek paměti v RAM, ale problém je že v této paměti mohla zůstat hodnota uložená z předchozího použití. To znamená, že v 99% případů bude proměnná obsahovat nějakou náhodnou hodnotu. Použití nové proměnné bez nastavení počátení hodnoty je častou příčinou mnoha programových chyb a z tohoto důvodu je velmi doporučováno proměnnou zároveň inicializovat na počíteční hodnotu:

```
<datový typ> <identifikator> = <hodnota>;
```

například:

```
int value = 42;
```

Přitom se stanou dvě věci:

- **Rezervace paměti**: Operační systém vyhradí v paměti RAM určitý počet paměťových buňek (v závislosti na použitém datovém typu).
- **Přiřazení jména**: Toto místo se propojí s předaným identifikátorem, aby bylo možné k vyhrazené paměti přistupovat, aniž by nutné si pamatovat složitou paměťovou adresu.

## 7.3 Paměťová reprezentace proměnných

Definice proměnné v jazyce C je pro systém pokynem k rezervování konkrétního počtu bajtů v operační paměti RAM. Každá proměnná tak získává svou unikátní adresu a vymezený prostor. V paměti RAM tak při vytvoření několika proměnných dojde k rozdělení paměti následujícím způsobem:

```
int value = 42;
```

```
char single_character = 'a';
```


**Obrázek 7.3.1** Schématické zobrazení RAM

## 7.4 Datové literály

V jazyce C je běžné do programu přímo vkládat pevné hodnoty, jako jsou **čísla**, **znaky** nebo **textové řetězce**. Tyto doslovně zapsané hodnoty se nazývají **literály**.

Literály se používají k inicializaci proměnných nebo k definování pevných pravidel v logice programu. Příkladem může být porovnání výsledku výpočtu s předem danou maximální nebo minimální hodnotou.

Při programování je klíčové rozlišovat různé druhy literálů, protože každý z nich má svůj datový typ, který určuje, jak s ním má počítač správně pracovat. V jazyce C se rozlišují čtyři základní typy:

- **Celočíselné literály**
- **Desetinné literály**
- **Znakové literály**
- **Řetězcové literály**

Znakové literály jsou svázány s datovým typem **char** a představují **jeden znak** v jednoduchých uvozovkách.

```
1 char single_character = 'a';
```

Celočíselné literály představují celá čísla a nejčastěji se vážou na datový typ **int**. Protože jsou ale znaky v počítači rovněž reprezentovány pomocí čísel, je možné číselné literály reprezentovat také pomocí datového typu **char**.

```
1 int number = 42;
2 char small_number = 27;
```

Desetinné literály představují **číselné hodnoty s desetinným místem**. Tyto hodnoty se v jazyce C zapisují s **desetinnou tečkou**. Desetinné literály se vážou s datovým typem **double** nebo **float**.

```
1 float decimal_number = 3.14;
2 double long_decimal_number = 3.141592;
```

Řetězcové literály reprezentují sekvenci znaků tvořící text, který je uzavřen v dvojitých uvozovkách. Technicky se jedná o ukazatel na pole znaků.

```
1 char * string = "Hello, World!";
```

## 7.5 Znaméknové a bezznaménkové datové typy

U celočíselných proměnných se rozlišují **znaménkové** a **bezznaménkové** varianty. Znaménková proměnná může nabývat kladných i záporných hodnot, zatímco bezznaménková proměnná reprezentuje pouze nezáporné hodnoty. Pro vytvoření bezznaménkové proměnné slouží klíčové slovo **unsigned**, které se zapíše před použitý datový typ:

```
1 unsigned int number = 42;
2 unsigned char small_number = 150;
```

V jazyce C existuje také klíčové slovo **signed**, které se používá stejným způsobem jako klíčové slovo signed a slouží k vinucení, že datový typ má být znaménkový. V jazyce C jsou ale primitivní celočíselné datové typy char, short, int a long ve výchozím stavu vždy znaménkové a proto klíčové slovo signed není v praxi téměř využíváno.

### 7.5.1 Reprezentace bezznaménkvých proměnných v paměti

Každý datový typ umožňuje rezervovat v paměti určitý počet bitů (respektive bytů), což určuje pevný rozsah hodnot, které do něj lze uložit. Například 8 bitů, které tvoří jeden byte, umožňuje reprezentovat hodnoty od 0 do 255. Čísla se ukládají v podobě kombinace jedniček a nul, přičemž každý z nich zabírá právě jeden bit.

U znaménkových čísel se jeden bit vyhradí pro uložení znaménka, což snižuje maximální hodnotu, kterou lze uložit. Pokud ale záporná čísla nejsou potřeba, je možné použít bezznaménkový datový typ, kde se všechny bity využijí pro samotnou hodnotu. Tím se maximální rozsah kladných čísel téměř zdvojnásobí. Volba správného typu tak pomáhá efektivněji využít paměť a předejít chybám.

## 7.6 Konstantní proměnné

V praxi často nastávají situace, kdy v paměti RAM potřebujeme vyhradit místo pro hodnotu, která se po celou dobu běhu programu **nesmí změnit**. Takovým proměnným se říká konstantní proměnné a k tomuto účelu slouží klíčové slovo const, které se zapíše před vybraný datový typ proměnné, Postup definice proměnné je pak stejný jako u běžné proměnné:

```
const int constant_number = 42;
```

V paměti RAM se v podstatě nic nemění, ale kompilátor vyvolá chybu překladu při jakémkoli pokusu o změnu hodnoty konstatní proměnné.

## 7.7 Typové přetečení

# 8 Operátory

## 8.1 Aritmetické operátory

## 8.2 Relační operátory

## 8.3 Logické operátory

# 9 Řídící konstrukce

**Řídicí konstrukce** jsou základní nástroje jazyka C, které slouží k popisu algoritmu. Zatímco proměnné představují data, řídicí konstrukce určují logiku, jakou se s těmito daty bude pracovat.

## 9.1 Blok kódu a rozsah platnosti

**Blok kódu**, vymezený složenými závorkami { a }, představuje v programování základní realizaci řídicí konstrukce sekvence. Jde o logický celek, ve kterém se instrukce vykonávají postupně. Tento strukturální prvek zároveň definuje ztv **rozsah platnosti** (scope), který ovlivňuje životnost proměnných, tedy úsek kódu ve kterém lze na danou proměnnou odkazovat.

Pokud se vytvoří proměnná *number* uvnitř daného bloku

```
1 {
2 int number = 42;
3 ...
4 }
```

K této proměnné lze pak přistupovat pouze uvnitř tohoto bloku ve kterém vznikla, za hranicemi bloku proměnná **number** již zaniká a nelze s ní již dál pracovat!

### 9.1.1 Vnořený blok kódu

Sekvence příkazů uvnitř bloku tvoří uzavřený kontext. Pokud se do sekvence vloží další dvojice složených závorek, vytvoří se tak **vnořený blok**. Tento mechanismus umožňuje systému efektivně členit sekvence kódu se kterými lze pracovat jako s celkem.

## 9.2 Větvení programu

### 9.2.1 Konstrukce if-else

### 9.2.2 Konstrukce switch-case

## 9.3 Iterace programu

### 9.3.1 Smyčka for

### 9.3.2 Smyčka while

## 9.4 Rekurze

# 10 Funkce

## 10.1 Lokální a globální proměnné

### 10.1.1 deklarace proměnné

# 11 Datové struktury

## 11.1 Typový alias (typedef)

## 11.2 Union

## 11.3 Enum

# 12 Paměťové ukazatelé

## 12.1 Pole

## 12.2 Řetězce

## 12.3 Rozložení paměti

## 12.4 Paměťové třídy

## 12.5 Ukazatel na funkce

# 13 Preprocesor