Character arrays (c-strings)

- Stored as an array of type `char`

  e.g. `char mychararr[] = "Hello";`

  is actually stored with the <u>null terminator</u>, `\0`, at the end:

H	e	l	l	o	\0

- In the "Hello" string example, the array is of size 6 bytes, and is sometimes referred to as a "null terminated character string".
- The null character is very important for many of the string manipulation functions.

Exception (error) handling - Motivation

- Programs need to handle unexpected behavior (exceptions) in a controlled and centralized manner

```
Exception: n1>100
Still running
```

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

int bar(int n) {
    if(n > 200) return -1;
    return 0;
}
int foo(int n1, int n2) {
    if (n1 > 100) return -1;
    if(bar(n2)==-1) return -2;
    return 0;
}

int main() {

    int err = foo(150, 150);
    switch (err) {
    case -1:
        cout << "Exception: n1>100" << endl;
        break;
    case -2:
        cout << "Exception: n2>200" << endl;
        break;
    }

    cerr << "Still running\n";

}
```

- This is the outer call
- If any of the nested calls returns an error, we need to propagate it to the outer-most call.
- An exception handler that prints a corresponding message, then exits.

Exception (error) handling - Motivation

- Propagating the return code through a higher level of nested calls becomes very cumbersome

```
Exception: n1>100
Still running
```

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

int bar(int n) {
    if(n > 200) return -1;
    return 0;
}
int foo(int n1, int n2) {
    if (n1 > 100) return -1;
    if(bar(n2)==-1) return -2;
    return 0;
}

int main() {

    int err = foo(150, 150);
    switch (err) {
    case -1:
        cout << "Exception: n1>100" << endl;
        break;
    case -2:
        cout << "Exception: n2>200" << endl;
        break;
    }

    cerr << "Still running\n";

}
```

- Some translation involved here, to propagate unique return codes for the centralized exception handler

- **Imagine having 5 or 10 levels of nested calls**
  - This is not atypical.
- **There must be a better way!**

Exception (error) handling - Motivation

C++ provides a better way!

- Propagating nested error codes →Throw an exception: send an indication that an error has occurred
- Parsing error codes → Catch/Handle an exception: process the exception;

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

void bar(int n) {
    if(n > 200) throw "Exception in bar(): n>200";
}
void foo(int n1, int n2) {
    if (n1 > 100) throw "Exception in foo(): n>100";
    bar(n2);
}

int main() {

    try {
        foo(150, 150);
    }
    catch (const char* msg) {
        cout << msg << endl;
    }

    cerr << "Still running\n";

}
```

```
Exception in foo(): n>100
Still running
```

Exceptions – Key Words

- `throw` – followed by an argument, is used to throw an exception
- `try` – followed by a block `{ }`, is used to invoke code that throws an exception
- `catch` – followed by a block `{ }`, is used to detect and process exceptions thrown in preceding `try` block.
    - `catch()` takes a **parameter that matches the type thrown**.

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

int main() {
   try
   {
      throw 20;
   }
   catch (int e)
   {
      cout << "An exception occurred. Exception Nr. " <<
            e << '\n';
   }

}
```

Exceptions – Flow of Control

1) A function that throws an exception is called from within a try block

2) If the function throws an exception, the <u>function terminates</u> and the <u>try block is immediately exited</u>.

3) A catch block to process the exception is searched for in the source code immediately following the try block.
   - If a catch block is found that matches the exception thrown, it is executed.
   - If no catch block that matches the exception is found, the program terminates.

Exceptions – example

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

// function that throws an exception
int totalDays(int days, int weeks)
{
    if ((days < 0) || (days > 7))
        // the argument to throw is the character string
        throw "invalid number of days";

    else
        return (7 * weeks + days);
}

int main() {
    int days = -1, weeks = 4;
    try{
        auto totDays = totalDays(days, weeks);
        cout << "Total days: " << days;
    }
    catch (const char* msg){
        cout << "Error: " << msg;
    }
    return 0;
}
```

```
Error: invalid number of days
```

- The argument **types** must match (const char* in this case)
- If an exception is thrown, the lines following the “throw” in the called function, and the rest of the “try” block will not be executed.

Exceptions – example

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

// function that throws an exception
int totalDays(int days, int weeks)
{
    if ((days < 0) || (days > 7))
        // the argument to throw is the character string
        throw "invalid number of days";

    else
        return (7 * weeks + days);
}

int main() {
    int days = 2, weeks = 4;
    try{
        auto totDays = totalDays(days, weeks);
        cout << "Total days: " << days;
    }
    catch (const char* msg){
        cout << "Error: " << msg;
    }
    return 0;
}
```

```
Total days: 2
```

- If an exception is not thrown:
  - Everything within the try block is executed normally (including nested calls)
  - The catch block is skipped

Exceptions not caught

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

// function that throws an exception
int totalDays(int days, int weeks)
{
    if ((days < 0) || (days > 7))
        // the argument to throw is the character string
        throw "invalid number of days";

    else
        return (7 * weeks + days);
}

int main() {
    int days = -1, weeks = 4;
    try{
        auto totDays = totalDays(days, weeks);
        cout << "Total days: " << days;
    }
    catch (char* msg){
        cout << "Error: " << msg;
    }
    return 0;
}
```

```
Exception thrown at 0x00007FFF2DA8CB69 in Lect_01_B.exe:
Microsoft C++ exception: char at memory location
0x00000090C273FBB0.
Unhandled exception at 0x00007FFF2DA8CB69 in
Lect_01_B.exe: Microsoft C++ exception: char at memory
location
```

- If the argument type doesn't match (const char*), we get an unhandled exception!

An exception will not be caught if

- It is thrown from outside of a `try` block
- There is no `catch` block that matches the data type of the thrown exception

If an exception is not caught, the program will terminate

Exception objects

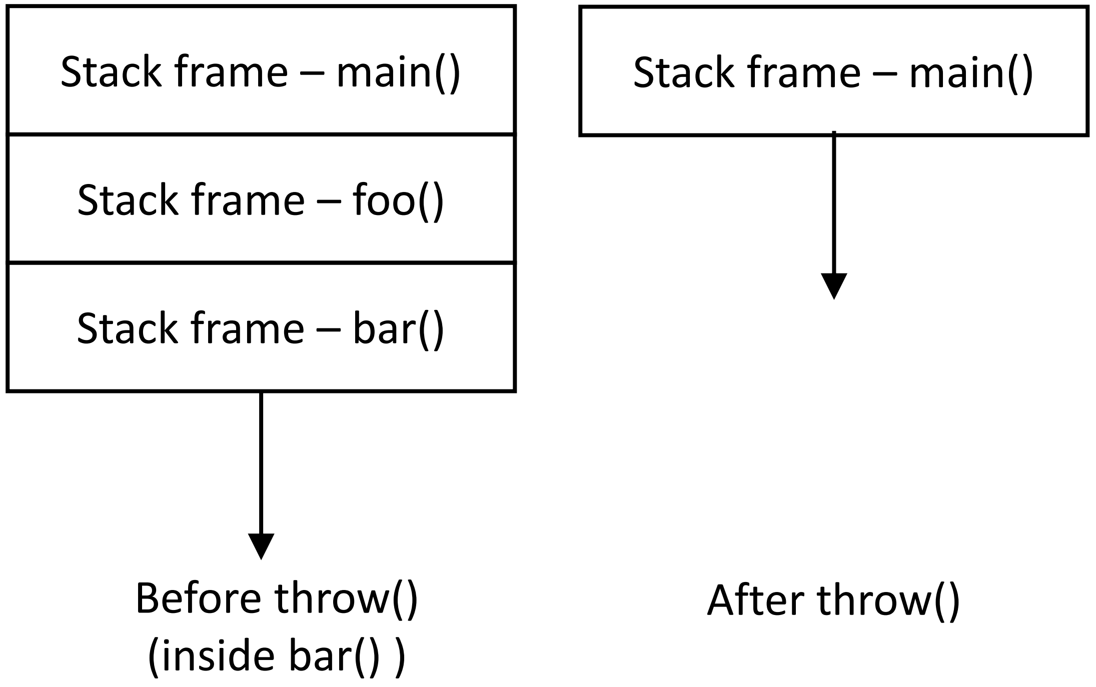
```
#include <iostream>
#include <exception>

using namespace std;

void bar(int n) {

    if (n > 200) {
        throw exception();
    }

}
void foo(int n) { bar(n); }

int main() {

    try {
        foo(300);
    }
    catch (exception& ex) {
        cerr << "Caught exception: " << ex.what() << endl;
    }

    cerr << "Still running\n";
}
```

```
Caught exception: Unknown exception
Still running
```

- An exception may be thrown using an object instead of intrinsic types.
- The object's class may be of type "**exception**" or derived from "exception"
  - Must #include <exception>

Unwinding the stack

- Once an exception is thrown, the program cannot return to throw point.
- All nested calls in `try` block, leading to the `throw` statement, terminate (e.g. both **foo()** and **bar()** in previous example).
- Stack frames are destroyed → This is part of unwinding the call stack
  - If objects were created within the `try` block or any of the nested calls within the try block), they are destroyed (with destructors invoked).
- While unwinding the stack, if a matching `catch` statement is found, the program resumes at that statement.



Exception objects

```
#include <iostream>
#include <exception>

using namespace std;

struct MyException : public exception {
    MyException(int n) : n(n) {}
    const char* what() const override { return "this is MyException";}
    int n = 0;
};

void bar(int n) {

    if (n > 200) {
        throw MyException(n);
    }

}
void foo(int n) { bar(n); }

int main() {

    try {
        foo(300);
    }
    catch (exception& ex) {
        cerr << "Caught exception: " << ex.what() << endl;
    }

    cerr << "Still running\n";
}
```

```
Caught exception: this is MyException
Still running
```

- An exception may be thrown using an object instead of intrinsic types.
- The method what() in class intrinsic may be overridden to implement a different behavior
- The object's class may be of type "**exception**" or **derived from "exception"**
    - Must #include <exception>
- The catch block can handle **"exception" or any of its derived classes**

Exception objects

```
#include <iostream>
#include <exception>

using namespace std;

struct MyException : public exception {
    MyException(int n) : n(n) {}
    const char* what() const override { return "this is MyException";}
    int n = 0;
};

void bar(int n) {

    if (n > 200) {
        throw MyException(n);
    }

}
void foo(int n) { bar(n); }

int main() {

    try {
        foo(300);
    }
    catch (exception ex) {
        cerr << "Caught exception: " << ex.what() << endl;
    }

    cerr << "Still running\n";
}
```

```
Caught exception: Unknown exception
Still running
```

- An exception may be thrown using an object instead of intrinsic types.
- The method what() in class intrinsic may be overridden to implement a different behavior
- The object's class may be of type "**exception**" or **derived from "exception"**
  - Must #include <exception>
- The catch block can handle **"exception" or any of its derived classes**
  - **If we pass by value, we loose the polymorphic behavior**

Exception objects

```
#include <iostream>
#include <exception>

using namespace std;

struct MyException : public exception {
    MyException(int n) : n(n) {}
    const char* what() const override { return "this is MyException"; }
    int n = 0;
};

void bar(int n) {

    if (n > 200) {
        throw MyException(n);
    }

}
void foo(int n) { bar(n); }

int main() {

    try {
        foo(300);
    }
    catch (...) {
        cerr << “default exception\n";
    }

    cerr << "Still running\n";
}
```

```
default exception
Still running
```

- A **catch-all** may be used instead

Exception objects

```
#include <iostream>
#include <exception>

using namespace std;

struct MyException : public exception {
    MyException(int n) : n(n) {}
    const char* what() const override {return "this is MyException";}
    int n = 0;
};

void bar(int n) {

    if (n > 200) {
        throw MyException(n);
    }

}
void foo(int n) { bar(n); }

int main() {

    try {
        foo(300);
    }
    catch (exception& ex) {
        cerr << "Caught exception: " << ex.what() << endl;
    }
    catch (const MyException& my) {
        cerr << "Caught a MyException: " << my.what() << endl;
        cerr << "my.n: " << my.n << endl;
    }
    catch (...) {
        cerr << "default exception\n";
    }

    cerr << "Still running\n";
}
```

```
Caught exception: this is MyException
Still running
```

Multiple handlers (i.e., catch expressions) can be chained.

The first handler whose argument type matches, is executed.

Exception objects

```
#include <iostream>
#include <exception>
#include <stdexcept>

using namespace std;

void bar(int n) {
    if (n > 200) {
        throw out_of_range("n>200");
    }

}
void foo(int n) { bar(n); }

int main() {

    try {
        foo(300);
    }
    catch (out_of_range oor) {
        cerr << "Caught out of range: " << oor.what() << endl;
    }

    cerr << "Still running\n";
}
```

```
Caught out of range: n>200
Still running
```

- We can throw an object of type “out_of_range”
  - Must #include <stdexcept> if not already included
  - “out_of_range” is derived from “logic_error” which is derived from “exception”

Exceptions thrown in STL – bounds check

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> v;

    v.at(17) = 42;
    cout << v.at(17) << endl;


}
```

```
Exception thrown at 0x00007FFF2DA8CB69 in Lect_01_B.exe:
Microsoft C++ exception: std::out_of_range at memory
location 0x000000DCF979F8F0.
Unhandled exception at 0x00007FFF2DA8CB69 in
Lect_01_B.exe: Microsoft C++ exception: std::out_of_range at
memory location 0x000000DC
```

- out-of-range access
- The implementation of vector throws an exception, which we have not caught, so our program terminated!
- <u>vector::at() has an out of bounds check</u>

<u>A side note:</u>

- vector::operator[] does not have out of bounds check
- Thus, should we have used `v[17]=42` instead of `v.at(17)=42`, we would've gotten a segmentation fault because the pointer to the vector's underlying array is nullptr at this point.
- Using "v.push_back(28); v[17]=42;" would've probably passed, but would still be erroneous because we would be still outside the bounds of our vector

Exceptions thrown in STL – bounds check

```
#include <iostream>
#include <vector>
#include <exception>

using namespace std;

int main() {
    vector<int> v;

    try {
        v.at(17) = 42;
        cout << v.at(17) << endl;
    }
    catch(exception &ex) {
        cout << "Caught an exception: " << ex.what() << endl;
    }
}
```

- Throws the std::out_of_range exception

Assertions

```
#include <iostream>

#undef NDEBUG
#include <cassert>

using namespace std;

void bar(int n) {
    assert(n <= 200);
}
void foo(int n) { bar(n); }

int main() {
    foo(300);
    cerr << "still running" << endl;
}
```

```
Assertion failed: n <= 200, file
C:\Dropbox\CS2124_OOP_2023_Spring\lect_co
de\12.Exceptions and
Assertions\exceptions_om.cpp, line 14
```

- Assertions are a C-language debug utility:
  - If condition is true → normal execution
  - If condition is false → program **prints** a statement to standard error and **aborts**
- Assertions may be used for testing and debugging by ensuring program produces correct results at various points within its execution.

- The symbol NDEBUG must NOT be defined
  - If NDEBUG exists, then the code implementing assert() is removed
  - Some IDE's define NDEBUG by default, so we either need to change the settings or undfine the NDEBUG, for assert functionality to be compiled-in
- #include <cassert> results in including the c-language header file <assert.h>

Assertions

```
#include <iostream>

#undef NDEBUG
#include <cassert>

using namespace std;

void bar(int n) {
    assert(n <= 200);
}
void foo(int n) { bar(n); }

int main() {
    foo(150);
    cerr << "still running" << endl;
}
```

```
still running
```

- Since the value is <=200 → assert condition is true → normal execution

- Assertions are a C-language debug utility:
  - <u>If condition is true</u> → normal execution
  - <u>If condition is false</u> → program **prints** a statement to standard error and **aborts**

- Assertions may be <u>used for testing and debugging</u> by ensuring program produces correct results at various points within its execution.

Assertions

```
#include <iostream>

#define NDEBUG
#include <cassert>

using namespace std;

void bar(int n) {
    assert(n <= 200);
}
void foo(int n) { bar(n); }

int main() {
    foo(300);
    cerr << "still running" << endl;
}
```

```
still running
```

- Assert() functionality is not compiled-in → not assertions will take place

- Assertions are a C-language debug utility:
  - If condition is true → normal execution
  - If condition is false → program **prints** a statement to standard error and **aborts**

- Assertions may be used for testing and debugging by ensuring program produces correct results at various points within its execution.