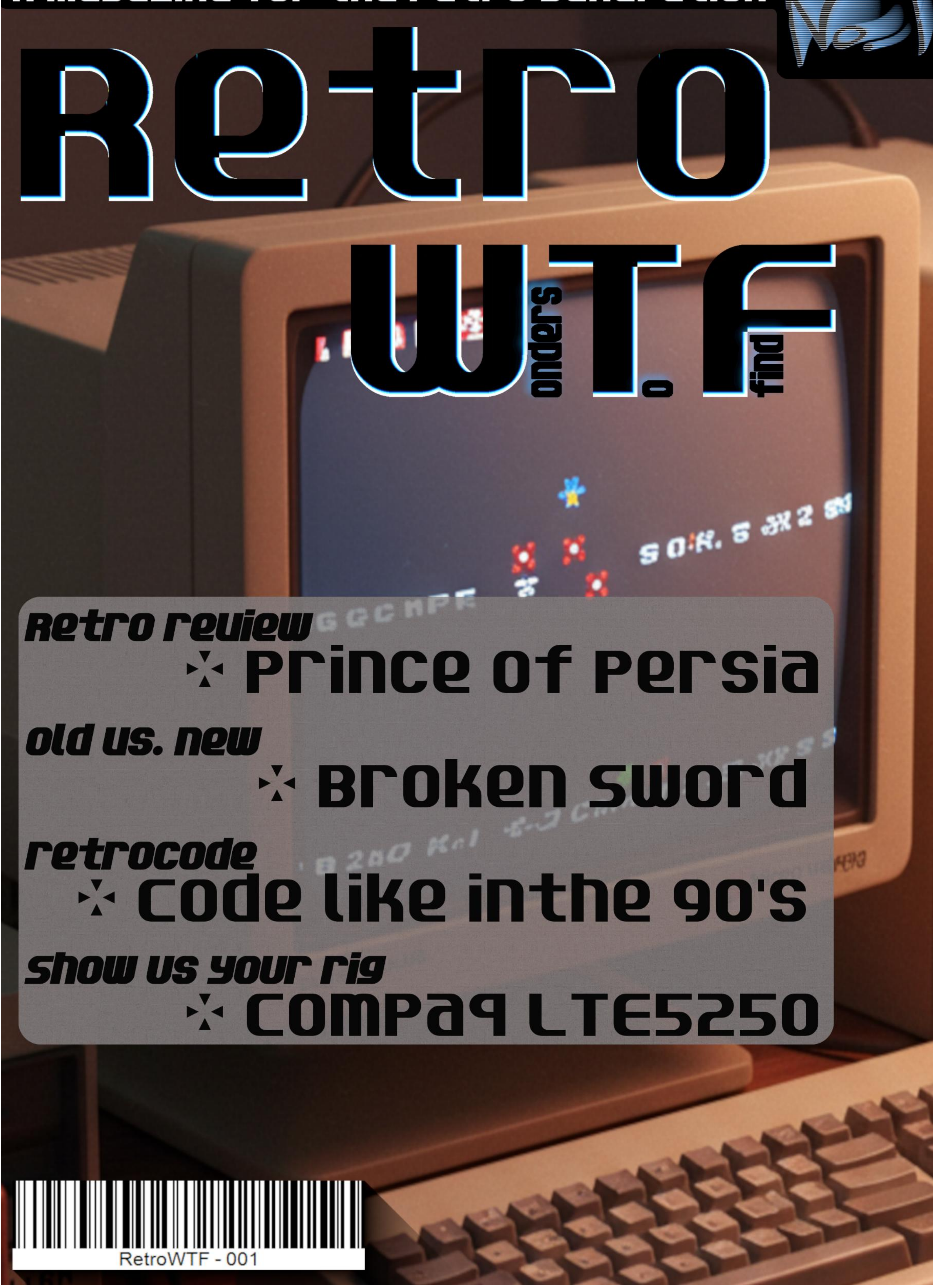
a magazine for the retro generation

No. 1

Retro WTF

Wonders To Find



Retro review

⁜ Prince of Persia

old vs. new

⁜ Broken Sword

retrocode

⁜ Code like inthe 90's

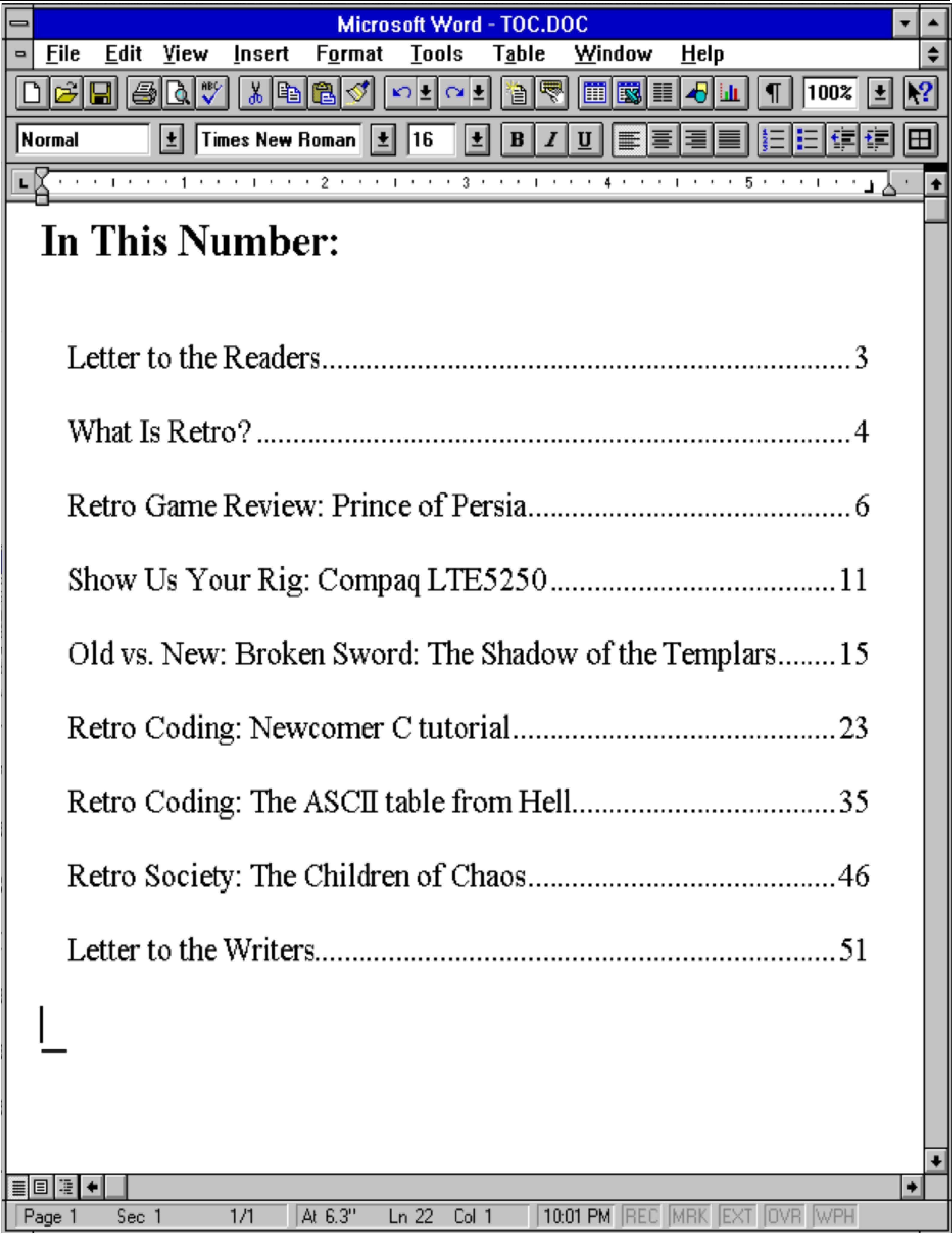
show us your rig

⁜ COMPAQ LTE5250

RetroWTF - 001

id INC
MICROPROSE
SYSTE
Prese
APOGEE
PRESENTS
MEMBER SOFTWARE PUBLISHERS ASSOCIATION (SPA)
Broderbu
S O F T W A
IDOS
E R A C T I V E
ACTIVISION
Presents
TRON
e Software
ENSEMBLE STUDIOS
®
BULLF
PRODUCTIO

TABLE OF CONTENTS


Microsoft Word - TOC.DOC
File Edit View Insert Format Tools Table Window Help
100%
Normal
Times New Roman
16
Page 1 Sec 1 1/1 At 6.3" Ln 22 Col 1 10:01 PM REC MRK EXT OVR WPH

In This Number:

Dear Readers ...

We all know that reading yet another article about Prince of Persia isn’ t what we truly crave. We devoured every interview, every behind-the-scenes detail, every retrospective, years ago. And no, replaying Indiana Jones and the Fate of Atlantis isn’ t our deepest desire – we’ ve already uncovered every secret, solved every puzzle, and watched the credits roll more times than we can count.

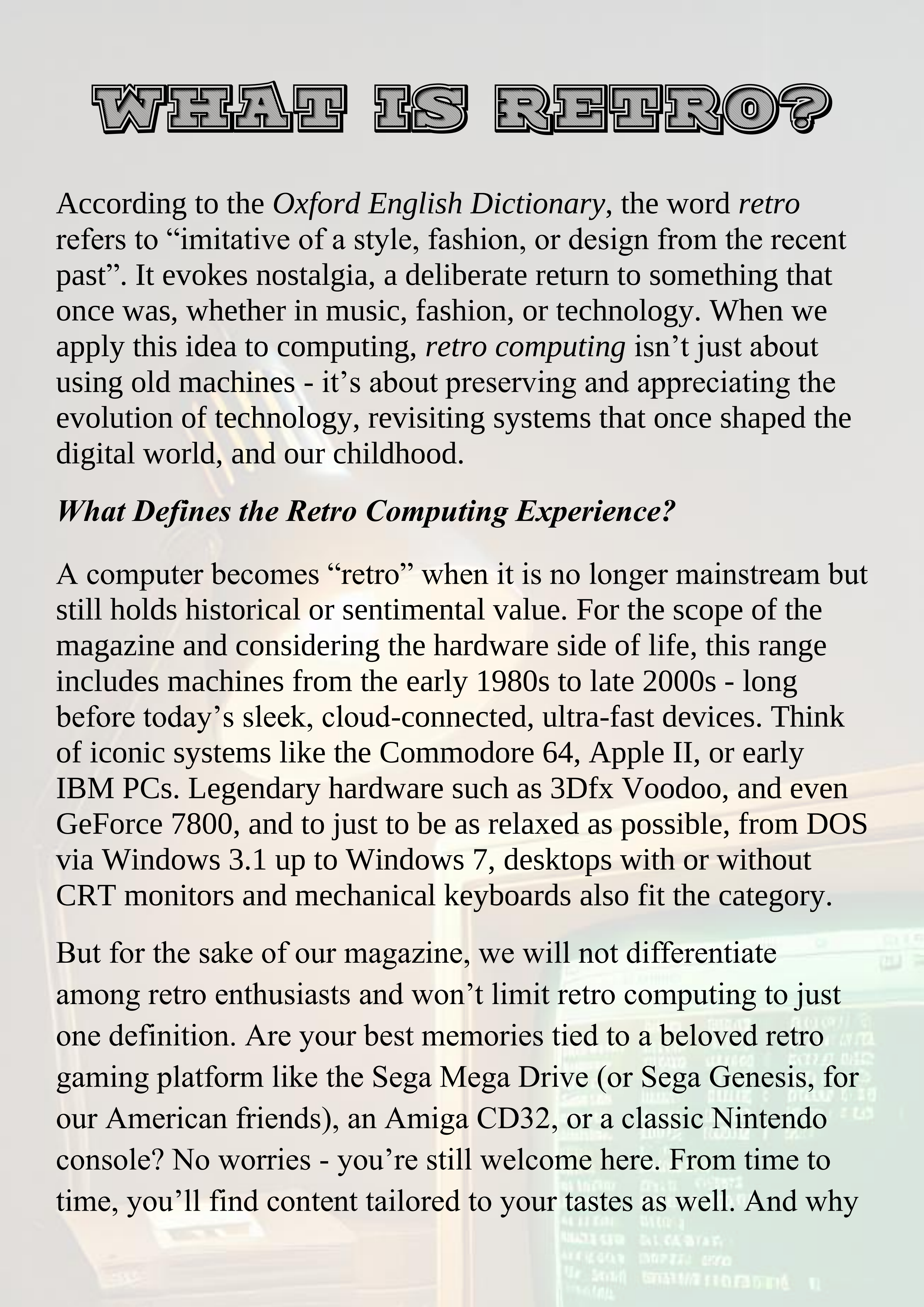
What we truly long for is to remember. To close our eyes and be transported back to the days when booting up a game felt like stepping into another world. When nothing mattered more than making that perfect jump, solving that one impossible riddle, or just spending hours exploring every pixel of a hand-crafted universe. When the hardest decision wasn’ t about work, bills, or responsibilities, but whether to wield a sword or a staff, whether to side with the Brotherhood or the Order, or whether to spend our last in-game gold on potions or better gear. Remembering a time when death meant, that you just restart from the last checkpoint and in best cases you got some loot of your dead body slain by Diablo. A time when taking care of your character used to be to drink a few health potions and continue the adventuring.

And yet, nostalgia isn’ t just about looking back – it’ s also about carrying forward what made those days special. It’ s about preserving the joy, the wonder, and the magic of gaming, even as technology evolves and our lives become more complicated.

That’ s why this magazine isn’ t just another publication – it’ s a love letter to everything that made us who we are. It’ s for the ones who still get chills hearing the Secret of Monkey Island theme, who still instinctively know what IDDQD means, and who still dream of finding the hidden Ferrari on the Strogg’ s planet.

Most importantly, it’ s for you. This isn’ t just a magazine – it’ s our magazine. A place to share your stories, your memories, and your passion. A place where our collective history as gamers can live on – not just as nostalgia, but as inspiration for what comes next. So, whether you want to reminisce, rediscover, or rekindle that old spark – welcome. Let’ s keep the adventure alive, together.

WHAT IS RETRO?

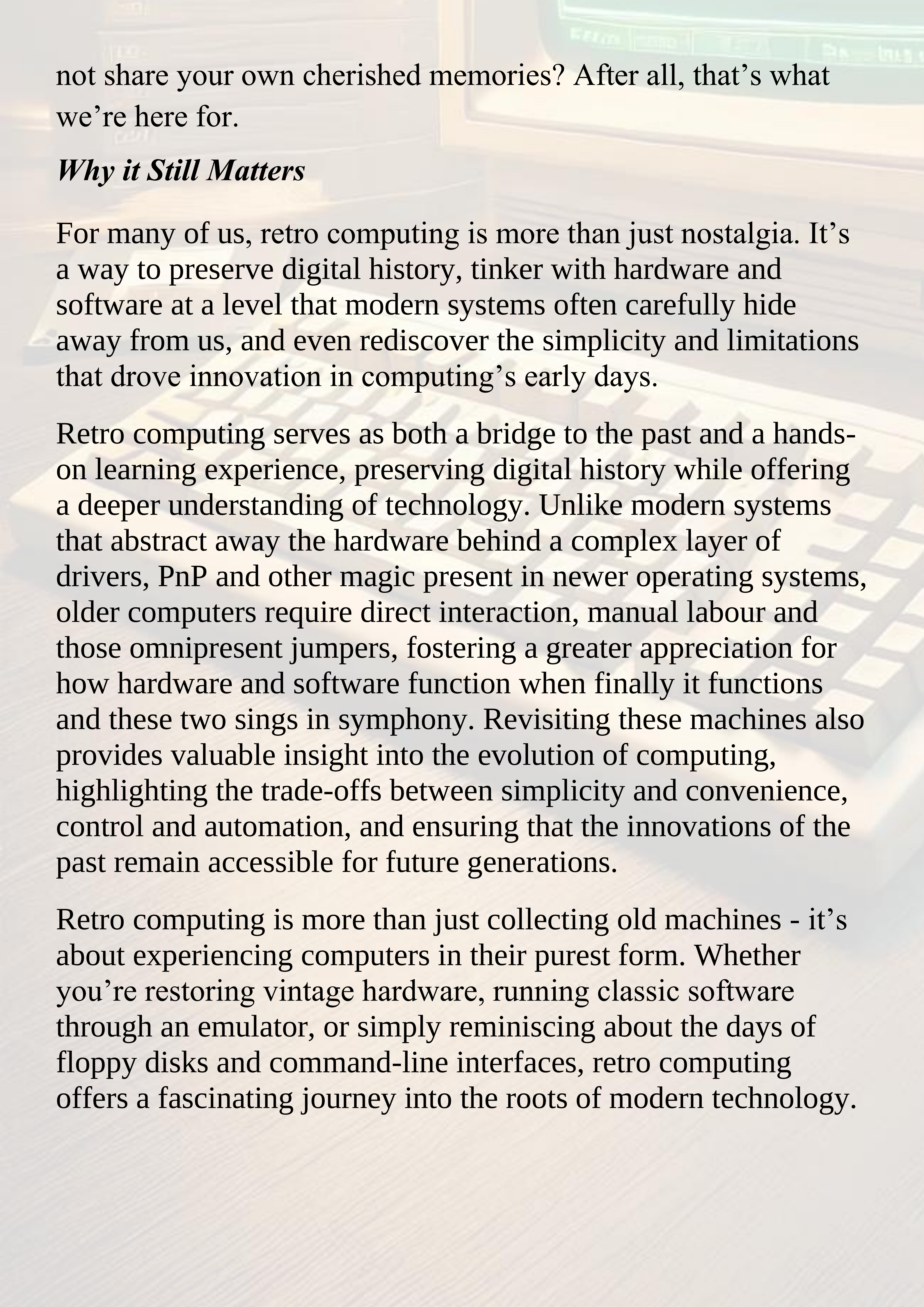


According to the *Oxford English Dictionary*, the word *retro* refers to “imitative of a style, fashion, or design from the recent past”. It evokes nostalgia, a deliberate return to something that once was, whether in music, fashion, or technology. When we apply this idea to computing, *retro computing* isn’t just about using old machines - it’s about preserving and appreciating the evolution of technology, revisiting systems that once shaped the digital world, and our childhood.

What Defines the Retro Computing Experience?

A computer becomes “retro” when it is no longer mainstream but still holds historical or sentimental value. For the scope of the magazine and considering the hardware side of life, this range includes machines from the early 1980s to late 2000s - long before today’s sleek, cloud-connected, ultra-fast devices. Think of iconic systems like the Commodore 64, Apple II, or early IBM PCs. Legendary hardware such as 3Dfx Voodoo, and even GeForce 7800, and to just to be as relaxed as possible, from DOS via Windows 3.1 up to Windows 7, desktops with or without CRT monitors and mechanical keyboards also fit the category.

But for the sake of our magazine, we will not differentiate among retro enthusiasts and won’t limit retro computing to just one definition. Are your best memories tied to a beloved retro gaming platform like the Sega Mega Drive (or Sega Genesis, for our American friends), an Amiga CD32, or a classic Nintendo console? No worries - you’re still welcome here. From time to time, you’ll find content tailored to your tastes as well. And why



not share your own cherished memories? After all, that’s what we’re here for.

Why it Still Matters

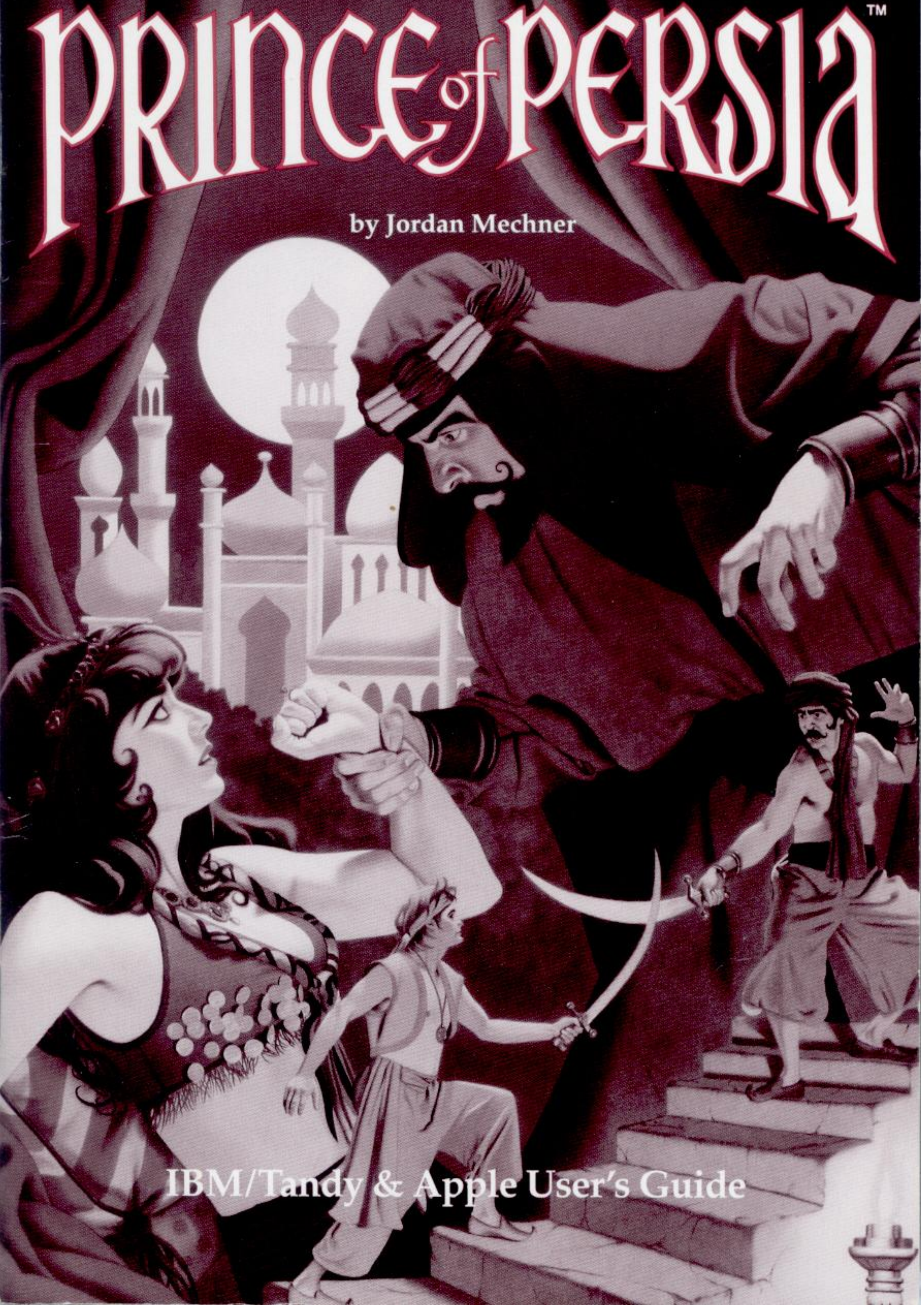
For many of us, retro computing is more than just nostalgia. It’s a way to preserve digital history, tinker with hardware and software at a level that modern systems often carefully hide away from us, and even rediscover the simplicity and limitations that drove innovation in computing’s early days.

Retro computing serves as both a bridge to the past and a hands-on learning experience, preserving digital history while offering a deeper understanding of technology. Unlike modern systems that abstract away the hardware behind a complex layer of drivers, PnP and other magic present in newer operating systems, older computers require direct interaction, manual labour and those omnipresent jumpers, fostering a greater appreciation for how hardware and software function when finally it functions and these two sings in symphony. Revisiting these machines also provides valuable insight into the evolution of computing, highlighting the trade-offs between simplicity and convenience, control and automation, and ensuring that the innovations of the past remain accessible for future generations.

Retro computing is more than just collecting old machines - it’s about experiencing computers in their purest form. Whether you’re restoring vintage hardware, running classic software through an emulator, or simply reminiscing about the days of floppy disks and command-line interfaces, retro computing offers a fascinating journey into the roots of modern technology.

PRINCE of PERSIA™

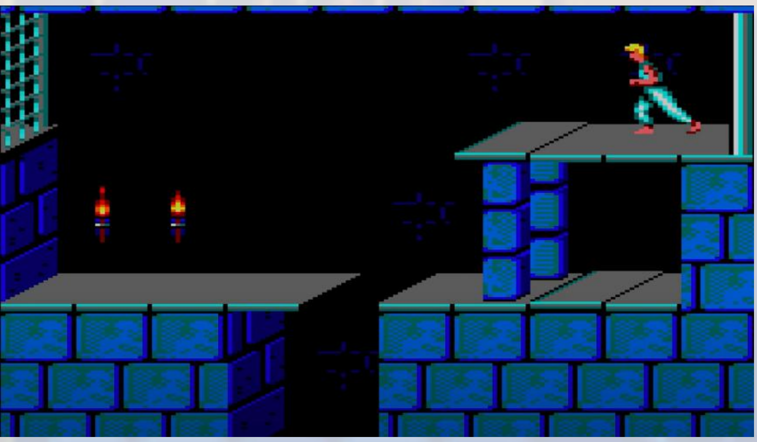
by Jordan Mechner



IBM/Tandy & Apple User's Guide

Retro-review: Prince of Persia (1989)

Prince of Persia is a timeless classic and a personal favorite of the author of these lines, that laid the foundation for his fascination for cinematic platformers, while the game continues to be remembered for its innovative design, fluid animations, and gripping gameplay. Released in 1989 and developed by Jordan Mechner, this game pushed the technical limits of its time and set new standards for storytelling in video games.



Running on an Amstrad CPC

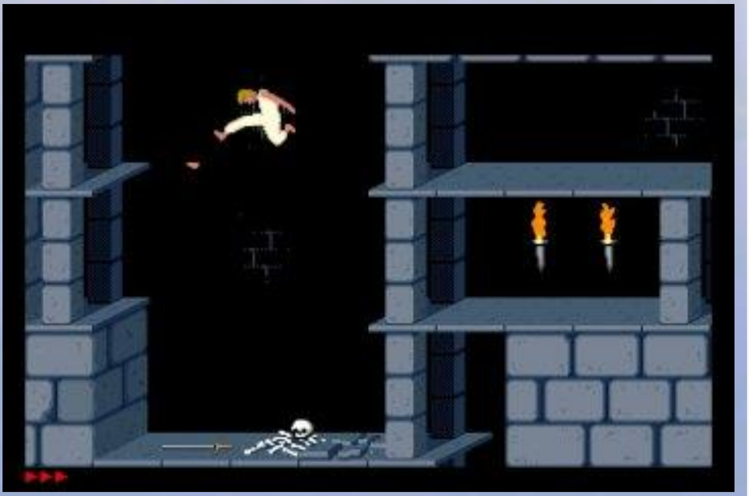
Gameplay

At its core, **Prince of Persia** is a side-scrolling action-adventure game that challenges players to navigate sparsely lit dungeons, decorated with torches, more or less alive skeletons, solve "push me to open the door" puzzles, and engage in sword combat against colorful guards on a strict diet, or not so, at least one of them.

The goal is simple yet intense: rescue the princess within 60 in-game minutes. This time constraint adds a layer of urgency that amplifies the tension throughout the experience, to not to mention that beside watching the clock, we also have to fight all the guards, avoid all the spiky traps, drink till you get healthy again and last but not least, kill the evil Jaffar at the end.

The game is notorious for its difficulty. The precise platforming mechanics require players to master timing and patience (fall to death in the current pit, or wait till the next), while the traps and enemies demand quick reflexes and a precise timing strategy that evolves over time. And nerves of steel, when trying to navigate the guillotines, which in some certain situation come in pairs of three.

Despite its challenging nature, the game rarely feels unfair—its design encourages perseverance and rewards players for improving their skills, and as the author of these lines can confirm, it is definitely doable in the 60 minutes that you have to allocate from your busy daily schedule.



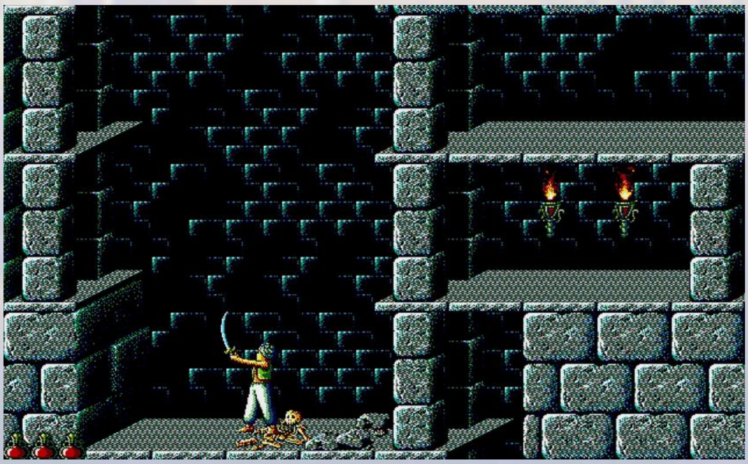
Jumping on the IBM PC

Animation and Graphics

One of the most striking features of **Prince of Persia** is its groundbreaking animation (Please, don't forget we are writing 1989!).

Mechner used rotoscoping, a technique that involves tracing over live-action footage, to create lifelike character movements. This level of realism was unprecedented in the late 1980s and gave the prince's movements—jumping, running, and fighting—a level of fluidity that still holds up today.

Rotoscoping is an animation technique where artists trace over live-action footage, frame by frame, to create realistic movement in an animated medium. This process allows animators to capture the fluidity and natural motion of real-life subjects, giving the resulting animation a lifelike and dynamic quality.



Acquiring the Sword on a Sharp X68000

The environments, while simple by modern standards (even when run on a GeForce 5090 with its hefty price tag), effectively evoke the atmosphere of a mystical Persian dungeon of an evil wizard, or the Far-East Castle of a good and beloved Sultan, whose daughter was sadly kidnapped by the aforementioned Grand Vizier.

However, we have some complaints concerning the safety of the caste, it seems the architect had forgot to put in some proper timbers, since walking on the floor sometimes can represent quite a danger. The dimly lit corridors, eerie traps, and minimalist aesthetic immerse players in the game's world, especially when falling down is as realistic as you would expect in real life.

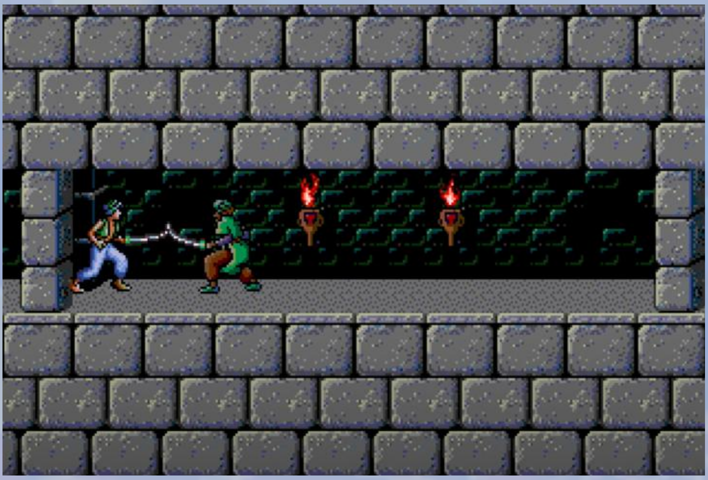
Sound And Music

The sound design complements the game's atmosphere, even for those players who could afford only the beeps of a PC Speaker. From the clang of swords during combat to the ominous creak of a falling tile, every sound enhances the tension and immersion. While the game lacks a continuous soundtrack, the sparse use of music during critical moments (such as near-death experiences or encounters with the villain Jaffar) adds emotional weight to the experience.

Legacy

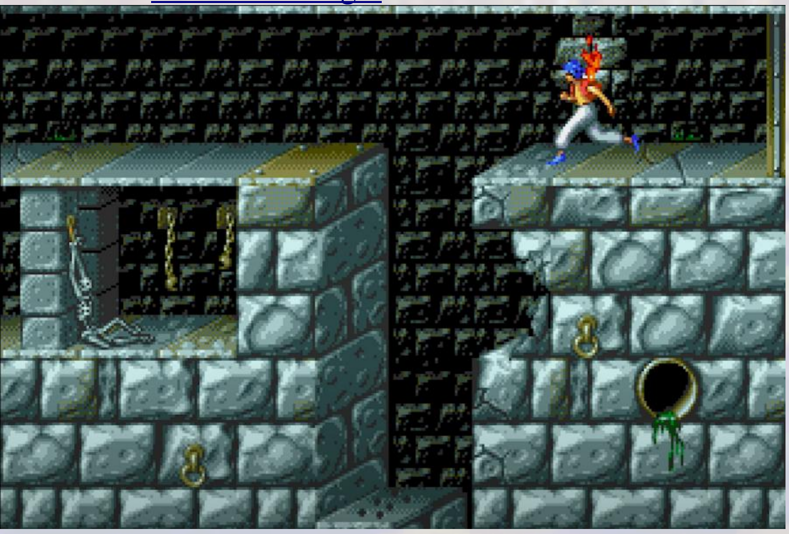
Not mentioning the fan made remake, 4D Prince of Persia which brings the difficulty to another levels, the game was ported to almost all platforms possible, some have graphics identical to the PC version (which came out in 1990, and boasted the full 256 color palette of the IBM PC, up from the 16, of the original version on Apple II) while some platforms have more fancier graphics. Almost all the platforms have identical level constructs to the original one, except a few ones, such as ports targeting consoles like SNES and Xbox.

For your delight we have included several screenshots of the program from various classic platforms in this article, and at the end of all, we will also add several online sources where you still can enjoy this favorite of ages.



The first fight on a TurboGrafx-CD

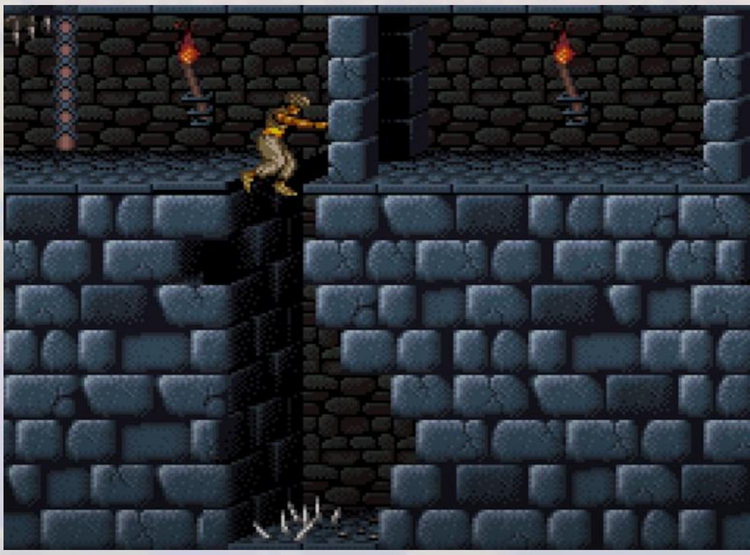
Prince of Persia 2, The Shadow ad the Flame is a direct descendant of the game with more traps, more tricks, better graphics and the wardrobe of the prince also suffered a decent upgrade, just like we can see on ports that have followed the original PC release. An extra bonus to this sequel is the cliffhanger which is explained on Jordan Mechner's site: https://www.jordanmechner.com/en/latest-news/#cliffhanger .



The graphics are beautiful on a Sega Genesis

The way we look back and reflect, it's better that we don't mention Prince of Persia 3D, however Prince of Persia: The Sands of Time and following episodes have actually updated the game up to be of a pretty decent quality, an as announced by Ubisoft, "The Sands of Time Remake" is an upcoming project that revisits the classic 2003 title. Initially slated for release in 2021, the project faced multiple delays to ensure quality improvements. As of now, the remake is scheduled for release in 2026, aiming to bring the beloved story and gameplay to a new generation with enhanced graphics and modernized mechanics.

Surprisingly good were the more recent releases of the game, such as Prince of Persia: The Lost Crown (2024) or The Rogue Prince of Persia (2024), but I think hardcore retro fans are waiting more of a remake of the classic ones, like The Shadow and Flame. Till this happens, feel free to play one of the many ports of the original game, we really can recommend the one that came out for Xbox 360 and PlayStation 3 a while ago: Prince of Persia Classic.



The SNES has also expanded levels.

Conclusion

Prince of Persia (1989) is more than just a game—it's a piece of video game history. Its innovative design, realistic for the time of release animation, and captivating gameplay make it a must-play for fans of retro games and platformers alike. Though it may test your patience, the sense of accomplishment from completing it is unparalleled.

Rating: 10/10
Verdict: A groundbreaking classic that remains a treasure for gamers even in 2025.

Play it in 2025

There are several sources online that have this game in their ever-growing library, such as, but not limited to:

- https://www.retrogames.cz/play_102-DOS.php
- https://playclassic.games/games/platform-dos-games-online/play-prince-of-persia-online/

Brøderbund

Show us your rig: The Compaq LTE 5250

COMPAQ LTE 5250

Ah, the Compaq LTE 5250 - a sweet little laptop from a time when "portable" meant "you can carry it, but you'll definitely regret it." Released in 1996, this bad boy was the dream machine for business professionals who wanted to look tech-savvy while lugging around what was essentially a high-tech cinder block.

Like all other laptops from those times, the LTE 5250 is also built like a lightweight tank, which means two things:

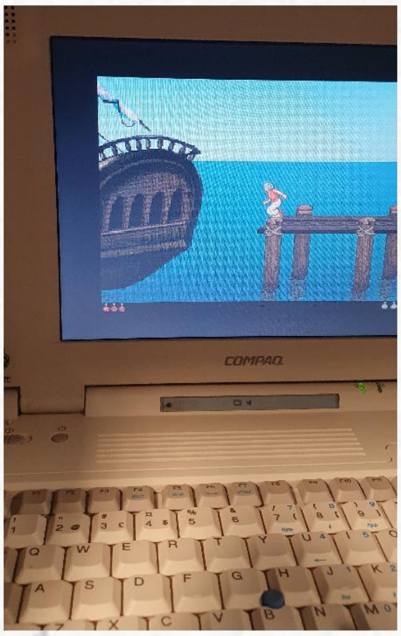
- Probably It could survive a nuclear apocalypse if stayed far enough from ground zero.

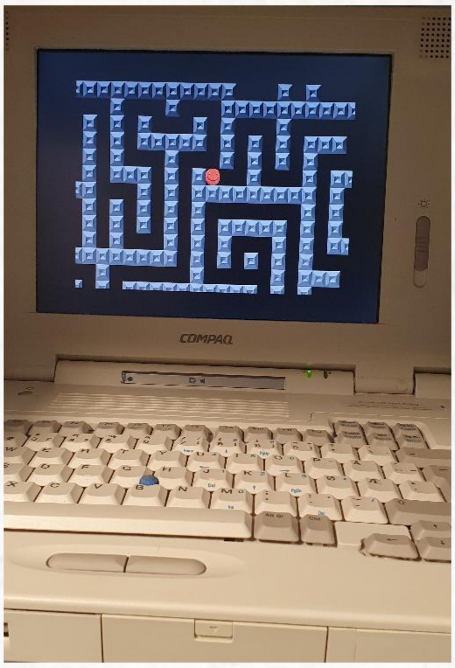
- It definitely can survive 20 years locked in the mouldy basement of the headquarters of and old defunct company from where I have rescued mine.

At 7 pounds (3.2 kg) without the hefty docking station, this thing wasn't just a laptop—it was a statement. A statement that said, "I don't care about my shoulders." The hinges? Rock solid. The chassis? A sturdy plastic masterpiece, which sadly shows signs of degradation over the last 40 years, like all old retro plastic does. The only thing missing was a built-in handle for easier transportation (or a warning label for back injuries).

The 10.4-inch screen comes in 800x600 resolution which was good enough for Windows 95, but let's be real, realistically it's about as crisp as an old VHS tape left in the sun. And sadly, this specific model does not accept the famous **Fn+T** key combination to stretch the screen to its full potential, thus leaving wonderful games like Heretic hanging in the middle of it.

Let's be real - this machine wrestles with Duke Nukem like it owes it money. Not because of the 120MHz Pentium - that little champ is actually pretty decent at mowing down aliens and securing pixelated babes - but thanks to the Cirrus graphics card, which apparently missed the memo on fast action games. Oddly enough, it breezes through Heroes of Might and Magic 2, probably because the 32MB of RAM lets it hoard monster sprites like a





medieval dragon sitting on its treasure.

The LTE 5250 is a time capsule of computing greatness—or frustration, depending on how nostalgic you feel about Windows 95's tendency to randomly throw blue screens at you. Mine came with Windows 95 pre-installed, meaning it has that classic startup chime that instantly teleports you to an era of beige computers and dial-up tones.

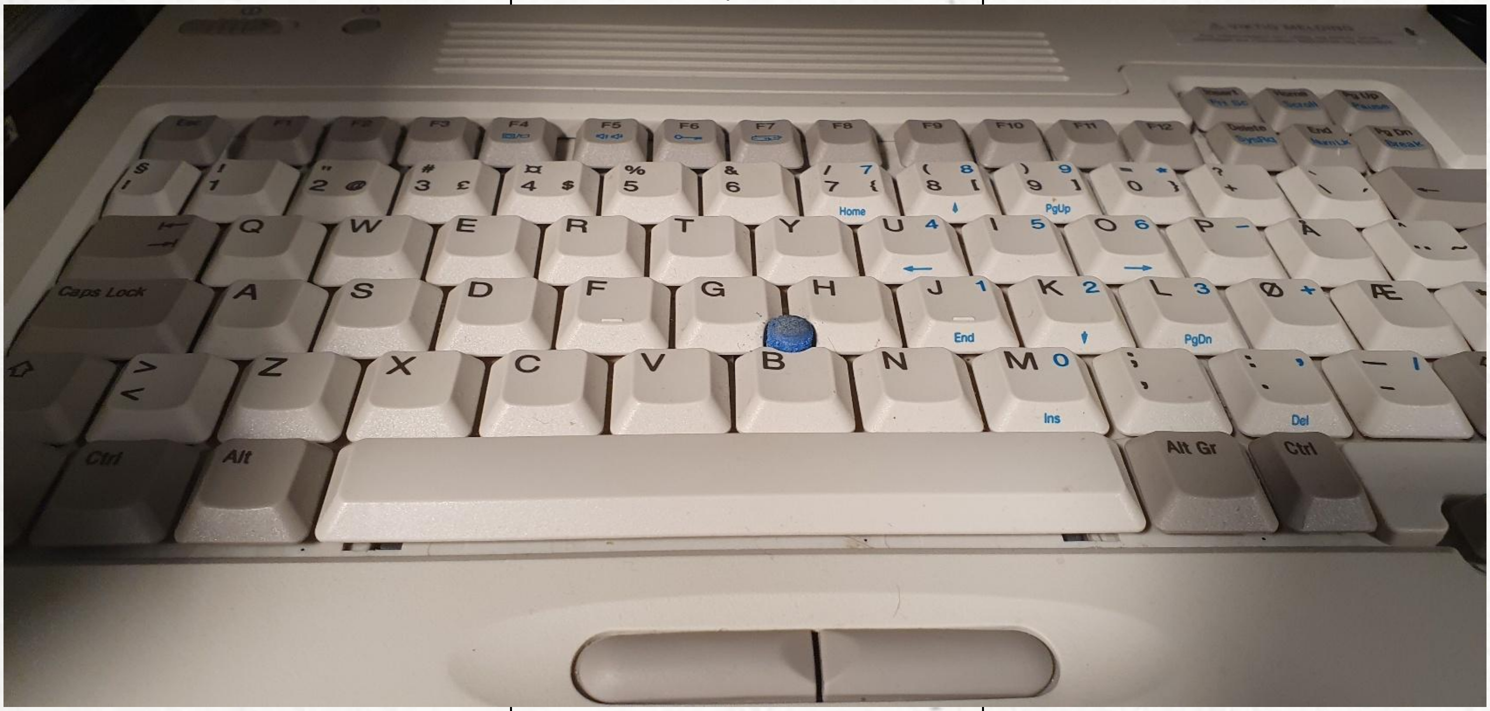
And while it may struggle with fast-paced FPS games, it's practically a command centre for classic productivity. Need to write a business proposal in Word 97? No problem! Want to browse the web? Nope… even if you enjoy waiting five minutes for a single page to load, and if you can find a browser that even works today paired up with a dial-up connection, we simply do not recommend getting online on a machine like this. Of course, running Oregon Trail or SimCity 2000 feels right at home - because let's be honest, the real reason anyone keeps these old machines around is to relive their gaming glory days, maybe by connecting two identical machines with a null-modem cable and play Doom, like it's 1995.

From the hardware point of view, there is a floppy drive exchangeable with a CD drive in the same slot, sadly due to this, only one of them can be used at the same time. The sound card is Sound Blaster compatible, so making beeps and bloops should not be a problem.

The keyboard? Surprisingly good. The keys have nice travel, making it feel like you're actually doing something important when typing. I might say, that this was the best keyboard I have ever experienced on a laptop, for me it is even beating those fancy IBM Thinkpads from the same age and time.

But the pointing device? Oh boy. It's one of those early trackpoint nubs, meaning you'll spend more time fighting your own cursor than getting work done. It's a great way to relive the frustration of early computing, like playing a video game where the boss fight is just your own patience. The hard drive is an absolute time machine in itself - mine came with a whopping 2GB drive, which in 1996 was considered excessive. "You'll never need more than that," they said. "What would you even fill it with?" they asked. Of course, in true 90s fashion, it sounds like a miniature jet engine when it spins up, reminding you that back then, performance came with a soundtrack. If the original drive ever fails (which it probably will), swapping it for a CompactFlash-to-IDE adapter is a great way to future-proof this old beast - because nothing screams "retro-futuristic" like booting Windows 95 from a 32GB CF card.

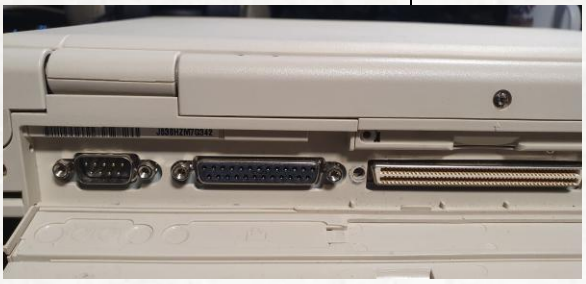
The LTE 5250's docking station is absurd. If the laptop itself is a cinder block, the dock is a full-on concrete slab. Weighing nearly as much as the laptop itself, it turns this "portable" machine into a full desktop replacement - because back then, mobility meant "carrying this thing between two desks". But on the positive side, it gives a plethora of expansion possibilities, such as an extra ISA slot, extra hard-drive, a second battery pack.

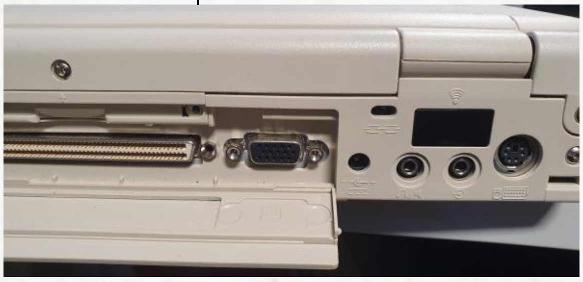


Like all old machines from those times, this specific machine is also not resistant to battery exhaustion, Thankfully, some brave souls on the internet still rebuild batteries (https://www.ebay.com/usr/80486sx) for these ancient machines, meaning you can actually run it without being tethered to a power outlet - though at this point, the charger is basically part of the aesthetic. If you're really feeling adventurous, you can even 3D-print replacement parts for the cracking plastics, ensuring that this 90s legend lives to fight another day.

Surprisingly this ancient machine has a growing fan club, even a private Facebook group exists for fans and owners of Compaq LTE and Armada owners, proving that everything has a place on this planet. The more than 300 page long manual, found online at https://archive.org/details/manualsbase-id-247638/mode/2up?view=theater has surprising amount of details, even down to the hardware level, on how to physically hack the machine together with details that we simply miss in the manuals of modern machines. Oh, you're right... what kind of manuals do even come with modern machines?

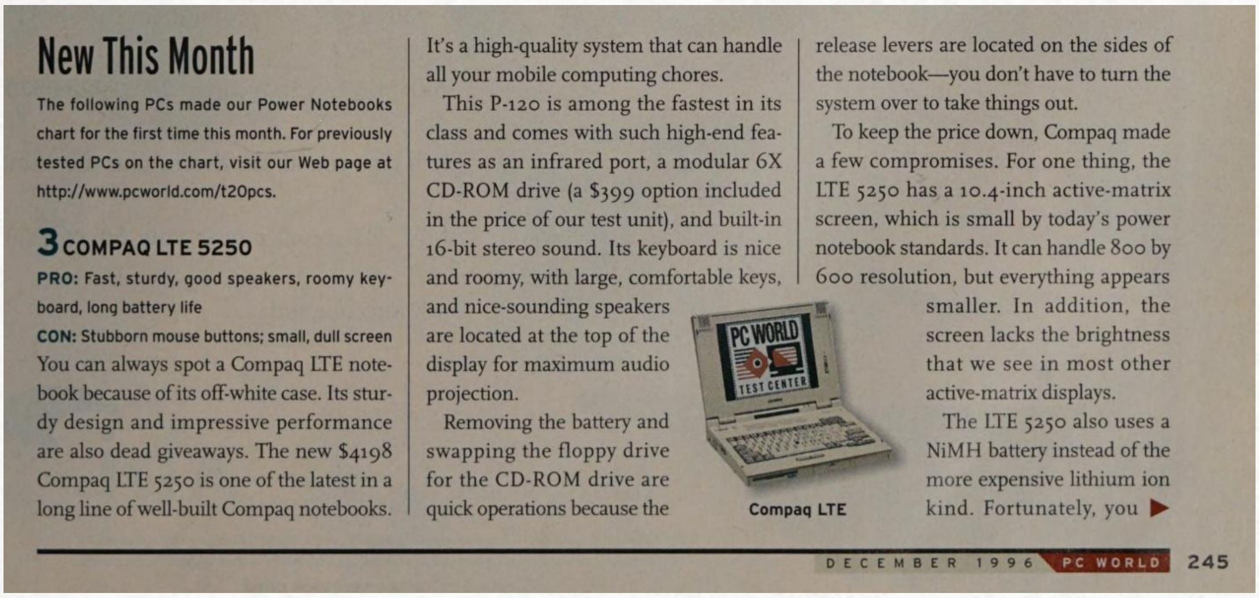
So, there you have it - the Compaq LTE 5250: part laptop, part museum exhibit, all nostalgia. Would I use it as a daily driver? Absolutely not. But would I fire it up to play Heroes of Might and Magic II while pretending it's still 1997? You bet.


The Right side of the backside


The Other Right side of the backside

One side of the sides of the computer

The other side



New This Month

The following PCs made our Power Notebooks chart for the first time this month. For previously tested PCs on the chart, visit our Web page at http://www.pcworld.com/t20pcs.

3 COMPAQ LTE 5250

PRO: Fast, sturdy, good speakers, roomy keyboard, long battery life

CON: Stubborn mouse buttons; small, dull screen

You can always spot a Compaq LTE notebook because of its off-white case. Its sturdy design and impressive performance are also dead giveaways. The new $4198 Compaq LTE 5250 is one of the latest in a long line of well-built Compaq notebooks. It's a high-quality system that can handle all your mobile computing chores.

This P-120 is among the fastest in its class and comes with such high-end features as an infrared port, a modular 6X CD-ROM drive (a $399 option included in the price of our test unit), and built-in 16-bit stereo sound. Its keyboard is nice and roomy, with large, comfortable keys, and nice-sounding speakers are located at the top of the display for maximum audio projection.

Removing the battery and swapping the floppy drive for the CD-ROM drive are quick operations because the release levers are located on the sides of the notebook—you don't have to turn the system over to take things out.

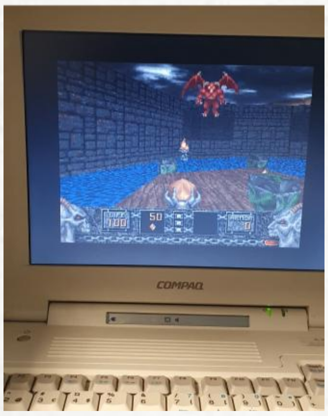
To keep the price down, Compaq made a few compromises. For one thing, the LTE 5250 has a 10.4-inch active-matrix screen, which is small by today's power notebook standards. It can handle 800 by 600 resolution, but everything appears smaller. In addition, the screen lacks the brightness that we see in most other active-matrix displays.

The LTE 5250 also uses a NiMH battery instead of the more expensive lithium ion kind. Fortunately, you ►

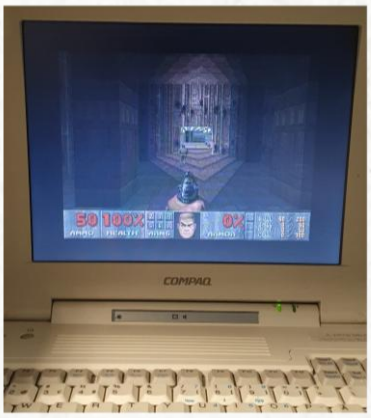
Compaq LTE

DECEMBER 1996 PC WORLD 245

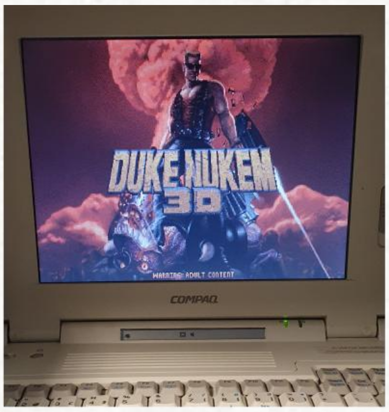
Original Review from December '96



Can

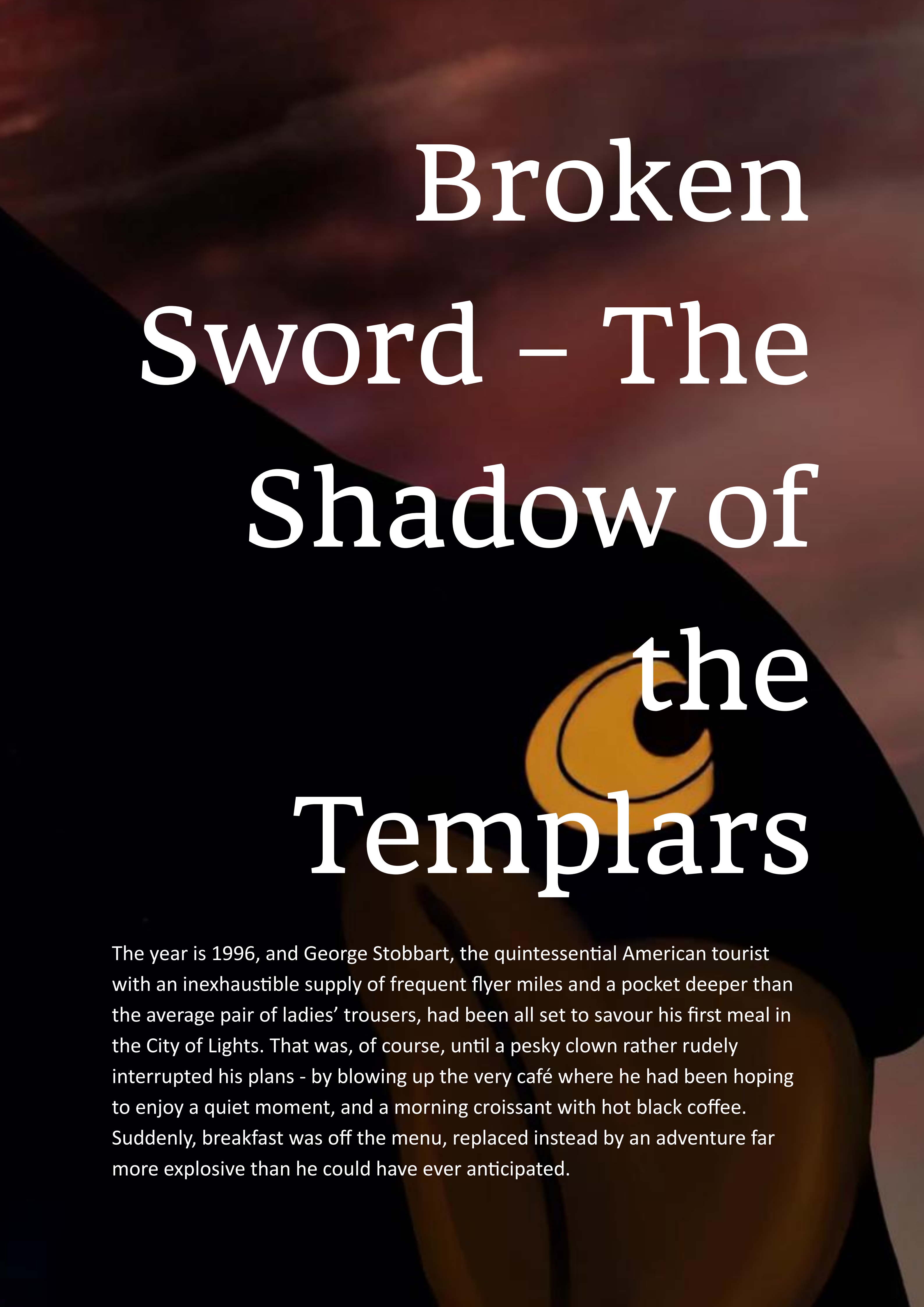


Can



Cannot

Old New



Broken Sword – The Shadow of the Templars

The year is 1996, and George Stobbart, the quintessential American tourist with an inexhaustible supply of frequent flyer miles and a pocket deeper than the average pair of ladies' trousers, had been all set to savour his first meal in the City of Lights. That was, of course, until a pesky clown rather rudely interrupted his plans - by blowing up the very café where he had been hoping to enjoy a quiet moment, and a morning croissant with hot black coffee. Suddenly, breakfast was off the menu, replaced instead by an adventure far more explosive than he could have ever anticipated.

This is how **Broken Sword: The Shadow of the Templars**, a rather splendid point-and-click adventure game starts. First unleashed upon the world in 1996 by the ever-talented Revolution Software the game is a masterclass in mystery, historical intrigue, and the fine art of getting caught up in other people's problems, while following George Stobbart, who has an unfortunate knack for being in the wrong place at the wrong time.

What follows the quite unpleasant incident at the cafe, is an utterly delightful romp through an elaborate conspiracy involving the Knights Templar, shadowy figures with questionable motives, and an unhealthy number of locked doors requiring obscure keys.

Fortunately, George is not alone in is endeavours. Enter Nicole Collard - journalist, sharp-witted Parisian, and the one person in this tale who seems to know what she's doing, unlike the two exquisite members of the Parisian police force: Inspector Rosso, a self-styled psychological detective with a penchant for abstract theories over actual evidence, and Sergeant Moue, his bumbling, sweat-drenched subordinate whose chief skills include blocking doorways and looking bewildered, form a delightfully incompetent duo who, through a mix of misplaced confidence and sheer inefficiency, manage to turn law enforcement into something of an art form - much to the benefit of any would-be conspirators and, indeed, our hapless hero.

Visually, Broken Sword is a hand-drawn delight, capturing the romance of Paris, the mystery of forgotten ruins, and the unrelenting danger of standing too close to suspicious individuals. Its script is razor-sharp, brimming with charm, humour, and the sort of dialogue that makes you wish real conversations were half as entertaining. The puzzles? Ingeniously crafted exercises over several locations, in lateral thinking or, if you prefer, wildly - clicking on everything until something happens across the entire Globe.

Solving these, our friend, George, embarks on a globe-trotting adventure, visiting a variety of locations steeped in history, mystery, and the occasional life-threatening peril.

In short, Broken Sword is a true gem of the adventure game genre—an effortlessly charming, wonderfully atmospheric tale of intrigue, history, and highly suspect clowns. If you've yet to experience its delights, do yourself a favour and rectify that immediately.

While Broken Sword: The Shadow of the Templars is rightly celebrated as a classic of the point-and-click adventure genre, it is not entirely without its quirks—some of which manifest as rather amusing (or infuriating) bugs.

One of the most infamous occurs in the original PC version, where some cutscenes occasionally fail to trigger if you visited certain locations before the ones the developers thought that are more logical, thus leaving poor George standing around awkwardly, utterly oblivious to the grand conspiracy unfolding around him.

George's Travel-Itinerary

Paris, France - The adventure begins in the City of Light, where George witnesses the infamous café bombing. From the bustling canals to the shadowy halls of a museum and a seedy hotel, home to a British Primadonna, Paris serves as the central hub for much of his investigation.

Ireland (Lochmarne) - George's travels lead him to a quiet Irish village, complete with a cosy pub, a suspiciously chatty farmer, and the ruins of an ancient castle hiding a very moody goat.

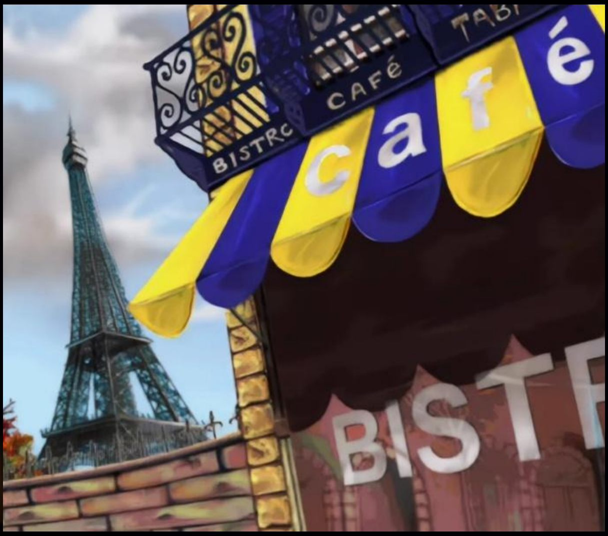
Syria (Marib) - In search of further clues, George journeys to a sun-soaked Middle Eastern town, in a country which has not felt the devastation of civil war yet, navigating a vibrant marketplace, evading his nemesis, and engaging in some questionable treasure hunting.

Spain (Villa de Vasconcellos) - The trail takes him to a remote Spanish villa, where a reclusive nobleman with a penchant for Templar history offers yet another piece of the puzzle.

Scotland (Bannockburn) - The final leg of George's journey brings him to an ancient Scottish site linked to the ... (no more spoilers, please).

The making of the remake

The game was highly praised in its days, thus a remake was something that everyone who ever fell in love with the original has eagerly waited, so enter the scene: Broken Sword: **The Shadow of the Templars, Reforged**.

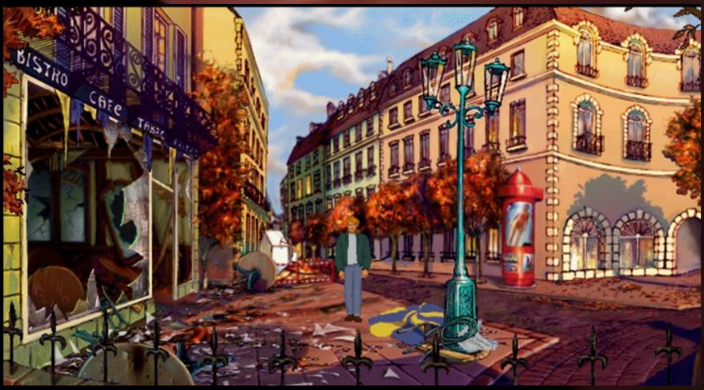


So, right, let's have a proper look at this "Reforged" version of "Broken Sword," shall we? It's quite the thing, really. They've certainly given it a good polish.

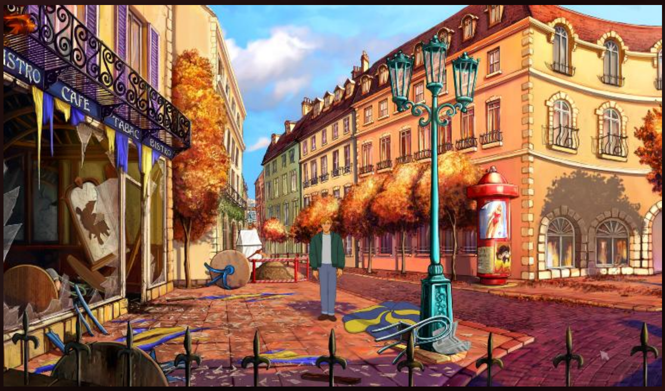
Visually, it's a marked improvement. The higher resolution that comes automatically with modern game releases brings out the detail in the backgrounds and characters, making it far more pleasing to the eye.

One can really appreciate the artistry and the incredible amount of work that went into the original designs, now that they're displayed with such clarity, thanks to the re-forged graphics, background and characters.

The old

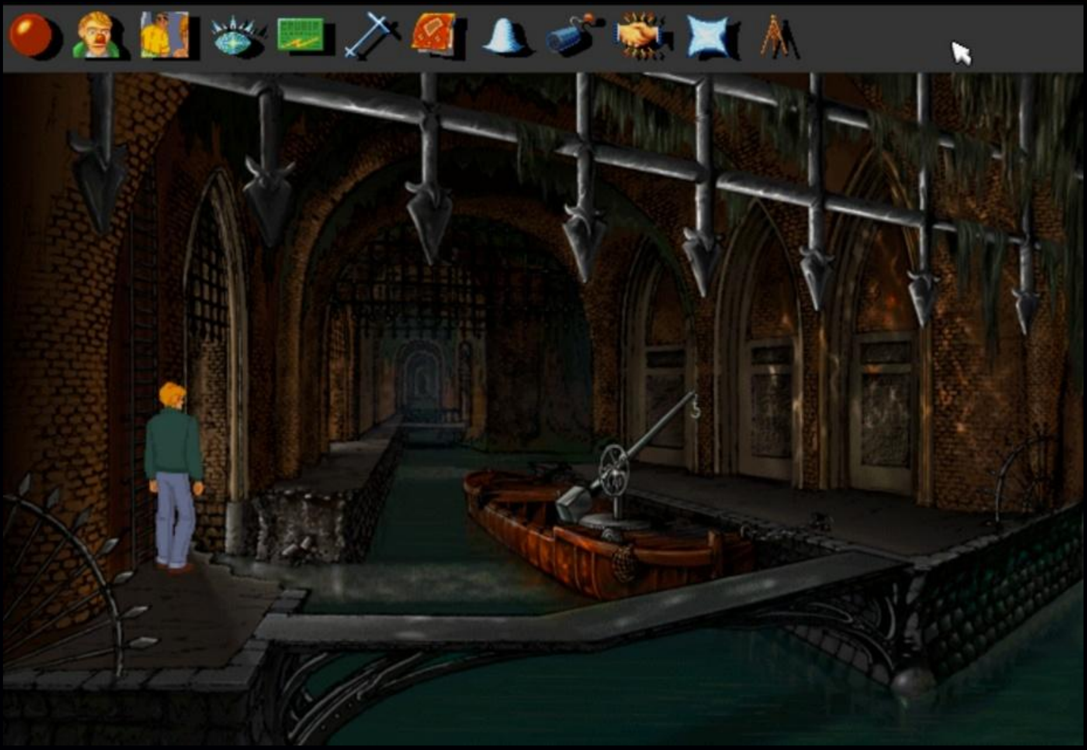


The new



While these images don't provide a visual advantage to either one, simply zoom in a bit - since this is a PDF, the differences will become instantly apparent.

The animations are smoother, too, which makes the whole experience feel more modern, but here we are not that sure whether this is due to the fact that it is run on computers hundred times faster than those in 1996, or well... just.



The author finds the old menu more player friendly

Reforged begins with a clear breakdown of how point-and-click games function when starting a new adventure. It offers two gameplay modes: Classic Mode and Story Mode, the latter designed to make the experience more accessible for newcomers. While I played primarily in Classic Mode, briefly exploring Story Mode to assess its effectiveness, it serves as a groundbreaking way to introduce younger and less experienced players to the point-and-click genre, because let's be honest: we were sort of deprived of a high-quality item from those in recent years.

The Reforged version also made quite an effort to streamline the user interface, which is a welcome change for some. It's more intuitive, and one doesn't find oneself fumbling about quite as much.

And the inclusion of a hint system, whilst perhaps a little hand-holding for the purists, is a sensible addition, but to be very honest, I preferred the older style UI where the menu was located on the top of the screen. Not that there is anything wrong with the new one, but it's not that much my cup of tea... or pint'o beer.

It's important to note that the game designers have opted to stick with the original 1996 version, meaning that the "Director's Cut" additions, including the Nico sections, are not included. This choice was made to preserve the authentic experience, a decision that was well-received by most fans of the game. Additionally, the developers have addressed the most frustrating bugs that, at times, could escalate the level of frustration to such an extent that players were forced to reload an earlier save and take a different route just to successfully complete the quest.

In essence, *Reforged* is a respectful and thoughtfully crafted update that pays homage to the original while refining it for a modern audience. It preserves the core gameplay mechanics, rich narrative, and nostalgic charm that made the original so cherished, ensuring that long-time fans still feel right at home. At the same time, it introduces visual enhancements, polished mechanics, and quality-of-life improvements that bring it in line with contemporary gaming expectations. By striking this delicate balance, *Reforged* serves as a fine example of how to revitalise a classic without compromising its essence or alienating its devoted fanbase.

We Approve 👍

The Retro Coder: Gentle Introduction to C

In an age of cutting-edge technology and ever-evolving programming languages, we, a unique group of enthusiasts choose to step back in time. We are known as “Retro Coders”. We dedicate our spare time after walking the dog, feeding the kids and tending to dirty dishes, to writing code for vintage computers, not out of necessity, but for the sheer joy of reviving the past. Whether it's crafting new games for the Commodore 64, optimizing assembly routines for an IBM PC, or breathing new life into an Amiga, we embrace the constraints and quirks of outdated hardware as a creative challenge.

Retro coding is more than just nostalgia; it's an art form that blends technical ingenuity with historical appreciation. By working within the limitations of classic machines, Retro Coders push the boundaries of what was once thought possible. Some of us create entirely new software, while others work to preserve, document, and enhance old programs, ensuring that the legacy of early computing continues to thrive.

The current embodiment of this series of the magazine will be an ever-evolving tutorial focusing on programming the IBM-PC (but not only) by providing insight on how to get what you want out from your ages old real-time environment by presenting techniques that were considered advanced 30 years ago, and downright extinct today.

To have a twist because certainly, without one nothing would work, we will be using modern technology and tools because to be honest, code editors and programming IDE's have evolved a lot during the last decades, and it would be border-line masochism to not to use them (on the other end, if you are vi master, feel free to use that, we don't mind). This article, sort of expects our readers to have an understanding of the generic programming notions, but not necessarily C nor C++, so we provide a quick introduction to the C programming language.

Since at this stage there is no homepage on the almighty internet dedicated to the home of RetroWTF, all the code will be placed in the pages of this edition, so please feel at home in the nineties, where you had to manually copy the code out from your magazine of choice. Strange, how times have a tendency to repeat themselves.

The Foundations of C Programming

C is a powerful and widely used programming language that forms the basis for many other languages, including C++, Java, and Python. It is extensively employed in system programming, embedded systems, and applications where performance is critical. Understanding C provides a strong foundation for learning other programming languages and understanding how computers work at a fundamental level.

For our readers who are not versed in the dark art of programming, we are providing a quick introduction to the C programming language, since we consider it to be the ideal candidate to write programs for our favorite retro platform, the IBM PC.

Our readers, who know how to conjure up lines of code without having to read through the introductory part of this article feel free to skip to the end of it, where we will jump deep, and conjure up some real dark lines of DOS covering direct screen access, and other mystical topics. Also, don't be afraid to read up on how to compile classical DOS program on modern systems.

What is Programming?

At its core, programming is about giving instructions to a computer. Think of it like writing a recipe, but for a machine. A "program" is a sequence of these instructions. Since C is a "compiled" language, meaning your code (source code) needs to be translated into machine-understandable code (executable) before it can run. We call this translation: compiling.

Writing a program

Before we can make a program, we actually need to write it, so we use either Text Editors or a full-fledged IDE.

Text Editors

These are the simple but effective tools for writing code, such as Notepad++ (Windows), Sublime Text (cross-platform), VS Code (cross-platform), and Atom (cross-platform).

Their usage almost always requires using the command line to compile and run programs.

Integrated Development Environments (IDEs)

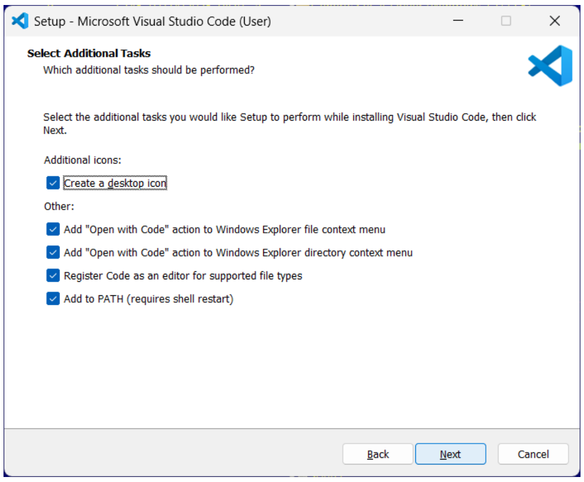
These tools provide a comprehensive environment for software development, including a text editor, compiler, debugger, and build tools.

Examples include Code::Blocks, Dev-C++ (Windows), Visual Studio (Windows), and Xcode (macOS).

IDEs streamline the development process, especially for larger projects, and allow you to escape the mayhem of command line if you are afraid of them.

VS Code

For our retrobright purpose we will use Visual Studio Code, which can be looked at as either a very advanced text editor or a very simple but flexible and extensible IDE. Getting it on your computer is easy, just follow the instructions here: there is nothing simpler than installing Visual Studio Code: just head over to https://code.visualstudio.com/download and press the download button for your platform. I chose the easier way, picked the Windows one, since I am writing this article on Windows.


Setup - Microsoft Visual Studio Code (User)

Select Additional Tasks

Which additional tasks should be performed?

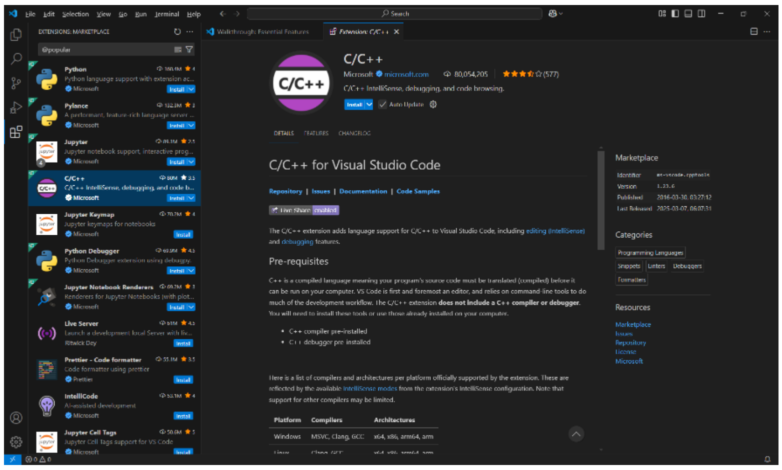
Select the additional tasks you would like Setup to perform while installing Visual Studio Code, then click Next.

Additional icons:

- [✓] Create a desktop icon

Other:

- [✓] Add "Open with Code" action to Windows Explorer file context menu
- [✓] Add "Open with Code" action to Windows Explorer directory context menu
- [✓] Register Code as an editor for supported file types
- [✓] Add to PATH (requires shell restart)



Python	160.4M ★ 4	Python language support with extension ac... – Microsoft
Pylance	132.3M ★ 3	A performant, feature-rich language server ... – Microsoft
Jupyter	89.1M ★ 2.5	Jupyter notebook support, interactive prog... – Microsoft
C/C++	80M ★ 3.5	C/C++ IntelliSense, debugging, and code b... – Microsoft
Jupyter Keymap	70.2M ★ 4	Jupyter keymaps for notebooks – Microsoft
Python Debugger	69.9M ★ 4.5	Python Debugger extension using debugpy. – Microsoft
Jupyter Notebook Renderers	69.2M ★ 3	Renderers for Jupyter Notebooks (with plot... – Microsoft
Live Server	61M ★ 4.5	Launch a development local Server with liv... – Ritwick Dey
Prettier - Code formatter	55.1M ★ 3.5	Code formatter using prettier – Prettier
IntelliCode	53.1M ★ 4	AI-assisted development – Microsoft
Jupyter Cell Tags	50.6M ★ 5	Jupyter Cell Tags support for VS Code – Microsoft

C/C++

Microsoft microsoft.com 80,054,205 ★★★★☆ (577)

C/C++ IntelliSense, debugging, and code browsing.

C/C++ for Visual Studio Code

Repository | Issues | Documentation | Code Samples

Live Share enabled

The C/C++ extension adds language support for C/C++ to Visual Studio Code, including editing (IntelliSense) and debugging features.

Pre-requisites

C++ is a compiled language meaning your program's source code must be translated (compiled) before it can be run on your computer. VS Code is first and foremost an editor, and relies on command-line tools to do much of the development workflow. The C/C++ extension **does not include a C++ compiler or debugger.** You will need to install these tools or use those already installed on your computer.

- C++ compiler pre-installed
- C++ debugger pre-installed

Here is a list of compilers and architectures per platform officially supported by the extension. These are reflected by the available IntelliSense modes from the extension's IntelliSense configuration. Note that support for other compilers may be limited.

Platform	Compilers	Architectures
Windows	MSVC, Clang, GCC	x64, x86, arm64, arm
Linux	Clang, GCC	x64, x86, arm64, arm

Marketplace

Identifier	ms-vscode.cpptools
Version	1.23.6
Published	2016-03-30, 03:27:12
Last Released	2025-03-07, 06:07:31

Categories

Programming Languages · Snippets · Linters · Debuggers · Formatters

Resources

Marketplace
Issues
Repository
License
Microsoft

Once it installed itself make sure to pick the C/C++ extension which will allow you to edit C/C++ code and you can move on to the next step: installing a compiler.

The Compilers

Now, on the other end, before we can run a C program, we need to compile it. A compiler is just another program that takes the source code you wrote and transforms it into the language the CPU of the computer can understand and execute.

There are several Compiler we can choose from:

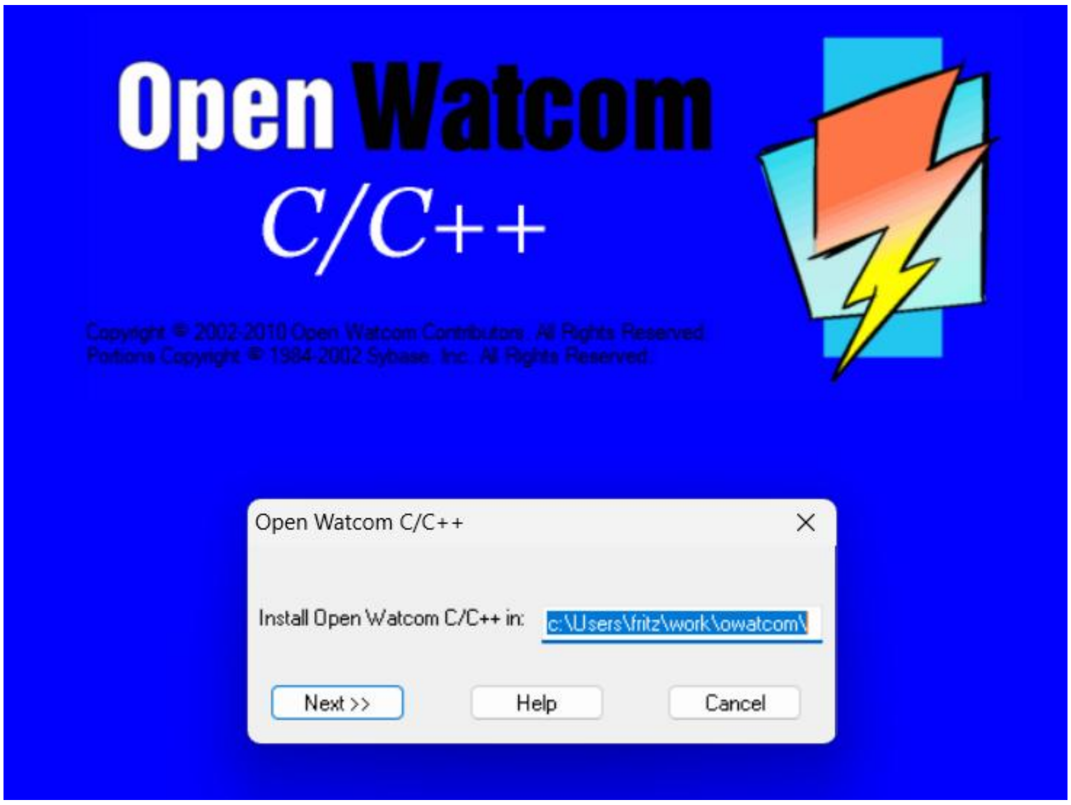
- **GCC** (GNU Compiler Collection): A highly popular and versatile compiler available for various operating systems.
- **Clang**: Another widely used compiler, particularly common on macOS and often used for its helpful error messages.
- **Microsoft Visual C++** (MSVC): The compiler provided by Microsoft as part of Visual Studio, primarily used on Windows.

For our specific purpose of writing retro programs we will use **OpenWatcom**, the once best compiler for DOS platforms (Remember, RETRO programming).

Our compiler of choice: OpenWatcom

OpenWatcom is an open-source compiler suite derived from the Watcom C, C++, and Fortran compilers, supporting DOS, Windows (16-bit and 32-bit), OS/2, and some embedded platforms. It features cross-compilation, DOS extender support, and a built-in debugger, making it useful for maintaining legacy software and retro development. While it was once popular for game and system programming, modern developers typically prefer GCC, Clang, or MSVC for contemporary projects.

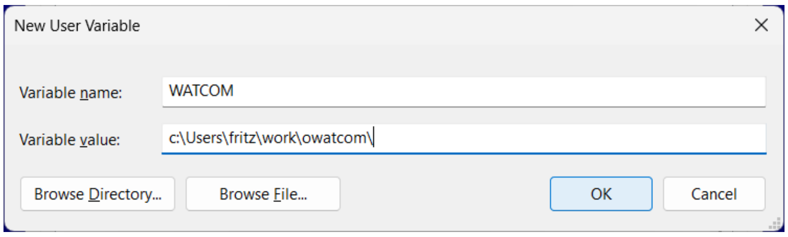
Installing it is as easy as downloading from https://openwatcom.org/ftp/install/ the latest official version (1.9) and running the installer. This should take care of all the default settings, so after this point you should be able to use the compiler as it is supposed to be used.


Open Watcom
C/C++
Copyright © 2002-2010 Open Watcom Contributors. All Rights Reserved.
Portions Copyright © 1984-2002 Sybase, Inc. All Rights Reserved.
Open Watcom C/C++
Next >>
Help
Cancel

Install Open Watcom C/C++ in: c:\Users\fritz\work\owatcom\

In case you feel adventurous and want to give a test ride to the bleeding edge, the latest snapshot can be obtained from https://github.com/open-watcom/open-watcom-v2/releases and in order to extract it, we'll need 7zip on our machine: https://www.7-zip.org/.

Once you have extracted to a specific folder (you might need to make beforehand) add the folder **binnt64** to your path: Can be found at: System Properties → Environment Variables and create a new environment variable, called WATCOM, just as the screenshot shows you.



Variable name: WATCOM
Variable value: c:\Users\fritz\work\owatcom\

Please note, these setting are system dependent, so if you are not on a 64-bit platform, please do not add the 64 bit directory, just the plain **binnt** one.

Pairing VS Code with OpenWatcom

It might be a bit cumbersome to get VSCode to fully cooperate with OpenWatcom, so to put it blandly and make this transition easy for everyone, let's just make it like this:

1. Create a `main.c` file in the directory you want to work on your retro code projects
2. Create a directory **.vscode** in that directory
3. Put the following `tasks.json` file in the **.vscode** directory after you have changed the two values of `c:/Users/fritz/work/owatcom` to the directory where you have installed your OpenWatcom (unless you are called fritz).

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "Build Using Watcom C++",
            "type": "shell",
            "command": "c:/Users/fritz/work/owatcom/binnt/wpp.exe ${file} -i=c:/Users/fritz/work/owatcom/h -oa -os -s
-xs -ml -fo=${workspaceFolder}/${fileBasenameNoExtension}.obj",
            "args": [],
            "problemMatcher": [],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        },
        {
            "label": "Link Using Watcom C++",
            "type": "shell",
            "command": "wlink.exe system dos option map option eliminate option stack=4096 name
${workspaceFolder}/${fileBasenameNoExtension}.exe file ${workspaceFolder}/${fileBasenameNoExtension}.obj",
            "args": [],
            "problemMatcher": [],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        }
    ]
}
```

From this point on when you have your main.c opened in the VSCode editor you simply press ***Ctrl+Shift+B***, which will bring up the Build Task To Run option, there select "Build Using Watcom C++" and if there are no errors, press again the "Link Using Watcom C++" option and this should generate the DOS executable for you. Running it is as easy as installing DosBox on your system, and passing in the compiled and linked executable to DosBox as an argument.

Don't worry if it makes not too much sense, https://code.visualstudio.com/docs/editor/tasks has a full explanation on this mystical tasks.json, feel free to browse it.

Your First C Program: the classical "Hello, World!"

Let's begin with the classic "Hello, World!" program, which is a fundamental starting point for learning any programming language.

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

And a few lines of Explanation

`#include <stdio.h>`: This is a preprocessor directive that includes the standard input/output library (`stdio.h`). This library provides functions for performing input and output operations, such as printing text to the console (`printf`) and reading input from the user (`scanf`).

`int main()`: This is the main function. Every C program must have a main function, as it is the entry point where the program's execution begins.

`int`: Specifies that the main function returns an integer value.

`return 0;`: Indicates that the program has executed successfully. A non-zero value typically indicates an error.

`printf("Hello, World!\n");`: This line uses the `printf` function to print the text "Hello, World!" to the console.

`\n`: This is an escape sequence that represents a newline character. It moves the cursor to the beginning of the next line, so any subsequent output will appear on a new line.

`"Hello, World!\n"`: This is a string literal, which is a sequence of characters enclosed in double quotes.

Variables and Data Types

Variables are used to store data in a program. Each variable has a name, a data type, and a value. The data type of a variable determines the type of data it can store.

Common Data Types in C

`int`: Used to store integer numbers (whole numbers without a fractional part), such as 10, -5, 0, and 1000.

`short int` or `short`: A shorter integer, using less memory.

`long int` or `long`: A longer integer, capable of storing a larger range of values.

`unsigned int`: An integer that can only store non-negative values.

`float`: Used to store single-precision floating-point numbers (numbers with a fractional part), such as 3.14, -2.5, and 0.001.

`double`: Used to store double-precision floating-point numbers. double variables can store a wider range of values and provide more precision than float.

`char`: Used to store single characters, such as 'A', 'b', and '5'. Characters are enclosed in single quotes.

`void`: A special data type that represents the absence of a value. It is often used with functions that do not return a value.

Declaring Variables

Before you can use a variable, you must declare it. Declaring a variable involves specifying its data type and giving it a name.

Syntax: `data_type variable_name;`

```
int age;
float height;
char grade;
```

You can also declare and initialize a variable in the same statement:

```
int age = 25;
float height = 5.9;
char grade = 'A';
```

The following Example Program is demonstrating Variables and Data Types

```
#include <stdio.h>
int main() {
    int age = 25;
    float height = 5.9;
    char grade = 'A';
    printf("Age: %d, Height: %.1f, Grade: %c\n", age, height, grade);
    return 0;
}
```

Where the magic is explained as:

`%d`: Format specifier for integers.

`%.1f`: Format specifier for floating-point numbers, with 1 digit after the decimal point.

`%c`: Format specifier for characters.

Functions

Functions are self-contained blocks of code that perform specific tasks and can be called more than one time without having to repeat the code performing the operation.

Declaring and Defining Functions

A function must be declared before it can be called. The declaration tells the compiler about the function's name, return type, and parameters. The definition provides the actual implementation of the function.

Function Declaration (Function Prototype):

```
return_type function_name(parameter_list);
```

Where:

`return_type`: The data type of the value that the function returns. If the function does not return a value, the return type is void.

`function_name`: The name of the function.

`parameter_list`: A comma-separated list of the function's parameters, along with their data types. If the function has no parameters, the parameter list is void or empty.

Function Definition:

```
return_type function_name(parameter_list) {
    // Function body (code to be executed)
    return value;  // Optional: return a value
}
```

The function definition includes the function header (the same as the declaration, but without the semicolon) and the function body, which is enclosed in curly braces `{}`.

Function Call:

To use a function, you call it by its name, followed by parentheses (). If the function takes any arguments, you pass them inside the parentheses.

```
function_name(arguments);
```

The following example program is demonstrating functions:

```c
#include <stdio.h>
// Function declaration
void greet();
int main() {
    greet(); // Function call
    return 0;
}
// Function definition
void greet() {
    printf("Hello from function!\n");
}
```

Function Parameters and Return Values

Functions can take parameters as input and return values as output.

Parameters:

Parameters are variables that are passed to a function when it is called. They allow you to provide data to the function so that it can perform its task.

Return Values:

A function can return a value to the caller using the return statement. The return value is the result of the function's computation, like we show in the following example.

```c
#include <stdio.h>

// Function declaration
int add(int a, int b);
int main() {
    int sum = add(3, 4); // Function call with arguments
    printf("Sum: %d\n", sum); // Print the returned value
    return 0;
}
// Function definition
int add(int a, int b) {
    return a + b; // Return the sum of a and b
}
```

Arrays

An array is a collection of elements of the same data type, stored in contiguous memory locations. Arrays provide a way to store and access multiple values of the same type using a single variable name.

Declaring Arrays

To declare an array, you specify the data type of the elements, the name of the array, and the number of elements in square brackets ``.

Syntax: `data_type array_name[size];`

```
int numbers[5]; // Declares an integer array of size 5
float temperatures[10]; // Declares a float array of size 10
char name[20]; // Declares a character array (string) of size 20
```

Initializing Arrays

You can initialize an array when you declare it by providing a comma-separated list of values enclosed in curly braces `{}`.

```
int numbers[5] = {1, 2, 3, 4, 5};
float temperatures[3] = {25.5, 28.0, 22.7};
char name[5] = {'J', 'o', 'h', 'n', '\0'}; // String initialization
```

If you provide fewer initializers than the size of the array, the remaining elements will be initialized to their default values (e.g., `0` for integers, `0.0` for floating-point numbers, and `\0` for characters).

For character arrays representing strings, the null terminator `\0` is used to mark the end of the string.

Accessing Array Elements

You can access individual elements of an array using the array name followed by the index of the element in square brackets `[]`. Array indices start at 0 and go up to `size-1`. Be warned: accessing array elements outside of this index interval might lead to program crashes and memory corruptions, and it is the main source of several similar problems.

```
int firstNumber = numbers[0]; // Accesses the first element (value: 1)
int thirdNumber = numbers[2]; // Accesses the third element (value: 3)
numbers[1] = 10; // Modifies the second element (value becomes 10)
```

The following example program is demonstrating arrays. Try to guess what is printed out on the screen before you run it.

```
#include <stdio.h>
int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        printf("%d ", numbers[i]); // Print each element
```

```c
    }
    printf("\n");
    return 0;
}
```

Pointers

A pointer is a variable that stores the memory address of another variable. Pointers are a powerful feature of C that allows for direct memory manipulation and efficient data handling.

Understanding Memory Addresses

Every variable in a program occupies a specific location in the computer's memory. This memory location is identified by a unique address. When you declare a variable, the compiler allocates a block of memory to store its value.

Declaring Pointers

To declare a pointer, you specify the data type of the variable that the pointer will point to, followed by an asterisk *, and then the name of the pointer.

Syntax: `data_type *pointer_name;`

```c
int *ptr; // Declares a pointer to an integer
float *fptr; // Declares a pointer to a float
char *chptr; // Declares a pointer to a character
```

The Address-of Operator (&)

The address-of operator & is used to obtain the memory address of a variable.

```c
int x = 10;
int *ptr = &x; // ptr now stores the memory address of x
```

The Dereference Operator (*)

The dereference operator * is used to access the value stored at the memory address pointed to by a pointer.

```c
int x = 10;
int *ptr = &x;
int value = *ptr; // value now contains the value of x (10)
```

Declaring and Using Pointers

```c
#include <stdio.h>
int main() {
    int x = 10;
    int *ptr = &x; // ptr stores the address of x
    printf("Value: %d, Address: %p\n", *ptr, ptr);
    return 0;
}
```

Where the magic is: `%p`: Format specifier for printing memory addresses.

Pointer Arithmetic

Pointer arithmetic is the dark art of performing arithmetic operations on pointers, since they hold memory addresses (which are just plain numbers in the end), performing arithmetic on them allows you to navigate and manipulate memory locations in an array or data structure. Pointer arithmetic works by adjusting the pointer's value (the memory address it points to) in a way that takes into account the size of the type it points to.

Pointer Increment and Decrement

When you increment a pointer, it moves to the next memory location of the same data type. When you decrement it, it moves to the previous memory location.

```
int arr= {10, 20, 30};
int *ptr = arr; // ptr points to the first element of arr (10)
ptr++; // ptr now points to the second element of arr (20)
ptr--; // ptr now points back to the first element of arr (10)
```

Adding an Integer to a Pointer

When you add an integer n to a pointer, it moves n memory locations forward, where each memory location has the size of the data type the pointer points to.

```
int arr= {10, 20, 30};
int *ptr = arr; // ptr points to the first element of arr (10)
ptr = ptr + 2; // ptr now points to the third element of arr (30)
```

Subtracting Pointers

You can subtract two pointers that point to elements of the same array. The result is the number of elements between the two pointers.

```
int arr= {10, 20, 30, 40, 50};
int *ptr1 = &arr[1]; // ptr1 points to the second element (20)
int *ptr2 = &arr[4]; // ptr2 points to the fifth element (50)
int diff = ptr2 - ptr1; // diff is 3 (the number of elements between ptr1 and ptr2)
```

Statements in C Language

A statement in C is an instruction that the compiler can transform into binary code. Statements define what actions a program should perform, such as assigning values, making decisions, or looping. In the C language each statement typically ends with a semicolon (`;`).

There are several type of statements in the language:

- Expression Statements – Assignments, function calls, or operations.
- Control Flow Statements – Change execution flow.
- Conditional Statements
- Looping Statements
- Compound Statements (Blocks) – A group of statements inside `{}`.
- Jump Statements – Alter program execution.

For the first part of this tutorial we will cover the two most basic complex statements, the `if` and the `for` statements and if there will be a next episode of the tutorial we will explore the other ones too.

The if statement

An if/else statement in C is a control structure that allows a program to make decisions based on conditions. It evaluates an expression and executes different blocks of code depending on whether the condition is true or false.

```
if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
```

For example:

```c
int num = 10;

if (num > 0) {
    printf("The number is positive.\n");
} else {
    printf("The number is not positive.\n");
}
```

Explanation:

If num is greater than 0, the program prints "The number is positive."

Otherwise, it prints "The number is not positive."

The for loop

The for loop in C is a control flow statement that allows you to repeat a block of code a specific number of times. It's commonly used when the number of iterations is known beforehand.

Here's the basic syntax of a for loop:

```
for (initialization; condition; increment) {
    // Code to execute during each iteration
}
```

The loop consists of three main components:

Initialization: Typically used to initialize a counter variable (e.g., `int i = 0`).

Condition: The loop continues to execute as long as this condition evaluates to true (e.g., `i < 10`).

Increment/Decrement: After each iteration, the counter variable is updated (e.g., `i++`).

With these lines we conclude the first part of this quick C tutorial, which provides a short explanation of the fundamental concepts of C programming necessary to understand the application we planned for this edition of the magazine: a full-fledged ASCII table. If there will be a second issue of the magazine, we will dig deeper both in the C language and will uncover some of the hidden internals of system programming under DOS.

The ASCII table from Hell

The **ASCII table** (American Standard Code for Information Interchange) is a character encoding standard that represents text in computers and other devices that work with text. It assigns numeric values to letters, digits, punctuation, and control characters.

The table is divided into two parts: the first 128 values (0-127) are known as **Standard ASCII**, which include control characters like line breaks and special symbols, and the uppercase and lowercase English alphabet. The **Extended ASCII** range (128-255) includes additional characters, such as accented letters, mathematical symbols, currency symbols, and other graphical characters, which vary depending on the character set used. These extended characters were introduced to address the need for characters in languages beyond English and for additional symbols required in specific regions or applications.

While ASCII itself is a **7-bit encoding**, Extended ASCII utilizes **8 bits** and was designed for compatibility with 8-bit systems, allowing up to 256 possible characters. The first 32 characters (0-31) of the table are control characters, and we'll talk about them shortly. Characters in the extended range above 127 include symbols such as the **Copyright (©)**, **Pound (£)**, and **Yen (¥)** signs, as well as graphical and accented characters used in various languages. Extended ASCII has been largely replaced by **UTF-8** and **Unicode**, which support a far wider range of characters, but the ASCII table remains foundational for understanding text encoding and is still widely used in computing.

With all this in mind, our first attempt to print an ASCII table to the screen can be summed up by the following lines:

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

int main() {
    for (int i = 0; i < 127; i += 6) { // Print six columns per row
        for (int j = 0; j < 6 && (i + j) < 127; j++) {
            printf("%3d %02X %c |  ", i + j, i + j, (char)(i + j));
        }
        printf("\n");
    }
    getch(); // Wait for key press before exiting
    return 0;
}
```

It is not a very complicated concoction, and it can be summed like a naïve attempt to generate an ASCII table. A quick breakdown of the code is like the following:

`#include <stdio.h>`: Standard library for input and output functions (`printf`).

`#include <conio.h>`: A console I/O library (specific to MS-DOS/Windows compilers like our beloved OpenWatcom but also Turbo C) providing functions like `getch()`. It is not part of the C standard and is not available in modern compilers like GCC. Hm… so are these modern or not in this case?

The interesting code happens in main, in the for loop: `for (int i = 0; i < 127; i += 6)`. This loops through ASCII values from 0 to 126 (since ASCII values range from 0 to 127), and increments `i` by 6 each time to process six characters per row.

The inner loop of this is: `for (int j = 0; j < 6 && (i + j) < 127; j++)` which runs up to 6 times per row (ensuring six characters are printed), and also by checking `(i + j) < 127` it prevents going beyond the basic ASCII range.

The line which looks like dark magic is: `printf("%3d %02X %c |  ", i + j, i + j, (char)(i + j));` where the values are:

`%3d`: Prints the decimal value right-aligned in a 3-character space.

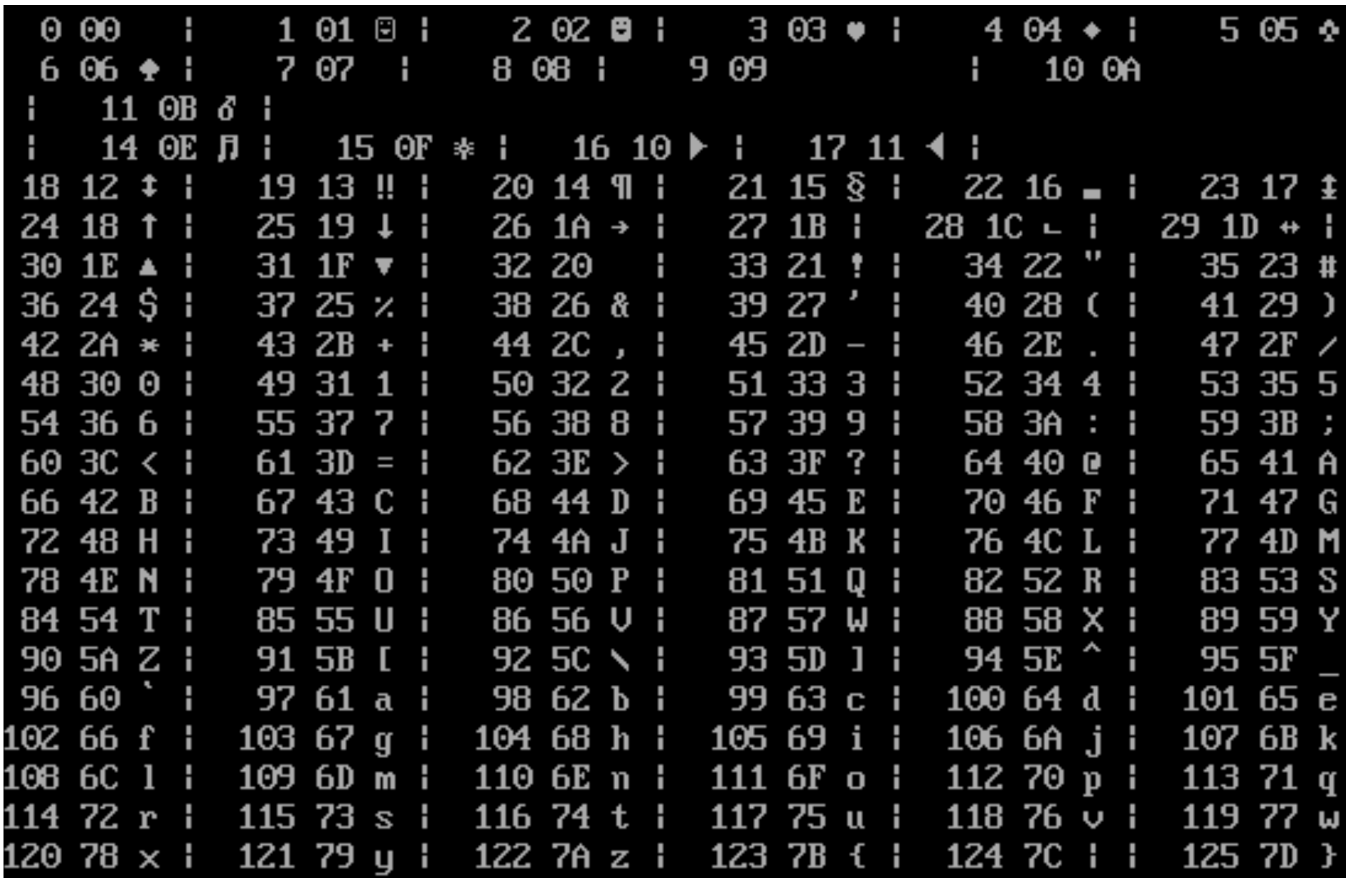
`%02X`: Prints the hexadecimal value using two uppercase digits (zero-padded if needed).

`%c`: Prints the corresponding ASCII character.

`|` : Separates each column for better readability.

So, in essence this prints on the screen one entry from the ASCII table, as we have envisioned: firstly, the hexadecimal code, then the decimal code, and then finally the character itself.

The `printf("\n");` statement is responsible for printing a newline character, ie. to make the cursor jump to the beginning of the new line. The `getch();` function call will just wait for a character to be pressed, and finally the `return 0;` will signal to the operating system that the program successfully ended. Which, upon executing, it produces the following wonderful result:



```
  0 00   ¦    1 01 ☺ ¦    2 02 ☻ ¦    3 03 ♥ ¦    4 04 ♦ ¦    5 05 ♣
  6 06 ♠ ¦    7 07  ¦   8 08 ¦    9 09        ¦   10 0A
¦   11 0B ♂ ¦
¦   14 0E ♫ ¦   15 0F ☼ ¦   16 10 ► ¦   17 11 ◄ ¦
 18 12 ↕ ¦   19 13 ‼ ¦   20 14 ¶ ¦   21 15 § ¦   22 16 ▬ ¦   23 17 ↨
 24 18 ↑ ¦   25 19 ↓ ¦   26 1A → ¦   27 1B ¦   28 1C ∟ ¦   29 1D ↔ ¦
 30 1E ▲ ¦   31 1F ▼ ¦   32 20   ¦   33 21 ! ¦   34 22 " ¦   35 23 #
 36 24 $ ¦   37 25 % ¦   38 26 & ¦   39 27 ' ¦   40 28 ( ¦   41 29 )
 42 2A * ¦   43 2B + ¦   44 2C , ¦   45 2D - ¦   46 2E . ¦   47 2F /
 48 30 0 ¦   49 31 1 ¦   50 32 2 ¦   51 33 3 ¦   52 34 4 ¦   53 35 5
 54 36 6 ¦   55 37 7 ¦   56 38 8 ¦   57 39 9 ¦   58 3A : ¦   59 3B ;
 60 3C < ¦   61 3D = ¦   62 3E > ¦   63 3F ? ¦   64 40 @ ¦   65 41 A
 66 42 B ¦   67 43 C ¦   68 44 D ¦   69 45 E ¦   70 46 F ¦   71 47 G
 72 48 H ¦   73 49 I ¦   74 4A J ¦   75 4B K ¦   76 4C L ¦   77 4D M
 78 4E N ¦   79 4F O ¦   80 50 P ¦   81 51 Q ¦   82 52 R ¦   83 53 S
 84 54 T ¦   85 55 U ¦   86 56 V ¦   87 57 W ¦   88 58 X ¦   89 59 Y
 90 5A Z ¦   91 5B [ ¦   92 5C \ ¦   93 5D ] ¦   94 5E ^ ¦   95 5F _
 96 60 ` ¦   97 61 a ¦   98 62 b ¦   99 63 c ¦  100 64 d ¦  101 65 e
102 66 f ¦  103 67 g ¦  104 68 h ¦  105 69 i ¦  106 6A j ¦  107 6B k
108 6C l ¦  109 6D m ¦  110 6E n ¦  111 6F o ¦  112 70 p ¦  113 71 q
114 72 r ¦  115 73 s ¦  116 74 t ¦  117 75 u ¦  118 76 v ¦  119 77 w
120 78 x ¦  121 79 y ¦  122 7A z ¦  123 7B { ¦  124 7C | ¦  125 7D }
```

Wonderful, ain't it? Except those few funny lines at the beginning, it almost works as expected. But every search we do on the almighty internet for a proper ASCII table has all those characters printed too. So what might the problem be? The reason for this oddity is historical, we have to look way behind in time, in prehistoric era, before retro was a word: control characters.

ASCII Control Characters (0-31)

ASCII control characters are special, non-printable characters in the ASCII table, ranging from **0 to 31**. They were originally designed for controlling various hardware. Most of these characters **do not produce visible symbols** when printed but instead perform control functions like moving the cursor, or ringing a bell.

The most Commonly Used ASCII Control Characters are summarized below, and are:

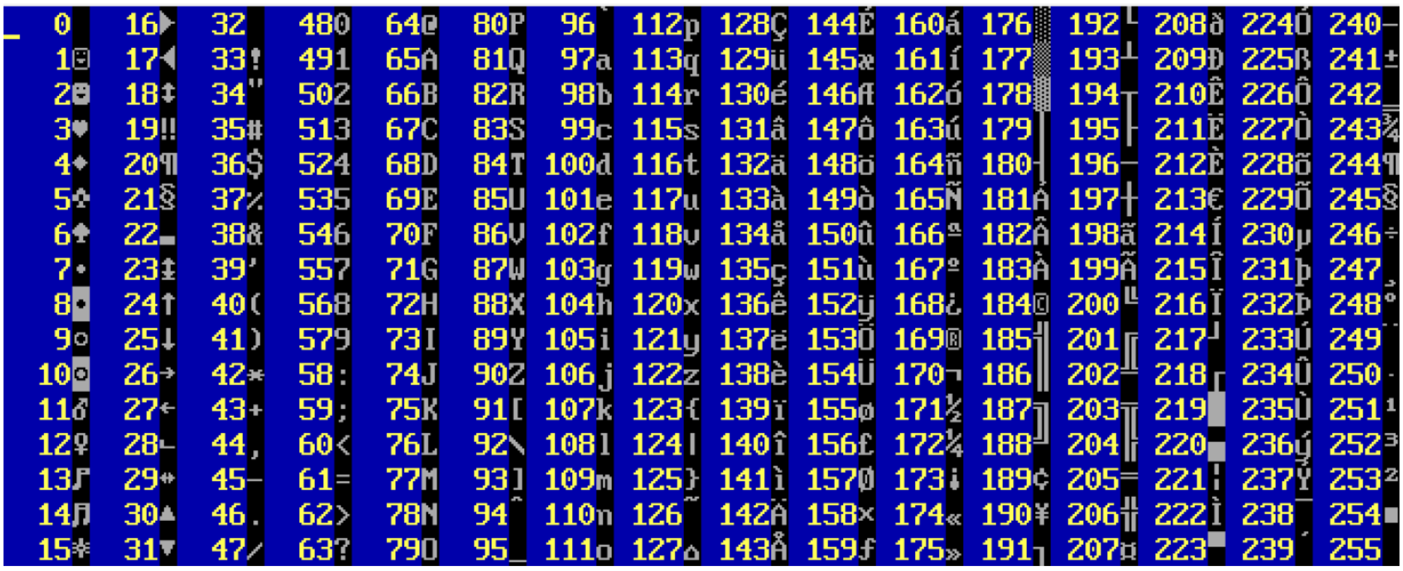
1. **NUL (0x00)**: Often used as a string terminator in C.
2. **BEL (0x07)**: Produces a sound (beep) in terminals.
3. **BS (0x08)**: Works as a backspace.
4. **TAB (0x09)**: Moves the cursor to the next tab stop.
5. **LF (0x0A) & CR (0x0D)**: Used for newlines (\n) and carriage returns (\r).
6. **ESC (0x1B)**: Used in ANSI escape sequences for formatting terminal output.

So, in essence ASCII control characters **do not display** like normal text but **perform control functions**. They were essential in early computing (teletype machines, terminals, printers). Some are still widely used in **serial communication, terminal formatting, and text processing**. Now, we see why we can't see a thing when we print some special characters onto the screen.

In order to circumvent this problem, we will need to assess the problem from a different point of view. But beware, dear reader, now we indeed will step into the domain of black magic programming, covering quite advanced topics. If you are a beginner, feel free to read on, but don't despair if there are some notions that are not clear from the beginning. We will try to explain these as detailed as possible though.

The ASCII table from Heaven

So, as a very first big drop, we will throw you, dear reader in the deep water, and show you the full ASCII table we intend to draw, with beautiful blue background for the numbers, black background for the characters and heavenly colors for the various other representable bits of the information (please ignore the cursor on the first line, it is there just to annoy us).



```
_   0    16►   32    480   64@   80P   96`  112p  128Ç  144É  160á  176░  192└  208ð  224Ó  240-
    1☺   17◄   33!   491   65A   81Q   97a  113q  129ü  145æ  161í  177▒  193┴  209Ð  225ß  241±
    2☻   18↕   34"   502   66B   82R   98b  114r  130é  146Æ  162ó  178▓  194┬  210Ê  226Ô  242‗
    3♥   19‼   35#   513   67C   83S   99c  115s  131â  147ô  163ú  179│  195├  211Ë  227Ò  243¾
    4♦   20¶   36$   524   68D   84T  100d  116t  132ä  148ö  164ñ  180┤  196─  212È  228õ  244¶
    5♣   21§   37%   535   69E   85U  101e  117u  133à  149ò  165Ñ  181Á  197┼  213€  229Õ  245§
    6♠   22▬   38&   546   70F   86V  102f  118v  134å  150û  166ª  182Â  198ã  214Í  230µ  246÷
    7•   23↨   39'   557   71G   87W  103g  119w  135ç  151ù  167º  183À  199Ã  215Î  231þ  247¸
    8◘   24↑   40(   568   72H   88X  104h  120x  136ê  152ÿ  168¿  184©  200╚  216Ï  232Þ  248°
    9○   25↓   41)   579   73I   89Y  105i  121y  137ë  153Ö  169®  185╣  201╔  217┘  233Ú  249¨
   10◙   26→   42*   58:   74J   90Z  106j  122z  138è  154Ü  170¬  186║  202╩  218┌  234Û  250·
   11♂   27←   43+   59;   75K   91[  107k  123{  139ï  155ø  171½  187╗  203╦  219█  235Ù  251¹
   12♀   28∟   44,   60<   76L   92\  108l  124|  140î  156£  172¼  188╝  204╠  220▄  236ý  252³
   13♪   29↔   45-   61=   77M   93]  109m  125}  141ì  157Ø  173¡  189¢  205═  221¦  237Ý  253²
   14♫   30▲   46.   62>   78N   94^  110n  126~  142Ä  158×  174«  190¥  206╬  222Ì  238¯  254■
   15☼   31▼   47/   63?   79O   95_  111o  127⌂  143Å  159ƒ  175»  191┐  207¤  223▀  239´  255
```

Wonderful, isn’t it? Our second big drop follows: here is the full source code of the program that draws it. Please churn through it initially, and it will be followed by a through explanation.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <graph.h>
#include <i86.h>
#include <stdint.h>
// The constants section
const int Black      = 0;
const int Blue       = 1;
const int White      = 7;
const int LightYellow= 14;
const int ROWS = 25;
const int COLS = 80;
// will clear the screen
void clearscr(void* scrSeg) {
    for(int i = 0;i<ROWS * COLS; i++) {
        *((uint16_t*)scrSeg  + i) = 0x0720;
    } // color, character
}
// writes a character at the given position, with the given colors
void writeChar(int x, int y, char bck, char col, char chh, void* scrSeg) {
    *((char*)scrSeg + (y * COLS + x) * 2) = chh;
    *((char*)scrSeg + (y * COLS + x) * 2 + 1) = bck << 4 | col ;
}
// writes a string at the given position, with the colors on the screen
void writeString(int x, int y, char bck, char col, const char* s, void* seg) {
    int l = strlen(s);
    for(int i=0; i<l; i++) {
        writeChar(x+i, y, bck, col, s[i], seg);
    }
}
// main entrypoint
int main(int argc, char* argv[]) {
    _setvideomode( _TEXTC80 );
    void __far * screen = MK_FP(0xb800, 0);
    int cc = 0;
    for (int i=0; i< 16; i++) {
        for (int j=0; j< 16; j++) {
            writeChar(i * 5 + 4, j, Black, White, cc, screen);
            char s[5] = {0};
            sprintf(s, "%4i", cc);
            writeString(i * 5, j, Blue, LightYellow, s, screen);
            cc ++;
        }
    }
    getch();
    clearscr(screen);
}
```

Now, dear reader, if you consider that you are an advanced level programmer, we still think this program could show you a few perky details, if not, we really admire your level of knowledge and would like to invite you to write an article for the next iteration of the magazine. However, for our beginner to intermediate level friends, here is a detailed, line by line explanation.

Inclusion of Header Files

The program starts by including a set of the most commonly used C header files, and also a few ones that are specific to OpenWatcom and also the DOS environment.

- `#include <stdio.h>`: This line includes the standard input/output library. In DOS, this provides functions like `sprintf` which is used later to format numbers into strings. These functions are typically implemented by the OpenWatcom C++ runtime library to interact with the DOS operating system for tasks like formatted output.

- `#include <stdlib.h>`: This includes the standard library, offering general-purpose functions. While not directly used in *this specific* program, it's often included as it contains functions like memory allocation (`malloc`, `free`) and conversions (`atoi`, `atol`) that might be needed in other programs, so we showcase it for future reference.

- `#include <string.h>`: This includes the string manipulation library. Here, the function `strlen` is used to determine the length of a string before printing it. In DOS, strings are typically null-terminated character arrays, and these functions operate on them.

- `#include <conio.h>`: This header is specific to console input/output in DOS and provides functions like `getch()`, which waits for a key press from the user. This library often interacts directly with the BIOS (Basic Input/Output System) of the PC to handle keyboard input.

- `#include <graph.h>`: This header is part of the OpenWatcom graphics library. While the name suggests graphics, in this particular program, only the function `_setvideomode` is used, and it's used to set the video mode to a *text* mode. This library provides a higher-level abstraction over direct hardware manipulation for video display.

- `#include <i86.h>`: This header provides functions that allow direct interaction with the Intel 8086 (and its successors like the 286, 386 in real mode, which DOS typically runs on) architecture. The function used here is `MK_FP`, which is used to create a far pointer. In DOS's segmented memory model, far pointers are essential for accessing memory outside the current data segment.

- `#include <stdint.h>`: This header defines standard integer types with specific widths, like uint16_t (unsigned 16-bit integer). This ensures portability and clarity about the size of integer variables.

Defining the Color Constants

The next important phase in the application is defining a few constants, because it is simply a bad practice to use magic numbers, it is more recommended to create constant having assigned the value of number which have specific meanings in your programs and name them to properly describe what they mean.

- `const int Black = 0;`: This declares a constant integer named Black and assigns it the value 0. This represents the color black in the standard VGA text mode palette.

- `const int Blue = 1;`: Similarly, this defines Blue as 1, representing the blue color.
- `const int White = 7;`: White is defined as 7, representing the white color.
- `const int LightYellow= 14;`: LightYellow is defined as 14, representing a bright yellow color. These integer values correspond to the color codes used in the attribute byte of the VGA text mode. Feel free to experiment with other colors, up to 16. Not that we cannot give you the entire color table, but it's more fun when you discover it.

Defining the Screen Size Constants

- `const int ROWS = 25;`: This defines the standard number of rows in a typical DOS text mode screen (25 lines).
- `const int COLS = 80;`: This defines the standard number of columns in a typical DOS text mode screen (80 characters per line).

The `clearscr` Function

The `void clearscr(void* scrSeg) { ... }` function is designed to clear the text mode screen. It takes a `void* scrSeg` as input, which is expected to be the segment address of the video memory. Don't worry if at this stage this makes no sense, we will explain in a tad these complex notions.

The loop `for(int i = 0; i < ROWS * COLS; i++) { ... }` iterates through each character cell on the screen. The total number of cells is `ROWS * COLS` (25 * 80 = 2000).

The magic happens in `*((uint16_t*)scrSeg  + i) = 0x0720;`.This is the core of the screen clearing logic. Let's break it down further:

- `scrSeg`: This is a pointer to the beginning of the video memory. In text mode, each character on the screen is represented by two consecutive bytes in memory.
- `(uint16_t*)scrSeg`: This casts the `void*` pointer `scrSeg` to a pointer to an unsigned 16-bit integer. This is done because we want to treat each character cell (character + attribute) as a single 16-bit unit.
- `+ i`: This adds the loop counter `i` to the pointer. Since the pointer is now a `uint16_t*`, adding `i` effectively moves the pointer i * 2 bytes forward in memory, addressing each character cell sequentially.
- `*((uint16_t*)scrSeg  + i)`: The asterisk `*` dereferences the pointer, meaning it accesses the 16-bit value at that memory location.
- `= 0x0720;`: This assigns the hexadecimal value `0x0720` to the current character cell.
  - `0x20`: This is the ASCII code for the space character. This will be the character displayed on the screen: the space character, the total voidness.
  - `0x07`: This is the attribute byte. In a standard color text mode, the attribute byte has the format `(background << 4) | foreground`. Here, `0x07` means a black background (0) and a white foreground (7).

    - Therefore, this line writes a white space character on a black background to each position on the screen, effectively clearing it.

The writeChar Function:

The `void writeChar(int x, int y, char bck, char col, char chh, void* scrSeg) { ... }` function writes a single character (`chh`) at a specified position (`x, y`) on the screen (`scrSeg`) with the given background (`bck`) and foreground (`col`) colors.

There are two magic lines in this piece of code:

- `*((char*)scrSeg + (y * COLS + x) * 2) = chh;`: This line writes the character itself to the video memory.
    - `(y * COLS + x)`: This calculates the linear index of the character cell on the screen. `y` is the row number (0-based), and `x` is the column number (0-based). Multiplying `y` by `COLS` gives the offset to the beginning of that row, and adding `x` gives the offset to the specific column in that row.
    - `* 2`: Since each character cell occupies two bytes (one for the character and one for the attribute), we multiply the linear index by 2 to get the byte offset from the beginning of the video memory.
    - `(char*)scrSeg`: This casts the void* pointer to a pointer to a single character (byte).
    - `*((char*)scrSeg + (y * COLS + x) * 2)`: This dereferences the pointer to access the first byte of the character cell.
    - `= chh;`: This assigns the character `chh` to this memory location.
- `*((char*)scrSeg + (y * COLS + x) * 2 + 1) = bck << 4 | col ;`: This line writes the attribute byte (colors) for the character.
    - The memory address calculation is the same as before, but we add + 1 to access the *second* byte of the character cell, which is the attribute byte.
    - `bck << 4`: This takes the background color code (`bck`) and shifts its bits four positions to the left. This places the background color in the higher four bits of the attribute byte.
    - `| col`: This performs a bitwise OR operation with the foreground color code (`col`). This places the foreground color in the lower four bits of the attribute byte.
    - The resulting byte, containing both background and foreground color information, is then written to the attribute byte location in video memory.

The writeString Function

The `void writeString(int x, int y, char bck, char col, const char* s, void* seg)` function writes an entire null-terminated string (`s`) at a specified position (`x, y`) with given colors at the given segment, representing the screen.

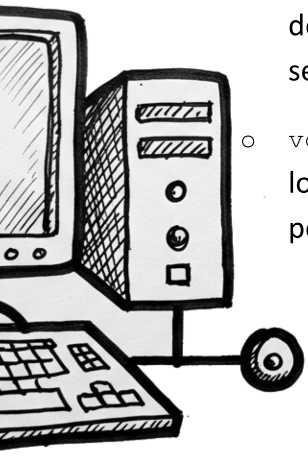
The statements in this function are not that magical, so we just show them here in their daily mundanity.

- `int l = strlen(s);`: This line calculates the length of the input string s using the `strlen` function. It is recommended good practice to not to call functions in a loop for values that can be pre-calculated before the body of the loop in order to optimize the code a bit, ie. to extract the information before. Modern compilers are good at optimizing these away, older compilers need help with these constructs.
- `for(int i=0; i<l; i++) { ... }`: This loop iterates through each character in the string.
- `writeChar(x+i, y, bck, col, s[i], seg);`: Inside the loop, we call the `writeChar` function for each character in the string.
  - `x+i`: The x-coordinate is incremented by `i` in each iteration, so the characters are written sequentially across the row.
  - `y`: The y-coordinate (row) remains the same for the entire string.
  - `bck, col`: The background and foreground colors are the same for the entire string.
  - `s[i]`: This accesses the character at the current index `i` in the string.
  - `seg`: The segment address of the video memory as we got from the caller function.

The main Function

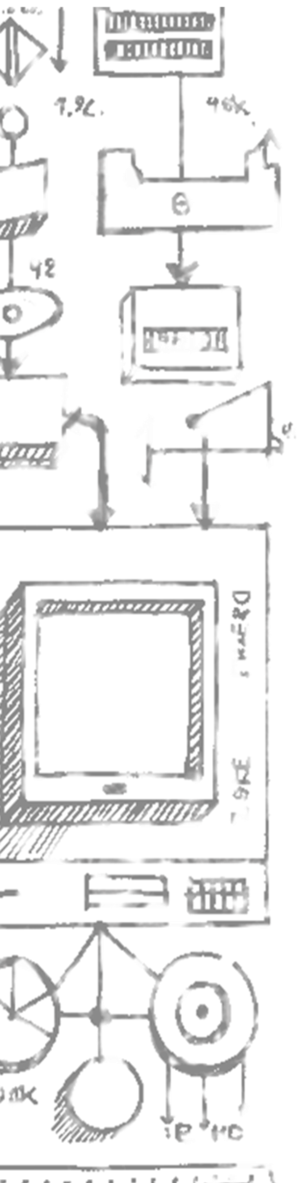
This is The Program's Entry Point, this where the execution of the program begins: `int main(int argc, char* argv) { ... }`where `argc` represents the number of command-line arguments, and `argv` is an array of strings containing those arguments. In this simple program, these are not used.

- `_setvideomode( _TEXTC80 );`: This is a function from the OpenWatcom graphics library, we included via the `graph.h` header. It sets the video mode of the display. `_TEXTC80` is a constant that specifies an 80-column color text mode. This ensures that the program operates in a standard text mode environment where it can directly manipulate the video memory.
- `void __far * screen = MK_FP(0xb800, 0);`: This is the most important line for accessing the video memory in DOS:
  - `0xb800`: This is the standard hexadecimal segment address in DOS for the video memory buffer when the system is in color text mode (like the one we just set with `_setvideomode`).
  - `0`: This is the offset within the segment. We want to start at the very beginning of the video memory for this segment.
  - `MK_FP(0xb800, 0)`: This macro, defined in `i86.h`, takes a segment and an offset as arguments and creates a *far pointer*. In DOS's segmented memory architecture, a far pointer consists of a 16-bit segment and a 16-bit offset, allowing access to memory outside the default data segment of the application, such as the video memory which resides at a fixed segment address: `0xB800:0x0000`.
  - `void __far * screen`: This declares a far pointer named screen that points to a memory location. The `__far` keyword is an OpenWatcom extension that explicitly indicates a far pointer.



    - So, the `screen` pointer now holds the address of the beginning of the video memory buffer.

- `int cc = 0;`: This initializes an integer variable `cc` (short for "character counter") to `0`. This certainly could have been an unsigned char too but then we would encounter some warnings later when using in `sprintf`, so let's make it `int`.
- `for (int i=0; i< 16; i++) { ... }`: This is the outer loop, iterating 16 times (from 0 to 15).
    - `for (int j=0; j< 16; j++) { ... }`: This is the inner loop, also iterating 16 times (from 0 t o 15). These nested loops will run a total of 16 * 16 = 256 times, just as many times as there are ASCII characters in the extended ASCII table. What a coincidence.
        - `writeChar(i * 5 + 4, j, Black, White, cc, screen);`: Inside the inner loop, this line writes a single character to the screen.
            - `i * 5 + 4`: This calculates the x-coordinate (column) for the character. It spaces out the characters horizontally by 5 columns and starts with an offset of 4.
            - `j`: This uses the inner loop counter j as the y-coordinate (row).
            - `Black, White`: The character will be displayed with a black background and a white foreground as defined in the constants section.
            - `cc`: The current value of the `cc` counter is cast to a `char` and displayed. As `cc` goes from 0 to 255, it will represent the ASCII codes of various characters.
            - screen: The pointer to the video memory.
        - `char s[5] = {0};`: This declares a character array named `s` of size 5 and initializes all its elements to the null terminator (`\0`). This array will be used to store the string representation of the cc value.
        - `sprintf(s, "%4i", cc);`: This function from `stdio.h` formats the integer value of `cc` into a string and stores it in the `s` array.
            - `%4i`: This format specifier tells `sprintf` to format the integer as a decimal number (i) and to pad it with spaces on the left if necessary so that it occupies a total of 4 characters.
            - `cc`: The value of `cc` is passed as the argument to be formatted.
            - The resulting string (e.g., " 0", " 1", ..., "255") is stored in the `s` array.
        - `writeString(i * 5, j, Blue, LightYellow, s, screen);`: This line writes the formatted string s to the screen.
            - `i * 5`: This calculates the x-coordinate for the string. It's placed 4 columns to the left of the single character written in the previous line.
            - j: The same row as the single character.



            - `Blue, LightYellow`: The string will be displayed with a blue background and a light yellow foreground.
            - s: The formatted string representation of the `cc` value.
            - `screen`: The pointer to the video memory.
        - `cc ++;`: This increments the cc counter for the next iteration of the inner loop.
- `getch();`: This function from `conio.h` waits for the user to press any key. The program will pause here until a key is pressed, allowing the user to see the output on the screen.
- `clearscr(screen);`: Once a key is pressed, this line calls the clearscr function to clear the screen. This is often done before the program exits, leaving the console in a clean state.
- `return 0;`: This indicates that the main function has executed successfully.

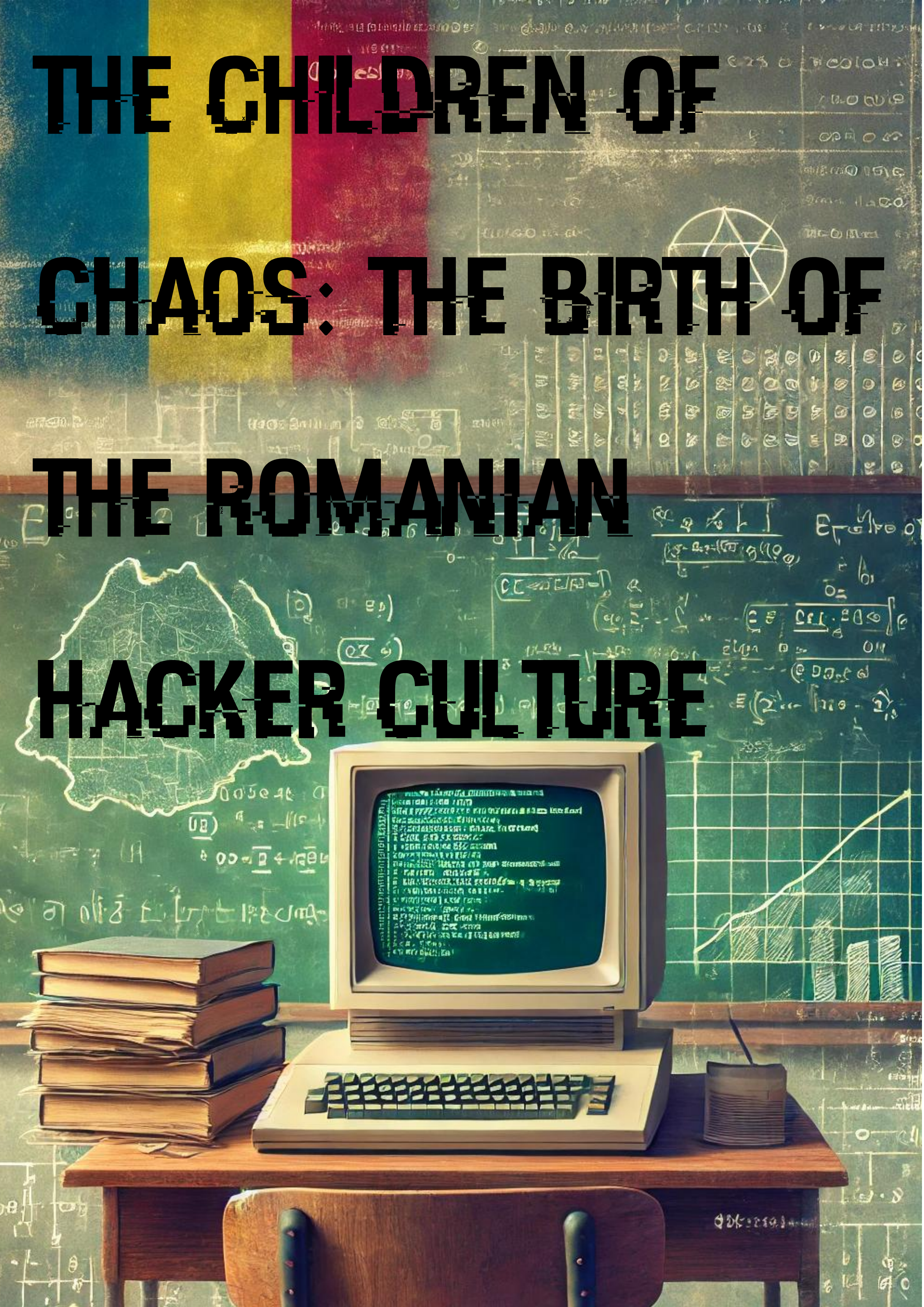
And that's all. Before we end our article, just a few lines about DOS and memory: DOS utilized a segmented memory model in its real-mode operation, where the 1MB of addressable memory was divided into 64KB segments. Memory locations were identified by a logical address consisting of a 16-bit segment address, stored in segment registers like CS, DS, SS, and ES, and a 16-bit offset within that segment.

The CPU calculated the physical address by shifting the segment address four bits to the left and adding the offset. This architecture, while allowing access to more memory than a single 16-bit address space, imposed a 64KB limit on the size of individual segments. This segmented approach introduced complexities for programmers, requiring them to manage segment boundaries and utilize near pointers for within-segment access and far pointers (segment:offset pairs) for accessing memory in different segments. Different memory models were developed to provide varying levels of abstraction and manage these segment limitations for different program needs. In contrast, modern operating systems typically employ a flat memory model, offering a simpler, contiguous address space.

With these lines, dear readers, we conclude our ASCII table from heaven, and thanks to the support provided by DOS's unhinged memory access to display a comprehensive ASCII table on the screen we achieve this by directly manipulating the video memory located at segment address 0xB800. Our heavenly program writes pairs of bytes to this memory region, where the first byte represents the ASCII character code and the second byte defines its display attributes, including foreground and background colors. Operating within the DOS environment, the program utilizes far pointers to access the video memory. This example illustrates a common technique in DOS programming for achieving efficient text-based output through direct hardware interaction.

So, keep on reading, and Happy coding!

THE CHILDREN OF CHAOS: THE BIRTH OF THE ROMANIAN HACKER CULTURE



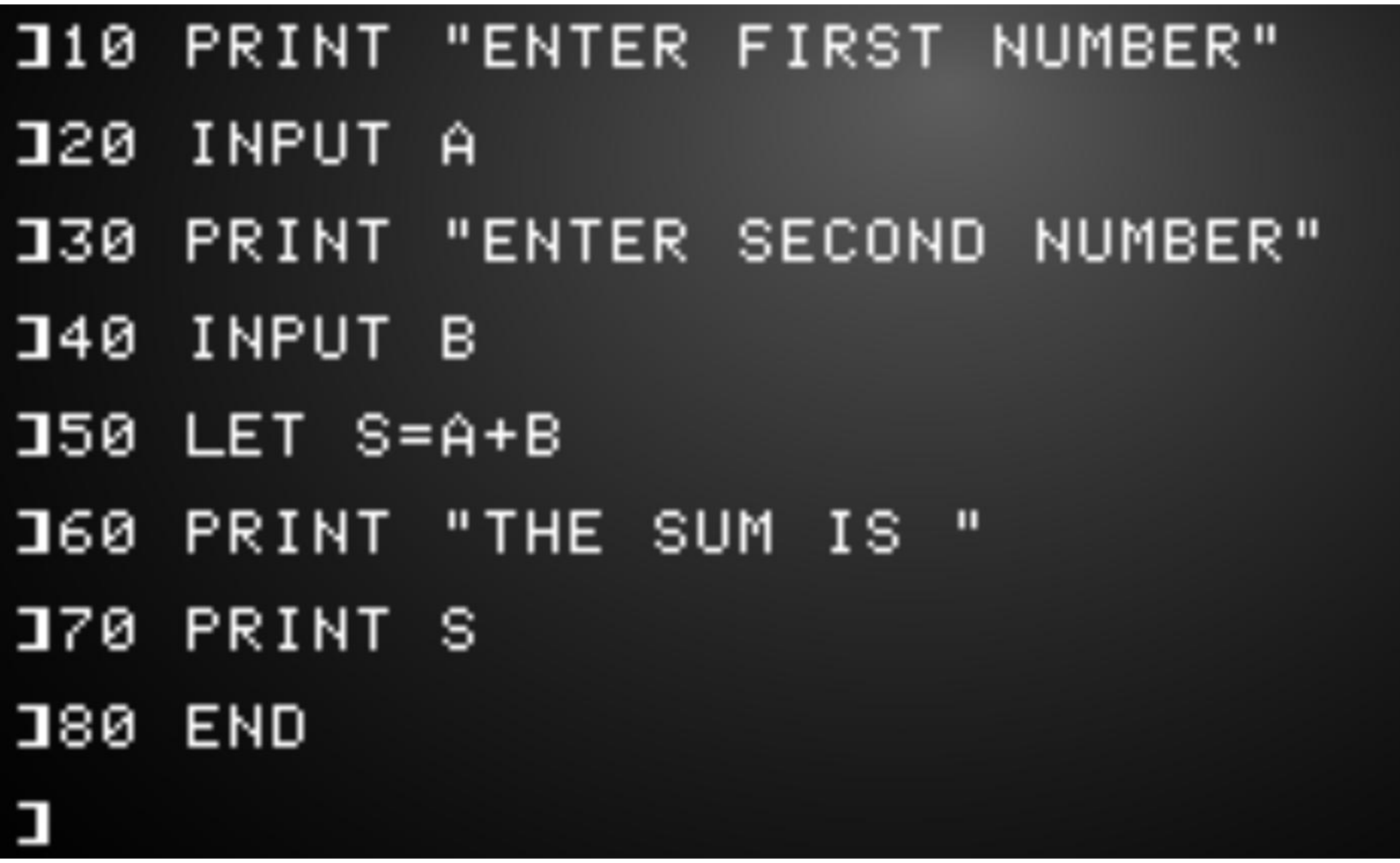
There were 23 of us. Crowded into a small room no larger than 3x4 meter, we were amongst the first generation of high school students officially granted the opportunity to earn the title of “Licentiate in Software Engineering” - provided we passed our final exam after four years. The year is 1993, and Romania is still struggling to find its path toward economic stability and social welfare while grappling with its past. Access to Western technology remains limited, but a growing curiosity about computers and rumors of this new thing, called the internet is taking hold among the youth.

While our small provincial town was still years away from the miraculous arrival of the internet, we were hardly deprived of technology. Our school was extravagantly stocked with a grand total of four fully functional Apple-IIc machines - and one massive, mysterious beast that hadn’t graced us with its presence the year before, back when our optional Computer Science class in junior high offered us a rare peek into the so-called future.

Following the timeless human instinct of “stick to what you know,” my 22 classmates eagerly rushed to claim the familiar Apple machines, ready to power them on like seasoned pros. Meanwhile, I, being perpetually late for everything, was left with the only remaining seat, right in front of the strange, intimidating beast.

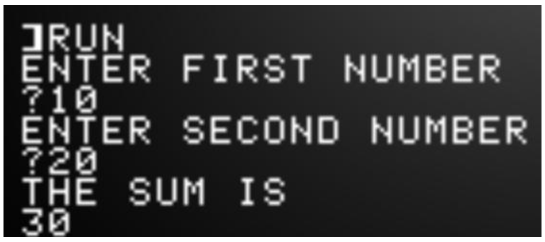
It was huge, almost twice the size of those sweet little Apples, it had a big red switch on the side, I anticipated to be the power button, the keyboard was larger than anything I have ever seen, with weird keys, like F1, F2, ... and I had no idea what it was... but since it looked like a computer, felt like a computer and after pressing the red switch it started acting like a computer, I guessed that it must be a computer.

Our first assignment for the hour was to write a small BASIC program, that would ask the user for two numbers, and add them together. Not an overly complicated task, considering that most of us had some introduction to Apple BASIC the previous year, so some of us quickly came up with the solution:



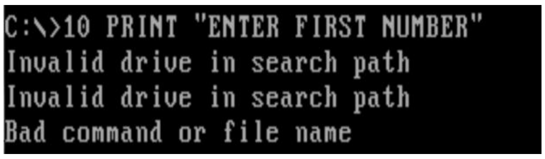
```
]10 PRINT "ENTER FIRST NUMBER"
]20 INPUT A
]30 PRINT "ENTER SECOND NUMBER"
]40 INPUT B
]50 LET S=A+B
]60 PRINT "THE SUM IS "
]70 PRINT S
]80 END
]
```

Elementary, ain't it, dear Reader? Just like charm, the program runs and does what you want it to do:



```
]RUN
ENTER FIRST NUMBER
?10
ENTER SECOND NUMBER
?20
THE SUM IS
30
```

And there was I sitting in front of the unfriendly beast, doing my best, annoying my peers with the loud clicks of the clikety-click keyboard:



```
C:\>10 PRINT "ENTER FIRST NUMBER"
Invalid drive in search path
Invalid drive in search path
Bad command or file name
```

All of this was completely new to me - even the language. We hadn't had a proper English class until then, and my entire knowledge of the language came from the set of basic BASIC commands from last years' course and of course watching American movies from the late '80s and early '90s - freshly out of Hollywood and straight onto our screens. They provided a much-needed escape from the ever-present face of our revered leader, the fearless Conducător of the country, whose eternal presence conveniently filled most of the meager two-hour daily TV schedule during the dark days of the communist regime. So naturally, saying *"Hasta la vista, baby"* rolled off my tongue far more easily than anything technical. The word command instantly made me think of *Commando*, and I missed *The Good and The Ugly* … and now suddenly, there I was, staring down this unforgiving beast as it bombarded me with cryptic errors about things I had never even heard of.

Thankfully, my teacher quickly caught on to my confusion.

"Hey, you see, this is an IBM machine. We got it in an aid package this summer. It's more complex than the Apples we've been using. Would you mind sitting with one of your classmates until we figure out what to do with it?"

I minded.

There were already five students crowded around each Apple, while here I had an entire, albeit unfamiliar wonder all to myself. Faced with the choice between abandoning something new and intriguing or retreating to the comfort of the familiar and share a keyboard for ten other hands, my stubbornness kicked in.

"Sorry, sir, I'd rather sit here"

He nodded. "Well, good then. Here's the book that came with it - figure out what to do with it", then left me figuring it out, focusing on other students who somehow got confused down the road. The book was the MSDOS Reference book, for version 3.3, which after a while I have learned came with the machine. I even was allowed to take the book home for the weekend, where armed with a tiny English-Hungarian dictionary I learned the famous VER command, slowly deciphered the mysteries of directories using DIR, and learned that it was the hard-drive making the bell like sounds when the machine started. All this in theory, of course, but I was instantly hooked.

By the time we returned to school the following rainy Monday morning, I already knew what to do with the machine. It was my moment to show off.

I powered it on, navigated through the directories, and, armed with the reference book, slowly identified various applications I recognized from its pages. Then, suddenly, a familiar word jumped out at me: `QBASIC.EXE`.

After all, we had learned BASIC - but the DOS book barely mentioned it, aside from acknowledging its existence as a programming language. I hesitated for a few minutes, unsure, before finally mustering the courage to type `QBASIC.EXE`. And what appeared on the screen was an entirely new universe dedicated to programming. Totally different from the Apple-II, no more line numbers before commands, the freedom to navigate through the code, and an entire Help system at my disposal… Though, I have to admit - the very first time when I pressed "by mistake" the universal help (F1) key in the QBasic editor, I panicked. My program vanished, strange text appeared, and for a terrifying moment, I had no idea how to bring it back. But with the help of my faithful dictionary which I have carried with me to the schools for several weeks on, we quickly remediated the situation and got back my lost program.

These were the humble beginnings, but things were about to change.

Soon, we got a real upgrade - a dozen or so 286 machines armed with amber screens, courtesy of our sister school in Hungary, set up to run in the confines of a novel Novell network. The trusty XT where I'd written my first lines of code was promoted to server status, and before long, we all had a real IPX network.

Computer science students were expected to help set the network up (in the end, it was for our own benefit), so we spent several afternoons wiring cables, fixing terminators, and verifying the functionality of the machines—all of it mostly by trial and error. There were no detailed instructions or guides; we only had one clear direction: "This is what it's supposed to do, this is how it's supposed to look; cable here, terminator there, put it in the network card. Make it work, and don't break anything". It was almost a technical miracle that we got it to work. But by the end, we knew our network inside out - like the back of our hand. Well, except for the most important detail: the supervisor password. That privilege was reserved for our teacher.

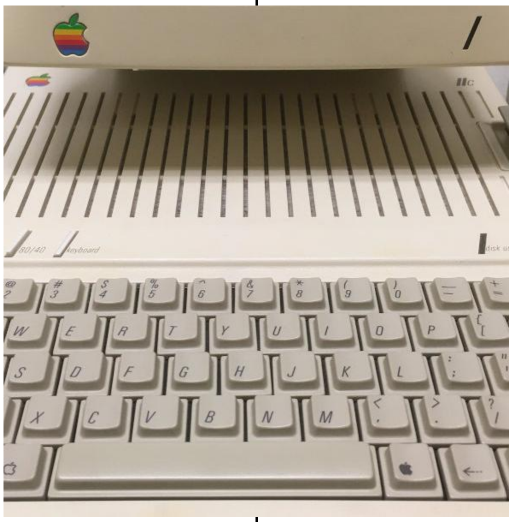
With the introduction of these cutting-edge new technologies (I mean, it's 1994, right?), we also decided to part ways with the beloved language of choice for beginners and graduate to something more decent: Turbo Pascal. It was around this time that our class began to shrink, with more than half of the students transferring to less demanding fields - coincidentally, right when pointers and memory management were introduced. Funny how that works, isn't it?

Looking back, the most interesting part is that, in those days, we had no manuals, no clear national curriculum for teaching computer science. Everything we learned came from a patchwork of university papers and courses, courtesy of our teacher—a man who had returned to his small provincial hometown to teach a bunch of high schoolers the mysteries of computing.

There was a sort of unofficial roadmap: first, you were supposed to learn Algorithms—because who doesn't love coding a bubble sort? Then, the following year, you'd move on to Graphics—because nothing was more refreshing than drawing a moving, ticking clock with the BGI library. And with each passing year, with each new subject we were exposed to, our numbers dwindled.

But those of us who stayed? We were a curious bunch. Stubborn, technical, like-minded, obsessed with computers and hungry for knowledge. And connected.

Because most importantly, we had contact with other schools. There were numerous competitions and Olympiads, mostly in the hard sciences - mathematics, computer science, physics. Whenever we visited another school, the first place we searched for was, of course, the computer lab. To our amazement, some of them had an entire room filled with color-screen computers. Meanwhile, in our school, we had exactly one. But then again, some schools we visited had only one or two computers total - even worse than our aging fleet of 286s.



But common to all of these schools was, that they had their own small circle of hungry minds - students eagerly working with computers, pushing the limits of what was possible, constantly experimenting. With no dedicated curriculum and no proper documentation and materials to explain the inner workings of a computer, everything was one massive, living, breathing experiment. An experiment that transcended classrooms, connecting students across schools, cities, and even counties.

While students in the West enjoyed the luxury of the internet, had email, access to forums with plethora of information, we had something more tangible: each other. Together with name, face, home addresses (not 127.0.0.1. but street, house, apartment), eventual phone numbers and envelopes just the right size to fit a floppy disk or two, carefully wrapped in aluminum foil for protection.

Someone, really bored at a school, one lazy afternoon discovered that calling interrupt 0x19 would instantly restart their computer. Because, really, what else would you do in your free time after you get your hands on some obscure documentation smuggled in from the decadent West. Meticulously go through each interrupt, write a small program to call it, observe what happens, and take notes—by hand, on actual paper. Then copy the paper several times, put them in envelopes and send to your friends. By the following week, students across the country were all doing the same thing: experimenting with interrupts like it was the next big thing in computing. For us it was. And then, across the country, baffled teachers were left wondering - why were computers suddenly restarting at random, with no one even touching them?

We didn't have access to the full Ralf Brown's interrupt list, nor did Peter Norton's award-winning book ever make its way into our hands. We didn't even know they existed. Our teachers were clueless about these details. But we *had* to figure it out, we *needed* to know, and we patched together the bits and bytes from various sources.

Some of us took this further. Suddenly sitting down to a computer was not what it used to be: simulated startup boot sequences, meticulously imitating a real boot, in the end asking for your password to log in were the norm for several weeks. Some of us took this even further: we learned how to write TSR programs (remember, these are still DOS days, and regardless that we didn't own a copy of Undocumented DOS, we still pieced it out using scratches of information gathered from several sources).

Now, no-one was safe: all keypresses were logged in a secret file, deeply burrowed in a claustrophobic hierarchy of intertwined folders, so that no-one finds them by mistake. We shared this knowledge with friends from other schools. As a special thank you gift, they shared with us what have they discovered: XOR based encryption. Now, we could keep all our secrets hiding in plain sight.

Around this time, the highly influential book "Applied Cryptography: Protocols, Algorithms, and Source Code in C" by Bruce Schneier was making waves, but we had no idea it even existed. Instead, we went directly against its most important piece of advice: "Never roll your own cryptography routines" - and invented our own algorithms.

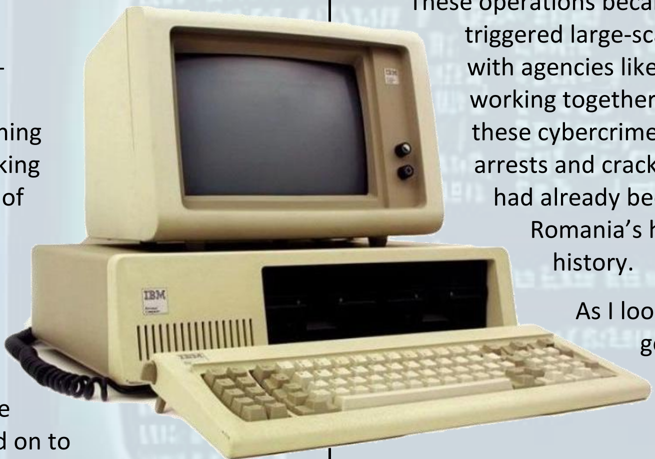
We didn't even call it cryptography. No, we dubbed it "secretizing" - because, at the time, we had no clue there was already a proper term for it. Honestly, we were just 15-year-old high school students, with a curriculum that didn't touch on any of these advanced topics. We were expected to solve endless Towers of Hanoi, and as long as we managed that, our passing grade was practically guaranteed.

But the flow of information between friends did not stop. And, sadly, with the information also other unwelcomed guests creeped into our computers. Not intentionally, or course. Or not. The mid-90s marked the rise of the real fear of computer viruses, and we weren't immune to it either. Back then, amongst us, information was exchanged exclusively hand-to-hand on floppy disks, long before emails and broadband internet made downloads the norm... We actually had no internet for at least the following 8 or 9 years, but regardless, it was an exciting time - friends would eagerly swap the latest games, homemade programs, tutorials, never thinking twice about what might be lurking unseen. But as the number of viruses grew, so did the paranoia. Every disk was a potential Trojan horse, and some of the most infamous viruses could spread like wildfire before you even realized something was wrong. We'd pass around antivirus tools alongside our favorite software, hoping they'd be enough to catch infections before they could wreak havoc. It was a game of trust, caution, and sometimes, unfortunate lessons learned the hard way.

But some viruses were simply too new, even for our latest antivirus software - how inconvenient. That's when we knew it was time to step up. If a well-known EXE started misbehaving, such as exposing longer than usual startup times, a size that was different than yesterday, or worse, if our once-trusty floppies suddenly refused to boot, we knew had a problem. Commercial antivirus? Sure, it caught last month's threats, but the real fun started when we had to play digital detectives ourselves.

Armed with DOS-based hex editors, disassemblers like Source, debuggers like Turbo Debugger or just plain debug, homemade honeypots, and an unhealthy amount of curiosity, we cracked open the infected files. We'd look for telltale patterns - strange jumps, code that makes no sense, or that all-too-familiar "INT 13h" call that meant a boot sector was getting trashed at some point. Sometimes, we'd find a crude signature left behind by the virus author, as if they expected us to appreciate their handiwork (Greetings, Windom Earle, author of the nondestructive Farside virus ... it took us 2 weeks to write a cleaner for programs infected with your virus ... and did not call that number).

Once we figured out how the virus spread - maybe hooking into the boot sector, maybe hijacking COMMAND.COM, maybe something else, it was time to fight back. We patched binaries, wrote tiny routines to nuke the infection, and sometimes even cobbled together our own crude "antivirus" tools in Turbo Pascal or C, which we picked up alongside the road from various sources, since again, we had no proper teaching material dedicated to it. These programs certainly were not exactly F-PROT-grade, but hey, they worked. Of course, by the time we got everything cleaned up, another virus was making the rounds. But that was just part of the game, wasn't it?



And as time rolled on, so did we. The cat-and-mouse game of viruses and catching them had been a thrilling playground, but eventually, our paths diverged. We slowly finished high school, moved on to university to get a diploma, fill our brain with mostly useless and outdated subjects, and forget all the interesting stuff we learned in high school. Some of us moved into entirely different fields beyond programming, finding our calling as architects, lawyers, priests, and teachers.

Then, others stayed with the computers and moved on to more interesting endeavors - after all, there were always new frontiers to explore. The writer of these lines, for instance, found a new obsession in the mesmerizing world of intros, demos, and graphical wizardry, where mathematics became the brush and the screen the canvas. Turning raw numbers into stunning visuals felt just as thrilling as dissecting malware, only now, the goal was to create new graphics effects never seen till then.

The most dedicated among us discovered a passion for security research, learning more and more advanced topics to break systems and then fix them. By the late '90s, international cybersecurity firms started noticing these talented individuals, and suddenly, the same skills that once got people in trouble were landing them jobs. By the 2000s, Romania had built a global reputation - not just for cybercrime, but for cybersecurity expertise. Today, many of us who once stayed up all night dissecting viruses and breaking computers are working for top firms, protecting the very systems you are using to read these lines.

And then, some others pursued the darker path, crossing the thin line between curiosity and something a little less innocent. Their names started showing up in places we wished they hadn't—local news reports, TV programs, even an article about a school network mysteriously crashing right before an important exam. Their deeds were whispered about in tight-knit circles, but everyone knew the truth: it was just a small mistake, born from curiosity and a lack of knowing where to stop

Finally, there were those who strayed too far. As e-commerce took off, some saw an opportunity not just to explore, but to profit. Credit card fraud became a booming underground industry, with hackers finding ways to steal credit card data through phishing schemes, compromised payment systems, and brute-force attacks on online stores. The more ambitious ones didn't just use stolen cards to make purchases - they built entire operations, selling card details in hidden online forums, creating fake storefronts to harvest credentials, or laundering money through increasingly sophisticated techniques.

These operations became so extensive that they triggered large-scale international investigations, with agencies like the FBI, Europol, and Interpol working together to track down and dismantle these cybercrime networks leading to high-profile arrests and crackdowns, but by then, the damage had already been done - and the legend of Romania's hackers was cemented in digital history.

As I look back, In the end, we were just a generation of kids with a relentless curiosity, growing up in a time and place where knowledge was scarce but determination was boundless. We built networks before we fully understood them, wrote programs before we knew the theory, and experimented with systems just to see what would happen. Some of us turned this passion into careers, helping secure the very technology we once broke, while others took a different path, lured by the opportunities that an unregulated digital world provided. But whether we became cybersecurity experts, software engineers, or infamous figures in underground circles, we all shared the same beginning - the thrill of discovery, the hunger for knowledge, and the unshakable belief that every locked door was just another challenge waiting to be solved.

Funny how things turn out.

Subject: Join Our Retro Computer Magazine as a Contributor!

Dear Prospective Contributor,

Are you passionate about classic video games, retro hardware, or the golden age of computer programming? Do you have a wealth of knowledge, personal stories, or insights that you’d love to share with fellow enthusiasts? If so, we invite you to become a contributor to our retro computing magazine!

Beware! This isn’t about making money - it’s about pure, unfiltered, button-mashing fun! No corporate overlords, no budget meetings, just a bunch of passionate individuals coming together to have some fun and make a mark on the world. If you’ve ever wanted to wax poetic about the majesty of Mode 0x13 graphics or debate the best Mega Man boss, this is your chance!

We are looking for articles on topics such as:

- Retrospectives on classic games and consoles
- Programming these retro machines
- Forgotten gems and overlooked classics
- Show us your precious hardware
- Personal experiences and gaming memories
- Restoration and preservation of retro hardware and software

As a contributor, you’ll have the chance to establish yourself as a voice in the retro gaming community, be part of an exciting project that honors gaming’s past while keeping its legacy alive - just for the sheer fun of it!

If you’re interested, please reach out to us, send your topic ideas, and any relevant writing samples or even a full article (if available). Since we’re not limited by the constraints of print, everything is digital, if your submission is a great fit, it will make its way into one of our future issues - assuming we keep this pixelated party going!

Let’s bring the joy of retro gaming to a wider audience together, one nostalgic power-up at a time. We look forward to hearing from you!

Have a sunny day!

retrowtf25@gmail.com

Next issue will be available: **When it's Done**

In our next issue you will be delighted with:

- Retro Review: **Ultima Underworld: The Stygian Abyss**
- Old vs. New: **Diablo II vs. Diablo II Resurrected**
- Retro coding: **C tutorial, Part 2**
- Playing Hide and Seek through the Ages: **Robin Hood vs. Desperados vs. Commandos**
- Tools of the trade: **Fight of the PC Emulators**
- Show us Your Rig: **The IBM Aptiva 2139**

... And some more

... or at least, this is the plan for now.

See you
soon!