Control flow

When a program is run, the CPU begins execution at the top of `main()`, executes some number of statements (in sequential order by default), and then the program terminates at the end of `main()`. The specific sequence of statements that the CPU executes is called the program's **execution path** (or **path**, for short) and for straight-line programs, the path is sequential, top to bottom and always follows the same path in every compilation.

Control flow statements (also called **flow control statements**), which are statements that allow the programmer to change the normal path of execution through the program.

Category	Meaning	Implemented in C++ by
Conditional statements	Causes a sequence of code to execute only if some condition is met.	if, else, switch
Jumps	Tells the CPU to start executing the statements at some other location.	goto, break, continue
Function calls	Jump to some other location and back.	function calls, return
Loops	Repeatedly execute some sequence of code zero or more times, until some condition is met.	while, do-while, for, ranged-for
Halts	Terminate the program.	std::exit(), std::abort()
Exceptions	A special kind of flow control structure designed for error handling.	try, throw, catch

If else

if – if - else

```
if (condition1) {
        // Code to execute if condition1 is true
}

if (condition2) {
        // Code to execute if condition2 is true, regardless of condition 1 running or not
}

else {
        //Code executes only if most recent if statement is false
}
```

if – else if - else

```
if (condition1) {
   // Code to execute if condition1 is true
} else if (condition2) {
   // Code to execute if condition2 is true and condition1 is false
} else {
   // Code to execute if neither condition1 nor condition2 is true
}
```

without usage of code blocks, **only one** statement after the if, else if, else will be executed according to condition, so in `if (condition) fun1(); fun2(0);` fun2 will always execute even if fun1 is on same line as if statement and fun2 next

always enclose statements, even if one, inside code blocks as if you were to comment out faulty code, the next statement that was supposed to run always , would become part of the conditional statement

Early Return

A return statement that is not the last statement in a function is called an **early return**. Such a statement will cause the function to return to the caller when the return statement is executed (before the function would otherwise return to the caller, hence, "early"). Any statement written after early return will **not** be executed and may as well be removed from the code.
Historically, early returns were frowned upon. However, in modern programming they are more accepted, particularly when they can be used to make a function simpler, or are used to abort a function early due to some error condition & they have a special case with if statements for conditioning the return statement.

```
// returns the absolute value of x
int abs(int x)
{
    if (x < 0)
        return -x; // early return (only when x < 0)

    return x;
}
```

Implicit blocks

If the programmer does not declare a block in the statement portion of an `if statement` or `else statement`, the compiler will implicitly declare one. Which means that if you define some variable in both the condition statements without using code blocks in a single statement, the definition will still go out of scope for later usage due to implicit assumed blocks added by compiler.

Null Statement

A **null statement** is an expression statement that consists of just a semicolon:

```
1 if (x > 10)
2     ; // this is a null statement
```

Be careful not to "terminate" your `if statement` with a semicolon, otherwise your conditional statement(s) will execute unconditionally (even if they are inside a block).

```
if (nuclearCodesActivated())
    ; // the semicolon acts as a null statement
blowUpTheWorld(); // and this line always gets executed!
```

constexpr if statement

```
constexpr double gravity{ 9.8 };

// reminder: low-precision floating point literals of the same type can be tested for equality
if (gravity == 9.8) // constant expression, always true
    std::cout << "Gravity is normal.\n";   // will always be executed
else
    std::cout << "We are not on Earth.\n"; // will never be executed

return 0;
```

Evaluating a constexpr conditional at runtime is wasteful (since the result will never vary). It is also wasteful to compile code into the executable that can never be executed. C++17 introduces the **constexpr if statement**, which requires the conditional to be a constant expression. The compiler will check the conditional statement and replace with the true or false statement for optimisation purposes.

```
if constexpr (gravity == 9.8)
```

Conditional Operator

A Statement of choice like

```
if (x > y)
    greater = x;
else
    greater = y;
```

can also be written as

```
greater = ((x > y) ? x : y);
```

This is particularly of use when we don't was to initialise identifiers before without error

```
constexpr bool inBigClassroom { false };
constexpr int classSize { inBigClassroom ? 30 : 20 };
```

has no if-else counterpart, as a attempt to do this

```
constexpr bool inBigClassroom { false };

if (inBigClassroom)
    constexpr int classSize { 30 };
else
    constexpr int classSize { 20 };
```

would return error because classSize is not defined, much like variables defined inside function die at the end of the function, variables defined inside a if-else chain, die at the end of the if or else statement and hence it can no longer be printed in later code tasks.

Conditional Operator also has very low precedence which needs to be kept in mind when using it.

Conditional Operator needs the 2nd and 3rd operand to be of the same type or convertible types like int and double and an error is unavoidable in case of conversion being impossible like unsigned and negative numbers *or* strings and numbers.

The conditional operator is most useful when doing one of the following:

- Initializing an object with one of two values.
- Assigning one of two values to an object.
- Passing one of two values to a function.
- Returning one of two values from a function.
- Printing one of two values.

Complicated expressions should generally avoid use of the conditional operator, as they tend to be error prone and hard to read.

Switch

Because testing a variable or expression for equality against a set of different values is common, C++ provides an alternative conditional statement called a **switch statement** that is specialized for this purpose.

We start a `switch statement` by using the `switch` keyword, followed by parentheses with the conditional expression that we would like to evaluate inside which could be a variable or expression granted that it will evaluate to a integral datatype (char short int long) or a enumerated datatype (covered later). Following the conditional expression, we declare a block. Inside the block, we use **labels** to define all of the values we want to test for equality.

The first kind of label is the **case label**, which is declared using the `case` keyword and followed by a constant expression. The constant expression must either match the type of the condition or must be convertible to that type.

```
void printDigitName(int x)
{
    switch (x) // x is evaluated to produce value 2
    {
    case 1:
        std::cout << "One";
        return;
    case 2: // which matches the case statement here
        std::cout << "Two"; // so execution starts here
        return; // and then we return to the caller
    case 3:
        std::cout << "Three";
        return;
    default:
        std::cout << "Unknown";
        return;
    }
}
```

here is no practical limit to the number of case labels you can have, but all case labels in a switch **must be unique.**

```
switch (x)
{
case 54:
case 54:  // error: already used value 54!
case '6': // error: '6' converts to integer value 54, which is already used
}
```

The second kind of label is the **default label** (often called the **default case**) and is executed when the expression does not match any case label. The default label is optional, and there can only be one default label per switch statement.

No matching case label and no default case

If the value of the conditional expression does not match any of the case labels, and no default case has been provided, then no cases inside the switch are executed and the whole switch codeblock is skipped in compilation.

return vs break

`return statements` exit the entire function block and any further code is not executed. A **break statement** (declared using the `break` keyword) tells the compiler that we are done executing statements within the switch, and that execution should continue with the statement after the end of the switch block.

```
#include <iostream>

void printDigitName(int x)
{
    switch (x) // x evaluates to 3
    {
    case 1:
        std::cout << "One";
        break;
    case 2:
        std::cout << "Two";
        break;
    case 3:
        std::cout << "Three"; // execution starts here
        break; // jump to the end of the switch block
    default:
        std::cout << "Unknown";
        break;
    }

    // execution continues here
    std::cout << " Ah-Ah-Ah!";
}

int main()
{
    printDigitName(3);
    std::cout << '\n';

    return 0;
}
```

Indentation

Labels are **not conventionally indented**, only their statement are indented one level

```
void printDigitName(int x)
{
    switch (x)
    {
        case 1: // indented from switch block
            std::cout << "One"; // indented from label (misleading)
            return;
```

while this is readable, it implies that the statement under each label is in nested scope which isn't the case.

```
void printDigitName(int x)
{
    switch (x)
    {
    case 1: // not indented from switch statement
        std::cout << "One";
        return;
```

This is the **preferred method**

Fallthrough

without a break/return statement, the switch statement will continuously run until end of codeblock

```
int main()
{
    switch (2)
    {
    case 1: // Does not match
        std::cout << 1 << '\n'; // Skipped
    case 2: // Match!
        std::cout << 2 << '\n'; // Execution begins here
    case 3:
        std::cout << 3 << '\n'; // This is also executed
    case 4:
        std::cout << 4 << '\n'; // This is also executed
    default:
        std::cout << 5 << '\n'; // This is also executed
    }

    return 0;
}
```

Since fallthrough is rarely desired or intentional, many compilers and code analysis tools will flag fallthrough as a warning which can be avoided by [[fallthrough]] attribute which modifies a `null statement` to indicate that fallthrough is intentional (and no warnings should be triggered).

```
switch (2)
{
case 1:
    std::cout << 1 << '\n';
    break;
case 2:
    std::cout << 2 << '\n'; // Execution begins here
    [[fallthrough]]; // intentional fallthrough -- note the semicolon to indicate the null statement
case 3:
    std::cout << 3 << '\n'; // This is also executed
    break;
}
```

Sequential case labels

```
bool isVowel(char c)
{
    return (c=='a' || c=='e' || c=='i' || c=='o' || c=='u' ||
        c=='A' || c=='E' || c=='I' || c=='O' || c=='U');
}
```

This program returns true for vowels and false for consonants. The problem is that, c is evaluated many times by the compiler similar to multiple evaluations in if-else statements.

```
bool isVowel(char c)
{
    switch (c)
    {
    case 'a': // if c is 'a'
    case 'e': // or if c is 'e'
    case 'i': // or if c is 'i'
    case 'o': // or if c is 'o'
    case 'u': // or if c is 'u'
    case 'A': // or if c is 'A'
    case 'E': // or if c is 'E'
    case 'I': // or if c is 'I'
    case 'O': // or if c is 'O'
    case 'U': // or if c is 'U'
        return true;
    default:
        return false;
    }
}
```

we can “stack” case labels to make all of those case labels share the same set of statements afterward. This is not considered fallthrough behavior, so use of comments or `[[fallthrough]]` is not needed here.

Implicit block not created

with switch statements, the statements after labels are all scoped to the main switch block. This means that declaration of variables can be done with a larger scope of use in switch blocks. However, initialization of variables *does* require the definition to execute at runtime making it illegal to initialise variables **except the last case**(because in other cases, the initialisation could be skipped if the switch expression does not hold, making further usage of variable in other cases undefined)**.** Initialization is also disallowed before the first case, as those statements will never be executed, as there is no way for the switch to reach them.

```
switch (1)
{
    int a; // okay: definition is allowed before the case labels
    int b{ 5 }; // illegal: initialization is not allowed before the case labels

case 1:
    int y; // okay but bad practice: definition is allowed within a case
    y = 4; // okay: assignment is allowed
    break;

case 2:
    int z{ 4 }; // illegal: initialization is not allowed if subsequent cases exist
    y = 5; // okay: y was declared above, so we can use it here too
    break;

case 3:
    break;
}
```

If a case needs to define and/or initialize a new variable, the best practice is to do so inside an explicit block underneath the case statement:

```
switch (1)
{
case 1:
{ // note addition of explicit block here
    int x{ 4 }; // okay, variables can be initialized inside a block inside a case
    std::cout << x;
    break;
}

default:
    std::cout << "default case\n";
    break;
}
```

Goto

In C++, unconditional jumps are implemented via a **goto statement**, and the spot to jump to is identified through use of a **statement label**. Just like with switch case labels, statement labels are conventionally not indented.

```
    double x{};
tryAgain: // this is a statement label
    std::cout << "Enter a non-negative number: ";
    std::cin >> x;

    if (x < 0.0)
        goto tryAgain; // this is the goto statement
```

In this way, we can continually ask the user for input until he or she enters something valid.

Statement labels utilize a third kind of scope(other than local and global) : **function scope**, which means the label is visible throughout the function even before its point of declaration. The goto statement and its corresponding `statement label` must appear in the same function.

goto statements can jump forward or backward

```
void printCats(bool skip)
{
    if (skip)
        goto end; // jump forward; statement label 'end' is visible here due to it having function scope

    std::cout << "cats\n";
end:
    ; // statement labels must be associated with a statement
}
```

you can't end a function with statement label with a colon and hence the null statement added

you can't jump forward over the initialization of any variable that is still in scope at the location being jumped to. you can jump backwards over a variable initialization, and the variable will be re-initialized when the initialization is executed

Use of `goto` is shunned in C++ (and other modern high level languages as well). The primary problem with goto is that it allows a programmer to jump around the code arbitrarily. This creates what is not-so-affectionately known as spaghetti code. **Spaghetti code** is code that has a path of execution that resembles a bowl of spaghetti (all tangled and twisted), making it extremely difficult to follow the logic of such code. Almost any code written using a goto statement **can be more clearly written using other constructs in C++,** such as if-statements and loops. One notable

exception is when you need to exit a nested loop but not the entire function -- in such a case, a goto to just beyond the loops is probably the cleanest solution.

Introduction to loops

And now the real fun begins -- in the next set of lessons, we'll cover loops. Loops are control flow constructs that allow a piece of code to execute repeatedly until some condition is met. Loops add a significant amount of flexibility into your programming toolkit, allowing you to do many things that would otherwise be difficult.

While statements

The **while statement** (also called a **while loop**) is the simplest of the three loop types that C++ provides, and it has a definition very similar to that of an if-statement:
A `while statement` is declared using the **while** keyword. When a while-statement is executed, the expression *condition* is evaluated. If the condition evaluates to `true`, the associated *statement* executes. once the statement has finished executing, control returns to the top of the while-statement and the process is repeated. This means a while-statement will keep looping as long as the condition continues to evaluate to `true`.

```
#include <iostream>

int main()
{
    int count{ 1 };
    while (count <= 10)
    {
        std::cout << count << ' ';
        ++count;
    }

    std::cout << "done!\n";

    return 0;
}
```

It is important to keep initial statement inside outer blocks to prevent errors and further name clash

Infinite loops

if the expression always evaluates to `true`, the while loop will execute forever. This is called an **infinite loop**. We can also use bool (true) to create intentional infinite loops.

```
while (true)
{
  // this loop will execute forever
}
```

The only way to exit an infinite loop is through a return-statement, a break-statement, an exit-statement, a goto-statement, an exception being thrown, or the user killing the program.

```
#include <iostream>

int main()
{

    while (true) // infinite loop
    {
        std::cout << "Loop again (y/n)? ";
        char c{};
        std::cin >> c;

        if (c == 'n')
            return 0;
    }

    return 0;
}
```

Loop variables

A **loop variable** is a variable that is used to control how many times a loop executes. Loop variables are often given simple names, with `i`, `j`, and `k` being the most common. However, if you want to know where in your program a loop variable is used, and you use the search function on `i`, `j`, or `k`, the search function will return half of the lines in your program! For this reason, some developers prefer loop variable names like `iii`, `jjj`, or `kkk`. An even better idea is to use "real" variable names, such as `count`, `index`, or a name that gives more detail about what you're counting

Integral loop variables should be signed

```
#include <iostream>

int main()
{
    unsigned int count{ 10 }; // note: unsigned

    // count from 10 down to 0
    while (count >= 0)
    {
        if (count == 0)
        {
            std::cout << "blastoff!";
        }
        else
        {
            std::cout << count << ' ';
        }
        --count;
    }

    std::cout << '\n';

    return 0;
}
```

It turns out, this program is an infinite loop. It starts out by printing `10 9 8 7 6 5 4 3 2 1 blastoff!` as desired, but then loop variable `count` overflows, and starts counting down from `4294967295` (assuming 32-bit integers). Why? Because the loop condition `count >= 0` will never be false! When count is `0`, `0 >= 0` is true. Then `--count` is executed, and count wraps around back to `4294967295`. And since `4294967295 >= 0` is `true`, the program continues. Because `count` is unsigned, it can never be negative, and because it can never be negative, the loop won't terminate.

Doing something every N iterations

Often, we want to do something every 2nd, 3rd, or 4th iteration, such as print a newline. This can easily be done by using the remainder operator on our counter:

```
int main()
{
    int count{ 1 };
    while (count <= 50)
    {
        // print the number (pad numbers under 10 with a leading 0 for formatting purposes)
        if (count < 10)
        {
            std::cout << '0';
        }

        std::cout << count << ' ';

        // if the loop variable is divisible by 10, print a newline
        if (count % 10 == 0)
        {
            std::cout << '\n';
        }

        // increment the loop counter
        ++count;
    }

    return 0;
}
```

This program produces the result:

```
01 02 03 04 05 06 07 08 09 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

Nested loops

It is also possible to nest loops inside of other loops. The nested loop (which we're calling the inner loop) and the outer loop each have their own counters. Note that the loop expression for the inner loop makes use of the outer loop's counter as well!

```
// outer loops between 1 and 5
int outer{ 1 };
while (outer <= 5)
{
    // For each iteration of the outer loop, the code in the body of the loop executes once

    // inner loops between 1 and outer
    int inner{ 1 };
    while (inner <= outer)
    {
        std::cout << inner << ' ';
        ++inner;
    }

    // print a newline at the end of each row
    std::cout << '\n';
    ++outer;
}
```

This program prints:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

or each iteration of the outer loop, the body of the outer loop will execute once. Because the outer loop body contains an inner loop, the inner loop is executed for each iteration of the outer loop.

First, we have an outer loop (with loop variable `outer`) that will loop 5 times (with `outer` having values `1`, `2`, `3`, `4`, and `5` successively).

On the first iteration of the outer loop, `outer` has value `1`, and then the outer loop body executes. Inside the body of the outer loop, we have another loop with loop variable `inner`. The inner loop iterates from `1` to `outer` (which has value `1`), so this inner loop will execute once, printing the value `1`. Then we print a newline, and increment `outer` to `2`.

On the second iteration of the outer loop, `outer` has value `2`, and then the outer loop body executes. Inside the body of the outer loop, `inner` iterates from `1` to `outer` (which now has value `2`), so this inner loop will execute twice, printing the values `1` and `2`. Then we print a newline, and increment `outer` to `3`.

This process continues, with the inner loop printing `1 2 3`, `1 2 3 4`, and `1 2 3 4 5` on successive passes. Eventually, `outer` is incremented to `6`, and because the outer loop condition (`outer <= 5`) is then false, the outer loop is finished. Then the program ends.

Do while statements

A **do while statement** is a looping construct that works just like a while loop, except the statement always executes at least once.

```
int main()
{
    // selection must be declared outside of the do-while-loop, so we can use it later
    int selection{};

    do
    {
        std::cout << "Please make a selection: \n";
        std::cout << "1) Addition\n";
        std::cout << "2) Subtraction\n";
        std::cout << "3) Multiplication\n";
        std::cout << "4) Division\n";
        std::cin >> selection;
    }
    while (selection < 1 || selection > 4);

    // do something with selection here
    // such as a switch statement

    std::cout << "You selected option #" << selection << '\n';

    return 0;
}
```

one important thing here is that, the selection variable must be created outside the do code block as it needs to be in scope for the expression being checked for true in while statement

In practice, do-while loops aren't commonly used. Having the condition at the bottom of the loop obscures the loop condition, which can lead to errors. Many developers

recommend avoiding do-while loops altogether as a result. We'll take a softer stance and advocate for preferring while loops over do-while when given an equal choice.

For statements

By far, the most utilized loop statement in C++ is the for-statement. The **for statement** (also called a **for loop**) is preferred when we have an obvious loop variable because it lets us easily and concisely define, initialize, test, and change the value of loop variables.

If a **while loop** is

```
{ // note the block here
    init-statement; // used to define variables used in the loop
    while (condition)
    {
        statement;
        end-expression; // used to modify the loop variable prior to reassessment of the condition
    }
} // variables defined inside the loop go out of scope here
```

Then a **for loop** can be written as

```
for (init-statement; condition; end-expression)
    statement;
```

Evaluation of for-statements

the initial statement is executed (only once) and is typically used for variable definition and initialisation. They have **loop scope** which is basically block scope and ends with the final loop statement.

The condition is evaluated for each loop run or iteration and the loop is only executed on the basis of the condition being true. Finally, after the statement is executed, the end-expression is evaluated. Typically, this expression is used to increment or decrement the loop variables defined in the init-statement.

```
#include <cstdint> // for fixed-width integers

// returns the value base ^ exponent -- watch out for overflow!
std::int64_t pow(int base, int exponent)
{
    std::int64_t total{ 1 };

    for (int i{ 0 }; i < exponent; ++i)
        total *= base;

    return total;
}
```

Off-by-one errors

One of the biggest problems that new programmers have with for-loops (and other loops that utilize counters) are off-by-one errors. **Off-by-one errors** occur when the loop iterates one too many or one too few times to produce the desired result.

```cpp
#include <iostream>

int main()
{
    // oops, we used operator< instead of operator<=
    for (int i{ 1 }; i < 5; ++i)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

This program is supposed to print `1 2 3 4 5`, but it only prints `1 2 3 4` because we used the wrong relational operator.

Although the most common cause for these errors is using the wrong relational operator, they can sometimes occur by using pre-increment or pre-decrement instead of post-increment or post-decrement, or vice-versa.

Omitting For loop Components

It is possible to write *for loops* that omit any or all of the statements or expressions like init statement or end expression. We add these components manually as u can see in example below

```cpp
int main()
{
    int i{ 0 };
    for ( ; i < 10; ) // no init-statement or end-expression
    {
        std::cout << i << ' ';
        ++i;
    }

    std::cout << '\n';

    return 0;
}
```

Infinite Forloop

Although you do not see it very often, it is worth noting that

```cpp
for (;;)
    statement;
```

Is a infinite loop, even if being a little unexpected as you'd probably expect an omitted condition-expression to be treated as `false`. However, the C++ standard explicitly (and inconsistently) defines that an omitted condition-expression in a for-loop should be treated as `true`. This is however recommended to be avoided and while(true) should be preffered

```cpp
while (true)
    statement;
```

For-loops with multiple initial statements

```cpp
for (int x{ 0 }, y{ 9 }; x < 10; ++x, --y)
    std::cout << x << ' ' << y << '\n';
```

This is about the only place in C++ where defining multiple variables in the same statement, and use of the comma operator is considered an acceptable practice.

Nested for-loops

For each iteration of the outer loop, the inner loop runs in its entirety.

```
#include <iostream>

int main()
{
    for (char c{ 'a' }; c <= 'e'; ++c) // outer loop on letters
    {
        std::cout << c; // print our letter first

        for (int i{ 0 }; i < 3; ++i) // inner loop on all numbers
            std::cout << i;

        std::cout << '\n';
    }

    return 0;
}
```

```
a012
b012
c012
d012
e012
```

Breaking a loop

In the context of a loop, a `break` statement can be used to end the loop early. Execution continues with the next statement after the end of the loop.

```
int sum{ 0 };

// Allow the user to enter up to 10 numbers
for (int count{ 0 }; count < 10; ++count)
{
    std::cout << "Enter a number to add, or 0 to exit: ";
    int num{};
    std::cin >> num;

    // exit loop if user enters 0
    if (num == 0)
        break; // exit the loop now

    // otherwise add number to our sum
    sum += num;
}

// execution will continue here after the break
std::cout << "The sum of all the numbers you entered is: " << sum << '\n';
```

It can also be used to exit intentional infinite loops

Continue vs Break

The **continue statement** provides a convenient way to end the current iteration of a loop without terminating the entire loop.

```cpp
#include <iostream>

int main()
{
    for (int count{ 0 }; count < 10; ++count)
    {
        // if the number is divisible by 4, skip this iteration
        if ((count % 4) == 0)
            continue; // go to next iteration

        // If the number is not divisible by 4, keep going
        std::cout << count << '\n';

        // The continue statement jumps to here
    }

    return 0;
}
```

This loop will not count 4 or 8 and skip over them and continue with the loop iterations whereas a break statement would end the loop completelty and return statement would end the function. The continue statement jumps to the next iteration of loop and hence if the end expression like incrementing by one is in the loop body (like its present in while loops) that statement won't be executed and the loop will go to infinity!

Many textbooks caution readers not to use `break` and `continue` in loops, both because it causes the execution flow to jump around, and because it can make the flow of logic harder to follow. For example, a `break` in the middle of a complicated piece of logic could either be missed, or it may not be obvious under what conditions it should be triggered.

However, used judiciously, `break` and `continue` can help make loops more readable by keeping the number of nested blocks down and reducing the need for complicated looping logic.

The std::exit() function

`std::exit()` is a function that causes the program to terminate normally. **Normal termination** means the program has exited in an expected way. Note that the term `normal termination` does not imply anything about whether the program was successful (that's what the `status code` is for). For example, let's say you were writing a program where you expected the user to type in a filename to process. If the user typed in an invalid filename, your program would probably return a non-zero `status code` to indicate the failure state, but it would still have a `normal termination`.

`std::exit()` performs a number of cleanup functions. First, objects with static storage duration are destroyed. Then some other miscellaneous file cleanup is done if any files were used. Finally, control is returned back to the OS, with the argument passed to `std::exit()` used as the `status code`.
Although `std::exit()` is called implicitly after function `main()` returns, `std::exit()` can also be called explicitly to halt the program before it would normally terminate. When `std::exit()` is called this way, you will need to include the `cstdlib` header.

```cpp
#include <cstdlib> // for std::exit()
#include <iostream>

void cleanup()
{
    // code here to do any kind of cleanup required
    std::cout << "cleanup!\n";
}

int main()
{
    std::cout << 1 << '\n';
    cleanup();

    std::exit(0); // terminate and return status code 0 to operating system

    // The following statements never execute
    std::cout << 2 << '\n';

    return 0;
}
```

```
1
cleanup!
```

Although in the program above we call `std::exit()` from function `main()`, `std::exit()` can be called from any function to terminate the program at that point.
The `std::exit()` function does not clean up local variables in the current function or up the call stack.

std::atexit

C++ offers the `std::atexit()` function, which allows you to specify a function that will automatically be called on program termination via `std::exit()`. std::atexit(cleanup) will print cleanup when we execute std::exit(0) saving us the hassle of typing it everytime

std::abort and std::terminate

The `std::abort()` function causes your program to terminate abnormally. **Abnormal termination** means the program had some kind of unusual runtime error and the program couldn't continue to run. We can either call this or the system may call it implicitly based on certain predicamenets like when we divide by zero. The `std::terminate()` function is typically used in conjunction with `exceptions` (we'll cover exceptions in a later chapter). Although `std::terminate` can be called explicitly, it is more often called implicitly when an exception isn't handled (and in a few other exception-related cases). By default, `std::terminate()` calls `std::abort()`.

Random Number Generation

The ability to generate random numbers can be useful in certain kinds of programs, particularly in games, statistical modelling programs, and cryptographic applications that need to encrypt and decrypt things. Take games for example -- without random events, monsters would always attack you the same way, you'd always find the same treasure, the dungeon layout would never change, etc... and that would not make for a very good game. In real life, we use randomization by flipping coins, rolling dices or shuffling a deck of cards all of which truly aren't random but the factors controlling it like gravity, wind etc etc are far too complex to predict or calculate for. Computers aren't designed to take advantage of such physical variables. Modern computers live in a controlled electrical world where everything is binary (0 or 1) and there is no in-between. By their very nature, computers are designed to produce results that are as predictable as possible. Consequently, computers are generally incapable of generating truly random numbers (at least through software). Instead, modern programs typically *simulate* randomness using an algorithm.

Pseudo-random number generators (PRNGs)

To simulate randomness, programs typically use a pseudo-random number generator. A **pseudo-random number generator (PRNG)** is an algorithm that generates a sequence of numbers whose properties simulate a sequence of random numbers.

It's easy to write a basic PRNG algorithm.

```
unsigned int LCG16() // our PRNG
{
    static unsigned int s_state{ 0 }; // only initialized the first time this function is called

    // Generate the next number

    // We modify the state using large constants and intentional overflow to make it hard
    // for someone to casually determine what the next number in the sequence will be.

    s_state = 8253729 * s_state + 2396403; // first we modify the state
    return s_state % 32768; // then we use the new state to generate the next number in the sequence
}
```

As it turns out, this particular algorithm isn't very good as a random number generator (note how each result alternates between even and odd -- that's not very random!). But most PRNGs work similarly to `LCG16()` -- they just typically use more state variables and more complex mathematical operations in order to generate better quality results.

The values are deterministic and produce same values over different runs for same initial states (**seeds**). Unfortunately, we can't use a PRNG to generate a random seed, because we need a randomized seed to generate random numbers.

PRNG with 128 bits of state can theoretically generate up to 2^128 (340,282,366,920,938,463,463,374,607,431,768,211,456) unique output sequences. That's a lot! However, which output sequence is *actually* generated depends on the initial state of the PRNG, which is determined by the seed. If a PRNG is not provided with enough bits of quality seed data, we say that it is **underseeded**. An underseeded PRNG may begin to produce randomized results whose quality is compromised in some way.
Developers who aren't familiar with proper seeding practices will often try to initialize a PRNG using a single 32-bit or 64-bit value (unfortunately, the design of C++'s standard Random library inadvertently encourages this). This will generally result in a significantly underseeded PRNG.

Seeding a PRNG with 64 bytes of quality seed data (less if the PRNGs state is smaller) is typically good enough to facilitate the generation of 8-byte random values for non-sensitive uses (e.g. not statistical simulations or cryptography).

Mersenne Twister

To access any of the randomization capabilities in C++, we include the `<random>` header of the standard library.

- `mt19937` is a Mersenne Twister that generates 32-bit unsigned integers
- `mt19937_64` is a Mersenne Twister that generates 64-bit unsigned integers

```cpp
#include <iostream>
#include <random> // for std::mt19937

int main()
{
    std::mt19937 mt{}; // Instantiate a 32-bit Mersenne Twister

    // Print a bunch of random numbers
    for (int count{ 1 }; count <= 40; ++count)
    {
        std::cout << mt() << '\t'; // generate a random number

        // If we've printed 5 numbers, start a new row
        if (count % 5 == 0)
            std::cout << '\n';
    }

    return 0;
}
```

```
3499211612      581869302       3890346734      3586334585      545404204
4161255391      3922919429      949333985       2715962298      1323567403
418932835       2350294565      1196140740      809094426       2348838239
4264392720      4112460519      4279768804      4144164697      4156218106
676943009       3117454609      4168664243      4213834039      4111000746
471852626       2084672536      3427838553      3437178460      1275731771
609397212       20544909        1811450929      483031418       3933054126
2747762695      3402504553      3772830893      4120988587      2163214728
```

Rolling a dice using Mersenne Twister

A 32-bit PRNG will generate random numbers between 0 and 4,294,967,295, but we do not always want numbers in that range. Unfortunately, PRNGs can't do this. They can only generate numbers that use the full range. While we could write a function to do this ourselves, doing so in a way that produces non-biased results is non-trivial. A **random number distribution** converts the output of a PRNG into some other distribution of numbers. a **uniform distribution** is a random number distribution that produces outputs between two numbers X and Y (inclusive) with equal probability.

```cpp
std::mt19937 mt{};

// Create a reusable random number generator that generates uniform numbers between 1 and 6
std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use std::uniform_int_distribution<> die6{ 1, 6 };

// Print a bunch of random numbers
for (int count{ 1 }; count <= 40; ++count)
{
    std::cout << die6(mt) << '\t'; // generate a roll of the die here

    // If we've printed 10 numbers, start a new row
    if (count % 10 == 0)
        std::cout << '\n';
}
```

```
3       1       3       6       5       2       6       6       1       2
2       6       1       1       6       1       4       5       2       5
6       2       6       2       1       3       5       4       5       6
1       4       2       3       1       2       2       6       2       1
```

However the values will repeat eventually Because we are value initializing our Mersenne Twister, it is being initialized with the same seed every time the program is run. And because the seed is the same, the random numbers being generated are also the same.
It turns out, we really don't need our seed to be a random number -- we just need to pick something that changes each time the program is run. Then we can use our PRNG to generate a unique sequence of pseudo-random numbers from that seed.

Seeding with the system clock

What's one thing that's different every time you run your program? Unless you manage to run your program twice at exactly the same moment in time, the answer is that the current time is different. Therefore, if we use the current time as our seed value, then our program will produce a different set of random numbers each time it is run. C and C++ have a long history of PRNGs being seeded using the current time (using the `std::time()` function), so you will probably see this in a lot of existing code.

```
#include <iostream>
#include <random> // for std::mt19937
#include <chrono> // for std::chrono

int main()
{
    // Seed our Mersenne Twister using steady_clock
    std::mt19937 mt{ static_cast<std::mt19937::result_type>(
        std::chrono::steady_clock::now().time_since_epoch().count()
        ) };

    // Create a reusable random number generator that generates uniform numbers between 1 and 6
    std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use std::uniform_int_distribution<> die6{ 1, 6 };

    // Print a bunch of random numbers
    for (int count{ 1 }; count <= 40; ++count)
    {
        std::cout << die6(mt) << '\t'; // generate a roll of the die here

        // If we've printed 10 numbers, start a new row
        if (count % 10 == 0)
            std::cout << '\n';
    }

    return 0;
}
```

Seeding with the random device

The random library contains a type called `std::random_device` that is an implementation-defined PRNG. Normally we avoid implementation-defined capabilities because they have no guarantees about quality or portability, but this is one of the exception cases.

```
#include <iostream>
#include <random> // for std::mt19937 and std::random_device

int main()
{
    std::mt19937 mt{ std::random_device{}() };

    // Create a reusable random number generator that generates uniform numbers between 1 and 6
    std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use std::uniform_int_distribution<> die6{ 1, 6 };

    // Print a bunch of random numbers
    for (int count{ 1 }; count <= 40; ++count)
    {
        std::cout << die6(mt) << '\t'; // generate a roll of the die here

        // If we've printed 10 numbers, start a new row
        if (count % 10 == 0)
            std::cout << '\n';
    }

    return 0;
}
```

Warming up a PRNG

When a PRNG is given a poor quality seed (or underseeded), the initial results of the PRNG may not be high quality. For this reason, some PRNGs benefit from being “warmed up”, which is a technique where the first N results generated from the PRNG are discarded. This allows the internal state of the PRNG to be mixed up such that the subsequent results should be of higher quality. Typically a few hundred to a few thousand initial results are discarded. The longer the period of the PRNG, the more initial results should be discarded.

Global Random Number Generator

```cpp
#ifndef RANDOM_MT_H
#define RANDOM_MT_H

#include <chrono>
#include <random>

// This header-only Random namespace implements a self-seeding Mersenne Twister.
// Requires C++17 or newer.
// It can be #included into as many code files as needed (The inline keyword avoids ODR violations)
// Freely redistributable, courtesy of learncpp.com (https://www.learncpp.com/cpp-tutorial/global-random-numbers-random-h/)
namespace Random
{
    // Returns a seeded Mersenne Twister
    // Note: we'd prefer to return a std::seed_seq (to initialize a std::mt19937), but std::seed can't be copied, so it can't be returned by value.
    // Instead, we'll create a std::mt19937, seed it, and then return the std::mt19937 (which can be copied).
    inline std::mt19937 generate()
    {
        std::random_device rd{};

        // Create seed_seq with clock and 7 random numbers from std::random_device
        std::seed_seq ss{
            static_cast<std::seed_seq::result_type>(std::chrono::steady_clock::now().time_since_epoch().count()),
                rd(), rd(), rd(), rd(), rd(), rd(), rd() };

        return std::mt19937{ ss };
    }

    // Here's our global std::mt19937 object.
    // The inline keyword means we only have one global instance for our whole program.
    inline std::mt19937 mt{ generate() }; // generates a seeded std::mt19937 and copies it into our global object

    // Generate a random int between [min, max] (inclusive)
    inline int get(int min, int max)
    {
        return std::uniform_int_distribution{min, max}(mt);
    }

    // The following function templates can be used to generate random numbers
    // when min and/or max are not type int
    // See https://www.learncpp.com/cpp-tutorial/function-template-instantiation/

     // You can ignore these if you don't understand them

     // Generate a random value between [min, max] (inclusive)
     // * min and max have same type
     // * Return value has same type as min and max
     // * Supported types:
     // *    short, int, long, long long
     // *    unsigned short, unsigned int, unsigned long, or unsigned long long
     // Sample call: Random::get(1L, 6L);             // returns long
     // Sample call: Random::get(1u, 6u);             // returns unsigned int
     template <typename T>
     T get(T min, T max)
     {
         return std::uniform_int_distribution<T>{min, max}(mt);
     }

     // Generate a random value between [min, max] (inclusive)
     // * min and max can have different types
     // * Must explicitly specify return type as template type argument
     // * min and max will be converted to the return type
     // Sample call: Random::get<std::size_t>(0, 6);  // returns std::size_t
     // Sample call: Random::get<std::size_t>(0, 6u); // returns std::size_t
     // Sample call: Random::get<std::int>(0, 6u);    // returns int
     template <typename R, typename S, typename T>
     R get(S min, T max)
     {
         return get<R>(static_cast<R>(min), static_cast<R>(max));
     }
}

#endif
```

function overloading

Function overloading allows us to create multiple functions with the same name, so long as each identically named function has different parameter types (or the functions can be otherwise differentiated). Each function sharing a name (in the same scope) is called an **overloaded function** (sometimes called an **overload** for short).

```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

double add(double x, double y)
{
    return x + y;
}

int main()
{
    std::cout << add(1, 2); // calls add(int, int)
    std::cout << '\n';
    std::cout << add(1.2, 3.4); // calls add(double, double)

    return 0;
}
```

overload differentiation

Overloaded functions that are not properly differentiated will cause the compiler to issue a compile error. **difference in number of parameters & datatype of parameters** (excluding typedef/alias) are key factors (also called **type signatures**) in differentiating functions .

```
int add(int x, int y)
{
    return x + y;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

```
int add(int x, int y); // integer version
double add(double x, double y); // floating point version
double add(int x, double y); // mixed version
double add(double x, int y); // mixed version
```

Return datatype is ignored for overload differentiation, hence the below code will be an error

```
int getRandomValue();
double getRandomValue();
```

ambigous matches and argument matching sequence

Step 1) The compiler tries to find an exact match. Either compiler finds an exact match and executes the function. If there is no exact match, then some trivial conversions will be made.

- **lvalue to rvalue** conversions
- qualification conversions (e.g. **non-const to const**)

```
void foo(const int)
{
}

void foo(const double&)
{
}

int main()
{
    int x { 1 };
    foo(x); // x trivially converted from int to const int

    double d { 2.3 };
    foo(d); // d trivially converted from double to const double&
```

  These matches are also considered “exact matches” and if there would be 2 matches, one with a non const and other with a const argument, it will be a ambiguous match error as the functions won’t be differentiated

- **non-reference to reference** conversions

Step 2) If no exact match is found, the compiler tries to find a match by applying numeric promotion to the argument which is the widening of narrow datatypes like char/bool to int and float to double.

```
void foo(int)
{
}

void foo(double)
{
}

int main()
{
    foo('a');  // promoted to match foo(int)
    foo(true); // promoted to match foo(int)
    foo(4.5f); // promoted to match foo(double)
```

Step 3) If no match is found via numeric promotion, the compiler tries to find a match by applying numeric conversions. **long can be converted to both int or double** and

hence ambiguous error is possible if both functions exist. similarly float and unsigned int have equal hierarchy to convert to int or double.

```
#include <string> // for std::string

void foo(double)
{
}

void foo(std::string)
{
}

int main()
{
    foo('a'); // 'a' converted to match foo(double)

    return 0;
}
```

Step 4) If no match is found via numeric conversion, the compiler tries to find a match through any user-defined conversions.

```
// We haven't covered classes yet, so don't worry if this doesn't make sense
class X // this defines a new type called X
{
public:
    operator int() { return 0; } // Here's a user-defined conversion from X to int
};

void foo(int)
{
}

void foo(double)
{
}

int main()
{
    X x; // Here, we're creating an object of type X (named x)
    foo(x); // x is converted to type int using the user-defined conversion from X to int

    return 0;
}
```

Step 5) If no match is found via user-defined conversion, the compiler will look for a matching function that uses ellipsis. (covered later)

Step 6) If no matches have been found by this point, the compiler gives up and will issue a compile error about not being able to find a matching function.

Multiple Matching Functions

If there are multiple arguments, the compiler applies the matching rules to each argument to find the best choice for function. The chosen function is the one with all arguments being matched just as well as all the other choices, with at least one argument matching better. In other words, it must provide the best match at least by one extra parameter margin.

Deleting a function call from a specific argument

In cases where we have a function call from a datatype argument that we explicitly do not want to be callable especially when the datatype doesn't match with the result we want, we can deny it to be able to be called.

```
#include <iostream>

void printInt(int x)
{
    std::cout << x << '\n';
}

void printInt(char) = delete; // calls to this function will halt compilation
void printInt(bool) = delete; // calls to this function will halt compilation

int main()
{
    printInt(97);   // okay

    printInt('a');  // compile error: function deleted
    printInt(true); // compile error: function deleted

    printInt(5.0);  // compile error: ambiguous match

    return 0;
}
```

`printInt(5.0)` is an interesting case, after using the delete specifier to forbid char and bool arguments to be called, they become candidates for function overload. Even if there is only one function with one parameter type of int, `printInt(5.0)` will give a ambiguous match as there is no best match available with int,char and bool to now choose from.

Default Arguments

A **default argument** is a default value provided for a function parameter.

```
void print(int x, int y=10) // 10 is the default argument
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

If the caller provides an argument, the value of the argument in the function call is used. If the caller does not provide an argument, the value of the default argument is used.

```
#include <iostream>

void print(int x=10, int y=20, int z=30)
{
    std::cout << "Values: " << x << " " << y << " " << z << '\n';
}

int main()
{
    print(1, 2, 3); // all explicit arguments
    print(1, 2); // rightmost argument defaulted
    print(1); // two rightmost arguments defaulted
    print(); // all arguments defaulted

    return 0;
}
```

C++ does not (as of C++23) support a function call syntax such as `print(,,3)` (as a way to provide an explicit value for `z` while using the default arguments for `x` and `y`. This has three major consequences:

1. In a function call, any explicitly provided arguments must be the leftmost arguments (arguments with defaults cannot be skipped).

2. If a parameter is given a default argument, all subsequent parameters (to the right) must also be given default arguments.

```
void print(int x=10, int y); // not allowed
```

3. If more than one parameter has a default argument, the leftmost parameter should be the one most likely to be explicitly set by the user.

Once declared, a default argument can not be redeclared in the same translation unit. So either declare it in forward declaration or the function definition but not both but its must be declared before it is used in the code hence its better to include it in forward declarations

such functions can lead to potentially ambiguous function calls.

```
void print(int x);
void print(int x, int y = 10);
void print(int x, double y = 20.5);
```

```
print(1, 2); // will resolve to print(int, int)
print(1, 2.5); // will resolve to print(int, double)
print(1); // ambiguous function call
```

Function templates

Function Templates is a way to write a single version of function that can work with all types of arguments. the author of the template doesn't have to try to anticipate all of the actual types that might be used. This means template code can be used with types that didn't even exist when the template was written!

```
#include <iostream>

template <typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // instantiates and calls function max<int>(int, int)

    return 0;
}
```

For a function that returns maximum of two numbers of any datatype.
Much like we often use a single letter for variable names used in trivial situations (e.g. `x`), it's a common convention to use a single capital letter (starting with `T`) when the meaning of that type is trivial or obvious.
If a type template parameter has a non-obvious usage or meaning, then a more descriptive name is warranted. There are two common conventions for such names:

- Starting with a capital letter (e.g. `Allocator`). The standard library uses this naming convention.
- Prefixed with a `T`, then starting with a capital letter (e.g. `TAllocator`). This makes it easier to see that the type is a type template parameter.

When the compiler encounters the function call, it will realise that it doesn't exist and use the template to create one. The process of creating functions (with specific types) from function templates (with template types) is called **function template instantiation** (or **instantiation** for short). When a function is instantiated due to a function call, it's called **implicit instantiation**. A function that is instantiated from a template is technically called a **specialization**, but in common language is often called a **function instance**. The template from which a specialization is produced is called a **primary template**. Function instances are normal functions in all regards.

during the compilation process involving a function call to a template. a version of the function with the datatype we provide is create for current and further calls with the same datatype while the template definition is removed.

Example 1)

```
#include <iostream>

// a declaration for our function template (we don't need the definition any more)
template <typename T>
T max(T x, T y);

template<>
int max<int>(int x, int y) // the generated function max<int>(int, int)
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // instantiates and calls function max<int>(int, int)

    return 0;
}
```

Example 2)

```
#include <iostream>

template <typename T>
T max(T x, T y) // function template for max(T, T)
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n';    // instantiates and calls function max<int>(int, int)
    std::cout << max<int>(4, 3) << '\n';    // calls already instantiated function max<int>(int, int)
    std::cout << max<double>(1, 2) << '\n'; // instantiates and calls function max<double>(double, double)

    return 0;
}
```

Instantiation converts the above program to code given below

```cpp
#include <iostream>

// a declaration for our function template (we don't need the definition any more)
template <typename T>
T max(T x, T y);

template<>
int max<int>(int x, int y) // the generated function max<int>(int, int)
{
    return (x < y) ? y : x;
}

template<>
double max<double>(double x, double y) // the generated function max<double>(double, double)
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n';    // instantiates and calls function max<int>(int, int)
    std::cout << max<int>(4, 3) << '\n';    // calls already instantiated function max<int>(int, int)
    std::cout << max<double>(1, 2) << '\n'; // instantiates and calls function max<double>(double, double)

    return 0;
}
```

One additional thing to note here: when we instantiate `max<double>`, the instantiated function has parameters of type `double`. Because we've provided `int` arguments, those arguments will be implicitly converted to `double`.

If the argument type used for instantiation and arguments given are of same datatype, it is unefficient to provide to same information regarding the datatype of function needed twice.
we can use **template argument deduction** to avoid this

```cpp
std::cout << max<>(1, 2) << '\n';
std::cout << max(1, 2) << '\n';
```

max<>(1,2) can only be deduced to `max<int>` template function, while max(1,2) provides a secondary alternative that can be used to prioritise a normal max function, if it exists and go with the to `max<int>` template function in the absence of any normal matching.

```cpp
#include <iostream>

template <typename T>
T max(T x, T y)
{
    std::cout << "called max<int>(int, int)\n";
    return (x < y) ? y : x;
}

int max(int x, int y)
{
    std::cout << "called max(int, int)\n";
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // calls max<int>(int, int)
    std::cout << max<>(1, 2) << '\n';    // deduces max<int>(int, int) (non-template functions not considered)
    std::cout << max(1, 2) << '\n';      // calls max(int, int)

    return 0;
}
```

Function templates with non-template parameters

It’s possible to create function templates that have both template parameters and non-template parameters.

```
// T is a type template parameter
// double is a non-template parameter
// We don't need to provide names for these parameters since they aren't used
template <typename T>
int someFcn (T, double)
{
    return 5;
}

int main()
{
    someFcn(1, 3.4); // matches someFcn(int, double)
    someFcn(1, 3.4f); // matches someFcn(int, double) -- the float is promoted to a double
    someFcn(1.2, 3.4); // matches someFcn(double, double)
    someFcn(1.2f, 3.4); // matches someFcn(float, double)
    someFcn(1.2f, 3.4f); // matches someFcn(float, double) -- the float is promoted to a double

    return 0;
}
```

Beware function templates with modifiable static local variables

When a static local variable(variables that retain value throughout the program and usually used for id generation) is used in a function template, each function with a different datatype will be instantiated from that template with a separate version of the static local variable.

```
#include <iostream>

// Here's a function template with a static local variable that is modified
template <typename T>
void printIDAndValue(T value)
{
    static int id{ 0 };
    std::cout << ++id << ") " << value << '\n';
}

int main()
{
    printIDAndValue(12);
    printIDAndValue(13);

    printIDAndValue(14.5);

    return 0;
}
```

This produces the result:

```
1) 12
2) 13
1) 14.5
```

14.5 is a double and hence will start the id generation separate from int ids already generated

Function templates with multiple template types

```cpp
#include <iostream>

template <typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max(2, 3.5) << '\n';  // compile error

    return 0;
}
```

This program won't compile as the types for both arguments not being same which causes a error because the template function with T type can only take one type.

static_cast can put a band-aid this issue on the different datatype

```cpp
    std::cout << max(static_cast<double>(2), 3.5) << '\n'; // convert our int to a double so we can call max(double, double)
```

explicit type arguments can also be used

```cpp
// we've explicitly specified type double, so the compiler won't use template argument deduction
std::cout << max<double>(2, 3.5) << '\n';
```

Multiple Template Types on a Function Template are by far, the best solution to this issue

```cpp
#include <iostream>

template <typename T, typename U>
auto max(T x, U y)
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max(2, 3.5) << '\n';

    return 0;
}
```

Using function templates in multiple files

add.h:

```
#ifndef ADD_H
#define ADD_H

template <typename T>
T addOne(T x) // function template definition
{
    return x + 1;
}

#endif
```

main.cpp:

```
#include "add.h" // import the function template definition
#include <iostream>

int main()
{
    std::cout << addOne(1) << '\n';
    std::cout << addOne(2.3) << '\n';

    return 0;
}
```

Only notable abnormality is that we have to include the file and can not just rely on linker to instantiate the function definition (as instantiation does not happen when there is no function call in the code file that it is defined in). also this doesn't violate ODR as templates can have identical definitions over files just like inline functions/variables.

It is important to include the file with the function templates and **not rely** on the linker to instantiate the function definition(as instantiation does not happen with no function call in the same code file it is defined in). This also does not violate ODR as template can have identical definitions over files just like inline functions/variables

Function templates may be overloaded

```cpp
#include <iostream>

// Add two values with matching types
template <typename T>
T add(T x, T y)
{
    return x + y;
}

// Add two values with non-matching types
// As of C++20 we could also use auto add(auto x, auto y)
template <typename T, typename U>
T add(T x, U y)
{
    return x + y;
}

// Add three values with any type
// As of C++20 we could also use auto add(auto x, auto y, auto z)
template <typename T, typename U, typename V>
T add(T x, U y, V z)
{
    return x + y + z;
}

int main()
{
    std::cout << add(1.2, 3.4) << '\n'; // instantiates and calls add<double>()
    std::cout << add(5.6, 7) << '\n';   // instantiates and calls add<double, int>()
    std::cout << add(8, 9, 10) << '\n'; // instantiates and calls add<int, int, int>()

    return 0;
}
```

the call to `add(1.2, 3.4)` will be directed to `add<T>` over `add<T, U>` even though both could possibly match. The rules for determining which of multiple matching function templates should be preferred are called “partial ordering of function templates” which in general will prefer more restrictive/specialised function templates

Non-type template parameters

A **non-type template parameter** is a template parameter with a fixed type that serves as a placeholder for a constexpr value passed in as a template argument.
A non-type template parameter can be any of the following types:

- An integral type
- An enumeration type
- `std::nullptr_t`
- A floating point type (since C++20)
- A pointer or reference to an object
- A pointer or reference to a function
- A pointer or reference to a member function
- A literal class type (since C++20)

integral type non type template parameter

```cpp
#include <iostream>

template <int N> // declare a non-type template parameter of type int named N
void print()
{
    std::cout << N << '\n'; // use value of N here
}

int main()
{
    print<5>(); // 5 is our non-type template argument

    return 0;
}
```

This example prints:

```
5
```

What are non-type template parameters useful for?

As of C++20, function parameters cannot be constexpr. This is true for normal functions and constexpr functions. it would be impossible to use expressions like static_assert that **require constants(and not normal variables)** to be evaluated for purposes like asserting whether a statement is true or not and only running the function if the function argument is desired.

```cpp
template <double D> // requires C++20 for floating point non-type parameters
double getSqrt()
{
    static_assert(D >= 0.0, "getSqrt(): D must be non-negative");

    if constexpr (D >= 0) // ignore the constexpr here for this example
        return std::sqrt(D); // strangely, std::sqrt isn't a constexpr function (until C++26)

    return 0.0;
}
```

we can easily ensure the function wont run for values we don’t want it to run.

Conclusion

Function templates do have a few drawbacks, and we would be remiss not to mention them. First, the compiler will create (and compile) a function for each function call with a unique set of argument types. So while function templates are compact to write, they can expand into a crazy amount of code, which can lead to code bloat and slow compile times. The bigger downside of function templates is that they tend to produce crazy-looking, borderline unreadable error messages that are much harder to decipher than those of regular functions. These error messages can be quite intimidating, but once you understand what they are trying to tell you, the problems they are pinpointing are often quite straightforward to resolve.

These drawbacks are fairly minor compared with the power and safety that templates bring to your programming toolkit, so use templates liberally anywhere you need type flexibility! A good rule of thumb is to create normal functions at first, and then convert them into function templates if you find you need an overload for different parameter types.