The C++ Field Guide

All my guides, combined and corrected. One source of truth. — From Mark

> Peter —
>
> I've sent you three PDFs over the last couple weeks. I screwed a few things up in the first two, fixed them in the third. This one rolls all of it into a single book so you're not flipping between files on your phone trying to figure out which version to trust.
>
> Here's the rule for this document: **this is the source of truth.** If it disagrees with the earlier PDFs, trust this one. I fixed the file-handling stuff (`fstream` instead of `ifstream` / `ofstream`), I fixed the `displayMenu` signature to match what your assignment actually requires, and I split the file prompt into two functions the way the spec wants.
>
> I also added a new section toward the back called *How to Code 101* — the thing nobody teaches you in class. It's the part where before you type anything, you write down the knowns and unknowns of the problem, then the nouns and verbs. That's 80% of the battle. I walk through it with a Blackjack game because it's fun and every concept you need shows up.
>
> Read it slow. Fix the two functions we went over. Ping me when you hit the next wall.
>
> — Mark

What's in here

Part 1 · C++ Basics

Variables, loops, if/else, functions — the stuff every program is built from.

1. What a C++ program actually is

A C++ program is a list of instructions the computer runs top to bottom. Every program starts at `main()`. That's the front door.

```cpp
// Every C++ program looks something like this
#include <iostream>     // bring in tools for printing/reading
using namespace std;

int main()
{
    cout << "Hello, Peter" << endl;
    return 0;       // 0 means "we finished clean"
}
```

Three things to notice:

- `#include` lines pull in libraries — toolboxes other people wrote. `iostream` gives you `cout` and `cin`.
- Every statement ends with a semicolon `;`. Forget one and the compiler will bark.
- Curly braces `{ }` group code into a block. Every function has a block. So does every loop and every `if`.

2. Variables — labeled buckets

A variable is a name for a spot in memory where you stash a value. You tell C++ the *type* of value that bucket holds, and C++ won't let you put the wrong thing in there.

```cpp
int iAge = 34;              // whole number
double dPrice = 9.99;       // decimal number
char cGrade = 'A';           // single character (note: single quotes)
bool bIsOpen = true;         // true or false, nothing else
string szName = "Magnolia";  // text (double quotes, needs <string>)

const double dSALES_TAX = 0.08;  // const = "can't be changed"
```

> **Naming style:** The `i`, `d`, `sz`, `b` prefixes in your code are called Hungarian notation. Old school, but your professor likes it, so keep doing it. Full story in the appendix.

3. Getting stuff in and out

```cpp
cout << "What is your name? ";
string szName;
getline(cin, szName);   // reads the whole line including spaces

int iAge;
cin >> iAge;              // reads just one word/number

cout << "Hi " << szName << ", you are " << iAge << endl;
```

`cout` pushes stuff out to the screen. `cin` pulls stuff in from the keyboard. The `<<` and `>>` arrows always point in the direction data is flowing.

> **The `cin.ignore()` gotcha:** When you mix `cin >> something` with `getline()`, the first one leaves a stray newline in the buffer and the `getline` eats it and returns empty. Fix: call `cin.ignore();` between them. You've already hit this — now you know why.

4. If / else — making decisions

```
if (iAge >= 65)
{
    cout << "Senior price";
}
else if (iAge >= 18)
{
    cout << "Adult price";
}
else
{
    cout << "Child price";
}
```

Operators you'll use constantly:

Comparisons

- `==` equal to *(not* `=`*)*
- `!=` not equal
- `<` `>` `<=` `>=`

Logic

- `&&` AND (both must be true)
- `||` OR (at least one)
- `!` NOT (flips true/false)

> **Watch the precedence.** `&&` binds tighter than `||`. So `a || b && c` means `a || (b && c)`, not `(a || b) && c`. If you ever feel unsure, wrap it in parentheses. Clarity beats cleverness every time. This bit you in the theater code — we'll look at that in Part 3.

5. Loops — doing something over and over

```
// while: check first, then run
while (iCount < 10)
{
    cout << iCount << endl;
    iCount++;    // shorthand for iCount = iCount + 1
}

// do-while: run first, then check (always runs at least once)
do
{
    cout << "Enter choice: ";
    cin >> iChoice;
} while (iChoice != -1);

// for: counter, condition, step — all in one line
for (int i = 0; i < 5; i++)
{
    cout << "step " << i << endl;
}
```

Rule of thumb: use `for` when you know how many times, `while` when you don't, and `do-while` when you need to run once before checking (menus are the classic example).

6. Functions — the piece you're working on

A function is a little machine. You give it inputs (parameters), it does a job, and sometimes it hands you back a result (return value).

```
// Recipe:  [return type]  [name]  ( [parameters] )  { [body] }

double calculateTax(double dPrice, double dRate)
{
    return dPrice * dRate;
}

int main()
{
    double dTax = calculateTax(19.99, 0.08);
    cout << "Tax: $" << dTax;
}
```

Things to lock in:

- **Return type** says what comes back out. `void` means nothing comes back.
- **Parameters** are inputs. They act like local variables inside the function.
- **One job per function.** If your function name has "and" in it, it's doing too much. A function called `promptAndValidateAndOpenFile` is three functions in a trench coat.
- **Declare the signature near the top** (or in a header file) so `main` knows it exists before it calls it.

> **The golden rule:** Every function should be small enough that when you read its name, you already know what it does. If you have to read the body to figure it out, the name is lying.

7. Pass by value vs pass by reference

This is the single concept that trips everyone up on their first functions assignment. Read it twice.

Pass by value — a photocopy

```
void addTen(int x)
{
    x = x + 10;
}

int main() {
    int n = 5;
    addTen(n);
    cout << n;  // still 5!
}
```

The function got a *copy*. Changing the copy doesn't touch the original.

Pass by reference — the real thing

```
void addTen(int& x)
{
    x = x + 10;
}

int main() {
    int n = 5;
    addTen(n);
    cout << n;  // now 15
}
```

The `&` means "give me the original, not a copy." Changes stick.

And there's a third flavor you'll see all over your assignment:

```
void displayProduct(const Product& prod)   // const reference
{
    cout << prod.szName;
}
```

`const` + `&` means: *don't copy it* (fast), and *don't let me accidentally change it* (safe). Use this for any big thing you only need to read — structs, strings, etc. It's the right default for "read-only" parameters.

> **Quick decision tree:**
>
> - Will the function *change* the variable? → Pass by reference: `Type& x`
> - Just reading, and it's small (int, double, char, bool)? → Pass by value: `Type x`
> - Just reading, but it's big (struct, string, array)? → Const reference: `const Type& x`

8. Structs — grouping related stuff

A struct is a custom type you invent. It's how you say "a theater is a *bundle* of a name, a number of screens, and some prices, all together."

```
struct Product
{
    string szName;
    string szSku;
    int iStock;
    double dPrice;
};   // <-- don't forget this semicolon

int main()
{
    Product pen;
    pen.szName = "Blue Pen";
    pen.iStock = 50;
    pen.dPrice = 1.49;

    cout << pen.szName << " costs $" << pen.dPrice;
}
```

You reach into a struct with a dot: `pen.szName`. That's it. That's the whole trick.

> **Why structs matter for functions:** instead of passing five separate variables (name, sku, stock, price, supplier), you pass one `Product&` and the function can see all of it. That's why your `Theater` struct exists — so every function can get at theater info by taking a single parameter.

9. Files — quick intro (CORRECTED: use `fstream`)

> **Heads up — I goofed in the earlier PDF.** I showed `ifstream` (read only) and `ofstream` (write only) as separate types. Your assignment locks the signatures to `fstream&`, which is the combo type that does both. You just pass a mode.

```
// The combo type — works for reading AND writing
fstream fileInput;
fileInput.open(szFilePath, ios::in);    // reading
// ...
fileInput.close();

// Writing (overwrite):
fstream fileOut;
fileOut.open("members.txt", ios::out);
fileOut << "Mark Ralston" << endl;
fileOut.close();

// Writing (append to end):
fileOut.open("members.txt", ios::app);
```

Everything else is the same — `.is_open()`, `.close()`, `>>`, `<<`, `getline()`.

The pattern is always: **open, check it opened, use it, close it.** If you skip the check you'll crash when the file's missing.

```
fstream fileInput;
fileInput.open("theater.txt", ios::in);
if (fileInput.is_open())
{
    string szLine;
    getline(fileInput, szLine);
    fileInput.close();
}
```

Part 2 · Dick's Office Supply Store

A small program, broken into functions the right way. Now with every correction from Sim 3 baked in.

The problem

Dick runs a small office supply store. He needs a program that lets him:

- See his current inventory
- Add stock when a shipment arrives
- Ring up a customer sale and get a total
- See the total value of what's on his shelves

Here's the data we care about — a single product:

```
struct Product
{
    string szName;
    string szSku;
    int iStock;
    double dPrice;
};
```

And we'll hold a handful of them in a regular C-style array. (Your class will get to vectors eventually; arrays are fine for now.)

Function 1: displayMenu (CORRECTED)

> **Heads up — earlier PDF had this wrong.** I showed `displayMenu` with empty parentheses and hardcoded choices. Your assignment locks it to the signature below. Same function gets called for the main menu and the admin menu, with different arrays passed in. Don't hardcode the choices.

```
void displayMenu(string szMenuName, string szChoicesArr[], int iChoices)
{
    cout << "\n" << szMenuName << endl;
    cout << "****************************" << endl;

    for (int i = 0; i < iChoices - 1; i++)
    {
        cout << (i + 1) << ". " << szChoicesArr[i] << endl;
    }
    cout << "-1. " << szChoicesArr[iChoices - 1] << endl;

    cout << "****************************" << endl;
    cout << "Enter choice: ";
}
```

Notice what's *not* in there: no `cin`, no `if` statements, no calls to other functions. That's intentional. The menu doesn't know what the choices do — that's `main`'s job. One job, one function.

Function 2: displayProduct

This one needs a product to show. It's only reading, so we pass by **const reference** — fast (no copy) and safe (can't accidentally change it).

```
void displayProduct(const Product& prod)
{
    cout << "  " << prod.szSku
         << "  " << prod.szName
         << "   stock: " << prod.iStock
         << "   $" << prod.dPrice << endl;
}
```

And if we want to show the whole inventory, we build it out of the single-product version. One function, one job.

```
void displayInventory(const Product inventory[], int iCount)
{
    cout << "\n--- Inventory ---\n";
    for (int i = 0; i < iCount; i++)
    {
        displayProduct(inventory[i]);
    }
}
```

> **Building up, not down:** The small function (`displayProduct`) doesn't know there's a bigger function using it. The bigger function reuses the small one instead of duplicating logic. This is how real programs get built — small, trustworthy pieces stacked together.

Function 3: addStock — this one *changes* things

When a shipment arrives, we want to bump the stock count. That means **modifying** a product, which means pass by `&` — no `const` this time, because we're going to change it.

```
void addStock(Product& prod, int iAmount)
{
    prod.iStock += iAmount;     // shorthand for: prod.iStock = prod.iStock + iAmoun
    cout << "Added " << iAmount << " of "
         << prod.szName << ". New stock: "
         << prod.iStock << endl;
}
```

Since the function got the *real* product (not a copy), when it changes `prod.iStock`, the change sticks in `main`'s array.

> **Compare it to value-passing:** If you wrote `void addStock(Product prod, ...)` without the `&`, the function would make a copy, bump the copy's stock, print the new number, and throw the copy away when it returned. Your real inventory would never change. That's a classic beginner bug — the program "works" (no error) but nothing happens. Always ask: "does this function need to change the argument?" If yes, use `&`.

Function 4: calculateLineTotal — *returns* a value

When a customer wants to buy 3 staplers, we want to hand back the price. `void` doesn't cut it — we need a `double` coming out.

```
double calculateLineTotal(const Product& prod, int iQuantity)
{
    return prod.dPrice * iQuantity;
}
```

Four lines. No printing. No input. Just: "given this product and a quantity, what's the price?" That's it. You can call it, you can use the result anywhere, you can test it by feeding it known values.

Function 5: findProductBySku

Given a SKU, walk the array and return the *index* of the matching product. Return `-1` if it's not there — a common convention for "not found."

```
int findProductBySku(const Product inventory[], int iCount,
                     const string& szSku)
{
    for (int i = 0; i < iCount; i++)
    {
        if (inventory[i].szSku == szSku)
        {
            return i;
        }
    }
    return -1;    // not found
}
```

Main — short, because the helpers do the work

This is what we were aiming for. `main` reads almost like the menu itself. No logic buried in it — it just calls the right helper for the user's choice.

```
int main()
{
    cout << fixed << setprecision(2);

    Product inventory[100];
    int iCount = 0;

    // Seed a few products (in a real program these would load from a file)
    inventory[iCount++] = {"Blue Pen",    "PEN-001", 50,  1.49};
    inventory[iCount++] = {"Legal Pad",   "PAD-002", 20,  3.99};
    inventory[iCount++] = {"Stapler",     "STP-003",  8, 12.99};

    // Menu choices stored in an array so displayMenu can print them
    string szChoices[] = {
        "Show inventory",
        "Add stock",
        "Ring up a sale",
        "Total inventory value",
        "Quit"
    };

    int iChoice;
    do
    {
        displayMenu("Dick's Office Supply", szChoices, 5);
        cin >> iChoice;
        cin.ignore();

        if (iChoice == 1)
        {
            displayInventory(inventory, iCount);
        }
        else if (iChoice == 2)
        {
            string szSku;
            int iAmount;
            cout << "SKU: ";     cin >> szSku;
            cout << "Amount: ";  cin >> iAmount;

            int i = findProductBySku(inventory, iCount, szSku);
            if (i == -1)   cout << "No such SKU.\n";
            else           addStock(inventory[i], iAmount);
        }
        else if (iChoice == 3)
        {
            string szSku;
            int iQty;
            cout << "SKU: ";       cin >> szSku;
```

```
            cout << "Quantity: ";  cin >> iQty;

            int i = findProductBySku(inventory, iCount, szSku);
            if (i == -1)    cout << "No such SKU.\n";
            else
            {
                double dTotal = calculateLineTotal(inventory[i], iQty);
                cout << "Total: $" << dTotal << endl;
                addStock(inventory[i], -iQty);    // negative = reduce stock
            }
        }
    } while (iChoice != -1);

    return 0;
}
```

Look at what `main` doesn't contain. No price math. No stock math. No SKU searching. No formatting. Every one of those lives in its own named function. That's the goal — `main` should read like a summary, not a novel.

A problem close to yours (CORRECTED)

Now let's tackle something that parallels the piece you're stuck on. In your theater program, you've got a function that prompts for a file path. Your assignment splits it into *two* functions on purpose.

> **Heads up — earlier PDF had this wrong too.** I mashed "get a valid filename" and "open the file" into one function. Your assignment splits them, and honestly the split is cleaner than what I showed you.

The split (what the spec actually wants)

> **promptForFilename()** — only checks the string *looks* like a filename (ends in `.txt`, no spaces). Doesn't try to open anything. Returns the filename, or the literal string `"EXIT"` if the user types `exit`.
>
> **processTheaterInformation()** — calls promptForFilename, then tries to open. If open fails, ask again. One function, one job.

Version 1 — the "shove it all in one function" way (BAD)

```
// Does: prompts, validates, opens file, reads products, returns count.
// Also: signals failure by... returning -1? or "EXIT"? it's unclear.
int loadProducts(Product inventory[])
{
    string szPath;
    fstream inputFile;

    cout << "File path: ";
    getline(cin, szPath);

    while (szPath != "exit" || !inputFile.is_open() &&
           szPath.find(".txt") != string::npos)
    {
        inputFile.open(szPath, ios::in);
        if (!inputFile.is_open())
        {
            cout << "Bad path, try again: ";
            getline(cin, szPath);
        }
        if (szPath == "exit") return -1;
    }

    int iCount = 0;
    while (inputFile >> inventory[iCount].szSku)
    {
        inputFile >> inventory[iCount].szName
                  >> inventory[iCount].iStock
                  >> inventory[iCount].dPrice;
        iCount++;
    }
    inputFile.close();
    return iCount;
}
```

What's wrong with this:

1. **Four jobs in one function:** prompting, validating, opening, reading. If any piece needs to change, you have to understand the whole blob.
2. **The loop condition is a trap.** `a || b && c` isn't `(a || b) && c` — precedence bites. Mixing three booleans in one `while` is a recipe for off-by-one and infinite loops.
3. **Magic return values.** `-1` means "user bailed." Who's supposed to remember that? Every caller has to check. Better to separate *asking* from *using*.
4. **Not reusable.** If later you want to load a *members* file, none of this helps you — it's welded to products and file-prompting together.

Version 2 — Broken up, one job each (GOOD)

Same feature, split into two small functions. Each one could be tested on its own, swapped out, or reused elsewhere. This is what your spec wants.

Step 1 — Just get a filename that *looks* valid. This function doesn't know or care what the file is for, or whether it exists. It prompts, checks the shape of the string (ends in `.txt`, no spaces), and hands the path back. If the user types `"exit"`, it returns the literal string `"EXIT"` — a clear signal.

```
string promptForFilename()
{
    string szFilePath;
    bool bValid = false;

    do
    {
        cout << "Please enter a valid file path: ";
        getline(cin, szFilePath);

        if (szFilePath == "exit")
        {
            return "EXIT";
        }

        // Valid means: ends in .txt AND has no spaces
        bool bEndsInTxt = szFilePath.length() >= 4 &&
            szFilePath.substr(szFilePath.length() - 4) == ".txt";
        bool bNoSpaces  = szFilePath.find(' ') == string::npos;

        if (bEndsInTxt && bNoSpaces)
        {
            bValid = true;
        }
        else
        {
            cout << "Invalid filename. Must end in .txt with no spaces."
                 << endl;
        }
    }
    while (!bValid);

    return szFilePath;
}
```

Step 2 — Loop until the file opens, then process it. This function calls `promptForFilename` — it doesn't reinvent the prompting logic. If the returned string is `"EXIT"`, bail. Otherwise try to open. If open fails, loop.

```
void processTheaterInformation(fstream& fileInput, Theater& myTheater)
{
    string szFilePath;
    bool bFileReady = false;

    do
    {
        szFilePath = promptForFilename();   // reuse it!

        if (szFilePath == "EXIT")
        {
            cout << "Exiting function due to early exit of file prompt"
                 << endl;
            return;
        }

        fileInput.open(szFilePath, ios::in);
        if (fileInput.is_open())
        {
            bFileReady = true;
        }
        else
        {
            cout << "File not found. Try again." << endl;
        }
    }
    while (!bFileReady);

    // --- now read the file ---
    // File has: theater name, number of rooms, starting funds,
    // and three sections (*Theater Rooms*, *Pricing*, *Employee
    // Information*) that can appear in ANY order. Loop through the
    // file, detect the marker lines, and branch on which one you hit.
    getline(fileInput, myTheater.szName);
    fileInput >> myTheater.iNumberScreens;
    fileInput >> myTheater.dFunds;    // you were missing this
    fileInput.ignore();

    string szLine;
    while (getline(fileInput, szLine))
    {
        if (szLine == "*Theater Rooms*")
        {
            // read iNumberScreens rooms
        }
        else if (szLine == "*Pricing*")
        {
            // read adult, senior, child, member, membership fee
```

```
        }
        else if (szLine == "*Employee Information*")
        {
            // read employees until EOF or next marker
        }
    }

    fileInput.close();
}
```

Why this version holds up better:

1. **One job, one function.** `promptForFilename` prompts. `processTheaterInformation` opens and reads. When something breaks, you know which box to open.
2. **Loop conditions are simple.** Each `while` checks one thing. No precedence puzzles.
3. **No magic strings or numbers.** `"EXIT"` means no file. Obvious at the call site.
4. **Reusable.** `promptForFilename` works for products, members, customers, anything. Write it once, use it anywhere.
5. **Testable.** You could call each function with a fake input and check the output. You can't test the monster in Version 1 without also typing into the console.

Part 3 · What's Broken in Your Code

Let me show you instead of telling you. Your version vs what it should be, side by side.

Your `promptForFilename`

YOUR VERSION

```
string promptForFilename()
{
    string szFilePath;
    bool valid = false;
    cout << "Please enter a valid file path ";
    getline(cin, szFilePath);
    do
    {
        if (szFilePath == "exit")
        {
            cout << "Exiting simulation early";
            return "EXIT";
            break;
        }
        else if (szFilePath.find(' ') != string::npos &&
                 szFilePath.find(".txt") != string::npos)
        {
            valid = false;
            cout << "Please enter a valid file path ";
            cout << endl << endl;
            cout << "Please enter theater file path: ";
            getline(cin, szFilePath);
        }
        else { valid = true; }
    }
    while (!valid);
    return szFilePath;
}
```

WHAT IT SHOULD BE — see the corrected `promptForFilename` in Part 2. Three specific things were wrong:

1. The validation check is backwards. You wrote:
`find(' ') != npos && find(".txt") != npos`
which translates to: "flag as bad if it has a space AND has .txt in it." But the spec says *valid* means ends with `.txt` AND has *no* spaces. Your check only catches `"my file.txt"` and lets garbage like `"theater"` or `"theater.exe"` through.

Fix next time: Write the *valid* condition first, separately, in plain English. Then translate each piece to code. If it feels twisty, negate the whole thing: `if (!(valid)) { reject; }`.

2. You prompt twice on the first run. There's a `cout << "Please enter..."` and a `getline` *before* the loop, then the same prompt inside the loop as the first thing. User sees it twice. A do-while prompts once per iteration — put the prompt inside, not before.

3. The `break` after `return "EXIT"` is dead code. Return leaves the function immediately. Anything after it never runs. Compiler might even warn you. Delete.

Your `processTheaterInformation`

YOUR VERSION (TRIMMED TO THE PROMPT SECTION)

```
void processTheaterInformation(fstream& fileInput, Theater& myTheater)
{
    string szFilePath;
    bool valid = true;
    cout << "Please enter file path ";
    getline(cin, szFilePath);
    do
    {
        if (szFilePath.find(' ') != string::npos &&
            szFilePath.find(".txt") != string::npos)
        {
            cout << "Please enter valid file path ";
            getline(cin, szFilePath);
            fileInput.open(szFilePath, ios::in);
        }
        else if (!fileInput.is_open()) { cout << "File not found... "; }
        else if (szFilePath == "exit")
        {
            cout << "Exiting function due to early exiting of file prompt";
            break;
        }
    } while (szFilePath != "exit" ||
             valid != false && !fileInput.is_open());

    // ... then only reads 2 fields out of a whole theater file ...
}
```

WHAT IT SHOULD BE — see the corrected `processTheaterInformation` in Part 2. Three specific things were wrong:

> **1. You didn't call `promptForFilename()`.** You wrote the function, then reimplemented filepath prompting from scratch inside this one. Two copies of the same broken logic. When you have a function that does a job, *call it*.

> **The instinct to build:** If you catch yourself writing the same kind of code twice, stop. That instinct — "I've seen this shape before, let me reuse it" — is 80% of being a decent programmer.

2. The loop condition is a precedence disaster. You wrote:
`while (szFilePath != "exit" || valid != false && !fileInput.is_open())`
Three conditions, mixed `||` and `&&`, no parentheses. Because `&&` binds tighter than `||`, this evaluates as `(szFilePath != "exit") || (valid != false && !fileInput.is_open())`, which is almost certainly not what you meant.

Fix next time: If a loop has more than two conditions, either parenthesize everything explicitly, or pull the logic into one named boolean before the loop. The compiler doesn't care but future-you will.

3. You only read two things out of a whole file. You got the theater name and number of screens, then stopped. The file also has starting funds, three theater rooms, five pricing values, and three employees — and the spec says those sections can appear in any order, so you need to watch for the marker lines (`*Theater Rooms*`, `*Pricing*`, `*Employee Information*`) and branch on which one you just read.

Fix next time: Before you write a function, list its steps as comments first. If the spec mentions a thing, it gets a comment. If sections can appear in any order, you already know you need a loop plus marker detection, not a fixed sequence of reads.

Part 4 · Strategy: Work Wide, Then Deep

> **You have nine parts and seven of them still say "not implemented."** Don't perfect part 2 before moving on. `displayTheaterInfo` is like ten lines. Get something *real* happening in every function, even if it's ugly, then circle back.
>
> Partial credit across all nine beats a beautiful part 2 with everything after it blank. I've watched people fail assignments by getting obsessed with the first problem.

Here's the order I'd go in on your assignment:

1. **Skeleton pass (1 hour).** Every function gets a `cout << "TODO: " << __func__ << endl;` inside it. The program compiles and runs. Every menu option "works" (prints a placeholder). You now have a frame.
2. **Easy pass (1 hour).** Do the short functions first — `displayTheaterInfo`, `displayMovies`, `displayMenu`. Each one is 5–15 lines. You'll knock out three or four of the nine fast.
3. **Medium pass (2 hours).** File prompt + process theater info. Member check. Ticket math.
4. **Hard pass (2 hours).** Employee menu, password updates, license fees.
5. **Polish pass.** Formatting, edge cases, the stuff the professor takes points for.

Every pass leaves you with a program that *runs*. If the deadline hits mid-pass, you still have something to submit.

Part 5 · How to Code 101

The thing nobody teaches you. Before you write a single line, write down the knowns, unknowns, nouns, and verbs. Every time.

The process

Here's a confession: I don't start typing code. I start with a blank page and a pen, and I write four lists. Only when those lists make sense does the keyboard come out.

1. Knowns — what do I have?

What facts does the problem hand me? What's the input? What files, numbers, rules, constraints are given? These are the things I don't have to figure out — they're just true.

2. Unknowns — what am I asked for?

What's the output? What does "done" look like? If I can't describe "done" in one sentence, I'm not ready to start. This is the single biggest reason assignments go sideways — people start coding before they know what finished looks like.

3. Nouns — what data do I need?

Every noun in the problem is a candidate for a variable or a struct. If two or three nouns naturally travel together ("a card has a suit, a rank, and a value"), that's a struct. If you have many of the same noun, that's an array.

4. Verbs — what actions happen?

Every verb in the problem is a candidate for a function. "Deal a card." "Check the score." "Ask for input." "Print the hand." Each of those is probably a function. If a verb is big enough that it has sub-verbs ("play a round" = "deal cards + ask hit-or-stand + check bust"), the big verb is `main` and the small ones are helpers.

Nouns and verbs map to data and functions. Once you have the lists, the program writes itself.

> **This is a word-problem method.** It's the same trick they taught you in middle-school algebra for word problems — "Sarah has 12 apples, she gives half to John, how many does John have?" You underline the nouns and circle the verbs. Programming is exactly the same.

Let's do it for real: Blackjack

Let's say you want to write a single-hand Blackjack game you can play in the terminal. Dealer vs. you, one deck, no splits or doubles, just hit/stand. Simple enough to hold in your head, rich enough to touch every concept in this guide.

Step 1 — Knowns (read the problem twice, underline facts)

- A deck has 52 cards.
- Cards have suits (hearts, diamonds, clubs, spades) and ranks (2–10, J, Q, K, A).
- Number cards are worth their face value. J, Q, K are worth 10. Ace is worth 1 or 11 — whichever keeps the player under or at 21.
- Player and dealer each get 2 cards. Player sees both of theirs, dealer shows 1.
- Player chooses: hit (take another card) or stand (stop). Can keep hitting until they bust (go over 21) or stand.
- Dealer then draws until they hit 17 or higher.
- Higher hand (closest to 21 without going over) wins. Tie = push.

Step 2 — Unknowns (what does "done" look like?)

A program that, on one run: shuffles a deck, deals to player and dealer, lets the player hit or stand in a loop, plays out the dealer's turn automatically, prints who won. End of program.

Two sentences. That's the whole goal. If I can't keep my scope that tight, I'll never finish.

Step 3 — Nouns (data)

Card	has a suit (string) and a rank (string or int). That's a struct.
Deck	an array of 52 Cards, plus an index pointing at the "next card to deal."
Hand	an array of Cards + how many are in it. Two hands: player, dealer.
Score	an int, computed from a hand. Not stored — re-derived each time.
Choice	a char ('h' or 's') or an int (1 or 2). User input.

Step 4 — Verbs (functions)

Every verb I can pull out of the problem. I write them as function signatures *before* I write any bodies. Names first, bodies later.

```
void   buildDeck(Card deck[]);                      // fills the 52 cards
void   shuffleDeck(Card deck[]);                    // randomizes order
Card   dealCard(Card deck[], int& iNextCard);      // takes top card, advances index
int    calculateHandValue(const Card hand[], int iCount);
void   displayHand(const Card hand[], int iCount,
                   const string& szWho, bool bHideFirst);
char   promptHitOrStand();                          // returns 'h' or 's'
void   playerTurn(Card deck[], int& iNextCard,
                  Card hand[], int& iCount);
void   dealerTurn(Card deck[], int& iNextCard,
                  Card hand[], int& iCount);
void   announceWinner(int iPlayerScore, int iDealerScore);
```

Read down the list. Each name should tell you what the function does without you reading the body. That's the test. "buildDeck" — builds the deck. "calculateHandValue" — given a hand, tells you what it's worth. No ambiguity.

Notice the parameters already encode everything from Part 1:

- `const Card hand[]` — read-only, don't copy big thing (Const reference rule).
- `int& iNextCard` — the function needs to *advance* the deck index, so pass by reference.
- `const string& szWho` — read-only string parameter.

> **This is the whole trick.** By the time you've written the four lists, 80% of the program is designed. The typing is just the last 20%.

Now the code — Blackjack, start to finish

Here's the full program. Not polished, not fancy, but every piece follows the rules from Parts 1 and 2. Read it with the four lists in mind. Every struct is a noun. Every function is a verb. `main` is a summary.

```cpp
#include <iostream>
#include <string>
#include <cstdlib>     // rand(), srand()
#include <ctime>       // time()
using namespace std;

// ----- Nouns -----
struct Card
{
    string szSuit;
    string szRank;
    int iBaseValue;   // 2..10, face=10, ace=11 (we handle soft/hard later)
};

const int iDECK_SIZE = 52;
const int iMAX_HAND  = 12;   // can't draw more than ~11 and stay under 22

// ----- Verbs (declarations) -----
void buildDeck(Card deck[]);
void shuffleDeck(Card deck[]);
Card dealCard(Card deck[], int& iNextCard);
int  calculateHandValue(const Card hand[], int iCount);
void displayHand(const Card hand[], int iCount,
                 const string& szWho, bool bHideFirst);
char promptHitOrStand();
void playerTurn(Card deck[], int& iNextCard,
                Card hand[], int& iCount);
void dealerTurn(Card deck[], int& iNextCard,
                Card hand[], int& iCount);
void announceWinner(int iPlayerScore, int iDealerScore);

// ============================================================
int main()
{
    srand(time(0));   // seed the random number generator ONCE

    Card deck[iDECK_SIZE];
    buildDeck(deck);
    shuffleDeck(deck);

    int  iNextCard = 0;
    Card playerHand[iMAX_HAND];
    Card dealerHand[iMAX_HAND];
    int  iPlayerCount = 0;
    int  iDealerCount = 0;

    // Opening deal: 2 to player, 2 to dealer
    playerHand[iPlayerCount++] = dealCard(deck, iNextCard);
```

```
    dealerHand[iDealerCount++] = dealCard(deck, iNextCard);
    playerHand[iPlayerCount++] = dealCard(deck, iNextCard);
    dealerHand[iDealerCount++] = dealCard(deck, iNextCard);

    displayHand(dealerHand, iDealerCount, "Dealer", true);   // hide one
    displayHand(playerHand, iPlayerCount, "You",    false);

    playerTurn(deck, iNextCard, playerHand, iPlayerCount);

    int iPlayerScore = calculateHandValue(playerHand, iPlayerCount);
    if (iPlayerScore > 21)
    {
        cout << "\nYou busted at " << iPlayerScore << ". Dealer wins.\n";
        return 0;
    }

    dealerTurn(deck, iNextCard, dealerHand, iDealerCount);

    int iDealerScore = calculateHandValue(dealerHand, iDealerCount);
    announceWinner(iPlayerScore, iDealerScore);
    return 0;
}
```

```
// ----- Build a standard 52-card deck -----
void buildDeck(Card deck[])
{
    string szSuits[]  = {"Hearts", "Diamonds", "Clubs", "Spades"};
    string szRanks[]  = {"2","3","4","5","6","7","8","9","10",
                         "J","Q","K","A"};
    int    iValues[]  = {2,3,4,5,6,7,8,9,10,10,10,10,11};

    int iIdx = 0;
    for (int s = 0; s < 4; s++)
    {
        for (int r = 0; r < 13; r++)
        {
            deck[iIdx].szSuit     = szSuits[s];
            deck[iIdx].szRank     = szRanks[r];
            deck[iIdx].iBaseValue = iValues[r];
            iIdx++;
        }
    }
}

// ----- Fisher-Yates shuffle -----
void shuffleDeck(Card deck[])
{
    for (int i = iDECK_SIZE - 1; i > 0; i--)
    {
        int j = rand() % (i + 1);
        Card tmp = deck[i];
        deck[i]    = deck[j];
        deck[j]    = tmp;
    }
}

// ----- Take the next card off the top; advance the index -----
Card dealCard(Card deck[], int& iNextCard)
{
    Card c = deck[iNextCard];
    iNextCard++;                // & in signature makes this stick in main
    return c;
}

// ----- Score the hand; soft-ace handling -----
int calculateHandValue(const Card hand[], int iCount)
{
    int iTotal  = 0;
    int iAces   = 0;
    for (int i = 0; i < iCount; i++)
    {
```

```
        iTotal += hand[i].iBaseValue;
        if (hand[i].szRank == "A") iAces++;
    }
    // If we're over 21 and we have aces counted as 11, demote them to 1
    while (iTotal > 21 && iAces > 0)
    {
        iTotal -= 10;              // ace becomes 1 instead of 11 → -10
        iAces--;
    }
    return iTotal;
}

// ----- Print a hand, optionally hiding the first card (dealer's hole) -----
void displayHand(const Card hand[], int iCount,
                 const string& szWho, bool bHideFirst)
{
    cout << szWho << ":  ";
    for (int i = 0; i < iCount; i++)
    {
        if (i == 0 && bHideFirst)
        {
            cout << "[hidden]";
        }
        else
        {
            cout << hand[i].szRank << " of " << hand[i].szSuit;
        }
        if (i < iCount - 1) cout << ",  ";
    }
    if (!bHideFirst)
    {
        cout << "   (" << calculateHandValue(hand, iCount) << ")";
    }
    cout << endl;
}

// ----- Ask the player: hit or stand? -----
char promptHitOrStand()
{
    char cChoice;
    do
    {
        cout << "(H)it or (S)tand? ";
        cin >> cChoice;
        cChoice = tolower(cChoice);
    } while (cChoice != 'h' && cChoice != 's');
    return cChoice;
}
```

```
// ----- Player: keep hitting until stand or bust -----
void playerTurn(Card deck[], int& iNextCard,
                Card hand[], int& iCount)
{
    while (true)
    {
        int iScore = calculateHandValue(hand, iCount);
        if (iScore >= 21) return;    // stop if 21 or bust

        char cChoice = promptHitOrStand();
        if (cChoice == 's') return;

        hand[iCount++] = dealCard(deck, iNextCard);
        displayHand(hand, iCount, "You", false);
    }
}

// ----- Dealer: draw until 17 or higher -----
void dealerTurn(Card deck[], int& iNextCard,
                Card hand[], int& iCount)
{
    cout << "\n--- Dealer's turn ---\n";
    displayHand(hand, iCount, "Dealer", false);

    while (calculateHandValue(hand, iCount) < 17)
    {
        hand[iCount++] = dealCard(deck, iNextCard);
        displayHand(hand, iCount, "Dealer", false);
    }
}

// ----- Compare final scores, print the result -----
void announceWinner(int iPlayerScore, int iDealerScore)
{
    cout << "\nFinal: You " << iPlayerScore
         << ", Dealer " << iDealerScore << ".\n";

    if      (iDealerScore > 21)            cout << "Dealer busts. You win!\n";
    else if (iPlayerScore > iDealerScore) cout << "You win!\n";
    else if (iPlayerScore < iDealerScore) cout << "Dealer wins.\n";
    else                                  cout << "Push (tie).\n";
}
```

What to notice

- Every function does one thing. Their names tell you what. No comments needed on most of them.

- `main` reads top to bottom like the rules of Blackjack. "Build deck. Shuffle. Deal. Player goes. Dealer goes. Announce." That's the summary-not-novel pattern.
- `calculateHandValue` is `const`-reference-safe — it never changes the hand, just reads it. Called three times in three different contexts. That's reuse.
- `iNextCard` is passed by `&` because each `dealCard` call needs to advance it. Pass-by-value would silently eat your progress.
- The soft-ace handling is four lines. Any time the score goes over 21 and an ace was counted as 11, demote it to 1 by subtracting 10. Simple once you see it; painful to invent on the fly.

> **Try this as your weekend project.** Type it in from scratch, get it running, then start adding things: betting chips, double-down, an opponent that plays too. Every feature you add, you'll *feel* where to put it because the bones are clean. That feeling is the whole point.

Part 6 · Debugging Habits

Use a real debugger.

CodeLLDB is in the VSCodium extension list for a reason. Click the gutter next to a line to set a breakpoint, hit F5, and your program pauses there. You can inspect every variable, step one line at a time, watch values change. Most beginners debug by adding `cout << "here 1"` everywhere and deleting them later. That works, but a real debugger is ten times faster and never leaves stray prints in your code.

Read compiler errors top to bottom.

When g++ spits out 40 lines of red, the *first* error is usually the real bug. Everything after it is dominoes — the compiler got confused by the first problem and cascaded. Fix the top one, recompile, see what's left. Don't try to fix all 40 at once.

Rubber duck debugging.

When you're stuck, explain the code out loud, line by line, to an inanimate object — a rubber duck on your desk, a coffee mug, whatever. Sounds absurd. Works constantly. The act of forcing yourself to narrate every step surfaces the bug about half the time before you even finish explaining. I'm not joking. Every senior engineer I know does this.

When you're truly stuck, walk away for 15 minutes.

Get water, go outside, don't think about it. Your brain keeps working in the background. Nine times out of ten, the answer shows up while you're doing something else. Staring at broken code for two hours straight is how you burn out and start making it worse.

Before writing a function, write the steps as comments.

Before any code: `// 1. prompt for path`, `// 2. if "exit", bail`, `// 3. try to open`, `// 4. if fails, retry`. Then you fill in each step. The comments become the outline, and half your bugs disappear because you thought before typing.

When it breaks, narrow down what changed.

"It worked 10 minutes ago." Great — what have you touched in the last 10 minutes? Version control (git) helps here: `git diff` tells you exactly what you changed. Without git, you're squinting at code trying to remember.

Part 7 · Real-World Pointers

Class will teach you the language. It won't teach you the craft. Here's what I wish somebody had told me at your stage.

1. Scratch your own itch

The fastest way to actually learn C++ — or any language — is to build something you personally want. Not another "calculate the area of a triangle" exercise. Something that solves a small annoyance in *your* life. A few ideas:

- **Workout logger.** You type in the day's lifts, it saves them to a file, and next week it prints what you did last time so you know the number to beat.
- **Grade calculator.** You plug in your current grades and assignment weights, it tells you exactly what you need on the final to hit an A.
- **Gas mileage tracker.** Log miles and gallons at each fill-up, get your running MPG.
- **D&D dice roller.** `d20 + 4` and it rolls it. Tiny, fun, teaches you strings and random numbers.
- **Flashcard drill.** Reads a text file of question/answer pairs, quizzes you in random order, tracks what you got wrong.
- **Blackjack** (see Part 5) — or any small card game. Teaches structs, arrays, randomness, loops, functions, all at once.

> **Your weekend project:** Build the Blackjack game from Part 5, or a text-file todo list. Program loads `todos.txt`, lets you add items, mark them done, delete them, save the file. That's it. You'll touch structs, arrays or vectors, file I/O, functions, pass-by-reference, strings, and loops — every concept from this guide. Budget a Saturday. Don't look up a tutorial; build it from scratch. When you finish, you'll have written about 200 lines of C++ and you'll *own* this material.

2. Use VSCodium, not Visual Studio

Your school probably pushes Visual Studio or VS Code. Both are fine to learn on, but they're Microsoft products and they track you. When you leave school, you don't want your muscle memory tied to one vendor's buttons.

VSCodium is VS Code with the Microsoft branding and telemetry stripped out. Same interface, same extensions (almost all of them), zero corporate tracking. It's free, open source, cross-platform. Install it once and you'll never have to learn a different editor.

Get it at `vscodium.com`. Same advice for compilers: learn **g++** (GCC) or **clang**, not Microsoft's MSVC. They're standard everywhere — Linux, Mac, even Windows (via MinGW or WSL). Your skills travel.

Editor tips that'll actually save you time

Once you've got VSCodium installed, these are the tricks I'd wire up on day one. None of them are fancy — they're the stuff senior engineers use every hour of every day.

Extensions to install (all free):

- **clangd** — real-time error squiggles, smart autocomplete, "go to definition" (F12), "find all references" (Shift+F12). Way better than Microsoft's C/C++ extension. Install `clangd` the command-line tool too (`sudo apt install clangd` or `brew install llvm`).
- **CodeLLDB** — a real debugger. Click the gutter next to a line to set a breakpoint, hit F5, watch your variables change step by step. Ten times better than adding `cout` statements everywhere to figure out what went wrong.
- **Rainbow Brackets** — colors matching `{ }` pairs so nested loops and ifs don't melt your brain. Your current theater code has a lot of nesting; this would help you see structure instantly.
- **GitLens** — hover any line and it tells you who wrote it, when, and in what commit. Becomes magical once you're working on a team.
- **Code Spell Checker** — flags typos in your variable names and comments. Small thing, saves embarrassment.

Keyboard shortcuts worth burning into your hands:

- `Ctrl+Shift+P` — the command palette. When you don't remember a shortcut, start here. Type what you want.
- `F12` — jump to where a function is defined. `Ctrl+Click` does the same.
- `Shift+F12` — find everywhere a function is called from. Invaluable when refactoring.
- `Ctrl+Shift+F` — search your whole project. Not just the file you're in.
- `Ctrl+D` — select the next occurrence of the word your cursor's on. Hit it three times, you've got three cursors. Type once, rename them all. Cleaner than find-and-replace for local changes.
- `Alt+Up` / `Alt+Down` — move the current line up or down. Way faster than cut-and-paste.

- `Ctrl+/` — comment or uncomment the selected lines. Great for temporarily disabling blocks of code while you debug.
- `Ctrl+`` (backtick) — open the integrated terminal. Never leave the editor to compile.

Settings worth turning on:

- **Format on save** — set `"editor.formatOnSave": true` in settings. Install `clang-format` and your code gets neatly indented every time you hit Ctrl+S. No more arguing with yourself about spacing.
- **Auto save** — set it to "afterDelay". You won't lose work to a crash. (Still commit to git.)
- **Word wrap** — `"editor.wordWrap": "on"`. So long lines don't force horizontal scrolling.
- **Bracket pair colorization** — `"editor.bracketPairColorization.enabled": true`. Built-in now, no extension needed.

Two tricks that feel like cheating:

- **Snippets.** Type `for` and press Tab — you get a full `for` loop skeleton with cursor placeholders. Same for `if`, `while`, `struct`, `main`. Saves thousands of keystrokes over a semester.
- **Multi-cursor.** Hold `Alt` and click multiple spots. Every cursor types at once. Use it to add the same prefix to five variable names, or fix the same typo in a dozen places.

Per-project setup (when you're ready): create a `.vscode/` folder in your project with a `tasks.json` that tells VSCodium how to compile your code (so Ctrl+Shift+B just works), and a `launch.json` for debugging. Google "vscode c++ tasks.json" when you want to set it up — takes ten minutes the first time, saves you hours after.

> **The meta-tip:** Don't try to learn all of this at once. Install VSCodium and clangd today. Use it for a week. When you notice yourself doing something repetitive or annoying, *then* go find the shortcut or extension for it. Tool-learning is best done just-in-time, not up front.

3. Get comfortable with the terminal

Right now you probably hit a green "play" button in your IDE and your code runs. That's fine for starting out, but you're a black box to yourself. You don't know *what* is being invoked when you hit that button. Learn what's under the hood.

```
# compile a program
g++ -Wall -Wextra -std=c++17 main.cpp -o store

# run it
./store

# check you're in the right folder
pwd

# list files
ls

# move into a folder / move back up
cd src
cd ..
```

That's 90% of what you need to know, day one. Once you can compile and run from the terminal, the IDE stops being a crutch and starts being a convenience. Big difference.

While you're at it, learn **git**. Even for solo projects. `git init`, `git add`, `git commit -m "message"`. Your code won't just live in a folder that can be nuked by a bad keystroke — every commit is a save point you can come back to.

4. Read other people's code

Once you've built a thing or two of your own, go on GitHub and read small C++ projects. Not the giant stuff — look for projects under 2000 lines. You'll start to see patterns: how people name functions, how they split files, how they handle errors. You'll steal ideas. That's how every programmer gets better — by seeing moves you didn't know were legal.

5. Don't rush the fundamentals

What you're learning right now — variables, functions, pass-by-reference, structs — is *not* "beginner stuff you'll move past." It's the foundation of every single program ever written, in

every language. A senior engineer at Amazon is using the exact same concepts you're using today. The difference is fluency, not vocabulary.

So don't rush past this to get to the "real" stuff. This *is* the real stuff.

> That's all of it. Fix the two functions we went over, get the rest of the stubs doing something, and ping me when you want to show me something you made.
>
> Try the Blackjack thing this weekend. Seriously. You'll be shocked how much clicks.
>
> You'll be fine.
>
> — Mark

Appendix

Footnote: Hungarian notation

Those `i`, `d`, `sz`, `b` prefixes on every variable — that's Hungarian notation. The prefix tells you what *type* the variable holds: `i` = integer, `d` = double, `sz` = "zero-terminated string" (an old C term for a string that ends in a zero byte), `b` = bool, `c` = char, `arr` = array, `p` = pointer.

A Microsoft programmer named Charles Simonyi invented it in the 1970s. He was Hungarian, his coworkers joked his variable names looked like a foreign language, and the nickname stuck. It spread through Microsoft in the 80s and 90s — the Windows API is drenched in it, which is how it ended up in every intro C++ course taught by someone who learned on Windows. Your professor is part of that lineage.

Back then, code editors were dumb. No hover tooltips, no "go to definition," no type checking as you typed. If you saw a variable named `count` halfway down a 500-line function, you had no fast way to tell if it was an int, a long, or something else. The prefix told you instantly. It was a workaround for missing tooling.

Most modern codebases don't use it. Editors got smart — hover a variable in VSCodium with clangd and it tells you the exact type. Google's C++ style guide, LLVM, most open-source C++ — none of them use Hungarian. You'll almost never see `iCount` in professional code written after about 2005. The prefixes also lie when someone changes a type from `int` to `long` and forgets to rename every `i` to `l`, which happens constantly.

There's a twist though. Simonyi originally meant the prefix to describe the variable's *purpose*, not its type — `rowIndex` vs `colIndex`, not `int` vs `int`. This is called "Apps Hungarian" and it's genuinely useful — it catches bugs like accidentally assigning a row to a column. The version that caught on — prefixing the C++ type — is called "Systems Hungarian" and is what most people criticize. Joel Spolsky wrote a famous essay ("Making Wrong Code Look Wrong") defending the original idea and arguing Systems Hungarian ruined the reputation of a good concept.

> **Practical takeaway:** Use Hungarian in this class because your professor grades on it. After this class, follow whatever convention the codebase you're working in uses. If you're starting your own project, skip Systems Hungarian and name things by what they *mean*, not what they *are* — `customerCount` beats `iCount` every time.

Cheat sheet

`int, double, char, bool, string`	The basic types. int = whole number, double = decimal, string = text.
`const double TAX = 0.08;`	A value that can't change. Use for fixed constants.
`cout << x << endl;`	Print x and move to the next line.
`cin >> x;`	Read one word/number into x.
`getline(cin, szName);`	Read a whole line (including spaces).
`cin.ignore();`	Eat the stray newline left by `cin >>`.
`if / else if / else`	Branch based on conditions.
`== != < > && || !`	Compare and combine conditions.
`while, do-while, for`	Loops. Pick the one that fits.
`void name(int x)`	Pass by value — copy. Changes don't stick.
`void name(int& x)`	Pass by reference — original. Changes stick.
`void name(const Type& x)`	Read-only reference. Fast AND safe. Use for big things you won't modify.
`return value;`	Hand something back to the caller and exit the function.
`struct Name { ... };`	Define a bundle of fields. Semicolon at the end!
`obj.field`	Reach into a struct.

`fstream f;`
`f.open("x.txt", ios::in);`
`if (f.is_open()) { ... }`
`f.close();`
	Open a file for reading, check it, close it when done. Use `ios::out` to write, `ios::app` to append.

`rand() % N`	Random int from 0 to N−1. Call `srand(time(0))` once in main first.