Úvod do programování

Petr Horáček

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 by jste 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.

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 Kontext/vnitřní stav algoritmu

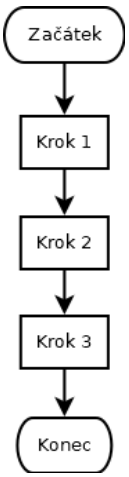
Algoritmus při svém průchodu vytváří tzv. **kontext** nebo také **vnitřní stav**. V reálném světě například v případě kuchařského receptu je kontext algoritmu stav v jakém se nachází vařené jídlo. V případě počítačového programu je kontext algoritmu tvořem hodnotami uloženými v paměti RAM. V průběhu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

2.4 Řídící konstrukce algoritmu

Každý algoritmus je tvořen kombinací tří základních konstrukcí, které určují, jak jsou jednotlivé kroky algoritmu prováděny.

2.4.1 Sekvence

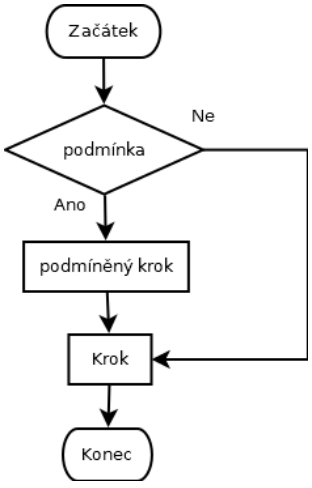
Sekvence je definovaná jako posloupnost kroků, pracující s daty, které v přesně stanovaném pořadí vedou k řešení problému.



Obrázek 2.4.1 Diagram sekvence

2.4.2 Selekce

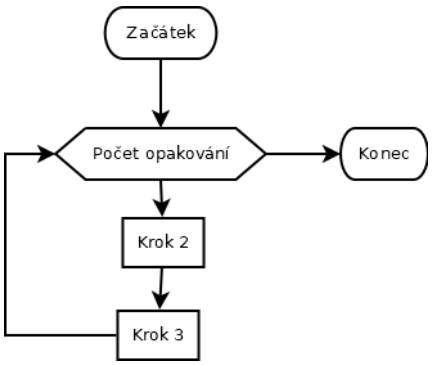
Selekce umožňuje rozhodování na základě podmínky, která část algoritmu se vykoná. Díky tomu lze vytvořit felexibilní chování, které reaguje na aktuální vnitřní stav algoritmu.



Obrázek 2.4.2 Diagram selekce (větvení)

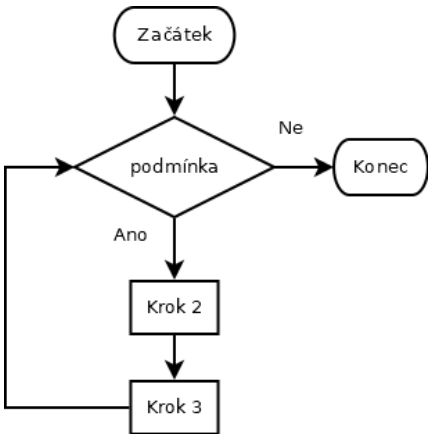
2.4.3 Iterace

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 2.4.3 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 2.4.4 Diagram podmíněné iterace

2.5 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.

2.5.1 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.

2.5.2 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 Programovací jazyk C

Jazyk C je jedním z nejstarších a nejvlivnějších programovacích jazyků v historii vývoje softwaru, který byl vyvinut v 70. letech 20. století v Bellových laboratořích. Díky svým vlastnostem, jednoduchosti a výkonnosti si získal široké uplatnění a stal se základem pro mnoho moderních programovacích jazyků, jako je C++, JavaScript a dokonce i některé jazyky vyšších úrovní jako Java a Python.

3.1 Vlastnosti jazyka C

- **Nízká úroveň abstrakce** - Jazyk C poskytuje abstrakci na použitým hardwarem (na úrovni jazyka programátora nezajímá jaký procesor programuje), ale zároveň umožňuje přímí přístup k paměti.
- **Vysoká optimalizace** - Díky své jednoduchosti je možné využít vysokou míru optimalizace výsledného programu. Díky tomu poskytují výsledné programy vysoký výkon.
- **Jednoduchost a přímočarost** - Syntaxe jazyka C je jednoduchá a přímočará (v porovnání s jazyky vyšší úrovně).

3.2 Použití jazyka C

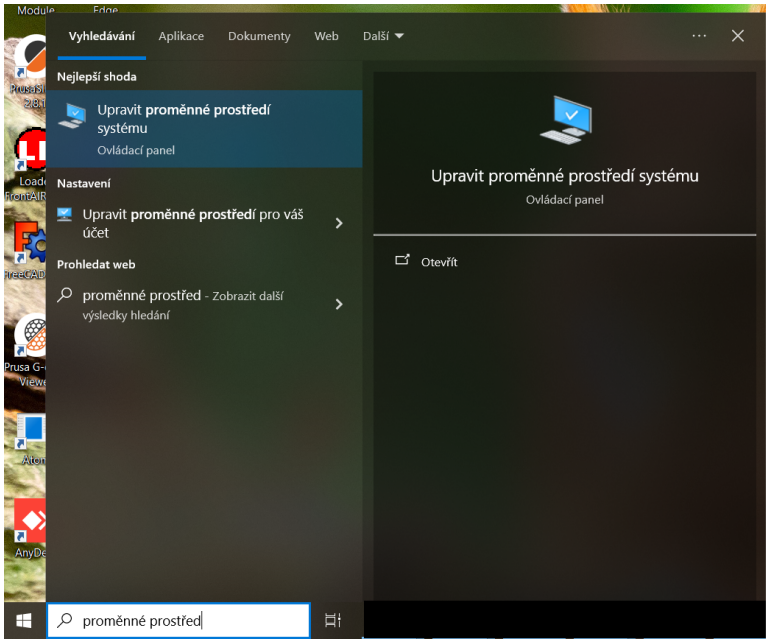
- **Systémové programování** - C je široce používán pro vývoj operačních systémů (například UNIX), ovladačů zařízení a dalších nízkoúrovňových aplikací, kde je vyžadován přímý přístup k hardwaru.
- **Aplikace s vysokým výkonem** - Vzhledem k tomu, že C poskytuje velkou kontrolu nad výkonem a pamětí, je ideální pro vývoj aplikací, kde jsou kladeny vysoké nároky na výkon, jako jsou grafické programy, hry, simulace nebo vědecké výpočty.
- **Embedded systémy** - C je jazykem první volby pro vývoj embedded (vestavěných) systémů, kde je efektivní využívání paměti a výkonu klíčové. Příkladem vestavěných systémů je například chytrý domácí spotřebič (robotický vysavač, pračka, ...)

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**, který má za úkol převést zdrojový kód na spustitelný soubor a nástroj pro usnadnění a automatizaci překladu **make**.

4.1 Příprava prostředí na MS Windows

Na systémech MS Windows je nejjednodušší způsob pro nastavení prostředí pro vývoj v jazyce C použít program **msys2** (`https://www.msys2.org/`). Tento program vytvoří strukturu nástrojů, které jsou k dispozici na systémech Linux. Po instalaci je třeba nejprve třeba nastavit systém, aby mohl vyhledat nástroje, které se do počítače nainstalovali. K tomu je nutné upravit **proměnnou prostředí PATH**, do které je třeba přidat dvě nové cesty. Proměnné prostředí je možně upravovat zadáním příkazu do nabídky start: *upravit proměnné prostředí systému*



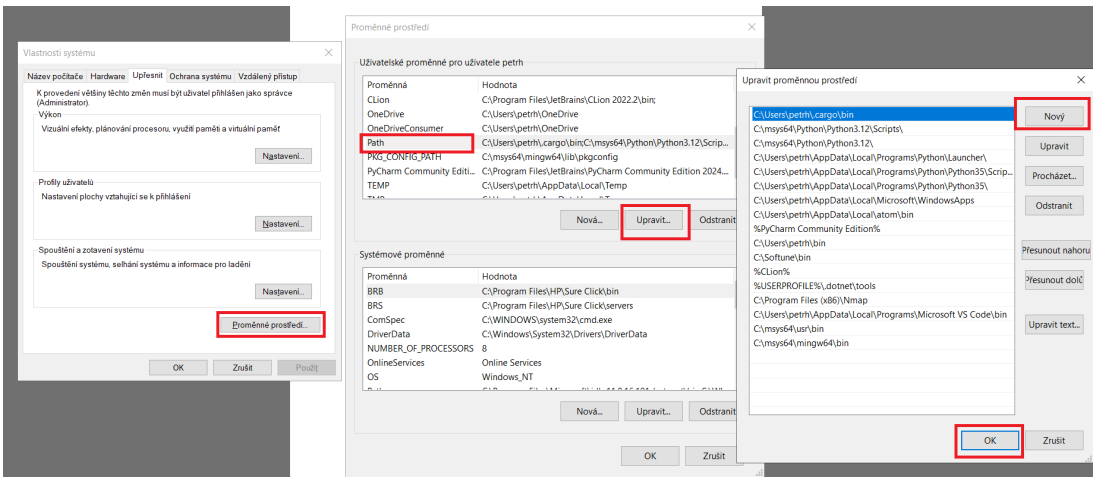
Obrázek 4.1.1 Nastavení proměnné prostředí Path

Poté se otevře okno pro nastavení proměnných prostředí ve kterém je tlačítko **Proměnné prostředí**, které otevře okno s proměnnými prostředí, které jsou v systému vytvořené. V sekce **Uživatelské proměnné** se nachází výčet proměnných prostředí pro aktuálně přihlášeného uživatele a mezi nimi se nachází proměnné **Path**. Po jejím označení je třeba stisknout tlačítko upravit. Následně by se mělo otevřít okno pro úpravu hodnot v této proměnné. Nyní je nutné přidat dvě nové cesty pomocí tlačítka **Nový** a nastavit jejich hodnotu na:

```
C:\msys2\usr\bin
```

```
C:\msys2\mingw64\bin
```

Následně stačí jen vše uložit stiskem tlačítka OK a cesty k nástrojům systému msys2 by měly být nastavené.



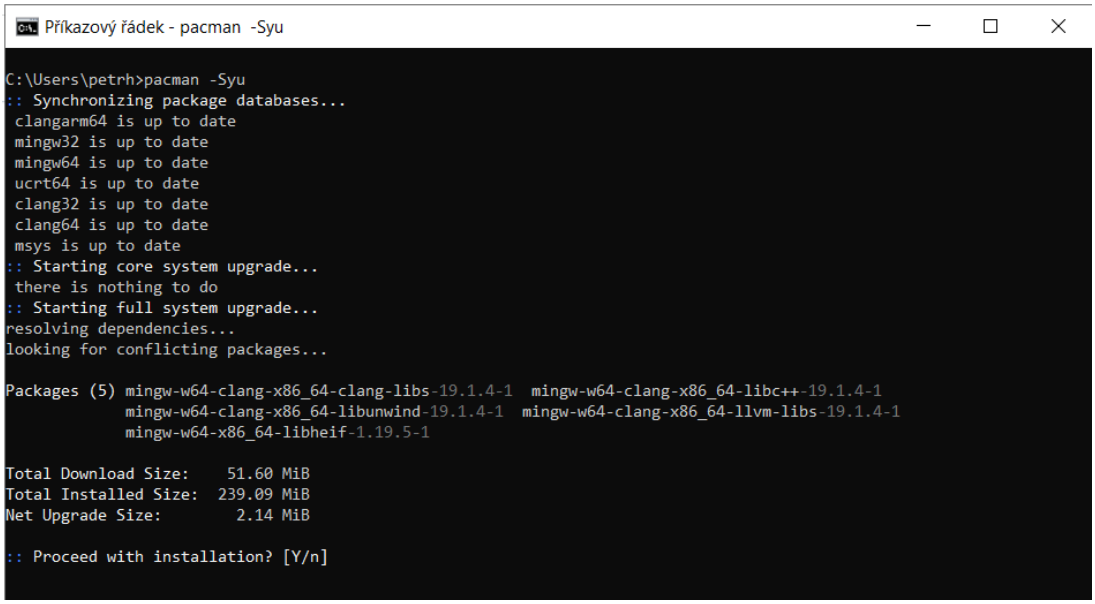
Obrázek 4.1.2 Okno pro nastavení proměnných prostředí

Nově nastavené cesty je možné věřit otevřením příkazové řádky pomocí zadání příkazu *cmd* do nabídky start a zadat příkaz, který by měl provést aktualizaci systému msys2:

```
$ pacman -Syu
```

(Prosím nekopírujte příkaz se znakem $, ten značí že se jedná o terminálový vstup)

V případě, že jsou cesty správně nastavené by se měl spustit balíčkový správce *pacman*, který slouží k instalaci a aktualizaci programů do prostředí msys2 a požádá vás o potvrzení spuštění instalace aktualizací:



Obrázek 4.1.3 Okno pro nastavení proměnných prostředí

Aktualizace a jiné instalace se potvrdí prostým stiskem klávesy *Enter*.

Po aktualizaci prostředí msys2 je potřeba nainstalovat kompilátor pro jazyk C **gcc**, který má za úkol převést zdrojový k na spustitelný a nástroj pro automatizaci překladu **make**. To se provede zadáním příkazu:

```
$ pacman -S gcc make
```

Některé knihovny, které jsou použity při programování jsou uloženy na specifické cestě v systému. Ta se může lišit v závislosti na použitém systému (MS Windows s prostředím

msys2, GNU Linux, Mac OS). Z tohoto důvodu se využívá pro vyhledání cest k instalovaným knihovnám nástroj **pkg-config**. Nástroj pkg-config využívá malé soubory, které osahují konfiguraci pro knihovnu instalovanou v lokálním systému, které mají příponu .pc a jsou uloženy v konfiguračním adresaři. Na systému msys2, ale nástroj pkg-config neví kde má tyto soubory hledat a je nutné mu je nastavit. K tomu slouží proměnná prostředí **PKG_CONFIG_PATH**, kterou je nutné nastavit stejně jako proměnnou prostředí path v předchozím bodě. V tomto případě, jsou konfigurační soubory uloženy v adresáři:

```
C:\msys2\ming64\lib\pkgconfig
```

4.2 Příprava prostředí na GNU Linux

Na systémech Linux bývá často nástroj gcc a make dostupný již v základní instalaci. Z tohoto důvodu je příprava prostředí na vývoj v jazyce C na systémech Linux snazší než na jiných systémech. Pokud nástroje gcc a make není v základní instalaci, záleží na distribuci instalovaného Linuxového systému.

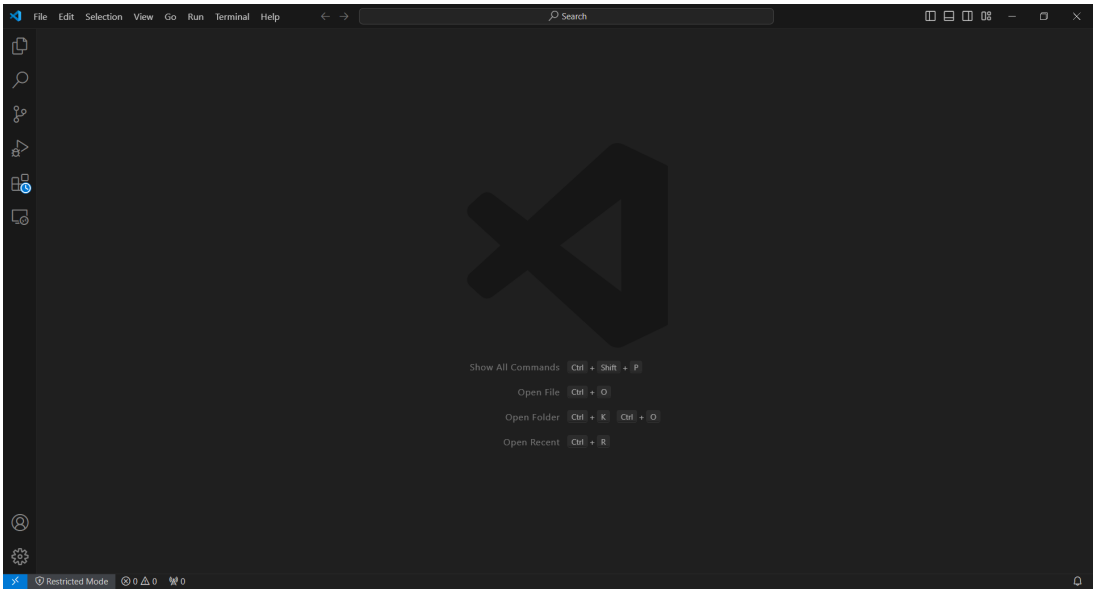
Na linuxové distribuci Ubuntu je možné nástroje gcc a make instalovat buď prostřednictvím nástroje **Ubuntu Software Center** a nebo pomocí příkazové rádky:

```
$ sudo apt install gcc make
```

4.3 Příprava prostředí na Mac OS

4.4 IDE

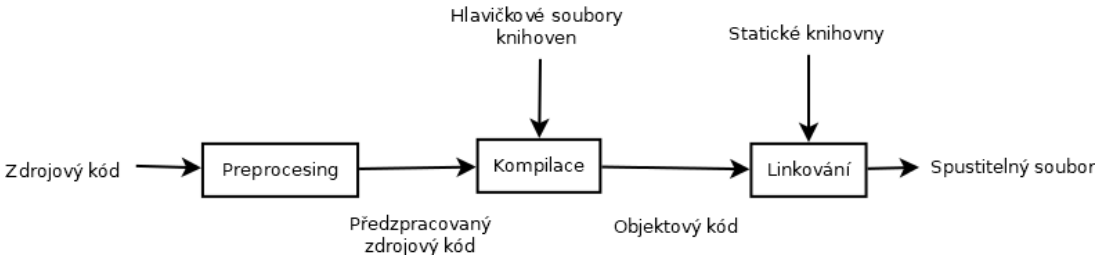
Poté co jsou v systému nainstalované nástroje gcc a make je prostředí pro vývoj programů vjazyce C připraveno k použití, ale je potřeba nainstalovat tzv. IDE (Integrated development environment), zjednodušeně řečeno textový editor, který obsahuje nástroje pro zjednodušení psaní zdrojových kódu. Nejjednodušší IDE je **Visual studio code** (`https://code.visualstudio.com/`). Jeho instalace je jednoduchá a přímočará.



Obrázek 4.4.1 Visual studio code

5 Proces překladu zdrojových kódů

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 5.0.1 Proces překladu zdrojových kódů

5.1 Fáze překladu

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

5.1.1 Preprocesor (úprava zdrojového kódu)

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

5.1.2 Překlad (kompilace do strojově nezávislého kódu)

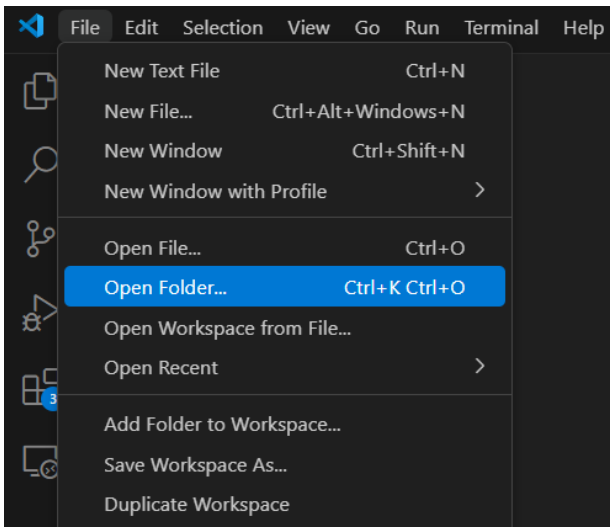
V této fázi překladač analyzuje a převádí zdrojový kód na tzv. **objektový kód**. Objektový kód obsahuje strojový kód procesoru, ale nejedná se ještě o spustitelný kód.

5.1.3 Linkování (spojování a vytváření spustitelného souboru)

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

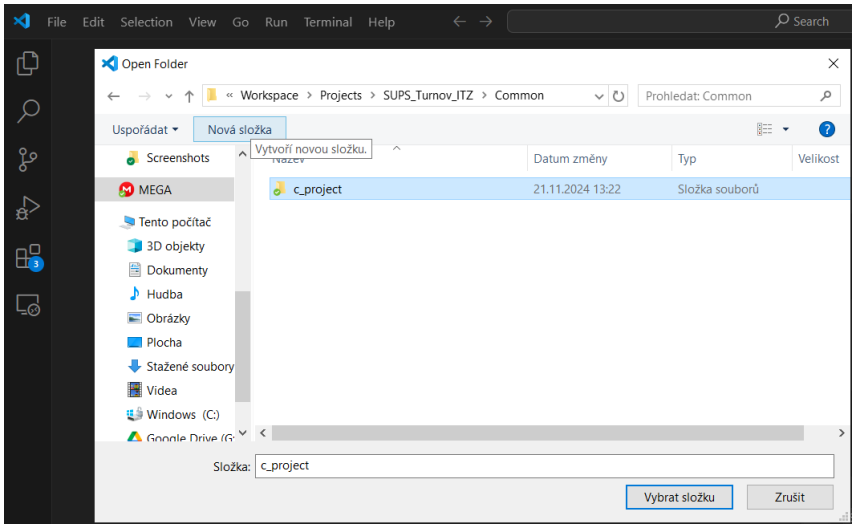
5.2 Založení projektu

Založení softwarového projektu v základu zanená vytvořit **projektovou složku** (složku ve které se nacházejí zdrojové kódy a další pomocné soubory). V editoru Visual Studio Code se projektový adresář vytvoří a otevře z hlavního menu: $File \rightarrow OpenFolder$



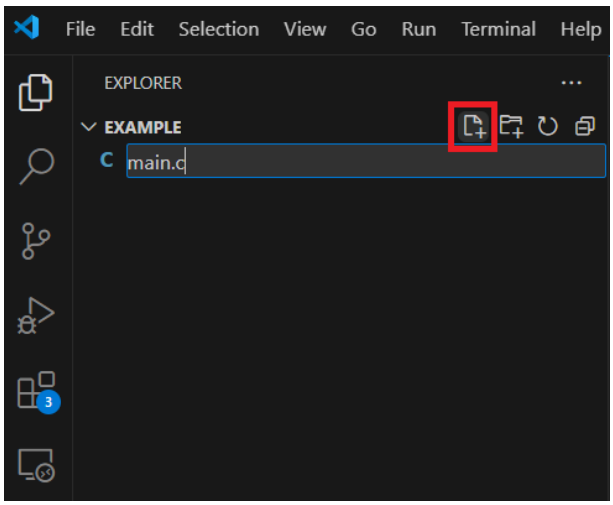
Obrázek 5.2.1 Otevření projektové složky

Otevře se okno pro výběr projektové složky. V tuto chvíli je možné buď otevřít existující projekt a pokračovat v práci a nebo je možné vytvořit složku pro založení nového projektu. Nová projektová složka se zpravidla jmenuje stejně jako nově založený projekt.



Obrázek 5.2.2 Výběr/založení projektové složky

Po otevření projektové složky se načte její obsah do přehledového okna Visual studia



Obrázek 5.2.3 Vytvoření nového souboru

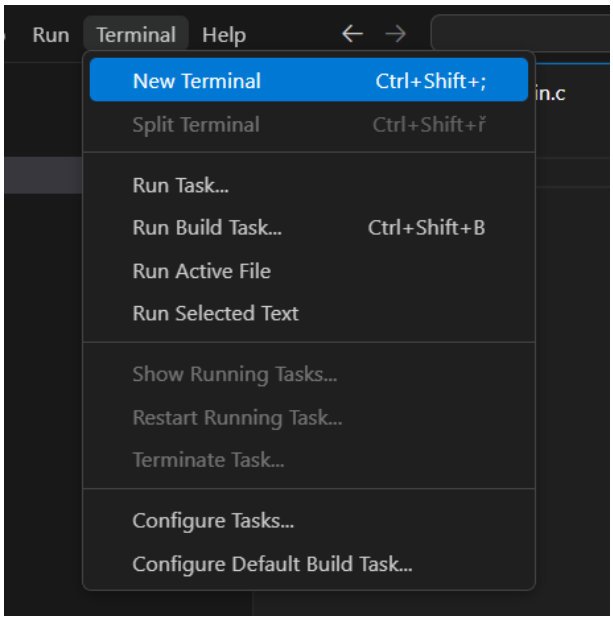
5.2.1 Základní struktura projektu v jazyce C

Každý projekt v jazyce C se skládá minimálně z jednoho zdrojového souboru, který se standardně navývá *main.c*. V tomto souboru se nachází tzv. **funkce main**, kterou začíná vykonávání spuštěného programu:

```
#include <stdio.h>

int main(void) {
    printf("Program exit..\n");
    return 0;
}
```

Pro přeložení zdrojového kódu je třeba nejprve otevřít v editoru VS Code terminálový vstup z hlavního menu: *Terminal* → *NewTerminal*



Obrázek 5.2.4 Otevření terminálového vstupu

K překladu zdrojového souboru main.c slouží kompilátor gcc, kterému je předán název vstupního souboru:

```
$ gcc main.c
```

V tuto chvíli, pokud je v souboru main.c správně zapsán a uložen kód funkce main, by výsledkem překladu vzniknout soubor pojmenovaný *a.exe* (na systémech GNU Linux a Mac OS *a.out*), který tvoří nově vytvořený spustitelný soubor. Tento soubor lze spustit z terminálového vstupu zavoláním jeho jména:

```
$ ./a.exe
```

Výstupní spustitelný soubor lze při překladu automaticky pojmenovat pomocí nastavení $-o < jméno >$, které se předá překladači gcc:

```
$ gcc main.c -o <jméno>
```

V tuto chvíli vznikne překladem spustitelný soubor, který bude pojmenovaný jménem zadaným při překladu. Protože každý projekt má svou vlastní strukturu, to znamená obsahuje různé zdrojové soubory a různě rozmístěné, je nutné překladači gcc předat různá nastavení a odkazy na soubory. Aby si programátor nemusel toto nastavení pametovat a složitě jej zadávat do terminálového vstupu, je možné vytvořit tzv. **překladová pravidla**, která se uloží do souboru **Makefile**. Jednotlivá překladová pravidla se zapisují do souboru Makefile ve formátu:

```
<název překladového pravidla>:
    <obsah překladového pravidla>
```

Obsah překladového pravidla musí být vždy odsazen od začátku řádku pomocí klávesy *Tab*. V případě, že by obsah překladového pravidla nebyl odsazen od začátku řádku a nebo byl odsazen pomocí mezer, program make, by hlásil chyby.

V případě, že v každém projektu budeme existovat soubor Makefile, který bude obsahovat stejně pojmenovaná pravidla, ale jejich obsah bude specifický pro daný projekt, získáme jednotný způsob jak překládat všechny naše projekty a nemusí nás zajímat jaké nastavení pro je potřeba. Standardně je dobré vytvořit tři překladová pravidla: build, exec a clean. Základní obsah souboru Makefile by měl být:

```
TARGET=jmeno_vystupniho_souboru

build:
    gcc main.c -o $(TARGET)

exec: build
    ./$(TARGET)

clean:
    rm -vf ./$(TARGET)
```

Soubor Makefile si pak načte program **make** a umožňuje spustit konkrétní překladové pravidlo, jehož název se zadá jako vstupní parametr:

```
$ make <název překladového pravidla>
```

Pro překlad projektu slouží příkaz:

```
$ make build
```

Pro překlad a současně sputění slouží příkaz:

```
$ make exec
```

A pro odstranění nepotřebných dočasných souborů v projektovém adresáři slouží příkaz:

```
$ make clean
```

6 Reprezentace dat v paměti PC

Počítače jsou postaveny na elektronických obvodech, které pracují se dvěma základními stavy:

- **Zapnuto** - proud prochází, což odpovídá logické hodnotě 1.
- **Vypnuto** - proud neprochází, což odpovídá logické hodnotě 0.

Tento dvoustavový systém je jednoduchý, levný na výrobu a velmi spolehlivý. Složitější systémy (například desetistavové) by vyžadovaly přesnější měření a byly by náchylnější k chybám kvůli šumu nebo odchylkám v signálu. Kombinací těchto nul a jedniček lze vytvořit složitější struktury, které reprezentují různé typy informací.

6.1 Jednotky paměti

6.1.1 Bit

Bit (zkratka z "binary digit") je základní jednotka informace v počítači. Bit může mít pouze dvě hodnoty: 0 nebo 1.

6.1.2 Bajt

Pokud se spojí dohromady 8 bitů, lze získat 256 kombinací hodnot jednotlivých bitů, které mohou kódovat hodnoty od 0 do 255. Spojení 8 bitů se říká **Bajt**. Pro reprezentaci paměťové kapacity úložných zařízení se běžně používají násobky bajtů:

- **Kilobajt** - 1024 bajtů
- **Megabajt** - 1024 kilobajtů
- **Gigabajt** - 1024 megabajtů
- ...

6.2 Číselné soustavy

Číselná soustava je způsob, jakým jsou číslné hodnoty zapisovány a reprezentována pomocí určitého počtu symbolů. Například běžně používaná desítková číselná soustava má 10 číslic (0, 1, .. 9). Každá soustava má **základ** (například 2 pro dvojkovou nebo 10 pro desítkovou), který určuje, kolik různých číslic se používá a jak se čísla skládají. V případě, že je třeba zapsat v číselné soustavě hodnotu, která je větší než jakou je možné vyjádřit pomocí základních číslic číselné soustavy, je nutné provést přechod do výšícho číselného řádu. **Číselný řád** označuje pozici číslice v čísle, která určuje její hodnotu podle mocniny základu číselné soustavy. Například v desítkové soustavě jsou číselné řády:

- **Jednotky** - $10^0 = 1$
- **Desítky** - $10^1 = 10$
- **Stovky** - $10^2 = 100$
- **Tisíce** - $10^3 = 1000$
- ...

Každé číslo pak lze zapsat v polynomiálním tvaru. **Polynomiální tvar** čísla je způsob, jak zapsat číslo jako součet jednotlivých číslic vynásobených jejich hodnotou podle jejich pozice (řádu). Například:

$$123_{(10)} = (10^2 \cdot 1) + (10^1 \cdot 2) + (10^0 \cdot 3)$$

Stejným způsobem lze definovat číselné řády také v binární soustavě, pouze jako základ zvolíme číslo 2:

- **Jednotky** - $2^0 = 1$
- **Desítky** - $2^1 = 2$
- **Stovky** - $2^2 = 4$
- **Tisíce** - $2^3 = 8$
- ...

Stejným způsobem pak můžeme zakódovanou hodnotu v dvojkové soustavě převést na hodnotu v desítkové soustavě:

$$1011_{(2)} = (2^3 \cdot 1) + (2^2 \cdot 0) + (2^1 \cdot 1) + (2^0 \cdot 1) = 8 + 0 + 2 + 1 = 11_{(10)}$$

6.3 Reprezentace dat v číselné podobě

Počítače pracují výhradně s čísly. Ať už jde o text, obrázky, zvuk, nebo jiné informace, vše musí být převedeno na čísla, protože pouze ta dokážou počítače zpracovat. Tento převod umožňuje počítačům být univerzálním nástrojem pro zpracování nejrůznějších druhů dat.

6.3.1 Text

Text v počítači je reprezentován jako sekvence znaků a každý znak je reprezentován určitou číselnou hodnotou. Aby každý počítač věděl jak interpretovat danou hodnotu reprezentující znak, vznikla standardizovaná tabulka znaků a k nim přiřazených číselných hodnot, které se říká ASCII tabulka.

Dec	Znak	Dec	Znak	Dec	Znak	Dec	Znak
0	NULL	32	Mezera	64	@	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	”	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	|	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

«««¡ HEAD Takže například text, ”Hello World” se v ascii zapíše pomocí sekvence čísel: 72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100.

6.3.2 Obrázky

Obrázky jsou složené z mřížky barevných bodů, kterým se říká pixely. Rozměr obrázku například 800x600 znameá, že obrázek je velký 800 pixelů na výšku a 600 pixelů na šířku. Celkový počet pixelů v obrázku je tedy roven součinu pixelové výšky a šířky například: $800 * 600 = 480000$ pixelů. Každý pixel obsahuje kombinaci tří základních barev RGB (Red, Green, Blue). Každá hodnota určující poměr základní barvy je reprezentována jedním bajtem, který umožňuje reprezentovat číselný poměr ve výsledném odstínu: $\{R = 200, G = 150, B = 10\}$. Pixel RGB, je tedy 3-bajtová (24-bitová) hodnota, která umožňuje

reprezentovat: $2^24 = 256 \cdot 256 \cdot 256 = 16777216$ unikátních barevných odstínů. ======= Takže například text, "Hello World" se v ascii zapíše pomocí sekvence čísel: 72, 101, 108, 108, 32, 87, 111, 114, 108, 100.

6.3.3 Obrázky

Obrázky jsou složené z mřížky barevných bodů, kterým se říká pixely. Rozměr obrázku například 800x600 znameá, že obrázek je velký 800 pixelů na výšku a 600 pixelů na šířku. Celkový počet pixelů v obrázku je tedy roven součinu pixelové výšky a šířky například: $800 * 600 = 480000$ pixelů. Každý pixel obsahuje kombinaci tří základních barev RGB (Red, Gree, Blue). Každá hodnota určující poměr základní barvy je reprezentována jedním bajtem, který umožňuje reprezentovat číselný poměr ve výsledném odstínu: $\{R = 200, G = 150, B = 10\}$. Pixel RGB, je tedy 3-bajtová (24-bitová) hodnota, která umožňuje reprezentovat: $2^24 = 256 \cdot 256 \cdot 256 = 16777216$ unikátních barevných odstínů. »»»¿ 3560f88c2cf79f35b4bd9d590535d84584e39adf

7 Proměnné

Paměť RAM slouží k uložení nejen strojových instrukcí spuštěného programu, ale také k uložení dočasných dat, které si program při svém vykonávání vytváří. Taková dočasná data můžou být mezivýsledky výpočtů, obsah načteného souboru z disku, … K práci s daty uloženými v paměti RAM slouží tzv. **programové proměnné**. V programování jsou proměnné jedním z nejzákladnějších a nejdůležitějších konceptů. Zjednodušeně si proměnnou lze představit jako pojmenované místo v RAM paměti, na které je možné data ukládat a následně opět přečíst.

7.1 Datové typy

7.2 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:

- **char** - 1 bajt - znak / celé číslo
- **short** - 2 bajty - celé číslo
- **int** - 4 bajty - celé číslo
- **long** - 8 bajtů - celé číslo
- **long long** - 16 bajtů - celé číslo
- **float** - 4 bajty - desetinné číslo
- **double** - 8 bajtů - desetinné číslo
- **long double** - 16 bajtů desetinné číslo

«««¡ HEAD Při rozhodování, který primitivní datový typ použít v jazyce C, je klíčové zvážit požadavky na paměť, rozsah hodnot a účel proměnné. Jednoduše řeščeno, 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 nebo long 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. Typ unsigned lze zvolit, pokud víte, že hodnota bude vždy kladná, čímž se maximalizuje rozsah. Správná volba datového typu nejen zlepší efektivitu programu, ale také zvýší jeho čitelnost a bezpečnost.

======= »»»¿ 3560f88c2cf79f35b4bd9d590535d84584e39adf

7.3 Deklarace a definice proměnné

7.4 Znaménkový celočíselný datový typ

8 Řídící konstrukce

9 Funkce

10 Datové struktury

11 Paměťové ukazatelé

12 Preprocesor

13 Soubory a řetězce