Ch5: Functions and Recursion

Global Functions

Global Functions

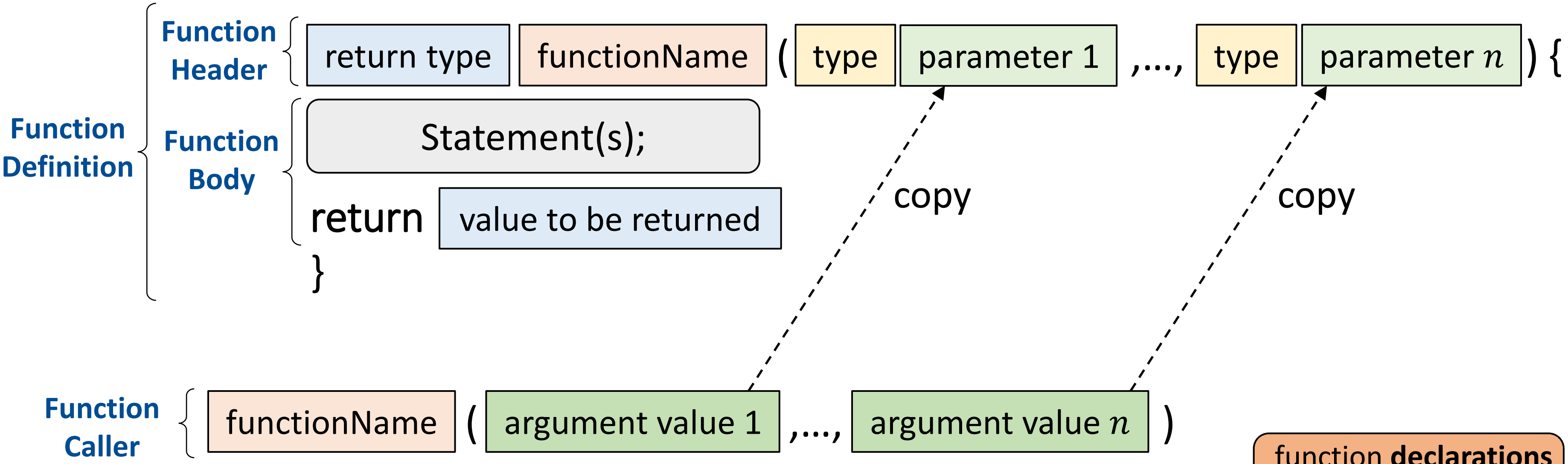
Functions allow you to modularize/divide a program by separating its tasks into self-contained units.

Advantages:

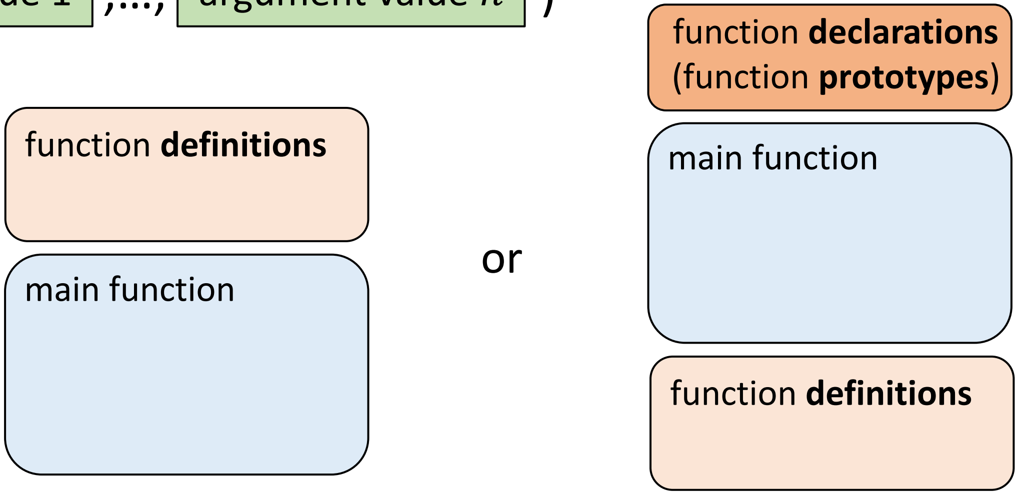
- Software reuse.
- Avoiding code repetition.
- Making the program easier to test, debug, and maintain.

❖ The functions that are not members of a class are called **Global Functions** (e.g., `main` function or all functions in the <cmath> header that perform common mathematical calculations).

Function Definition and Function Caller

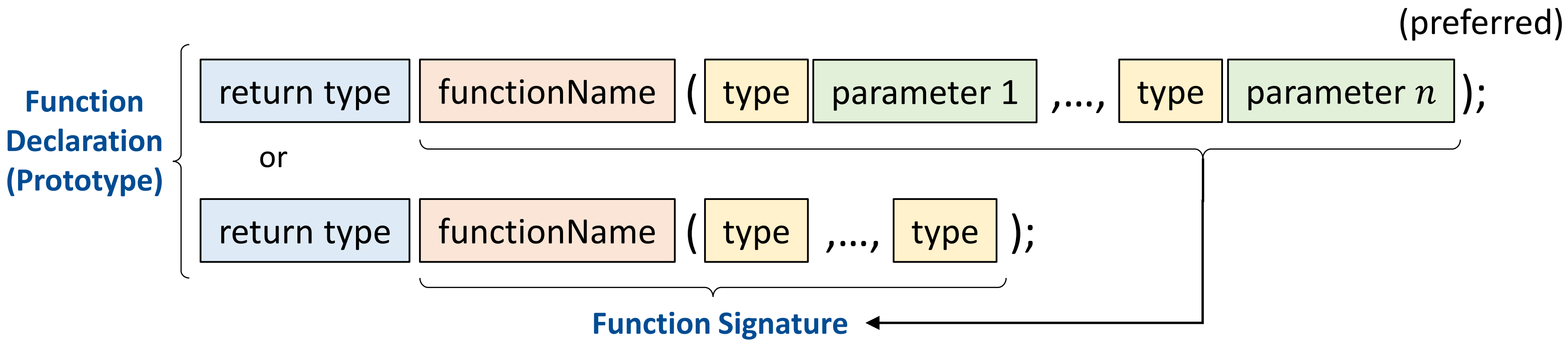


Usage: Global Function must be either **defined** or **declared** (by **function prototype**) before the main function.



Function Declaration/Prototype and Function Signature

A **function prototype/declaration** is the same as the first line of the corresponding function definition but ends with a required semicolon.



(preferred)

Function Declaration (Prototype)

return type functionName (type parameter 1 ,..., type parameter n);

or

return type functionName (type ,..., type);

Function Signature

- Parameter names in function prototypes are optional but recommended.
- The portion of a function prototype that includes the name of the function and the types of its arguments (but not return type) is called the **function signature**.
- Functions in the same scope must have unique signatures.
- Function arguments (in function caller) may be constants, variables, or more complex expressions.
- **Best Practice**: Always provide function prototypes. Providing the prototypes avoids tying the code to the order in which functions are defined.

Remarks

- If a function does not require any parameter to perform a task, its parameter list must be empty as ().
- The **return type** specifies the type of data the function returns to its caller after performing its task. If a function does not return any information to its caller, its type must be void.
- The **argument types** in the function call must be **consistent** (not necessarily identical) with the types of the corresponding parameters in the function's definition, otherwise, the compiler attempts perform Implicit Type Conversion (Narrowing Conversions or Promotion) to convert the arguments to those types. Narrowing conversions can result in incorrect values.

```
#include <iostream>
double addDouble(double x, double y) {
  return x + y;
}
int addInt(int x, int y) {
  return x + y;
}
int main() {
   std::cout << "3.4 + 4 = " << addDouble(3.4, 4) << '\n'; // promotion in 4
   std::cout << "3.4 + 4 = " << addInt(3.4, 4) << '\n'; // narrowing in 3.4
}
```

- Commas in function calls are not comma operators. The order of evaluation of a function's arguments is not specified by the C++ standard. Thus, different compilers can evaluate function arguments in different orders.

Sample Program: maximum Function

A user-defined function called maximum that returns the largest of its three int arguments.

```
// maximum function with a function prototype.
#include <iostream>

int maximum(int x, int y, int z); // function prototype

int main() {
  std::cout << "Enter three integer values: ";
  int int1, int2, int3;
  std::cin >> int1 >> int2 >> int3;

  // invoke maximum
  std::cout << "The maximum integer value is: "
    << maximum(int1, int2, int3) << std::endl;
}
```

```
// returns the largest of three integers
int maximum(int x, int y, int z) {
  int maximumValue{x}; // assume x is the largest to start

  // determine whether y is greater than maximumValue
  if (y > maximumValue) {
    maximumValue = y;
  } // make y the new maximumValue

  // determine whether z is greater than maximumValue
  if (z > maximumValue) {
    maximumValue = z; // make z the new maximumValue
  }

  return maximumValue;
}
```

Function Declarations and Definitions in Separate Files (Best Practices)

- You should never put function definitions in header files (except **template functions**). Header files should contain function declarations/prototypes.
- Function definitions can be in a separate .cpp file. Each .cpp file should have #include directives for any functions that it calls.
- All these files (separate header containing function declarations and .cpp containing function definitions) must be into the project folder.

main.cpp

```
#include <iostream>
#include "Functions.h"

int main() {
   std::cout << "3 + 4 = " << add(3, 4) << '\n';
   std::cout << "3^2 + 4^2 = " << sumOfSquares(3, 4) << '\n';
}
```

Functions.h

```
int add(int x, int y);
int sumOfSquares(int x, int y);
int pow2(int x);
```

MyFunctions.cpp

```
#include "Functions.h"
int add(int x, int y) {
   return x + y;
}
int sumOfSquares(int x, int y) {
    return pow2(x) + pow2(y);
}
int pow2(int x) {
    return x*x;
}
```

const Function Parameters

- Function parameters can be made constants via the const keyword, without initialization.
- Making a function parameter constant inform the compiler that the parameter's value is not accidentally changed inside the function.
- The const qualifier should be used to enforce the principle of least privilege. Using this principle to properly design software can greatly reduce debugging time and improper side effects and can make a program easier to modify and maintain.

```
#include <iostream>
int add(const int x, int y) {
  // x++; // error
  ++y;
  return x + y;
}
int main() {
   std::cout << "Result: " << add(3, 4) << "\n";
}
```

Scope Rules

Block Scope and Local Variable

- The portion of a program where an <u>identifier</u> can be used is known as its **Scope**.
- Identifiers declared inside a block ({}) have **block scope**, which begins at the identifier's declaration and ends at the terminating right brace (}) of the enclosing block (the blocks can appear in **all control structures** and **functions**).
- **Local variables** have block scope, and can be referenced only in that block and in <u>blocks nested</u> within that block (inner blocks).

```
#include <iostream>
int main() {
  int y{5}; // local variable to main
  std::cout << "y = " << y << "\n";
  {
    int z{7}; // local variable in nested block
    // "y" is accessible here
    std::cout << "y = " << y << "\n" << "z = " << z << "\n";
  }
  // "z" is not accessible here
}
```

- Variables declared in a particular function's body are **local variables** which can be used only in that function. When a function terminates, the values of its local variables are lost.
- Parameters of a function also are **local variables** of that function.

static Local Variable

Local variables in a function can be declared static. Such variables not only have block scope, but also retains its value when the function returns to its caller. The next time the function is called, the static local variable contains the value it had when the function last completed execution.

Note: All static local variables of numeric types are initialized to zero by default.

```
#include <iostream>

void demo() {
   static int count; // static variable (it is initialized to zero by default)
   ++count; // value is updated and will be carried to next function calls
   std::cout << count << "\n";
}

int main() {
   for (int i{0}; i < 5; ++i){
      demo();
   }
}
```

▶

Global Namespace Scope: Global Variables

- An identifier declared outside any function (including `main`) or class definition has **global namespace scope**. Such an identifier is “known” in all functions from the point at which it is declared until the end of the file.
- Function definitions, function prototypes placed outside a function, class definitions, and global variables all have global namespace scope.

```
#include <iostream>
int y{5}; // global variable
int Foo(int x) {
   y++; // function can read & modify global variables!
   return x + y;
}
int main() {
  std::cout << "y = " << y << "\n";
  std::cout << "Function = " << Foo(2) << "\n";
  std::cout << "y = " << y << "\n";
}
```

Unary Scope Resolution Operator

A global variable can be accessed directly; however, C++ provides the **Unary Scope Resolution Operator** (::) to access a global variable when a local variable of the same name is in scope.

```
#include <iostream>
int y{5}; // global variable
int Foo(int x) {
    ::y++; // function can read & modify global variables!
    return x + ::y;
}
int main() {
    int y{0}; // local variable
    std::cout << "y = " << y << "\n";
    std::cout << "y = " << ::y << "\n";
    std::cout << "Function = " << Foo(2) << "\n";
    std::cout << "y = " << ::y << "\n";
}
```

Local variable y is prioritized. ←

Best Practices:

- In general, variables should be declared in the narrowest scope in which they need to be accessed. Thus, global variables should be avoided.
- Avoid using variables of the same name for different purposes in a program. Although this is allowed in various circumstances, it can lead to errors.
- Always use (::) to refer to global variables (even if there is no collision with a local-variable name) to make it clear that you are intending to access a global variable rather than a local variable.

Recursion

Recursion

A recursive function is a function that calls itself, either directly, or indirectly (through another function).

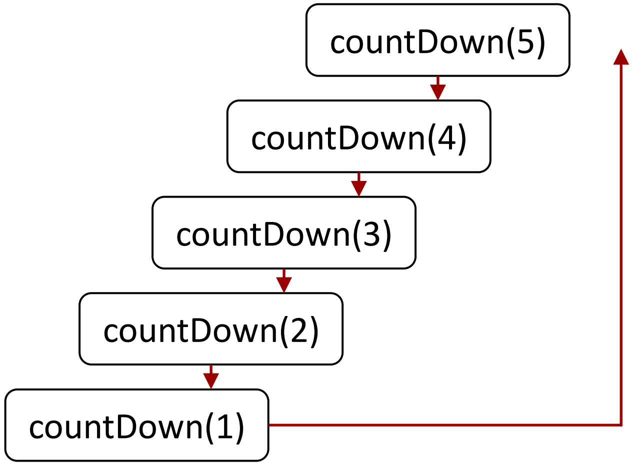
Recursive function calls generally work just like normal function calls. However, they need a recursive termination condition to stop calling itself (usually using an if statement), or they will run forever (actually, until the call stack runs out of memory).

```
#include <iostream>

void countDown(int i) {
  std::cout << "push " << i << '\n';
  if (i > 1) { // termination condition
    countDown(i - 1);
  }
  std::cout << "pop " << i << '\n';
}

int main() {
  countDown(5);
}
```

```
push 5
push 4
push 3
push 2
push 1
pop 1
pop 2
pop 3
pop 4
pop 5
```


countDown(5)
countDown(4)
countDown(3)
countDown(2)
countDown(1)

Example: Fibonacci Numbers

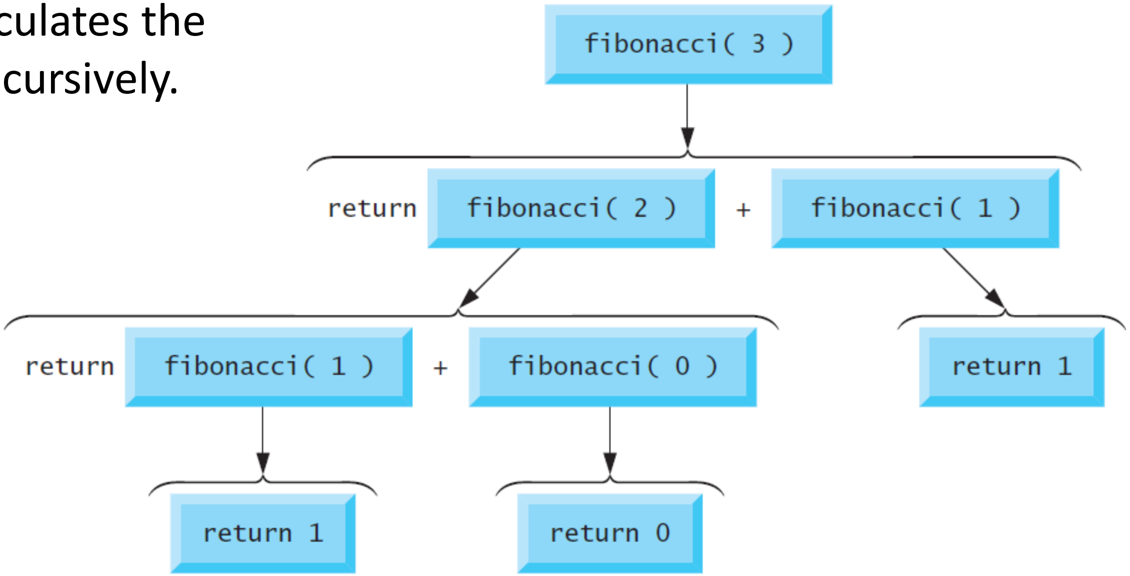
Fibonacci numbers are defined mathematically as:

$$f(n) = \begin{cases} 0 & \text{if} \quad n = 0 \\ 1 & \text{if} \quad n = 1 \\ f(n-1) + f(n-2) & \text{if} \quad n > 1 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, …

(begins with 0 and 1 and each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers)

Write a function that calculates the nth Fibonacci number recursively.



fibonacci(3)

return fibonacci(2) + fibonacci(1)

return fibonacci(1) + fibonacci(0)

return 1

return 1

return 0

Example: Fibonacci Numbers

```
#include <iostream>
unsigned long fibonacci(unsigned long number); // function prototype
int main() {
  // calculate the fibonacci values of 0 through 10
  for (unsigned int n{0}; n <= 10; ++n) {
    std::cout << "fibonacci(" << n << ") = " << fibonacci(n) << std::endl;
  }

  // display higher fibonacci values
  std::cout << "fibonacci(30) = " << fibonacci(30) << std::endl;
}

// recursive function fibonacci
unsigned long fibonacci(unsigned long number) {
  if ((0 == number) || (1 == number)) { // base case (termination condition)
    return number;
  }
  else { // recursion step
    return fibonacci(number - 1) + fibonacci(number - 2);
  }
}
```

Recursion vs Iteration

- Any problem that can be solved recursively can also be solved iteratively (non-recursively).
- A recursive approach is normally chosen when the recursive approach results in a program that is easier to understand and debug. However, recursive calls take time and consume additional memory.

```
#include <iostream>
unsigned long fibonacci (unsigned long number); // function prototype
int main() {
  // calculate the fibonacci values of 0 through 10
  for (unsigned int n{0}; n <= 10; ++n) {
    std::cout << "fibonacci(" << n << ") = " << fibonacci(n) << std::endl;
  }
  // display higher fibonacci values
  std::cout << "fibonacci(30) = " << fibonacci(30) << std::endl;
}

unsigned long fibonacci (unsigned long number) {
  int t1{0}, t2{1}, nextTerm{0};
  // The first two terms.
  if(number == 0)
    return t1;
  if(number == 1)
    return t2;
  for (int n{2}; n <= number; ++n) {
    nextTerm = t1 + t2;
    t1 = t2;
    t2 = nextTerm;
  }
  return nextTerm;
}
```

Function Overloading

Function Overloading

Function Overloading is used to create several functions of the **same name** (that perform similar tasks), as long as they have **different signatures**.

- The C++ compiler selects the proper function to call by examining the <u>number</u>, <u>types</u>, and <u>order</u> of the arguments in the call.

```cpp
#include <iostream>

int add(int x, int y);
double add(double x, double y);
int add(int x, int y, int z);

int main() {
  std::cout << add(1, 2) << "\n"; // calls int version
  std::cout << add(1.2, 2.3) << "\n"; // calls double version
  // std::cout << add(1, 2.3) << "\n"; // error: call of overloaded
                                   // 'add(int, double)' is ambiguous
  std::cout << add(1, 2, 3) << "\n"; // calls int version
}
```

```cpp
// function add for int values
int add(int x, int y) {
  return x + y;
}

// function add for double values
double add(double x, double y) {
  return x + y;
}

// function add for int values
int add(int x, int y, int z) {
  return x + y + z;
}
```

Default Arguments

Default Arguments

A function can have a parameter with default argument. When making a function call, the caller can optionally provide an argument for any function parameter that has a **Default Argument**.

- 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>

int add(int x, int y=4){ // 4 is the default argument
  return x + y;
}

int main() {
  std::cout << "x + y = " << add(1, 2) << '\n'; // y will use user-supplied argument 2
  std::cout << "x + y = " << add(3) << '\n'; // y will use default argument 4
}
```

Note: The equals sign must be used to specify a default argument. Using brace initialization will not work:

```
int add(int x, int y{4}) // compile error
```

Multiple Default Arguments

A function can have multiple parameters with default arguments:

```
#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
}
```

Note: Default arguments can only be provided for the rightmost parameters.

```
void print(int x = 10, int y = 20, int z) // not allowed
```

Because C++ (as of C++20) does not support a function call syntax such as `print(,,3)`

Default Arguments and Function Declaration

Once declared, a default argument can not be redeclared. Therefore, the default argument can be declared in either the function declaration or the function definition, but not both.

```
#include <iostream>

int add(int x, int y=4); // function declaration (function prototype)

int main() {
  std::cout << "x + y = " << add(1, 2) << "\n"; // y will use user-supplied argument 2
  std::cout << "x + y = " << add(3) << "\n"; // y will use default argument 4
}

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

Best practice: Declare the default argument in the function declaration and not in the function definition (particularly if it is in a header file).

Default Arguments and Function Overloading

Functions with default arguments may be overloaded. However, such functions can lead to potentially **ambiguous function calls**.

```
#include <iostream>

int add(int x){
   return x;
}

int add(int x, int y = 4){
   return x + y;
}

double add(int x, double y = 4.6){
   return x + y;
}

int main() {
   std::cout << "add(1, 2) = " << add(1, 2) << "\n"; // will resolve to add(int, int)
   std::cout << "add(1, 2.5) = " << add(1, 2.5) << "\n"; // will resolve to add(int, double)
   std::cout << "add(1) = " << add(1) << "\n"; // ambiguous function call
}
```

```
#include <iostream>

void print(std::string string) {
    std::cout << string << "\n";
}

void print(char ch = 'd') {
   std::cout << ch << "\n";
}

int main() {
    print("Hello, World"); // resolves to print(std::string)
    print('a'); // resolves to print(char)
    print(); // resolves to print(char)
}
```

Function Templates

Introduction to Function Templates

Consider the following code. To calculate the addition of two integer and flouting-point numbers, we must create overloaded versions of function add with parameters of type int and double (similarly, long, long double, and even new types that you've created).

```
#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) << "\n";
  std::cout << add(1.2, 2.2) << "\n";
}
```

Overloaded functions are normally used to perform similar operations that involve **different** program logic on different data types. If the program logic and operations are **identical** for each data type, we can instead create and maintain a single version of function (**template**) that can work with arguments of any type.

Function Template

```
#include <iostream>

template <typename T> // or template <class T>. This is the template parameter declaration
T add(T x, T y) { // this is the function template definition
  return x + y;
}

int main() {
  std::cout << add<int>(1, 2) << "\n"; // instantiates and calls function add<int>(int, int)
  std::cout << add<double>(1, 2.2) << "\n"; // instantiates and calls function add<double>(double, double)
                                            // 1 will be implicitly converted to double
}
```

- When we create our function template, we use **placeholder types** (also called **type template parameters**) for fundamental types or user-defined types. These placeholders are used to specify the parameter types, return types, or types used in the function body that we want to be specified later.

- The type in angled brackets (called a template argument) specifies the actual type that will be used in place of template type (T) by the compiler.

- **Best Practice**: Use a single capital letter (starting with T) to name your type template parameters (e.g., T, U, V, …).

Template Argument Deduction

In cases where the type of the arguments match the actual type we want, we do not need to specify the actual type; instead, we can use template argument deduction to have the compiler deduce the actual type that should be used from the argument types in the function call.

```
#include <iostream>

template <typename T> // this is the template parameter declaration
T add(T x, T y) { // this is the function template definition
  return x + y;
}

int main() {
  std::cout << add<>(1, 2) << "\n"; // calls function add<int>(int, int)
  std::cout << add(1, 2) << "\n"; // calls function add<int>(int, int)
  std::cout << add(1.0, 2.2) << "\n"; // calls function add<double>(double, double)
  // std::cout << add(1, 2.2) << "\n"; // syntax error: no matching function for call to "add(int, double)"
}
```

Best Practice: Favor the normal function call syntax when using function templates.

Template Argument Deduction & Non-Template Functions

```
#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'; // selects 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 function max(int, int)
}
```

▶

Implicite Type Conversion

```
#include <iostream>

template <typename T> // this is the template parameter declaration
T add(T x, T y) { // this is the function template definition
  return x + y;
}

double add2(double x, double y) {
   return x + y;
}

int main() {
  std::cout << add<double>(1, 2.2) << "\n"; // calls function add<double>(double, double)
                                            // 1 will be implicitly converted to double
  std::cout << add(1.0, 2.2) << "\n";  // calls function add<double>(double, double)
  // std::cout << add(1, 2.2) << "\n"; // syntax error: no matching function for call to "add(int, double)"
  std::cout << add(static_cast<double>(1), 2.2) << "\n"; // calls function add<double>(double, double)
  std::cout << add2(1, 2.2) << "\n"; // int argument will be implicitly converted to a double
}
```

Note: Implicit type conversions can be done when calling non-template functions. However, when performing template argument deduction, implicit type conversion are not allowed.

Functions Template Declaration

Functions templates can be declared (by function prototypes) before function main and defined after function main, but this definition cannot be placed in a separate source (.cpp) file. The best practice is to put all your function templates in a header (.h) file (without additional declarations) instead of a separate source (.cpp) file, and then, #included wherever needed.

```
#include <iostream>

template <typename T>
T addOne(T x); // function template declaration

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

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

≡

addone.h

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

```
#include <iostream>
#include "addone.h"

int main() {
  std::cout << addOne(1) << "\n";
  std::cout << addOne(2.3) << "\n";
}
```

Example: maximum Function Template

```
#include <iostream>
#include "maximum.h"

int main() {
   std::cout << "Input three integer values: ";
   int int1, int2, int3;
   std::cin >> int1 >> int2 >> int3;
   std::cout << "The maximum integer value is: "
      << maximum(int1, int2, int3);

   std::cout << "\n\nInput three double values: ";
   double double1, double2, double3;
   std::cin >> double1 >> double2 >> double3;
   std::cout << "The maximum double value is: "
      << maximum(double1, double2, double3);

   std::cout << "\n\nInput three characters: ";
   char char1, char2, char3;
   std::cin >> char1 >> char2 >> char3;
   std::cout << "The maximum character value is: "
      << maximum(char1, char2, char3) << std::endl;
}
```

maximum.h

```
template <typename T>
T maximum(T value1, T value2, T value3) {
   T maximumValue{value1};

   if (value2 > maximumValue) {
      maximumValue = value2;
   }

   if (value3 > maximumValue) {
      maximumValue = value3;
   }

   return maximumValue;
}
```

Function Templates with Non-Template Parameters

It is possible to create function templates that have both template parameters and non-template parameters. The type template parameters can be matched to any type, and the non-template parameters work like the parameters of normal functions.

```
#include <iostream>

// T is a type template parameter
// y is a non-template parameter
template <typename T>
double add(T x, double y) {
   return x + y;
}

int main() {
   std::cout << add(1, 3.4) << "\n"; // matches add(int, double)
   std::cout << add(1.2, 3.4) << "\n"; // matches add(double, double)
}
```

Function Templates with Multiple Template Type Parameters

Rather than using one template type parameter T, we can use more (e.g., T, U, V, …) to resolve the types independently.

```
#include <iostream>
template <typename T, typename U> // using two template type parameters T and U
T add(T x, U y) { // x can resolve to type T, and y can resolve to type U
  return x + y;
}
int main() {
  std::cout << add(2.2, 1) << "\n"; // works fine, prints 3.2
  std::cout << add(1, 2.2) << "\n"; // narrowing conversion problem! prints 3
}
```

In the cases that narrowing conversion is an issue, we can use auto keyword for return type to let the compiler deduce what the return type should be from the return statement.

```
#include <iostream>
template <typename T, typename U>
auto add(T x, U y) {
  return x + y;
}
int main() {
  std::cout << add(2.2, 1) << "\n"; // prints 3.2
  std::cout << add(1, 2.2) << "\n"; // prints 3.2
}
```

Abbreviated Function Templates

C++20 introduces a new use of the auto keyword: When the auto keyword is used as a parameter type in a normal function, the compiler will automatically convert the function into a function template with each auto parameter becoming an independent template type parameter. This method for creating a **function template** is called an **abbreviated function template** that makes your code more concise and readable.

```
#include <iostream>

template <typename T, typename U>
auto add(T x, U y) {
  return x + y;
}

int main() {
  std::cout << add(2.2, 1) << "\n"; // prints 3.2
  std::cout << add(1, 2.2) << "\n"; // prints 3.2
}
```

≡

```
#include <iostream>

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

int main() {
  std::cout << add(2.2, 1) << "\n"; // prints 3.2
  std::cout << add(1, 2.2) << "\n"; // prints 3.2
}
```