C++ 1 Les 6

Friends

Slicing

Exception Handling

Leerdoelen

Aan het eind van deze workshop kun je:

- Friend classes en functions gebruiken in C++
- Slicing herkennen en voorkomen in C++
- Exception handling correct toepassen in C++
- Code schrijven die geen memory leaks geeft als er een exception optreedt in de constructor.

Friends

Friend Class/Function

- Een **friend class** heeft toegang tot alle private en protected delen van de class.
- Bij een **friend function** heeft alleen die functie toegang tot alle private en protected delen van de class.

"Friends can touch your private parts."

Friend Class Voorbeeld

```
class A
{
  private:
  int a,b;
  friend class B;
}
```

Friend Function Voorbeeld

```
class A
{
  private:
  int a,b;
  friend int B::Fun();
}
```

Slicing

Slicing (1)

```
class A {
      int i;
};

class B : public A {
      int j;
};

int main()
{
   B b;
   A a = b; //Slicing, maar kan geen kwaad
}
```

Slicing (2)

```
class A {
     int i;
};

class B : public A {
     int j;
};

int main()
{
  B b1;
  b1.i = b1.j = 1;

  B b2;
  b2.i = b2.j = 2;

  A& A_ref = b1;
  A_ref = b2;
}
```

Wat is de waarde van b1.i en b1.j ?

Slicing (2)

```
class A {
     int i;
};

class B : public A {
     int j;
};

int main()
{
  B b1;
  b1.i = b1.j = 1;

  B b2;
  b2.i = b2.j = 2;

  A& A_ref = b1;
  A_ref = b2; //Slicing: b1.i=2 en b1.j=1
}
```

Exception Handling

Simpele Exception Handling (1)

```
int main ()
{
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "Exception nr" << e << '\n';
  }
  return 0;
}
```

Je gooit een exception van het type “int”

Hier vang je int exceptions af

Simpele Exception Handling (2)

```
try
{
  //code
}
catch (int param)
{ cout << "int exception"; }
catch (char param)
{ cout << "char exception"; }
catch (...)
{ cout << "default exception"; }
```

Iedere exception die nog niet is afgevangen

Standard Exception

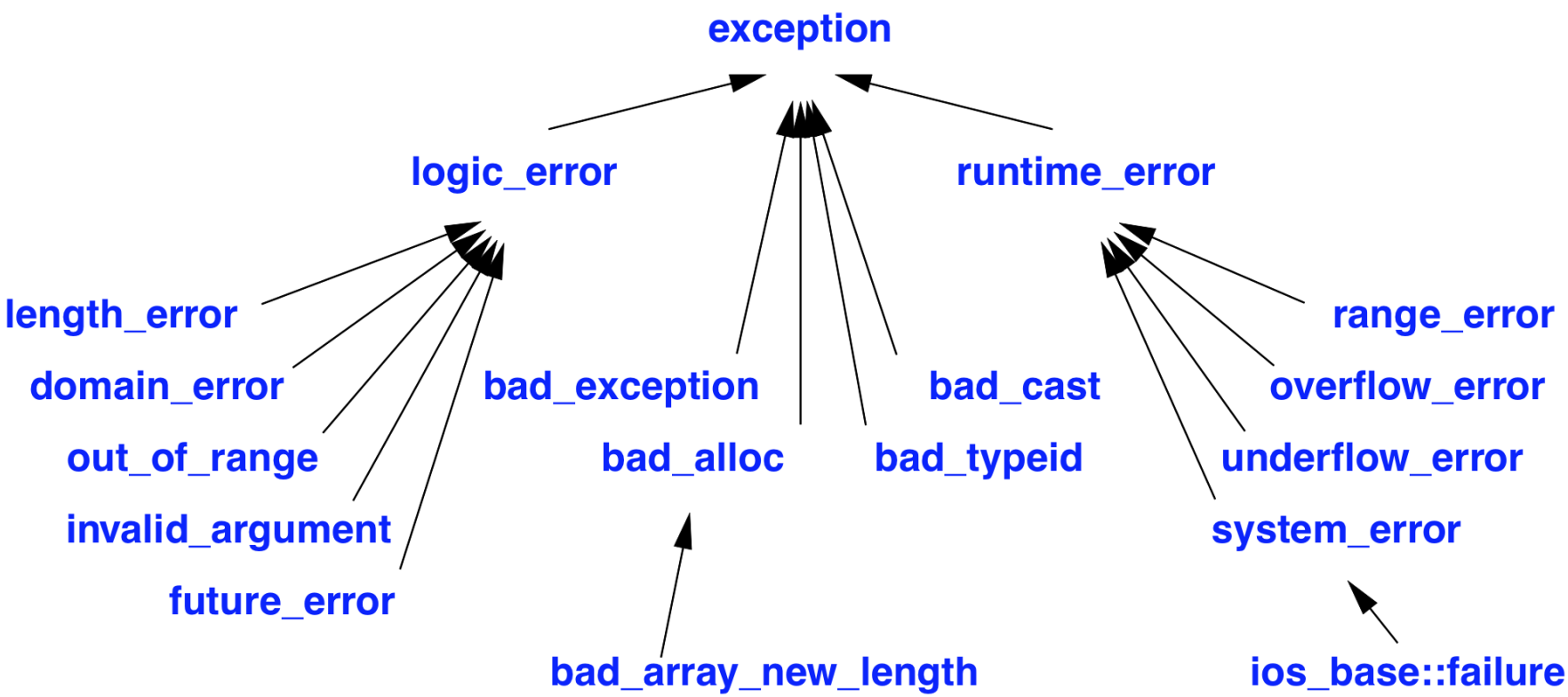
- `std::exception`
- Komt uit C++ Standard Library
- `#include <exception>`
- Beschrijving van de exception in de `what()` virtual function
- Alle exceptions uit de Standard Library gooien exceptions afgeleid van deze klasse

Standard Exception (voorbeeld)

```
#include <cstdlib>
#include <iostream>

int main(int argc, const char* argv[])
{
    int exit_code = EXIT_SUCCESS;
    try {
        // ...
    } catch (const std::exception& ex) {
        std::cerr << argv[0] << ": " << ex.what() << '\n';
        exit_code = EXIT_FAILURE;
    }
    return exit_code;
}
```

Standard Exception Hierarchy



Standard Exception Voorbeeld

```
class myexception :
  public exception
{
  virtual const char* what() const noexcept
  {
    return "My exception happened";
  }
}

int main(){
try {
  throw myexception();
}
catch (const exception& e) {
  cerr << e.what() << '\n';
}
```

Reference
(anders mogelijke bad_alloc
door copy constructor)

Geen *new*
(anders mogelijke
bad_alloc)

Exception Klasse (1)

```
class exception {
public:
  exception () throw();
  exception (const exception&) throw();
  exception& operator= (const exception&) throw();
  virtual ~exception() throw();
  virtual const char* what() const throw();
}
```

Je mag geen exceptions meer gooien in deze functies.

C++98, deprecated!

Exception Klasse (2)

```
class exception {
public:
  exception () noexcept;
  exception (const exception&) noexcept;
  exception& operator= (const exception&) noexcept;
  virtual ~exception();
  virtual const char* what() const noexcept;
}
```

C++11

Constructor Exceptions

- Bij exception in constructor: destructor van de klasse zelf wordt niet aangeroepen, maar wel de destructors van zijn base classes. Waarom?
- Object is **niet** geïnstantieerd.
- Is niet verkeerd. Een throw kan een keuze zijn om aan te geven dat het maken of initialiseren niet correct is gegaan.

Constructor Exception Voorbeeld

```
class A
: public B, public C
{
public:
    D sD;
    E sE;

    A()
    :B(),C(),sD(),sE()
    {
    }
};
```

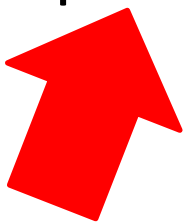
Volgorde:

```
B
C (exception)
~B
```

```
B
C
sD
sE (exception)
~sD
~C
~B
```

Destructor Exceptions

- Geen throw gebruiken in destructor:
  - Bij normale deallocation volgen potentiele memory leaks
  - Bij stack unwinding na een eerdere exception zal bij een nieuwe exception de applicatie termineren



Hier kun je omheen programmeren,
maar het is zo zeldzaam dat, tenzij je
Dit persé nodig hebt, het beter is
om het niet te doen.

Exception Unsafe Code (1)

```
void foo() {
    throw "Exception";
}

void bar() {
    int* pi = new int(0);
    foo();
    delete pi;
}

int main() {
    try {
        bar();
    }
    catch(...) {
    }
}
```

Wordt niet aangeroepen dus:
memory leak

Exception Safe Code (lelijk)

```
void bar() {
  int* pi = nullptr;
  try {
      pi = new int(0);
      foo();
      delete pi;
  } catch (...) {
      delete pi;
      throw; // rethrow!
  }
}
```

RAII
(Resource Acquisition is Initialization)

```
template<typename T>
class smart_ptr {
public:
  smart_ptr(T* ding) { p = ding; }
  ~smart_ptr() { delete p; }

private:
  T* p;
};
```

Code wordt nu clean + correct

```
void bar() {
   std::unique_ptr<int> pi(new int(0));
   foo();
}
```

- Standard Library kent smart pointers die je kunt gebruiken:
  - std::unique_ptr
  - std::shared_ptr

Conclusies

- throw exception object (géén **new**)
- catch exception object **reference**

- gebruik **smart pointers**
- schrijf eigen **RAII-classes** bij aanmaken/opruimen van iets