SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220126<A|D>

Replace <A|D> with this section's letter

More C++ Basics

CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

Agenda

- C++ Intro (continued)
- Functions in C++
- References and const types

C++ Introduction (cont)

Loops: `while`

```
int main() {
    int i_num = 10;
    while (i_num > 0) {
        cout << i_num << ' ';
        --i_num;
    }
    cout << endl;
}
```

- condition must be in parenthesis
- code for loop located in block
- `--` is the pre-decrement operator
    - Python: `i_num -= 1`

TurningPoint

SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220126<A|D>

Replace <A|D> with this section's letter

What would be the output of the code?

```
int main() {
    int i_num = 10;
    while (i_num > 0) {
        cout << i_num << ' ';
        --i_num;
    }
    cout << endl;
}
```

Loops: `while`

```
int main() {
    int i_num = 10;
    while (i_num > 0) {
        cout << i_num << ' ';
        --i_num;
    }
    cout << endl;
}
```

- condition must be in parenthesis
- code for loop located in block
- `--` is the pre-decrement operator
  - Python: `i_num -= 1`

```
10 9 8 7 6 5 4 3 2 1
```

Loops: `for`

```
int main() {
    for (int i = 10; i > 0; --i) {
         if (i == 5) continue;
         // yes, C++ has continue and break
         cout << i << ' ';
    }

    cout << endl;

}
```

- `for` loop has 3 components specified in parentheses and separated by semicolons (`;`)
  - initialization: first value assigned to loop variable
    - `i` initialized to `10`
  - test: condition determining when to enter loop
    - `i` must be greater than `0`
  - update: modification of loop variable's value
    - `i` decremented
- often more concise than equivalent `while` loop
- **scope** of loop variable (`i`) limited to `for` loop

Loops: do-while

```
int main() {
    int i_num = 10;
    do {
        cout << i_num << ' ';
        --i_num;
    } while (i_num > 0); // remember the semicolon
    cout << endl;
}
```

- always executes loop body at least once
- `for` and `while` loops used more often

What would happen if `i_num` initialized to -1?

```
int main() {
    int i_num = 10;
    do {
        cout << i_num << ' ';
        --i_num;
    } while (i_num > 0); // remember the semicolon
    cout << endl;
}
```

Collections: vector

```
int main() {
   vector<int> vec; // vec can only hold ints
   cout << "vec.size(): " << vec.size() << endl;

   vec.push_back(17);
   vec.push_back(42);
   cout << "vec.size(): " << vec.size() << endl;

   for (size_t i = 0; i < vec.size(); ++i) {
       cout << vec[i] << ' ';
   }

   cout << endl;
   vec.clear();
   cout << "vec.size(): " << vec.size() << endl;

}
```

- available after `#include <vector>`
- full name: `std::vector<type>`
- similar to Python `list` and Java's `ArrayList`
- *generic* type
- other useful methods:
  - `back()`
  - `pop_back()`
  - `capacity()`

Collections: vector initialization

```
int main() {
    // Initialize the size of the vector and the value of that all the
    // entries will start with.
    // The vector v1 will be of size 7, with every entry equal to 42
    vector<int> vec1(7, 42); // parentheses

    // specify the distinct values to initialize each entry to
    vector<int> vec2{1, 1, 2, 3, 5, 8, 13, 21}; // curly braces
}
```

What is the value of `vec1[3]`?

```
int main() {
    // Initialize the size of the vector and the value of that all the
    // entries will start with.
    // The vector v1 will be of size 7, with every entry equal to 42
    vector<int> vec1(7, 42); // parentheses

    // specify the distinct values to initialize each entry to
    vector<int> vec2{1, 1, 2, 3, 5, 8, 13, 21}; // curly braces
}
```

What is the value of `vec2[3]`?

```
int main() {
    // Initialize the size of the vector and the value of that all the
    // entries will start with.
    // The vector v1 will be of size 7, with every entry equal to 42
    vector<int> vec1(7, 42); // parentheses

    // specify the distinct values to initialize each entry to
    vector<int> vec2{1, 1, 2, 3, 5, 8, 13, 21}; // curly braces
}
```

Loops: ranged `for`

```
int main() {
    vector<int> vec;
    ...
    for (size_t i = 0; i < vec.size(); ++i) {
        cout << vec[i] << ' ';
    }
    cout << endl;
    ...
}
```

Loops: ranged `for`

```
int main() {
    vector<int> vec;
    ...

    // for (size_t i = 0; i < vec.size(); ++i) {
    //     cout << vec[i] << ' ';
    // }

    for (int value : vec) {
        cout << value << ' ';
    }
    cout << endl;
    ...
}
```

- don't need `i`?
  - easier to use a *ranged* `for`
- similar to Python's `for`
- available since C++11

Loops: ranged `for` (continued)

```
int main() {
    vector<int> vec{2, 3, 5, 7, 11};

    for (int value : vec) {
        value = 42;
    }


    for (int value : vec) {
        cout << value << ' ';
    }
    cout << endl;
}
```

What is output by this code?

```
int main() {
    vector<int> vec{2, 3, 5, 7, 11};

    for (int value : vec) {
        value = 42;
    }

    for (int value : vec) {
        cout << value << ' ';
    }
    cout << endl;
}
```

Loops: ranged `for` (continued)

```
int main() {
    vector<int> vec{2, 3, 5, 7, 11};

    for (int value : vec) {
        value = 42;
    }

    for (int value : vec) {
        cout << value << ' ';
    }
    cout << endl;
}
```

- line of 42's not printed
- fix for this next time

strings

- `#include <string>`
- string **literals** use *double* quotes
  - `string str = "this is a string";`
- `string` variable acts much like a `vector` of characters
  - e.g. `push_back()`, `pop_back()`, `size()`, `clear()`
- `string` has some additional useful features
  - e.g. input to/output from `string` possible (not possible for `vector`)
- characters are a separate type from strings
  - character literals use *single* quotes
  - `char c = 'q';`

`string` example

```
#include <string>
using namespace std;

int main() {
    string str1 = "Twas brillig and";
    cout << str1 << endl;
    for (char ch : str1) {
        cout << ch << ' ';
    }
    cout << endl;

    string str2 = " the slithy toves";
    string str3 = str1 + str2;
    cout << str3 << endl;
}
```

- `#include <string>` needed
- full name: `std::string`
- `string` literals are in double quotes
- `char` literals use single quotes
- `char` type holds a single character
- `char` is a type of integer
- strings can be output with the `<<` operator
- strings can be looped over (like vectors)
- plus (`+`) operator concatenates strings

```
T w a s b r i l l i g a n d
Twas brillig and the slithy toves
```

C++ strings are mutable

```
int main() {
    string animal = "bat";

    animal[0] += 1;

    cout << animal << endl;
}
```

What is the type of `animal[0]`?

```
int main() {
    string animal = "bat";

    animal[0] += 1;

    cout << animal << endl;
}
```

What will be output by the code?

```
int main() {
    string animal = "bat";

    animal[0] += 1;

    cout << animal << endl;
}
```

C++ strings are mutable

```
int main() {
    string animal = "bat";

    animal[0] += 1;

    cout << animal << endl;
}
```

```
cat
```

- elements of a string are of type `char`
- chars are ints so we can do arithmetic with them
- chars hold the ascii value of the character
- strings are mutable (we can modify their contents)

File I/O

- `#include <fstream>`
- functionality includes:
  - opening and closing files
  - testing for successful open
  - reading input
  - looping over lines in a file

File input example: read token

```
int main() {
    ifstream jab("jabberwocky");
    if (!jab) {
        cerr << "failed to open jabberwocky";
        exit(1);
    }

    string something;
    jab >> something;
    cout << something << endl;
    jab.close();

}
```

- `ifstream` used for working with input file
- `ifstream` evaluates to `0` when file opening fails
- full name: `std::ifstream`

jabberwocky

Twas brillig and the slithey toves
did gyre and gymble in the wabe.
All mimsy were the borogroves
and the momeraths outgrabe.

Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the JubJub bird and shun
The frumious Bandersnatch!

...

```
Twas
```

File input example: read line

```
int main() {
    ifstream jab("jabberwocky");
    if (!jab) {
        cerr << "failed to open jabberwocky";
        exit(1);
    }

    string something;
    getline(jab, something);
    cout << something << endl;
    jab.close();

}
```

jabberwocky

Twas brillig and the slithey toves
did gyre and gymble in the wabe.
All mimsy were the borogroves
and the momeraths outgrabe.

Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the JubJub bird and shun
The frumious Bandersnatch!

...

Twas brillig and the slithey toves

File input example: read token by token

```
int main() {
    ifstream jab("jabberwocky");
    if (!jab) {
        cerr << "failed to open jabberwocky";
        exit(1);
    }

    string something;
    while (jab >> something) {
        cout << something << endl;
    }
    jab.close();
}
```

- read operation located in while condition
- loop stops when nothing left to read
- each read operations gets the next **whitespace delimited** token
  - `' '` (space)
  - `\n` (newline)
  - `\t` (tab)

Twasbrilligandtheslitheytoves ...

File input example: creating a vector of lines

```
int main() {
    ifstream jab("jabberwocky");
    string line;
    vector<string> lines;

    while (getline(jab, line)) {
        lines.push_back(line);
    }

    jab.close();

    // print the contents of the vector
    for (size_t i = 0; i < lines.size(); ++i) {
        cout << lines[i] << endl;
    }

    // print them in reverse?
    for (size_t i = lines.size(); i > 0 ; --i) {
        cout << lines[i-1] << endl;
    }

}
```

- test of open omitted to save space
- file closed as soon as contents read
- `while` loop continues so long as `getline()` successfully extracts line
- full name: `std::getline()`

File input example: accessing each character in line

```
int main() {
     ifstream jab("jabberwocky");
     string line;
     vector<string> lines;

     while (getline(jab, line)) {
          lines.push_back(line);
     }
     jab.close();
     // print the contents of the vector
     for (size_t i = 0; i < lines.size(); ++i) {
         for (size_t j = 0; j < lines[i].size(); ++j) {
             cout << lines[i][j] << ' ';
         }

     }
     cout << endl;
}
```

```
Twas brillig and the slithey toves
did gyre and gymble in the wabe.
All mimsy were the borogroves
and the momeraths outgrabe.

Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the JubJub bird and shun
The frumious Bandersnatch!

...
```

2D vector of vectors

```
int main() {
    const int ROWS = 10;
    const int COLS = 10;
    vector<vector<int>> mat;

    for (int row = 0; row < ROWS; ++row) {
        mat.push_back(vector<int>(COLS));
        for (int col = 0; col < COLS; ++col) {
            mat[row][col] = row * ROWS + col;
        }
    }

    for (int row = 0; row < ROWS; ++row) {
       for (int col = 0; col < COLS; ++col) {
           cout << mat[row][col] << ' ';
       }
       cout << endl;
    }

}
```

- filling 2d “matrix” with values from 0 to 99
- constants commonly in all caps
- **Note**: vector created and pushed onto `mat`

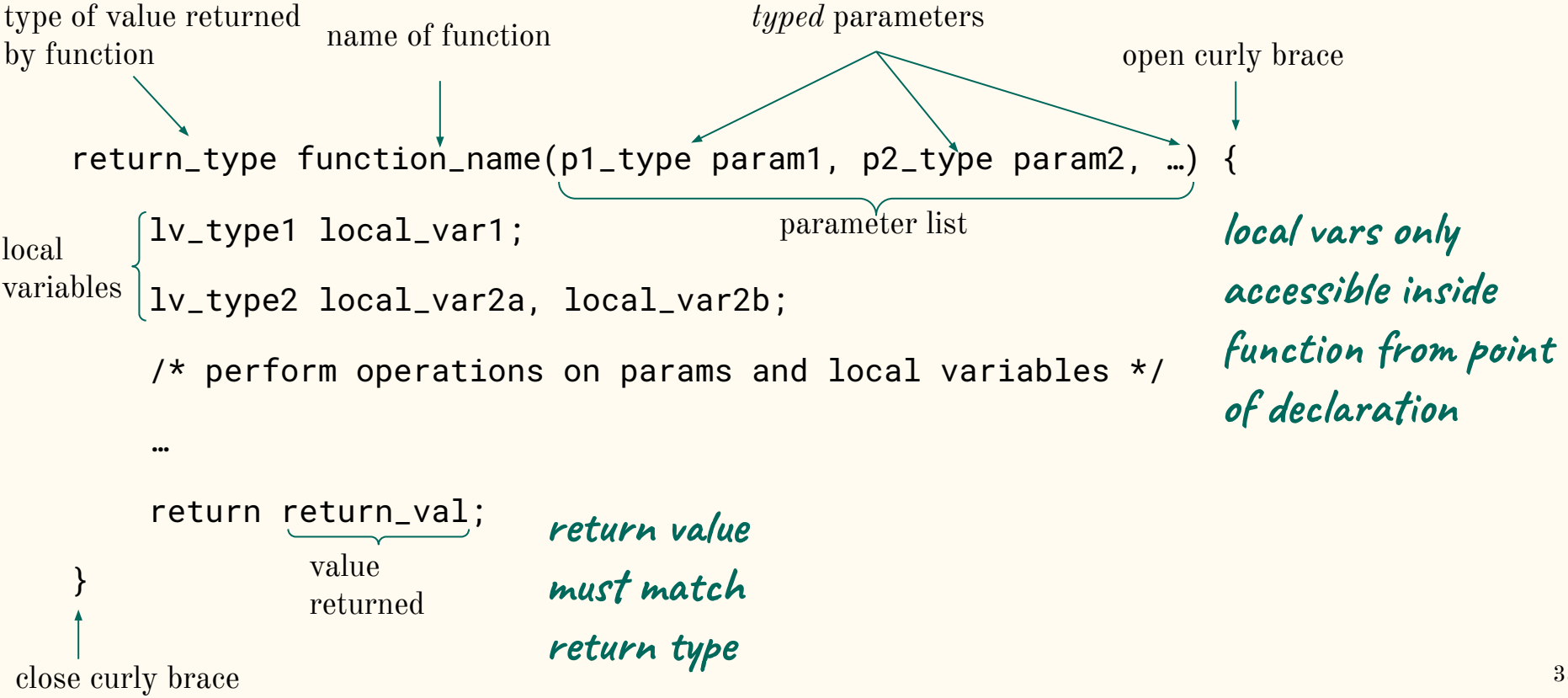
```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

Functions

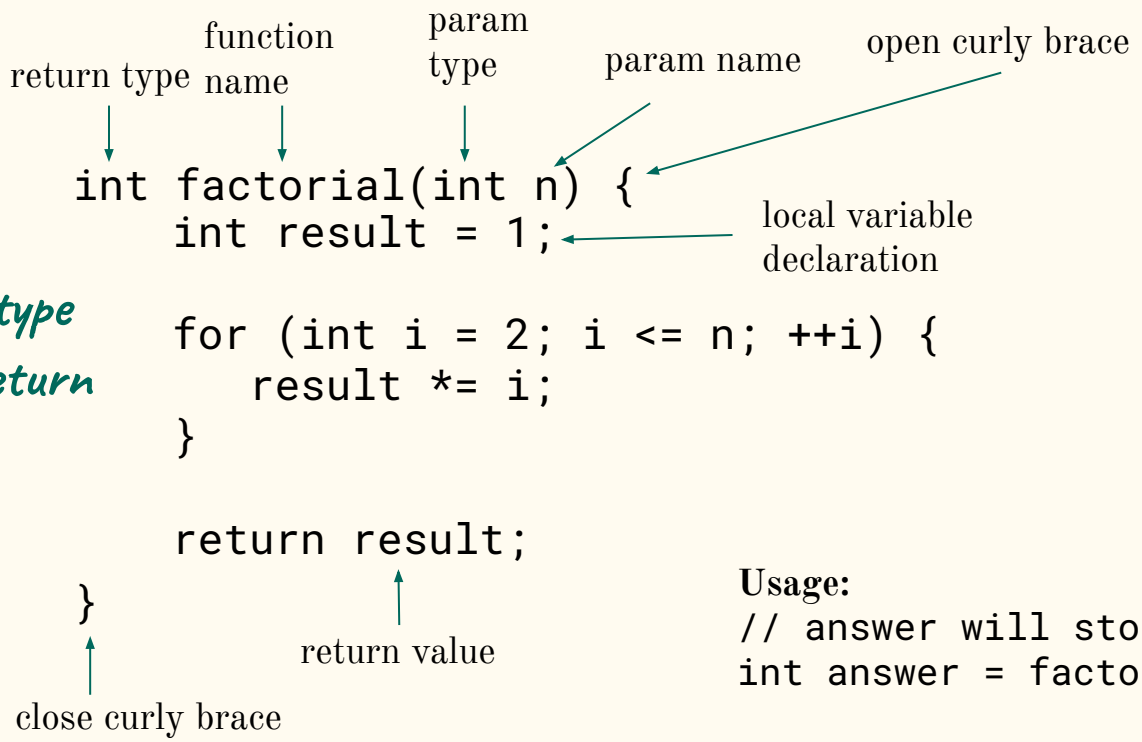
Functions

- help organize code
  - break problems into smaller pieces
  - produces easier to read code
- enable code reuse
- `main()` function starting point of `C++` programs

C++ functions


type of value returned by function
name of function
typed parameters
open curly brace
return_type function_name(p1_type param1, p2_type param2, …) {
parameter list
local variables
lv_type1 local_var1;
lv_type2 local_var2a, local_var2b;
/* perform operations on params and local variables */
…
return return_val;
value returned
}
close curly brace
local vars only accessible inside function from point of declaration
return value must match return type

C++ functions: an example



Note: return type and type of return value match

Usage:

```
// answer will store value 120 (5!)
int answer = factorial(5);
```

C++ functions: a full example

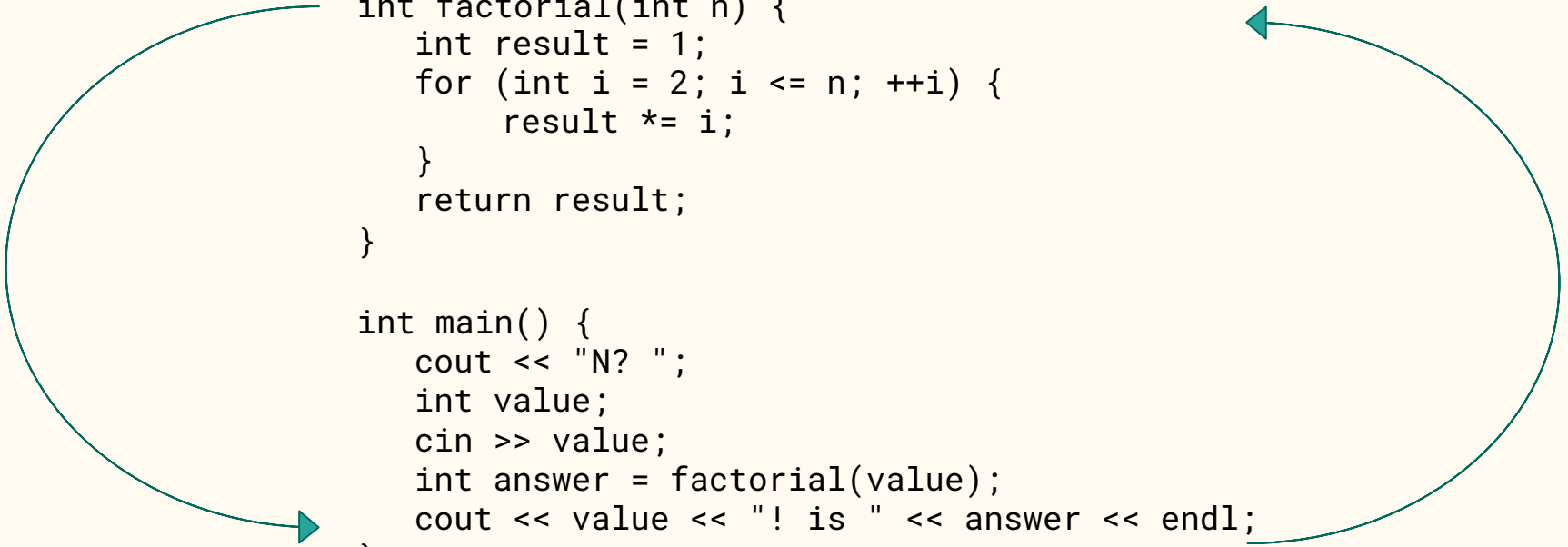
```
#include <iostream>
using namespace std;

int factorial(int n) {
   int result = 1;
   for (int i = 2; i <= n; ++i) {
        result *= i;
   }
   return result;
}

int main() {
   cout << "N? ";
   int value;
   cin >> value;
   int answer = factorial(value);
   cout << value << "! is " << answer << endl;
}
```

```
N? 4
4! is 24
```

Using function prototypes

function must be declared before using

```
#include <iostream>
using namespace std;

int factorial(int n) {
   int result = 1;
   for (int i = 2; i <= n; ++i) {
        result *= i;
   }
   return result;
}

int main() {
   cout << "N? ";
   int value;
   cin >> value;
   int answer = factorial(value);
   cout << value << "! is " << answer << endl;
}
```



Using function prototypes

```
#include <iostream>
using namespace std;

int main() {
   cout << "N? ";
   int value;
   cin >> value;
   int answer = factorial(value);
   cout << value << "! is " << answer << endl;
}

int factorial(int n) {
   int result = 1;
   for (int i = 2; i <= n; ++i) {
        result *= i;
   }
   return result;
}
```

function must be declared before using

factorial () function unknown to compiler when invoked

```
error: use of undeclared identifier 'factorial'
```

Using function prototypes

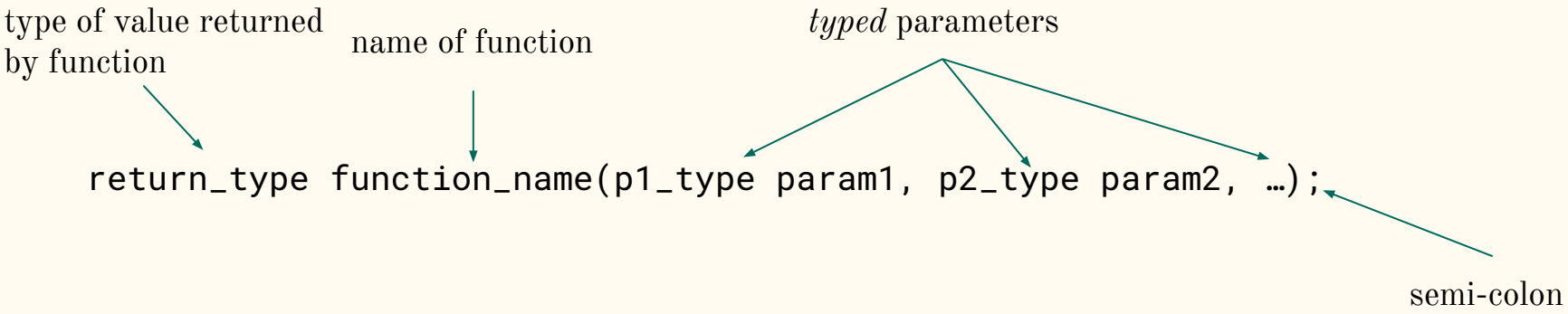
```
#include <iostream>
using namespace std;

int factorial (int n);

int main() {
   cout << "N? ";
   int value;
   cin >> value;
   int answer = factorial(value);
   cout << value << "! is " << answer << endl;
}

int factorial(int n) {
   int result = 1;
   for (int i = 2; i <= n; ++i) {
       result *= i;
   }
   return result;
}
```

function prototype

informs compiler a definition will be provided

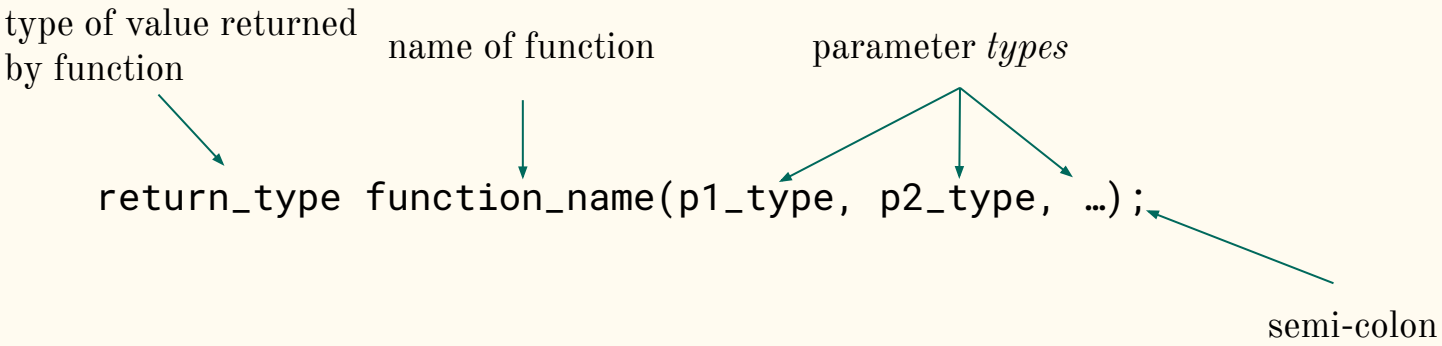
Using function prototypes

- function prototypes provide important information
  - return type
  - function name
  - function parameter *type(s)*



Using function prototypes

- function prototypes provide important information
  - return type
  - function name
  - function parameter *type(s)*



Using function prototypes

```
#include <iostream>
using namespace std;

int factorial (int);

int main() {
   cout << "N? ";
   int value;
   cin >> value;
   int answer = factorial(value);
   cout << value << "! is " << answer << endl;
}

int factorial(int n) {
   int result = 1;
   for (int i = 2; i <= n; ++i) {
       result *= i;
   }
   return result;
}
```

Functions without a return type

- functions not required to return value
- return type can be defined as `void`
  - indicates that function will not return a value

```
// helper function for displaying n!
void display_factorial(int n) {
    cout << n << "! is " << factorial(n) << endl;
}
```

more explicitly ↓

```
void display_factorial(int n) {
    cout << n << "! is " << factorial(n) << endl;
    return;
}
```

Overloading functions

- functions can have same name in C++
  - parameter lists must differ

```
void print_int(int an_int) {
     cout << "Int value is " << an_int << endl;
}
void print_string(string a_str) {
     cout << "String value is " << a_str << endl;
}
```

alternatively...

```
void print(int an_int) {
     cout << "Int value is " << an_int << endl;
}
void print(string a_str) {
     cout << "String value is " << a_str << endl;
}
```

Passing arguments to functions

- `C++` allows programmer to determine how arguments provided to function
- 3 options
  - pass-by-value (default)
  - pass-by-reference
  - pass-by-constant-reference

Pass-by-value

```
// swaps two integers
void swap_nums(int num1, int num2) {
    int tmp;

    tmp = num1;
    num1 = num2;
    num2 = tmp;

}

int main() {
    int num_1 = 5, num_2 = -3;

    cout << "Before swap…" << endl;
    cout << "Number 1: " << num_1 << endl;
    cout << "Number 2: " << num_2 << endl << endl;

    swap_nums(num_1, num_2);

    cout << "After swap.." << endl;
    cout << "Number 1: " << num_1 << endl;
    cout << "Number 2: " << num_2 << endl;
}
```

- arguments to functions are passed by value (default)
- parameter assigned a copy of the value passed

```
Before swap...
Number 1: 5
Number 2: -3

After swap..
Number 1: 5
Number 2: -3
```

not what we want

Pass-by-reference

```
// swaps two integers
void swap_nums(int num1, int num2) {
    int tmp;

    tmp = num1;
    num1 = num2;
    num2 = tmp;
}
```

Pass-by-reference

```
// swaps two integers
void swap_nums(int& num1, int& num2) {
    int tmp;

    tmp = num1;
    num1 = num2;
    num2 = tmp;

}

int main() {
    int num_1 = 5, num_2 = -3;

    cout << "Before swap…" << endl;
    cout << "Number 1: " << num_1 << endl;
    cout << "Number 2: " << num_2 << endl << endl;

    swap_nums(num_1, num_2);

    cout << "After swap.." << endl;
    cout << "Number 1: " << num_1 << endl;
    cout << "Number 2: " << num_2 << endl;
}
```

- alternative naming of objects known as a *reference*
  - reference can be thought of as an alias
- ampersand (`&`) indicates a reference to argument
- changes made inside function extend to caller

```
Before swap...
Number 1: 5
Number 2: -3

After swap..
Number 1: -3
Number 2: 5
```

Pass-by-reference

```
// swaps two integers
void swap_nums(int& num1, int& num2) {
    int tmp;

    tmp = num1;
    num1 = num2;
    num2 = tmp;

}

int main() {

    swap_nums(5, -3);
}
```

reference parameters need arguments with a memory address (*l-value*)

```
error: no matching function for call to 'swap_nums'
    swap_nums(5, -3);
    ^~~~~~~~~
note: candidate function not viable: expects an l-value for 1st
argument
void swap_nums(int& num1, int& num2) {
     ^
1 error generated.
```

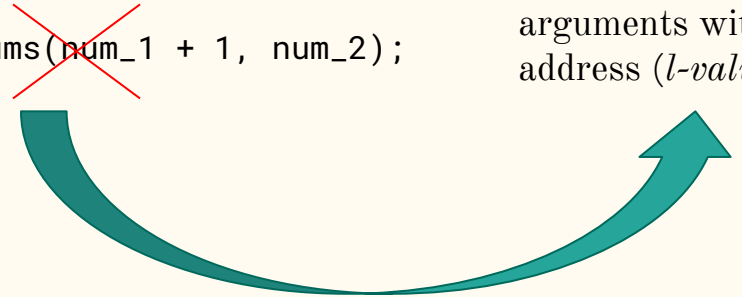
Pass-by-reference

```
// swaps two integers
void swap_nums(int& num1, int& num2) {
    int tmp;

    tmp = num1;
    num1 = num2;
    num2 = tmp;
}
```

reference parameters need arguments with a memory address (*l-value*)

Pass-by-reference

```
// swaps two integers
void swap_nums(int& num1, int& num2) {
    int tmp;

    tmp = num1;
    num1 = num2;
    num2 = tmp;
}

int main() {
    int num_1 = 5, num_2 = -3;

    swap_nums(num_1 + 1, num_2);
}
```

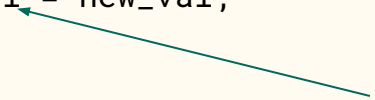


reference parameters need arguments with a memory address (*l-value*)

References and `const` types

Functions with vector parameters

```
void update_vector(vector<int> int_vec, int new_val) {
     for (int val : int_vec) {
          val = new_val;
     }
}

 int main() {
     vector<int> nums{0, 1, 2, 3};

     update_vector(nums, -1);
     for (int num; nums) {
          cout << num << ' ';
     }
     cout << endl;
 }
```



`val` assigned a copy of each value in `int_vec`

```
0 1 2 3
```

Object references

```
void update_vector(vector<int> int_vec, int new_val) {
    for (int& val : int_vec) {
        val = new_val;
    }
}

int main() {
    vector<int> nums{0, 1, 2, 3};

    update_vector(nums, -1);
    for (int num; nums) {
        cout << num << ' ';
    }
    cout << endl;
}
```

& is reference operator

```
0 1 2 3
```

Object references

```
void update_vector(vector<int> int_vec, int new_val) {
    for (int& val : int_vec) {
        val = new_val;
    }
}

int main() {
    vector<int> nums{0, 1, 2, 3};

    update_vector(nums, -1);
    for (int num; nums) {
        cout << num << ' ';
    }
    cout << endl;
}
```

copy of original vector

```
0 1 2 3
```

Object references

```
void update_vector(vector<int>& int_vec, int new_val) {
    for (int& val : int_vec) {
        val = new_val;
    }
}

int main() {
    vector<int> nums{0, 1, 2, 3};

    update_vector(nums, -1);
    for (int num; nums) {
        cout << num << ' ';
    }
    cout << endl;
}
```

the **original** vector (*pass-by-reference*)

```
-1 -1 -1 -1
```

Constant object references

- object copies passed as function arguments (pass-by-value)
- operating on a copy useful when modifications unwanted/disallowed

```
void update_vector(vector<int>& int_vec, int new_val) {
    for (int& val : int_vec) {
        val = new_val;
    }
}

int main() {
    vector<int> nums{0, 1, 2, 3};

    update_vector(nums, -1);
    for (int num; nums) {
        cout << num << ' ';
    }
    cout << endl;
}
```

Constant object references

- object copies passed as function arguments (pass-by-value)
- operating on a copy useful when modifications unwanted/disallowed

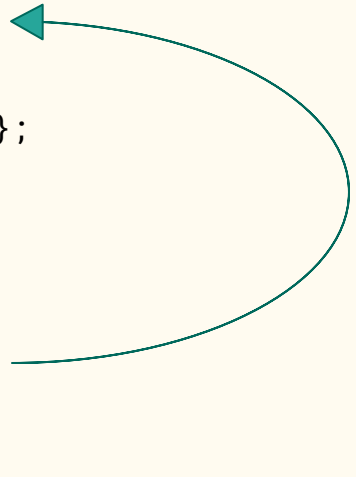
```
void update_vector(vector<int>& int_vec, int new_val);
```

```
int main() {
    vector<int> nums{0, 1, 2, 3};

    update_vector(nums, -1);
    for (int num; nums) {
         cout << num << ' ';
    }
    cout << endl;
}
```

Constant object references

- object copies passed as function arguments (pass-by-value)
- operating on a copy useful when modifications unwanted/disallowed

```
void update_vector(vector<int>& int_vec, int new_val);

void print_vector(vector<int> int_vec);


int main() {
    vector<int> nums{0, 1, 2, 3};

    update_vector(nums, -1);
    for (int num; nums) {
         cout << num << ' ';
    }
    cout << endl;
}
```



Constant object references

- object copies passed as function arguments (pass-by-value)
- operating on a copy useful when modifications unwanted/disallowed

```
void update_vector(vector<int>& int_vec, int new_val);

void print_vector(vector<int> int_vec);
```

```
int main() {
    vector<int> nums{0, 1, 2, 3};

    update_vector(nums, -1);
    print_vector(nums);
}
```

Constant object references

- object copies passed as function arguments (pass-by-value)
- operating on a copy useful when modifications unwanted/disallowed

```
void update_vector(vector<int>& int_vec, int new_val);

void print_vector(vector<int> int_vec) {
    for (int num: int_vec) {
         cout << num << ' ';
    }
    cout << endl;
}
int main() {
    vector<int> nums{0, 1, 2, 3};

    update_vector(nums, -1);
    print_vector(nums);
}
```

```
-1 -1 -1 -1
```

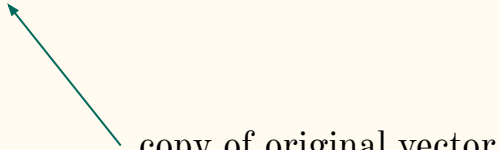
still works!!!

What if nums contained many more integers?

Constant object references

- object copies passed as function arguments (pass-by-value)
- operating on a copy useful when modifications unwanted/disallowed

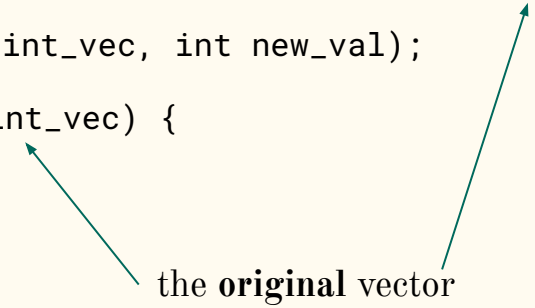
```
void update_vector(vector<int>& int_vec, int new_val);

void print_vector(vector<int> int_vec) {
    for (int num: int_vec) {
        cout << num << ' ';
    }
    cout << endl;
}
int main() {
    vector<int> nums(1000000000000, 0);

    update_vector(nums, -1);
    print_vector(nums);
}
```



Constant object references

- object copies passed as function arguments
- operating on a copy useful when modifications unwanted/disallowed

```
void update_vector(vector<int>& int_vec, int new_val);

void print_vector(vector<int>& int_vec) {
    for (int num: int_vec) {
        cout << num << ' ';
    }
    cout << endl;
}
int main() {
    vector<int> nums(1000000000000, 0);

    update_vector(nums, -1);
    print_vector(nums);
}
```



reference allows modification of original object

Constant object references

- object copies passed as function arguments (pass-by-value)
- operating on a copy useful when modifications unwanted/disallowed

```
void update_vector(vector<int>& int_vec, int new_val);

void print_vector(const vector<int>& int_vec) {
    for (int num: int_vec) {
        cout << num << ' ';
    }
    cout << endl;
}
int main() {
    vector<int> nums(1000000000000, 0);

    update_vector(nums, -1);
    print_vector(nums);
}
```

the **original** vector
(pass-by-constant-reference)

modifications prohibited

`const` keyword

- specifies that object cannot be modified
- `const + &`
  - access to original object
  - no modification permitted