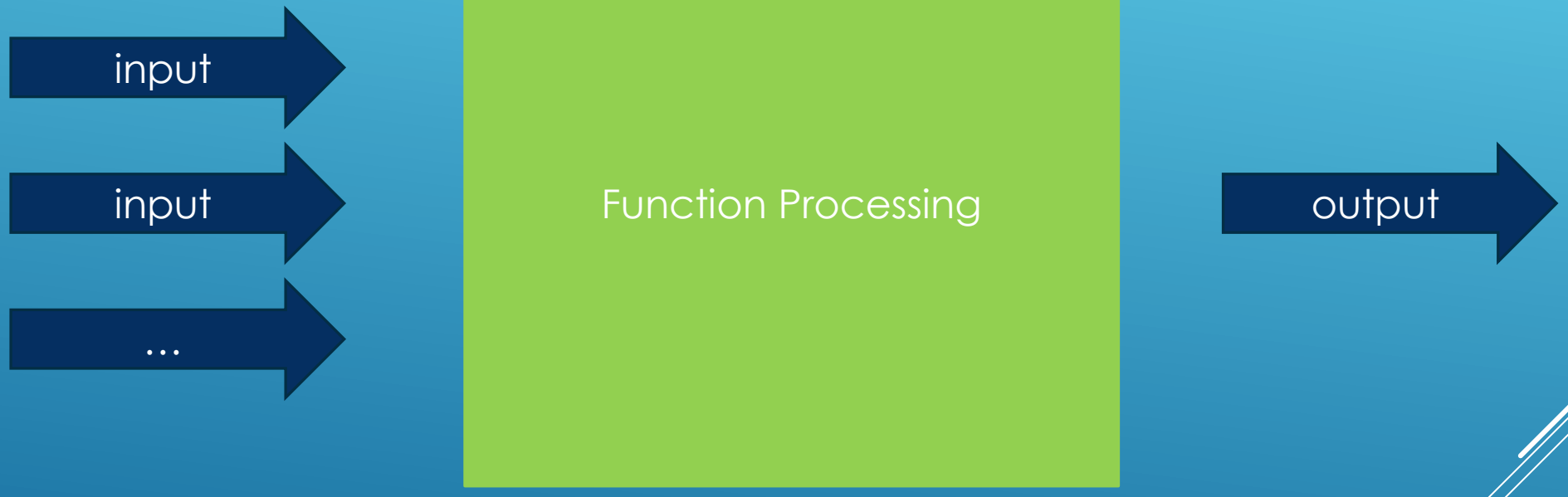
Slides

Section :Getting things out of functions

Slide intentionally left empty

Getting things out of functions : Introduction


input
input
...
Function Processing
output

Slide intentionally left empty

Input and Output parameters

Input and output function parameters

```
void max_str(const std::string& input1, const std::string input2,
             std::string& output)
{
    if(input1 > input2){
        output = input1;
    }else{
        output = input2;
    }
}

void max_int( int input1, int input2,int& output){
    if(input1 > input2){
        output = input1;
    }else{
        output = input2;
    }
}

void max_double( double input1, double input2,double* output){
    if(input1 > input2){
        *output = input1;
    }else{
        *output = input2;
    }
}
```

Input and output function parameters

- Output parameters should be passed in such a way that you can modify the arguments from inside the function. Options are passing by reference or by pointer. References are preferred in C++
- Input parameters shouldn't be modifiable from inside a function. The function really need to get input (read) from the arguments. You enforce modification restrictions with the const keyword. Options are passing by const reference, passing by pointer to const, or even passing by const pointer to const

Slide intentionally left empty

Returning from functions

Input and output function parameters

```
void max_str(const std::string& input1, const std::string input2,
             std::string& output)
{
    if(input1 > input2){
        output = input1;
    }else{
        output = input2;
    }
}

void max_int( int input1, int input2,int& output){
    if(input1 > input2){
        output = input1;
    }else{
        output = input2;
    }
}

void max_double( double input1, double input2,double* output){
    if(input1 > input2){
        *output = input1;
    }else{
        *output = input2;
    }
}
```

Returning from functions (Default is by value)

```
int sum ( int a , int b){
    int result = a + b;
    std::cout << "In : &result(int) :  " << &result << std::endl;
    return result;
}

int main(int argc, char **argv)
{

    int a {34};
    int b {16};

    int result = sum(a,b);
    std::cout << "Out : &result(int) :  " << &result << std::endl;
    std::cout << "sum : " << result << std::endl;

    return 0;
}
```

Compiler optimizations

In modern compilers, return by value is commonly optimized out by the compiler when possible and the function is modified behind your back to return by reference, avoiding unnecessary copies!

Compiler optimizations to return by reference(GOOD FOR YOU!)

```
std::string add_strings(std::string str1, std::string str2){
    std::string result =  str1 + str2;
    std::cout << "In : &result(std::string) :  " << &result << std::endl;
    return result;
}

int main(int argc, char **argv)
{
    std::string str_result = add_strings(std::string("Hello"),std::string(" World!"));
    std::cout << "Out : &result(std::string) :  " << &str_result << std::endl;
    std::cout << "str_result : " << str_result << std::endl;

    return 0;
}
```

Slide intentionally left empty

Returning by reference

Return by reference

```
int& max_return_reference(int& a, int& b)
{
    if(a > b) {
        return a;
    } else {
        return b;
    }
}
```

```
int a {15};
int b {13};

std::cout << "Before function call : " << std::endl;
std::cout << "a : " << a << std::endl;
std::cout << "b : " << b << std::endl;

int& ref_max = max_return_reference(a,b);
int val = max_return_reference(a,b);//Value in returned reference
                                    // is stored in val. val is not
                                    // a reference.Just a regular variable.
std::cout << "max : " << ref_max << std::endl;
std::cout << "val : " << val << std::endl;

++ref_max;

std::cout << std::endl;
std::cout << "After function call : " << std::endl;
std::cout << "a : " << a << std::endl;
std::cout << "b : " << b << std::endl;
```

References to local variables

```
int& sum( int& a, int& b){
    int result = a + b;
    return result;// Reference to local variable returned
}


int& max_input_by_value (int a, int b)
{
    if(a > b) {
        return a; // Reference to local variable returned
    } else {
        return b; // Reference to local variable returned
    }
}
```

References to local variables

```
//Beware of references to local variables
int& bad = sum(a,b); // CRASH ON MY SYSTEM
std::cout << "bad : " << bad << std::endl;

int & max = max_input_by_value(a,b);//CRASH ON MY SYSTEM
std::cout << "max : " << max << std::endl;
```

Slide intentionally left empty

Returning by pointer

Returning by pointer

```
int* max_return_pointer(int* a, int* b)
{
    if(*a > *b) {
        return a;
    } else {
        return b;
    }
}
```

```cpp
int a {15};
int b {13};

std::cout << "Before function call : " << std::endl;
std::cout << "a : " << a << std::endl;
std::cout << "b : " << b << std::endl;

int* p_max = max_return_pointer(&a,&b);

std::cout << "max : " << *p_max << std::endl;

++(*p_max);

std::cout << std::endl;
std::cout << "After function call : " << std::endl;
std::cout << "a : " << a << std::endl;
std::cout << "b : " << b << std::endl;
```

Pointers to local variables

```
int* sum( int* a, int* b){
    int result = *a + *b;
    return &result;// Pointer to local variable returned
}


int* max_input_by_value (int a, int b)
{
    if(a > b) {
        return &a; // Pointer to local variable returned
    } else {
        return &b; // Pointer to local variable returned
    }
}
```

Pointers to local variables

```
//Beware of pointers to loca variables
int* bad = sum(&a,&b); // CRASH ON MY SYSTEM
std::cout << "bad : " << *bad << std::endl;

int* max = max_input_by_value(a,b);//CRASH ON MY SYSTEM
std::cout << "max : " << *max << std::endl;
```

Slide intentionally left empty

Returning array element address by pointer

```cpp
double* find_max_address(const double scores[], size_t count){
//const double* find_max_address(const double scores[], size_t count){

    size_t max_index{};
    double max{};

    for(size_t i{0}; i < count ; ++i){
        if(scores[i] > max){
            max = scores[i];
            max_index = i;
        }
    }
    return &scores[max_index];
}
```

Slide intentionally left empty

Bare type deduction with auto

Naked auto deduction

```
//Type deduction with auto : Just a copy
double some_var{55.5};

auto x = some_var;

std::cout << "sizeof(some_var) : " << sizeof(some_var) << std::endl;
std::cout << "sizeof(x) : " << sizeof(x) << std::endl;
std::cout << "&some_var : " << &some_var << std::endl;
std::cout << "&x : " << &x << std::endl;
```

Naked auto reference deduction

```
//Type deduction with references : a copy is made with the naked auto
std::cout << std::endl;
std::cout << "type deduction with references - case1 : " << std::endl;
some_var = 55.5; //double

double& some_var_ref {some_var};

auto y = some_var_ref; // y is not deduced as a reference to double
                       // it's just a double that contains the value
                       // in some_var_ref
++y;

std::cout << "sizeof(some_var) : " << sizeof(some_var) << std::endl;
std::cout << "sizeof(some_var_ref) : " << sizeof(some_var_ref) << std::endl;
std::cout << "sizeof(y) : " << sizeof(y) << std::endl;
std::cout << "&some_var : " << &some_var << std::endl;
std::cout << "&some_var_ref : " << &some_var_ref << std::endl;
std::cout << "&y : " << &y << std::endl;
std::cout << "some_var : " << some_var << std::endl;
std::cout << "some_var_ref : " << some_var_ref << std::endl;
std::cout << "y : " << y << std::endl;
```

Proper reference deduction

```
//Deducing the reference. Insert a & after the auto keyword.
std::cout << std::endl;
std::cout << "type deduction with references - case2 : " << std::endl;
some_var = 55.5; //double

auto& z = some_var_ref; // z is deduced as a double reference
++z;


std::cout << "sizeof(some_var) : " << sizeof(some_var) << std::endl;
std::cout << "sizeof(some_var_ref) : " << sizeof(some_var_ref) << std::endl;
std::cout << "sizeof(z) : " << sizeof(z) << std::endl;
std::cout << "&some_var : " << &some_var << std::endl;
std::cout << "&some_var_ref : " << &some_var_ref << std::endl;
std::cout << "&z : " << &z << std::endl;
std::cout << "some_var : " << some_var << std::endl;
std::cout << "some_var_ref : " << some_var_ref << std::endl;
std::cout << "z : " << z << std::endl;
```

Constness preservation with deduced references

```
//Constness is preserved with properly deduced references.
const double some_other_var {44.3};

const double& const_ref {some_other_var};

auto& p = const_ref;

std::cout << "some_other_var : " << some_other_var << std::endl;
std::cout << "const_ref : " << const_ref << std::endl;
std::cout <<  "p : " << p << std::endl;

//++p; // Compiler error

std::cout << "some_other_var : " << some_other_var << std::endl;
std::cout << "const_ref : " << const_ref << std::endl;
std::cout <<  "p : " << p << std::endl;
```

Non reference naked auto deductions : constness doesn't matter

```
//Constness doesn't matter with non reference deduction
const double i_am_const {71.2};
auto q = i_am_const; // q is a separate variable, initialized with the
                     // value in i_am_const.
++q;// Can modify q without a problem. It's a copy.
```

Slide intentionally left empty

Return Type deduction

Let the compiler deduce the return type of a function, judging from return statements in the function

```
auto process_number(int value){

    if(value < 10){
        return 22; // returning int literal
    }else{
        return 55; // returning int literal
    }
}
```

Things gone wrong

```
auto process_number(int value){

    if(value < 10){
        return 22; // returning int literal
    }else{
        return 33.1; // returning double literal
    }
}
```

Things gone wrong

```
auto process_number(int value){

    if(value < 10){
        return 22; // returning int literal
    }else{
        return 33.1; // returning double literal
    }
}
```

Possible fix

```
double process_number(int value){

    if(value < 10){
        return 22; // returning int literal
    }else{
        return 33.1; // returning double literal
    }
}
```

Possible fix

```
auto process_number(int value){

    if(value < 10){
        //return 22; // returning int literal
        return static_cast<double>(22);
    }else{
        return 33.1; // returning double literal
    }
}
```

Slide intentionally left empty

Return Type deduction with references

Naked reference auto return type deduction

```
auto max(int& a, int& b){
    if(a>b){
        return a;
    }else{
        return b; // Will return a copy.
    }
}
```

Naked reference auto return type deduction

```
int x{4};
int y{5};


//Naked auto reference return type deduction : A copy is returned
std::cout << std::endl;
std::cout << "Naked auto reference return type deduction : " << std::endl;

//int& result = max(x,y); // Error : Can not treat return value as a reference
                          // It's jut a bare separate variable with a value inside.

int result = max(x,y); // A copy of the max is returned

++ result;

std::cout << "x :" << x << std::endl;
std::cout << "y :" << y << std::endl;
std::cout << "result : " << result << std::endl;
```

Proper auto reference return type deduction

```
auto& max_ref(int& a, int& b){
    if(a>b){
        return a;
    }else{
        return b; // Will return a true reference.
    }
}
```

Proper auto reference return type deduction

```
//Proper reference return type deduction : A true reference is returned.
std::cout << std::endl;
std::cout << "Proper reference return type deduction : " << std::endl;

int p{10};
int q{11};

int& result_ref = max_ref(p,q);

++result_ref;

std::cout << "p : " << p << std::endl;
std::cout << "q : " << q << std::endl;
std::cout << "result_ref : " << result_ref << std::endl;
```

Slide intentionally left empty

Function definition and return type deduction

```
auto max(int a, int b){
    if(a > b){
        return a;
    }else{
        return b;
    }
}


auto& max_ref(int& a, int& b){
    if(a > b){
        return a;
    }else{
        return b;
    }
}
```

Declaration & Definition

```
auto max(int a, int b);

int main(int argc, char **argv)
{
    int x{4};
    int y{5};

    int result = max(x,y);

    return 0;
}

auto max(int a, int b){
    if(a > b){
        return a;
    }else{
        return b;
    }
}
```

Slide intentionally left empty

Optional output from functions

Function could return something useful, or fail!

```
int find_character_v0(const std::string & str, char c){
    //If found , return the index, else return -1
    int not_found {-1};
    for (size_t i{} ; i < str.size()  ; ++i){
        if(str.c_str()[i] == c){
            return i;
        }
    }
    return not_found;
}
```

Finding a character

```
std::string str1 {"Hello World in C++20!"};
char c{'C'};

std::cout << std::endl;
std::cout << "Judging by return value (-1) : " << std::endl;

//std::cout << "index : " << find_character_v0(str1,c) << std::endl;

if(find_character_v0(str1,c)!= -1){
    std::cout << "found the character" << std::endl;
}else{
    std::cout << "couldn't find the character" << std::endl;
}
```

Function could return something useful, or fail!

```
void find_character_v1(const std::string & str, char c, bool & success){
    //If found set success to true, else set to false
    for (size_t i{} ; i < str.size()  ; ++i){
        if(str.c_str()[i] == c){
            success = true;
            return;
        }
    }
    success = false;
}
```

Finding a character

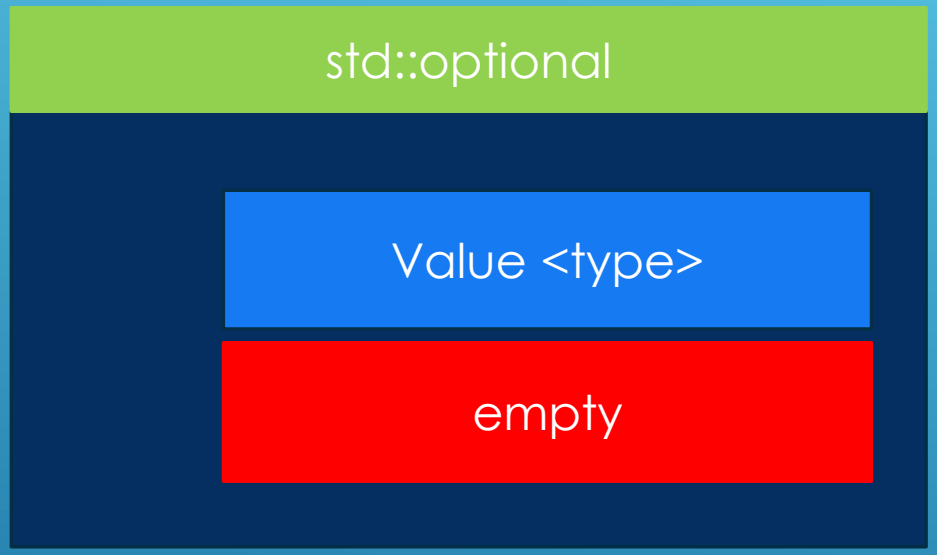
```
std::string str1 {"Hello World in C++20!"};
char c{'C'};

std::cout << std::endl;
std::cout << "Using an input success flag : " << std::endl;

bool success{};

find_character_v1(str1,c,success);

if(success){
    std::cout << "found the character" << std::endl;
}else{
    std::cout << "couldn't find the character" << std::endl;
}
```

Slide intentionally left empty

Introducing std::optional

std::optional

A type introduced in C++17 help handle optional output from functions and some other things


std::optional
Value <type>
empty

Declare and initialize

```
//Declare and initialize
std::optional<int> items{3};
std::optional<std::string> name{"Daniel Gakwaya"};
std::optional<double> weight {77.2};
std::optional<char> character{'s'};


//Declare and empty initialize
std::optional<std::string> dog_name {}; // Initializes to std::nullopt
std::optional<int> age {std::nullopt}; // std::nullopt is the null or zero equivalent
                                       // for std::optional
std::optional<char> letter = std::nullopt;  // Triggers unused warning
std::optional<double> height = {};          //Initializes to std::nullopt
                                            // Triggers unused warning.
```

Reading

```
//Reading values : can use the value method, or the dereference operator

std::cout << "items : " << items.value() << std::endl;
std::cout << "items : " << *items << std::endl; // Personaly find this confusing as
                                                // it's not a pointer, so this the
                                                // first and last time I use it in the
                                                // course
std::cout << "name : " << name.value() << std::endl;
std::cout << "weight : " << weight.value() << std::endl;
std::cout << "character : " << character.value() << std::endl;
```

Setting values

```
//Setting values : assigning to std::optional
//you do it as if you're assigning to the wrapped type
std::cout << std::endl;
std::cout << "Setting values : " << std::endl;

dog_name = "Fluffy";
age = 26;

std::cout << "dog_name : " << dog_name.value() << std::endl;
std::cout << "age : " << age.value() << std::endl;
```

Reading bad data

```
//Trying to use std::nullopt std::optional variable will throw an exception
//and crash your program. We'll see ways around this later in the course
//when we learn about exception handling.
std::cout << letter.value() << std::endl; // Throws exception and crashes program
std::cout << height.value() << std::endl; // Trhows exception and crashes program
```

Playing it safe

```
//Conditional reading
std::cout << std::endl;
std::cout << "Conditional reading" << std::endl;
//has_value() method. Recommended. Self documenting
if(letter.has_value()){
    std::cout << "letter contains a useful value" << std::endl;
}else{
    std::cout << "letter contains std::nullopt" << std::endl;
}

//Checking against std::nullopt
if(letter != std::nullopt){
    std::cout << "letter contains a useful value" << std::endl;
}else{
    std::cout << "letter contains std::nullopt" << std::endl;
}


if(height!=std::nullopt){
    std::cout << "height contains a useful value" << std::endl;
}else{
    std::cout << "height contains std::nullopt" << std::endl;
}
```

General recommendation

```
//Always recommended to perform such checks before
//reading from a std::optional
if(dog_name.has_value()){
    std::cout << "dog_name : " << dog_name.value() << std::endl;
}else{
    std::cout << "dog_name contains nullopt" << std::endl;
}

dog_name = std::nullopt;

if(dog_name.has_value()){
    std::cout << "dog_name : " << dog_name.value() << std::endl;
}else{
    std::cout << "dog_name contains nullopt" << std::endl;
}
```

Slide intentionally left empty

Optional output with std::optional

Find a character in a string

```
std::optional<int> find_character_v2(const std::string & str, char c){
    //If found set the return index, else return an empty std::optional
    for (size_t i{} ; i < str.size()  ; ++i){
        if(str.c_str()[i] == c){
            return i;
        }
    }
   return {}; //Or std::nullopt
}
```

Find a character in a string

```
std::string str1 {"Hello World in C++20!"};
char c{'C'};

std::cout << std::endl;
std::cout << "Handling optional output with std::optional" << std::endl;
//auto result = find_character_v2(str1,c);
std::optional<int> result = find_character_v2(str1,c);

if(result.has_value()){
    std::cout << "Found " << c  << " at index " << result.value() << std::endl;
}else{
    std::cout << "Could not find " << c  << " in the string : " << str1 << std::endl;
}
```

Specify default search parameter [UGGLY]

```
std::optional<int> find_character_v3(const std::string & str,
                                     std::optional<char> c = std::nullopt){
    //If found set return index, else return empty
    //If c is specified, find it else just find 'z'
    char char_to_find;
    if(c.has_value()){
        char_to_find = c.value();
    }else{
        char_to_find = 'z'; // Will find z by default
    }

    for (size_t i{} ; i < str.size()  ; ++i){
        //std::cout << "str[i] : " << str.at(i) << " , c : " << c << std::endl;
        if(str.c_str()[i] == char_to_find){
            return i;
        }
    }
   return {};// Or std::nullopt
}
```

Find a character in a string

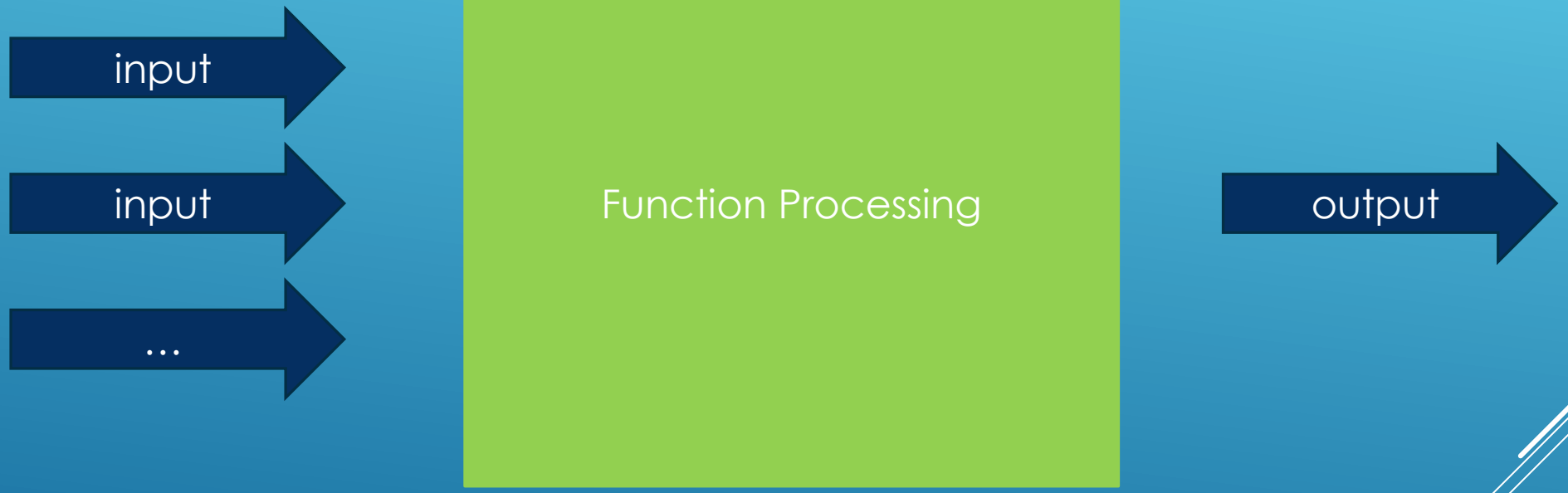
```
std::cout << std::endl;
std::cout << "Specifying a default character to search for[Uggly syntax]" << std::endl;

str1 = "Hello Worldz in C++20!";

auto result1 = find_character_v3(str1); //Will search for 'z' if you don't specify
                                        // the character so search for
if(result1.has_value()){
    std::cout << "Found character 'z' at index " << result1.value() << std::endl;
}else{
    std::cout << "Could not find character 'z' in the string : " << str1 << std::endl;
}
```

Specify default search parameter [PRETTY]

```
std::optional<int> find_character_v4(const std::string & str,
                                     std::optional<char> c= std::nullopt){
    //If found set return index, else return empty
    //If c is specified, find it else just find a

    /*
     //Replace this crazy logic with something more sane
    char char_to_find;
    if(c.has_value()){
        char_to_find = c.value();
    }else{
        char_to_find = 'z'; // Will find z by default
    }
    */
    //If c has a valid value, get it and assign it to char_to_find,
    //otherwise use the default 'z'.Cool right ?
    char char_to_find = c.value_or('z');

    for (size_t i{} ; i < str.size()  ; ++i){
        //std::cout << "str[i] : " << str.at(i) << " , c : " << c << std::endl;
        if(str.c_str()[i] == char_to_find){
            return i;
        }
    }
   return {};// Or std::nullopt
}
```

Slide intentionally left empty

Getting things out of functions : Summary


input
input
...
Function Processing
output

Input output parameters

```
void max_str(const std::string& input1, const std::string input2,
             std::string& output)
{
    if(input1 > input2){
        output = input1;
    }else{
        output = input2;
    }
}

void max_int( int input1, int input2,int& output){
    if(input1 > input2){
        output = input1;
    }else{
        output = input2;
    }
}

void max_double( double input1, double input2,double* output){
    if(input1 > input2){
        *output = input1;
    }else{
        *output = input2;
    }
}
```

Returning from functions (Default is by value)

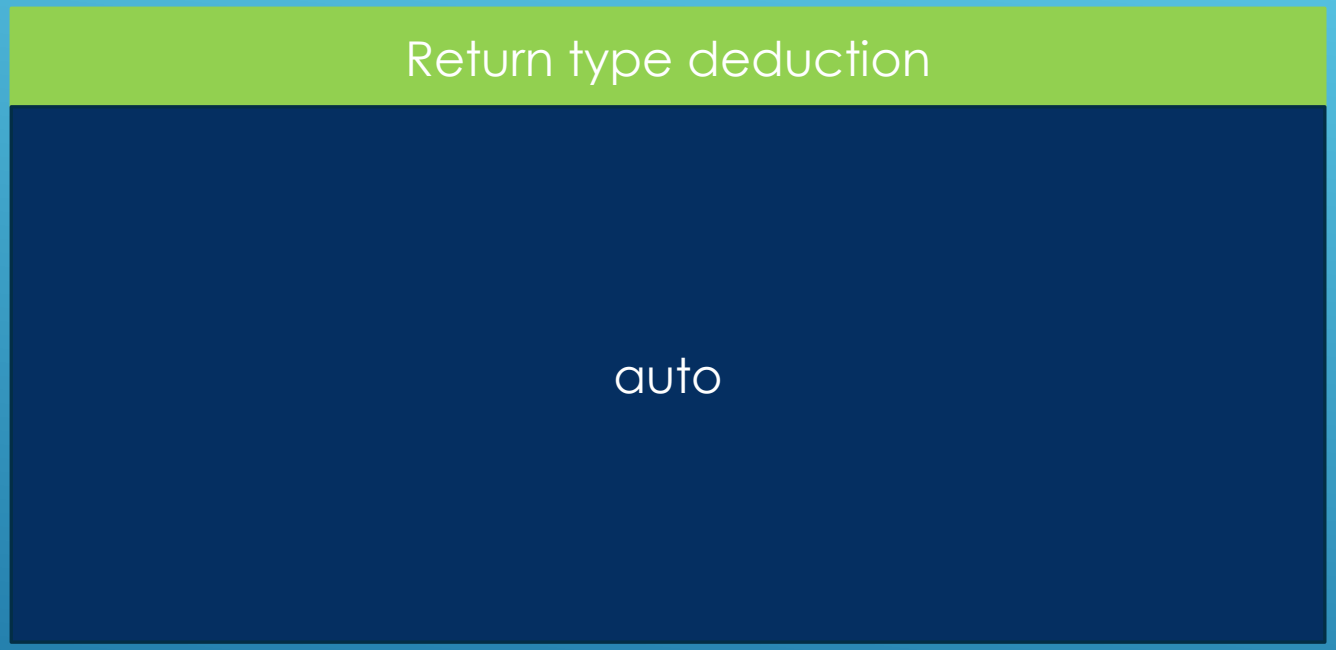
```
int sum ( int a , int b){
    int result = a + b;
    std::cout << "In : &result(int) :  " << &result << std::endl;
    return result;
}

int main(int argc, char **argv)
{

    int a {34};
    int b {16};

    int result = sum(a,b);
    std::cout << "Out : &result(int) :  " << &result << std::endl;
    std::cout << "sum : " << result << std::endl;

    return 0;
}
```

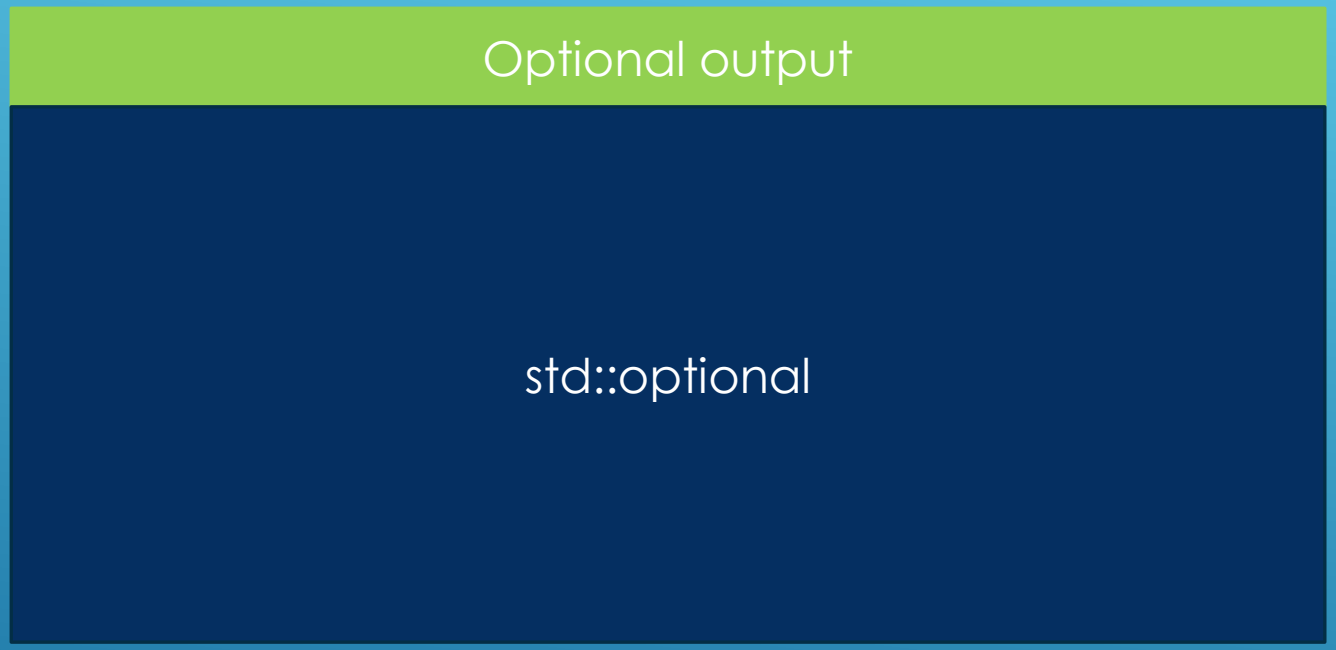
Return by reference

```
int& max_return_reference(int& a, int& b)
{
    if(a > b) {
        return a;
    } else {
        return b;
    }
}
```

Returning by pointer

```
int* max_return_pointer(int* a, int* b)
{
    if(*a > *b) {
        return a;
    } else {
        return b;
    }
}
```


Return type deduction
auto


Optional output
std::optional

Slide intentionally left empty