Basic C++

2

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

http://gsd.web.elte.hu

Expressions

- Expressions: operands(literals/variables) with operators
- Operators have
  - Type and value category
  - Determined by precedence and associativity

```cpp
#include <iostream>

int main()
{
    int t[] = {1,2,3,4}; // 4 element integer array
    int *p = &t[0];      // p points to t[0]
    std::cout << (!++*++p+1==42) << '\n'; // ((!(++(*(++p))))+1)==42
    return 0;
}
```

Operators 1

precedence	operator	description	example
1 L->R	`::`	scope	`std::cout  complex_t::re  ::globname`

2 L->R	`i++    i--`
	`type(t) type{t}`
	`f( )`
	`t[ i ]`
	`s.m`
	`ptr->m`
		postfix increment
		type conversion
		function call
		array indexing
		member access
		member access
			`i++               ptr--`
			`double(1)         int{3.14}`
			`fahr2cels(2)      sin(0.5)`
			`t[i] = 42         t[i][j] = 1;`
			`complex_t c;      c.re = 3.14;`
			`complex_t *ptr = &c;  c->im = 41.5;`

3 R->L	`++i    --i`
	`+i     -i`
	`!i     not i`
	`~i`
	`(type)t`
	`*ptr`
	`&i`
	`sizeof()`
	`new    new[]`
	`delete delete[]`
		prefix increment
		unary sign
		logical NOT
		binary NOT
		C-style conversion
		pointer indirection
		address of
		size of
		heap allocation
		heap deallocation
			`++i   ++ ++i   --i   -- --i` ~~`x+++++y`~~
			`+i`
			`!x  !isspace(ch)`
			`~x`
			`(int)3.14`
			`*ptr  (*ptr).im == ptr→im`
			`ptr = &c;`
			`sizeof(complex_t) sizeof(c) sizeof(*ptr)`
			`ptr = new complex_t;  qtr = new int[3];`
			`delete ptr;            delete [] qtr;`

4 L->R	`x.*mp  ptr->*mp`	member pointer	

Operators 2

precedence	operator	description	example
5 L→R	a*b a/b a%b	multiplicative	a*b/c == (a*b)/c b = !x%2
6 L->R	a+b a-b	additive	a+2-c == (a+2)-b
7 L->R	a<<b a>>b	bitwise shift	a<<b a>>3 u>>3 ~(~0u >> 1)
8 L->R	a <=> b	spaceship operator	(C++20)
9 L->R	a<b a<=b a>b a>=b	relational operator	ptr=&t[0]; qtr=&t[2]; ptr<qtr
10 L->R	a==b a!=b	equal non-equal	i==42 ptr!=nullptr ch!='\0'
11 L->R	a&b	bitwise and	1 == (x>>3 & 0x1) // 1100110000111110
12 L->R	a^b	bitwise exclusive or	(x ^ x) == 0
13 L->R	a|b	bitwise or	::open(“file”,O_CREAT|O_RWONLY,umask)
14 L→R !!	a&&b a and b	logical and	ptr!=nullptr && *ptr == 42
15 L→R !!	a||b a or b	logical or	j==42 && i<10 || j!=42 && i>=10

Operators 3

precedence	operator	description	example
16 R->L !!	a ? b : c	ternary conditional	max=a>b?a:b std::cout<<(last?’\n’:’ ‘)
	throw ex	throwing exception	throw std::out_of_range{“index error”};
	co_yield	yield (co-routine)	
	i=e	assignment	i = 42
	i+=e i-=e	compound assignments	i += 42
	a*=b a/=b a%=b		t[f(i)] += 42
	a<<=b a>>=b		
	a&=b a^=b a|=b		
17 L->R !!	a,b	comma (sequence)	if (++a, *ptr == t[a])

Operators 4

- Some more operators:
  - cast operators for conversions
  - typeid(), sizeof...(), noexcept(), alignof()

```
void f()
{
   int i = 42;
   const int *cip = &i;
   int *ip = const_cast<int *>(cip);          // add, remove constness
   i = static_cast<int>(3.14);                // between relative types
   Derived& dp = dynamic_cast<Dertived&>(bp); // may throw std::bad_cast
   byte bp = reinterpret_cast<byte>(ip);      // reinterpret bytes

   bool b = typeid(*ip) == typeid(i);         // static/dynamic type
}
```

Expression evaluation

- Expressions are defined by
  - precedence
  - associativity
- Evaluation order is not defined
  - except shortcut rules for && || ?: , operators

```
void f()
{
    int i = i;

    std::cout << i << ++i << '\n';
}
```

Expression evaluation

```
#include <iostream>

int f() { std::cout << 'f'; return 2; }
int g() { std::cout << 'g'; return 1; }
int h() { std::cout << 'h'; return 0; }

int main()
{
    std::cout << ( f() == g() == h() ) << '\n';

    return 0;
}
```

Expression evaluation

```cpp
#include <iostream>

int f() { std::cout << 'f'; return 2; }
int g() { std::cout << 'g'; return 1; }
int h() { std::cout << 'h'; return 0; }

int main()
{
    std::cout << ( f() == g() == h() ) << '\n';  // (f() == g()) == h()

    return 0;
}
```

Expression evaluation

```cpp
#include <iostream>

int f() { std::cout << 'f'; return 2; }
int g() { std::cout << 'g'; return 1; }
int h() { std::cout << 'h'; return 0; }

int main()
{
    std::cout << ( f() == g() == h() ) << '\n';  // ( 2  ==  1 ) ==  0

    return 0;
}
```

Expression evaluation

```
#include <iostream>

int f() { std::cout << 'f'; return 2; }
int g() { std::cout << 'g'; return 1; }
int h() { std::cout << 'h'; return 0; }

int main()
{
    std::cout << ( f() == g() == h() ) << '\n';  // ( 2  ==  1 ) ==  0
                                                 //       0      ==  0
    return 0;                                    //              1
}
```

Expression evaluation

```
#include <iostream>

int f() { std::cout << 'f'; return 2; }
int g() { std::cout << 'g'; return 1; }
int h() { std::cout << 'h'; return 0; }

int main()
{
    std::cout << ( f() == g() == h() ) << '\n';  // ( 2  ==  1 ) ==  0
                                                //       0      ==  0
    return 0;                                   //              1
}

$ ./a.out
$ fgh1
```

Expression evaluation

```
#include <iostream>

int f() { std::cout << 'f'; return 2; }
int g() { std::cout << 'g'; return 1; }
int h() { std::cout << 'h'; return 0; }

int main()
{
    std::cout << ( f() == g() == h() ) << '\n';  // ( 2  ==  1 ) ==  0
                                                 //        0      ==  0
    return 0;                                    //               1
}

$ ./a.out
$ fgh1   # or  hgf1  or  gfh1  or...
```

Traps and pitfalls

```
#include <iostream>

int f(int value)
{
    unsigned int mask = 0xf;   // 0000...00001111
    return value & mask == 0;  // check if lower 4 bits are all 0s ?
}

int g(char value)
{
    if ( value = '\0' )    // check the value is terminal zero char ?
        return 0;
    else
        return 1;
}

int h(int val1, int val2)
{
    return val1 *= val2 - 1;  // val1 = val1*val2 - 1 ?
}
```

Traps and pitfalls

```cpp
#include <iostream>

int f(int value)
{
    unsigned int mask = 0xf;   // 0000...00001111
    return value & mask == 0;  // ~~check if lower 4 bits are all 0s~~ ?
}                              // always false

int g(char value)
{
    if ( value = '\0' )    // ~~check the value is terminal zero char~~ ?
        return 0;          // assign '\0' to value and false
    else
        return 1;
}

int h(int val1, int val2)
{
    return val1 *= val2 - 1;  // ~~val1 = val1*val2 - 1~~ ?
}                             // val1 = val1*(val2-1)
```

Traps and pitfalls

```cpp
#include <iostream>

int f(int value)
{
    unsigned int mask = 0xf;     // 0000...00001111
    return (value & mask) == 0;  // check if lower 4 bits are all 0s
}

int g(char value)
{
    if ( value == '\0' )    // check the value is terminal zero char
        return 0;
    else
        return 1;
}

int h(int val1, int val2)
{
    return --(val1 *= val2);    // val1 = val1*val2 - 1
}
```

Traps and pitfalls

```
#include <iostream>

int f(int value)
{
    unsigned int mask = 0xf;     // 0000...00001111
    return (value & mask) == 0;  // check if lower 4 bits are all 0s
}

int g(char value)
{
    if ( '\0' == value )    // Yoda condition
        return 0;
    else
        return 1;
}

int h(int val1, int val2)
{
    val1 *= val2;
    return --val1;    // val1 = val1*val2 - 1
}
```

Traps and pitfalls

```
#include <iostream>

void f()
{
    int t[10];
    int i = 0;

    while ( i < 10 )
    {
        t[i] = i++;   // fill t := {0,1,2,3,4,5,6,7,8,9} ?
    }
}


int g(int a, int b)
{
    if ( a++ < b++  &&  a % b == 0 )   // correct?
        return a;
    else
        return b;
}
```

Traps and pitfalls

```
#include <iostream>

void f()
{
    int t[10];
    int i = 0;

    while ( i < 10 )
    {
        t[i] = i;      // fill t := {0,1,2,3,4,5,6,7,8,9}
        ++i;           // i := i + 1
    }
}

int g(int a, int b)
{
    if ( a++ < b++  &&  a % b == 0 )   // correct!
        return a;
    else
        return b;
}
```

Standard and explicit conversions

- Standard conversion consists of the following steps
  - Optional trivial conversion e.g., array decay
  - Promotion short -> int, int -> double, …, double -> int
  - Function pointer conversion noexcept)
  - Qualification conversion int * -> const int *)
- Explicit conversions
  - Cast operators static_cast<int>(3.14)
  - Conversion operators int(3.14), double{1}, …)
  - Constructors
  - Conversion operators

Implicit conversions

- When expression E with type T1 is used in context where
  - T1 is not accepted, but T2 is accepted
  - and E is convertible to T2
- Samples
  - Passing parameter E to formal argument T2
  - E is used as operands where the operator accept T2
  - Initializing object of T2 (including return statement)
  - Assignment
  - Switch statement (T2 integral type)
  - If and Loop statements (T2 bool type)

Implicit conversions

- The conversion happens in the following procedure
  - Zero or One standard conversion
  - Zero or One user defined conversion
  - Zero or One standard conversion

```
void f()
{
    long l = 1;   // int -> long conversion
    l = 'A' + 1L; // char->int promotion ->long conversion, result is long
    if ( 'A' + 1L ) ;   // same as above, then ->bool context
    while ( std::cin >> i ) ;  // std::cin -> bool user defined conversion
}
```

Statements

- Expression statements
- Compound statements
- Control structures
- Declaration statements
- Try blocks

```
extern void g();

void f()
{
    int i;
    i = 5;  // expression statement
    g(i);   // expression statement
    ;       // null statement
}
```

Compound statement (block)

- Sequence of statements
- Collects statements into one unit
- Controls scope and lifetime

```
void f(int x)
{
    if ( x > 10 )
    {
        int i = 5;
        std::osfstream of{"out.txt"};
        std::mutex mut;
        {
            std::lock_guard{mut};  // lock mut
            of << i << '\n';
        }  // lock is released here
    }  // scope of i and of end here, of is flushed and closed
}
```

Compound statement (block)

- Sequence of statements
- Collects statements into one unit
- Controls scope and lifetime

```
void f(int x)
{
    if ( x > 10 )
    {
        int i = 5;
        std::osfstream of{"out.txt"};
        std::mutex mut;
        {
            std::lock_guard{mut};  // ~~lock mut~~ temporary!
            of << i << '\n';
        }  // ~~lock is released here~~
    }  // scope of i and of end here, of is flushed and closed
}
```

Compound statement (block)

- Sequence of statements
- Collects statements into one unit
- Controls scope and lifetime

```
void f(int x)
{
    if ( x > 10 )
    {
        int i = 5;
        std::osfstream of{"out.txt"};
        std::mutex mut;
        {
            std::lock_guard{mut} lck;  // lock mut
            of << i << '\n';
        }  // lock is released here
    }  // scope of i and of end here, of is flushed and closed
}
```

Condition statement

- Dangling **else** belongs to the closest preceding **if**

```
if ( x < 10 )
    if ( y > 5 )
        std::cout << "x<10 and y>5" << '\n';
else
    std::cout << "x<10 and y<=5" << '\n';
```

Condition statement

- Dangling **else** belongs to the closest preceding **if**

```
if ( x < 10 )
    if ( y > 5 )
        std::cout << "x<10 and y>5" << '\n';
else
    std::cout << "x<10 and y<=5" << '\n';


if ( x < 10 )   // equivalent to
{
    if ( y > 5 )
    {
        std::cout << "x<10 and y>5" << '\n';
    }
    else
    {
        std::cout << "x<10 and y<=5" << '\n';
    }
}
```

Condition statement

- Dangling **else** belongs to the closest preceding **if**

```
if ( x < 10 )
    if ( y > 5 )
        std::cout << "x<10 and y>5" << '\n';
else
    std::cout << "x<10 and y<=5" << '\n';


if ( x < 10 )   // different from
{
    if ( y > 5 )
    {
        std::cout << "x<10 and y>5" << '\n';
    }
}
else
{
    std::cout << "x>=10" << '\n';
}
```

Condition statement

- Sometimes we have to chain is-else-if-… constructs

```
if ( x < 10 && y > 5 )
{
    std::cout << "x<10 and y>5" << '\n';
}
else if ( x < 10 && y <= 5 )
{
    std::cout << "x<10 and y<=5" << '\n';
}
else if ( x >= 10 && y > 5 )
{
    std::cout << "x>=10 and y>5" << '\n';
}
else if ( x >= 10 && y <= 5 )
{
    std::cout << "x>=10 and y<=5" << '\n';
}
else
{
    std::cout << "this is unlikely" << '\n';
}
```

Selection statement

- Sometimes we have to chain is-else-if-… constructs

```
void print_day(int day_of_week)  // Sunday == 1, ..., Saturday == 7
{
    switch ( day_of_week )
    {

    case  1: std::cout << "Week-end";  break;
    case  2: std::cout << "Monday";    break;
    case  3: std::cout << "Tuesday";   break;
    case  4: std::cout << "Wednesday"; break;
    case  5: std::cout << "Thursday";  break;
    case  6: std::cout << "Friday";    break;
    case  7: std::cout << "Week-end";  break;
    }
}
```

Selection statement

- Sometimes we have to chain is-else-if-… constructs

```
void print_day(int day_of_week)  // Sunday == 1, ..., Saturday == 7
{
    switch ( day_of_week )
    {

    case  2: std::cout << "Monday";    break;
    case  3: std::cout << "Tuesday";   break;
    case  4: std::cout << "Wednesday"; break;
    case  5: std::cout << "Thursday";  break;
    case  6: std::cout << "Friday";    break;
    case  1:
    case  7: std::cout << "Week-end";  break;
    }
}
```

Selection statement

- Sometimes we have to chain is-else-if-… constructs

```
void print_day(int day_of_week)  // Sunday == 1, ..., Saturday == 7
{
    switch ( day_of_week )
    {

    case  2: std::cout << "Monday";    break;
    case  3: std::cout << "Tuesday";   break;
    case  4: std::cout << "Wednesday"; break;
    case  5: std::cout << "Thursday";  break;
    case  6: std::cout << "Friday";    break;
    case  1: [[ fallthrough ]] ;
    case  7: std::cout << "Week-end";  break;
    }
}
```

Selection statement

- Sometimes we have to chain is-else-if-… constructs

```
void print_day(int day_of_week)  // Sunday == 1, ..., Saturday == 7
{
    switch ( day_of_week )
    {
    default: std::cout << "Bad value in day_of_week"; break;
    case  2: std::cout << "Monday";    break;
    case  3: std::cout << "Tuesday";   break;
    case  4: std::cout << "Wednesday"; break;
    case  5: std::cout << "Thursday";  break;
    case  6: std::cout << "Friday";    break;
    case  1: [[ fallthrough ]] ;
    case  7: std::cout << "Week-end";  break;
    }
}
```

While statement

- Looping on condition

```
int find_neg(const std::vector<int> &v)
{
    int i = 0;
    int neg = 0;
    bool found = false;

    while ( i < std::ssize(v)  &&  !found )
    {
        if ( v[i] < 0 )
        {
            neg = v[i];
            found = true;
        }
        ++i;
    }
    return neg;   // found negative item or 0
}
```

While statement

- Looping on condition

```
std::vector<int> read()
{
    std::vector<int> v;
    int i;

    while ( std::cin >> i )   // while std::cin.good()
    {
        v.push_back(i);
    }
    return v;   // likely move, so not bad performance
}
```

While statement

- Looping on condition

```
int find_neg(const std::vector<int> &v)
{
    int i = 0;
    int neg = 0;
    bool found = false;

    while ( i < std::ssize(v)  &&  !found )
    {
        if ( v[i] < 0 )
        {
            return v[i]; // found v[i]
            found = true;
        }
        ++i;
    }
    return 0;   // not found
}
```

For statement

- for(opt-exp1; opt-expr2; opt-expr3) statement
- for(decl-stmt; opt-expr2; opt-expr3) statement
- If expr2 is missing -> true
- Loop variable is visible and live only in the loop

```cpp
int find_neg(const std::vector<int> &v)
{
    for (int i = 0; i < std::ssize(v); ++i)
    {
        if ( v[i] < 0 )
        {
            return v[i]; // found v[i]
        }
    }
    return 0;   // not found
}
```

For statement

- for(opt-exp1; opt-expr2; opt-expr3) statement
- for(decl-stmt; opt-expr2; opt-expr3) statement
- If expr2 is missing -> true
- Loop variable is visible and live only in the loop

```
int find_neg(const std::vector<int> &v)
{
    for (int i = 0; i < std::ssize(v); ++i)
    {
        if ( v[i] < 0 )
        {
            return v[i]; // found v[i]
        }
    }
    return 0;   // not found
}
```

```
int i = 0;
while (i < std::ssize(v))
{
    /*
      Same statements
    */
    ++i;
}
```

(Range) for statement

- Looping on a data collection
- Variable can be reference so we can modify the value
- But we must not modify the range

```
int find_neg(const std::vector<int> &v)
{
    for (int val : v)  // applicable to anything with begin() end()
    {
        if ( val < 0 )
        {
            return val; // found v[i]
        }
    }
    return 0;   // not found
}
```

(Range) for statement

- Looping on a data collection
- Variable can be reference so we can modify the value
- But we must not modify the range

```
int find_neg(const std::vector<int> &v)
{
    for (auto val : v)  // applicable to anything with begin() end()
    {
        if ( val < 0 )
        {
            return val; // found v[i]
        }
    }
    return 0;   // not found
}
```

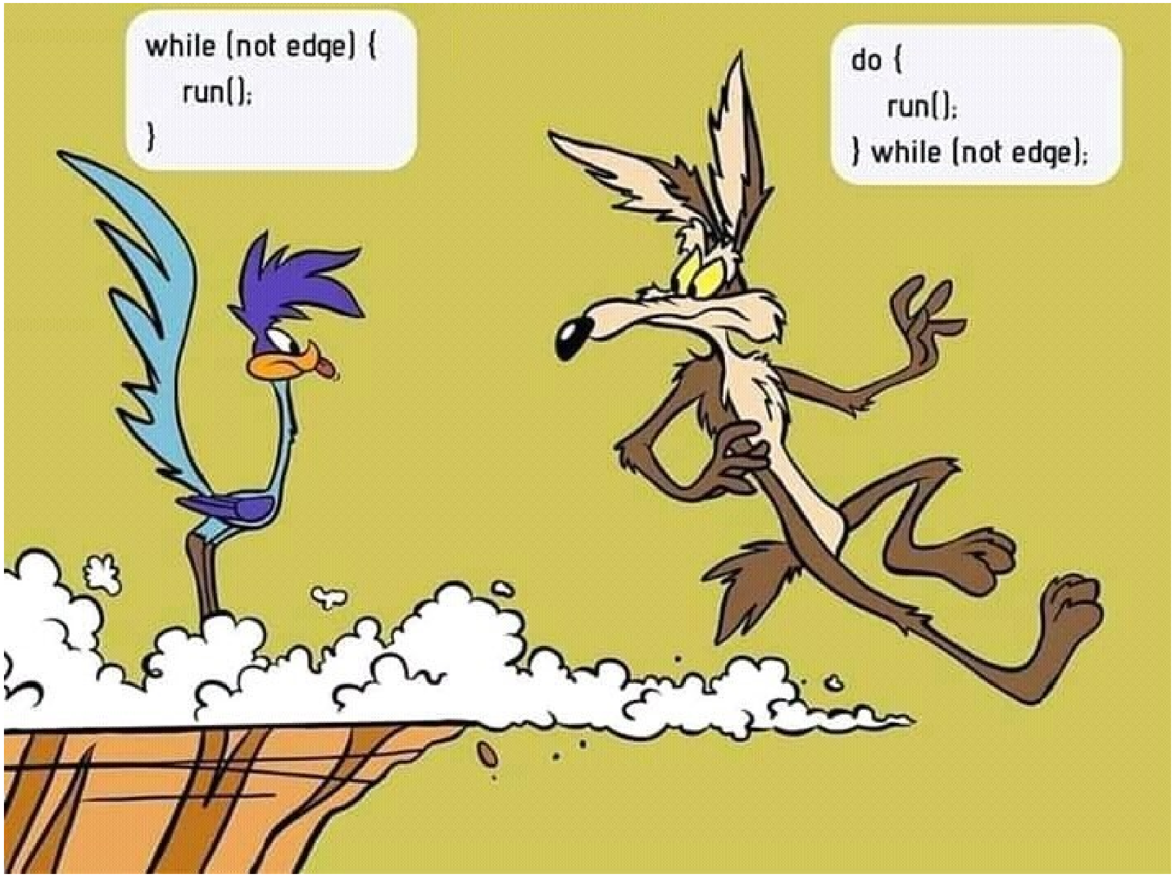
Do while

- Condition checked after the execution of the statement
- Always enters into the loop at least once

```
int find_neg(const std::vector<int> &v)
{
    int i = 0;
    do  // We know that the vector.size() > 0
    {
        if ( v[i] < 0 )
        {
            return val; // found v[i]
        }
    }
    while (++i < std::ssize(v));
    return 0;   // not found
}
```

Do -- while

- Condition checked after the execution of the statement
- Always enters into the loop at least once



```
while (not edge) {
    run();
}
```

```
do {
    run();
} while (not edge);
```

Return, break and continue

- Return: leaves the function (return value is converted if needed)
- Break: leaves the innermost loop and continues after the loop
- Continue: jumps over the rest of the core and reiterate condition

```
int find_neg(const std::vector<int> &v)
{
    for (auto &val : v)  // reference parameter: can change element
    {
        if ( val < 0 )
            return val;       // found v[i]
        if ( 0 == val )
            break;            // found zero, leave the loop
        if ( val % 3 )
            continue;         // skip is not dividable by 3
        --val;                // decrease positive multiples of 3
    }
    return 0;   // not found
}
```

Selection statements with initializers

```
/* C language, before C99 */
{
  int i;
  for ( i = 0; i < 10; ++i) {
     /* use i here */
  }
  /* i still visible here */
}
```

```
/* C++ language, C since C99 */
{
  for ( int i = 0; i < 10; ++i) {
     /* use i here */
  }
  /* i is not visible here */
}
```

Selection statements with initializers

```
/* C++, since the beginning */
{
  if ( const char *path = std::getenv("PATH") ) {
     /* use path here */
  }
  else {
     /* path is also available here, nullptr */
  }
  /* path not available here */
}

{
  if ( auto sp = wp.lock() ) { /* shared_ptr from weak_ptr */
     /* use sp here */
  }
  /* sp is destructed here */
}
```

Selection statements with initializers

- Not works well, when
  - it is not the declared variable we depend on
  - the success/fail is not usual int/bool/ptr != 0

```
std::set<int> s;

auto p = s.insert(42);
if ( p.second ) {
  std::cerr << "insert ok" << ‘\n’;
}
else {
  std::cerr << "insert failed" << ‘\n’;
}


std::mutex mut1, mut2, mut3;

int ret = std::try_lock( mut1, mut2, mut3 );  // many OS functions
if ( -1 == ret ) {
  std::cerr << "locks done" << ‘\n’;
}
```

Selection statements with initializers

- Declaration is allowed in if and switch statements
  - The scope of declared variable is not “leaking” out
  - More flexibility for the condition

```
std::set<int> s;

// auto p = s.insert(42);
if ( auto p = s.insert(42); p.second ) {
  std::cerr << "insert ok" << ‘\n’;
}
else {
  std::cerr << "insert failed" << ‘\n’;
}


std::mutex mut1, mut2, mut3;

// int ret = std::try_lock( mu1t, mut2, mut3 );
if ( int ret = std::try_lock( mu1t, mut2, mut3 ); -1 == ret ) {
  std::cerr << "locks done" << ‘\n’;
}
```

Selection statements with initializers

- Declaration is allowed in if and switch statements
  - The scope of declared variable is not “leaking” out
  - More flexibility for the condition

```
std::set<int> s;

// auto p = s.insert(42);
if ( auto p = s.insert(42); p.second ) {
  std::cerr << "insert ok" << ‘\n’;
}
else {
  std::cerr << "insert failed" << ‘\n’;
}


std::mutex mut1, mut2, mut3;

// int ret = std::try_lock( mu1t, mut2, mut3 );
if ( int ret = std::try_lock( mu1t, mut2, mut3 ); -1 == ret ) {
  std::cerr << "locks done" << ‘\n’;
}  // unlock????
```

Selection statements with initializers

- Use lock_guard, unique_lock, scoped_lock, ...

```
std::mutex       mut;
std::deque<int>  data;


// producer
{
  std::lock_guard sl(mut);
  data.push_back(i);
}


// consumer
if ( std::lock_guard sl(mut); !data.empty() ) {
  int i = data.front();
  data.pop_front();
}
```

Selection statements with initializers

- Don’t trick yourself!!!

```
std::mutex       mut;
std::deque<int>  data;


// producer
{
  std::lock_guard sl(mut);
  data.push_back(i);
}


// consumer
if ( std::lock_guard(mut); !data.empty() ) {  // bad, temporary!!!
  int i = data.front();
  data.pop_front();
}
```

Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
  switch ( argc )
  {


    case 1: std::cout  << "1"      << '\n';   break;
    case 2: std::cout  << "2"      << '\n';   break;
    default: std::cout << "d"      << '\n';   break;
  }
  return 0;
}
```

Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
  switch ( argc )
  {
    int x;

    case 1: std::cout  << "1" << x << '\n';   break;
    case 2: std::cout  << "2" << x << '\n';   break;
    default: std::cout << "d" << x << '\n';   break;
  }
  return 0;
}
```

Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
  switch ( argc )
  {
    int x;

    case 1: std::cout  << "1" << x << '\n';   break; // undefined beh.
    case 2: std::cout  << "2" << x << '\n';   break;
    default: std::cout << "d" << x << '\n';   break;
  }
  return 0;
}
```

Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
  switch ( argc )
  {
    int x = argc;

    case 1: std::cout  << "1" << x << '\n';   break;
    case 2: std::cout  << "2" << x << '\n';   break;
    default: std::cout << "d" << x << '\n';   break;
  }
  return 0;
}
```

Selection statements with initializers

- Switch is more interesting than you think.

```cpp
#include <iostream>

int main(int argc, char *argv[])
{
  switch ( argc )
  {
    int x = argc;

    case 1: std::cout  << "1" << x << '\n';   break;
    case 2: std::cout  << "2" << x << '\n';   break;
    default: std::cout << "d" << x << '\n';   break;
  }
  return 0;
}
```

```
error: jump to case label XXX crosses initialization of int x
```

Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
  switch ( int x = argc )
  {
    // works even in “old” C++

    case 1: std::cout  << "1" << x << '\n';   break;
    case 2: std::cout  << "2" << x << '\n';   break;
    default: std::cout << "d" << x << '\n';   break;
  }
  return 0;
}
```

Selection statements with initializers

- Switch is more interesting than you think.

```
#include <iostream>

int main(int argc, char *argv[])
{
  switch ( int x = argc; ++x )
  {
    // works since C++17

    case 1: std::cout  << "1" << x << '\n';   break;
    case 2: std::cout  << "2" << x << '\n';   break;
    default: std::cout << "d" << x << '\n';   break;
  }
  return 0;
}
```

Selection statements with initializers

- Declaration list is allowed

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

int main()
{
  std::vector v = { 1, 2, 3 };  // CTAD, C++17

  if (int s = v.size(), c = v.capacity(); s < c ) {
    std::cerr << "s < c" << '\n';
  }
  else {
    std::cerr << "s == c" << '\n';
  }
  return 0;
}
```

Selection statements with initializers

- A bit more interesting case

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

int main()
{
  std::vector v = { 1, 2, 3 };  // CTAD, C++17

  if (int s = v.size(), it = v.begin(); s > 0 && s < *it ) {
    std::cerr << "s < c" << '\n';
  }
  else {
    std::cerr << "s == c" << '\n';
  }
  return 0;
}
```

```
error: v.begin() is not convertible to int
```

Selection statements with initializers

- Auto deduction must be consistent

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

int main()
{
  std::vector v = { 1, 2, 3 };  // CTAD, C++17

  if (auto s = v.size(), it = v.begin(); s > 0 && s < *it ) {
    std::cerr << "s < c" << '\n';
  }
  else {
    std::cerr << "s == c" << '\n';
  }
  return 0;
}
```

```
error: inconsistent deduction for ‘auto’
```

Selection statements with initializers

- Structured binding helps

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

int main()
{
  std::vector v = { 1, 2, 3 };  // CTAD, C++17

  if (auto [s,it] = std::pair{ v.size(),v.begin()}; s > 0 && s < *it){
    std::cerr << "s < c" << '\n';
  }
  else {
    std::cerr << "s == c" << '\n';
  }
  return 0;
}
```

works fine

Selection statements with initializers

- Ideally, we should allow multiple statements

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

int main()
{
  std::vector v = { 1, 2, 3 };  // CTAD, C++17

  if (auto s = v.size(); auto it = v.begin(); s > 0 && s < *it){
    std::cerr << "s < c" << '\n';
  }
  else {
    std::cerr << "s == c" << '\n';
  }
  return 0;
}

error: parse error
```

Error handling, exceptions

- Handling exceptional cases: errno, assert, longjmp
- Goals of exception handling
- Handlers and exceptions
- Standard exceptions
- Exception safe programming
- C++11 noexcept
- Exception_ptr
- Expected (C++23)

C++ Errno

```
#include <cerrno>
#include <cstdio>   // std::fopen
#include <cstring>  // std::strerror

struct record { ... };
struct record rec;

extern int errno;  /* preprocessor macro: thread-local since C++11 */
int myerrno;       /* my custom error code */

std::FILE *fp;

if ( NULL == (fp = std::fopen( "fname", "r")) ) /* try to open the file */
{
    std::fprintf( stderr, "can't open file %s\n", "fname");
    std::fprintf( stderr, "reason: %s\n", std::strerror(errno)); /* perror(NULL) */
    myerrno = 1;
}
else if ( ! std::fseek( fp, n*sizeof(rec), SEEK_SET) ) /* pos to record */
{
    std::fprintf( stderr, "can't find record %d\n", n);
    myerrno = 2;
}
else if ( 1 != std::fread( &r, sizeof(r), 1, fp) ) /* try to read a record */
{
    std::fprintf( stderr, "can't read record\n");
    myerrno = 3;
}
else   /* everything was successful up to now */
{
    ...
}
```

Iostream error handling

```cpp
void f()
{
    std::ifstream file("input.txt");

    if ( ! file )    /* before C++11: void*, since C++11: bool */
    {
        std::cerr << "file opening failed\n";
        return;
    }

    for( int n; file >> n; )  /* while ( ! cin.fail() ) */
    {
        std::cout << n << '\n';
    }

    if ( file.bad() )
    {
        std::cerr << "i/o error while reading\n";
    }
    else if ( file.eof() )
    {
        std::cerr << "eof reached\n";
    }
    else if ( file.fail() )
    {
        std::cerr << "non-integer\n";
    }
}
```

Assert

```
#include <cassert>   /* assert.h in C */

void open_file(std::string fname)
{
   assert(fname.length() > 0);


   std::ifstream f(fname.c_str());
   . . .
}
```

- Run-time error!

Static assert (C++11)

```
#include <type_traits>

template <typename T>
void swap(T &x, T &y)
{
   static_assert( std::is_nothrow_move_constructible<T>::value, &&
                  std::is_nothrow_move_assignable<T>::value, "Swap may throw" );

   auto tmp = x;
   x = y;
   y = tmp;
}



#if __STDC_HOSTED__ != 1
#  error "Not a hosted implementation"
#endif

#if __cplusplus >= 202302L
#  warning "Using #warning as a standard feature"
#endif
```

Goals of exception handling

- Type-safe transmission of arbitrary data from throw-point to handler
- Every exceptions should be caught by the appropriate handler
- No extra code/space/time penalty if not used
- Grouping of exceptions
- Work fine in multithreaded environment
- Cooperation with other languages (like C)

Setjmp/longjmp

```c
#include <setjmp.h>
#include <stdio.h>

jmp_buf x;

int main()
{
    int i = 0;

    if ( (i = setjmp(x)) == 0 ) // try
    {
        f();
    }
    else    // catch
    {
        switch( i )
        {
        case  1: handler1(); break;
        case  2: handler2(); break;
        default: fprintf( stdout, "error code = %d\n", i); break;
        }
    }
    return 0;
}
```

```c
// perhaps in another source file

#include <setjmp.h>
extern jmp_buf x;

void f()
{
    // ...
    g();
}

void g()
{
    if ( something_wrong() )
    {
        longjmp(x,2);  // throw
    }
}
```

Setjmp/longjmp

```
#include <setjmp.h>
#include <stdio.h>

jmp_buf x;

int main()
{
    int i = 0;

    if ( (i = setjmp(x)) == 0 ) // try
    {
        f();
    }
    else    // catch
    {
        switch( i )
        {
        case  1: handler1(); break;
        case  2: handler2(); break;
        default: fprintf( stdout, "error code = %d\n", i); break;
        }
    }
    return 0;
}
```

```
// perhaps in another source file

#include <setjmp.h>
extern jmp_buf x;

void f()
{
    // ...
    g();
}

void g()
{
    if ( something_wrong() )
    {
        longjmp(x,2);  // throw
    }
}
```

Exceptions in C++

-
  ```
  try  // dangerous area
  {
    f();   // someth
  }
  catch (T1 e1) { /* handler for T1 */ }
  catch (T2 e2) { /* handler for T2 */ }
  catch (T3 e3) { /* handler for T3 */ }

  void f()
  {
    //...
    T e;
    throw e;    /* throws exception of type T */

    // or:
    throw T();  /* throws default value of T */
  }
  ```

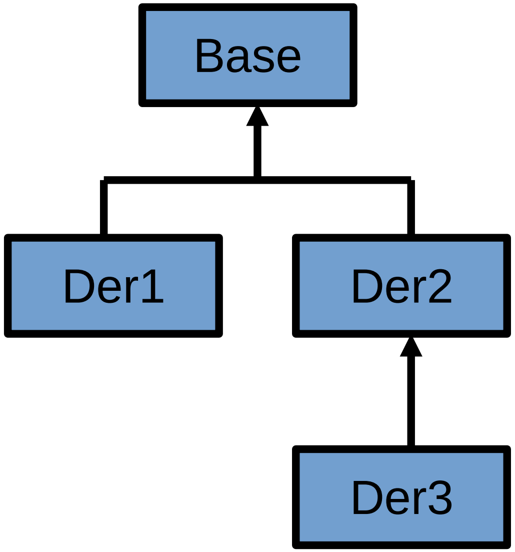
Which handler?

A handler of type H catches the exception of type E if

- H and E is the same type
- H is unambiguous base type of E
- H and E are pointers or references and some of the above stands

Exception hierarchies

- Resolved run-time
- Not overloading!



```
class Base { … };
class Der1 : public Base { … };
class Der2 : public Base { … };
class Der3 : public Der2 { … };

try
{
  f();
  // …
}
catch (Der3 &e1)  { /* handler for Der3 */ } /* the most derived handler first */
catch (Der2 &e2)  { /* handler for Der2 */ }
catch (Der1 &e3)  { /* handler for Der1 */ }
catch (Base &e3)  { /* handler for Base */ } /* the base handler last */

void f()
{
  if ( ... )
    throw Der3();  /* throw the most derived type */

}
```

Re-throw

- Re-throw works only inside a catch block
- Re-throws the original object, not the (possible sliced) one

```
class Base { … };
class Der1 : public Base { … };

void g()
{
  throw Der1;  // throw derived exception Der1
}

void f()
try  // function block itself can be try block
{
  g();
}
catch ( Base b )  // catch Base by value: copied, since C++11 can be moved
{
  must_do_at_exception(b);
  throw;  // re-throw original exception Der1
}
catch ( ... )  // catch all
{
  must_do_at_exception();
  throw;  // re-throw original exception
}
```

std exception hierarchy

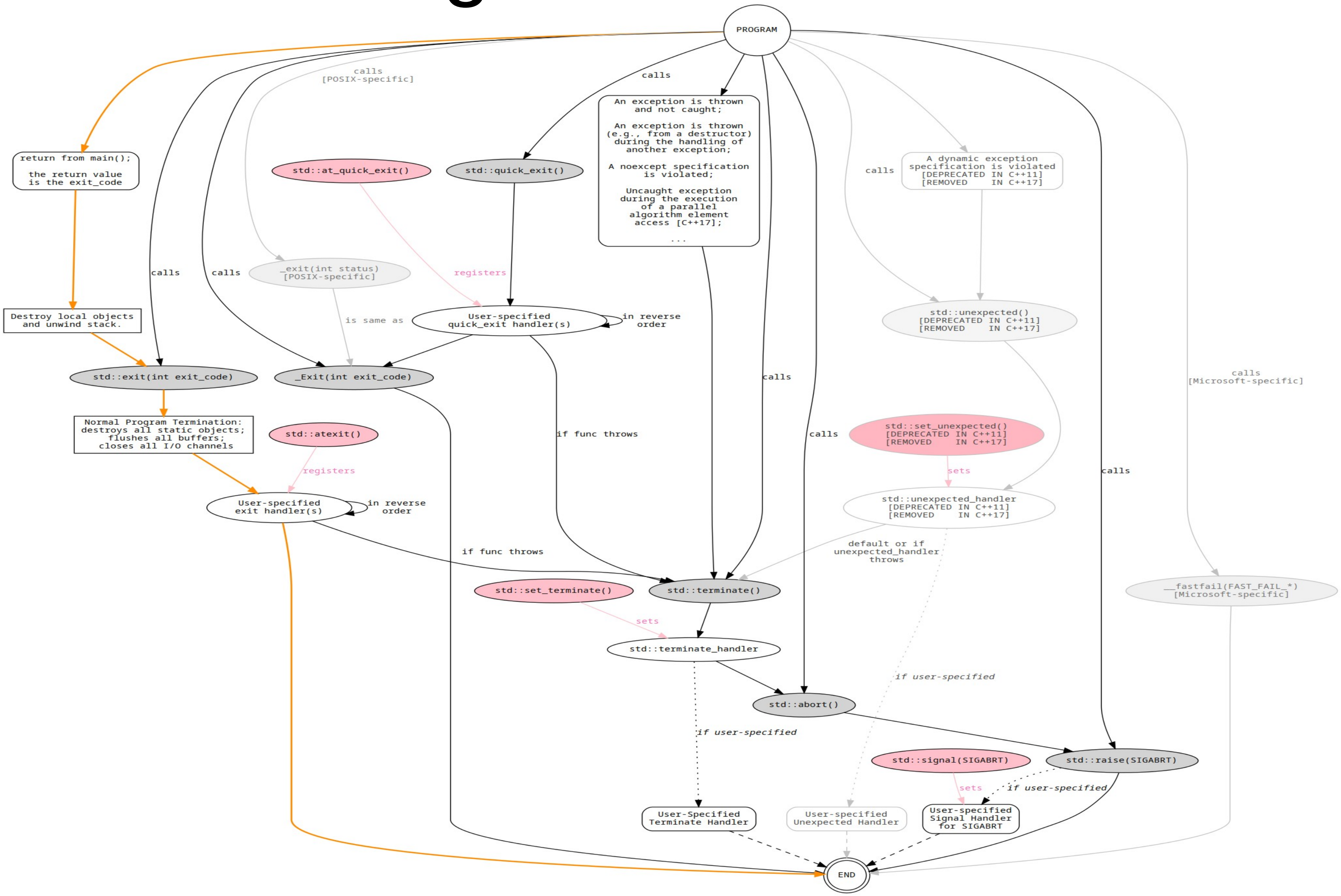
```
class exception {};                            // in <exception>

class bad_exception : public exception {};     // calls unexpected()
class bad_weak_ptr : public exception {};      // C++11 weak_ptr -> shared_ptr
class bad_function_call : public exception {}; // C++11 function::operator()
class bad_typeid : public exception {};        // typeid(0)
class bad_cast   : public exception {};        // dynamic_cast
  class bad_any_cast : public bad_cast {};     // C++17
class bad_variant_access : exception {}        // C++17
class bad_optional_access : exception {}       // C++17
class bad_alloc  : public exception {};        // new  <new>
  class bad_array_new_length : bad_alloc {}    // C++11, new T[-1]

class logic_error : public exception {};
  class domain_error : public logic_error {}; // domain error,std::sqrt(-1)
  class invalid_argument : public logic_error {}; // bitset char != 0 or 1
  class length_error : public logic_error {}; // length str.resize(-1)
  class out_of_range : public logic_error {}; // bad index in container or string
  class future_error : public logic_error {}; // C++11: promise abandons the shared state

class runtime_error : public exception {};
  class range_error     : public runtime_error {}; // floating point ovf or unf
  class overflow_error  : public runtime_error {}; // int overflow INT_MAX+1
  class underflow_error : public runtime_error {}; // int underflow INT_MIN-1
  class system_error    : public runtime_error {}; // e.g. std::thread constr.
    class ios_base::failure : public system_error {}; // C++11
    class filesystem::filesystem_error : public system_error {}; // C++17
    class nonexistent_local_time : public system_error // C++20
    class ambigous_local_time : public system_error    // C++20
    class format_error : public system_error           // C++20
```

Program termination


PROGRAM
calls [POSIX-specific]
calls
return from main(); the return value is the exit_code
std::at_quick_exit()
std::quick_exit()
An exception is thrown and not caught;
An exception is thrown (e.g., from a destructor) during the handling of another exception;
A noexcept specification is violated;
Uncaught exception during the execution of a parallel algorithm element access [C++17];
...
calls
A dynamic exception specification is violated [DEPRECATED IN C++11] [REMOVED IN C++17]
calls
calls
_exit(int status) [POSIX-specific]
registers
Destroy local objects and unwind stack.
User-specified quick_exit handler(s)
in reverse order
is same as
std::unexpected() [DEPRECATED IN C++11] [REMOVED IN C++17]
std::exit(int exit_code)
_Exit(int exit_code)
calls
calls [Microsoft-specific]
Normal Program Termination: destroys all static objects; flushes all buffers; closes all I/O channels
std::atexit()
if func throws
calls
std::set_unexpected() [DEPRECATED IN C++11] [REMOVED IN C++17]
registers
sets
calls
User-specified exit handler(s)
in reverse order
std::unexpected_handler [DEPRECATED IN C++11] [REMOVED IN C++17]
if func throws
default or if unexpected_handler throws
std::set_terminate()
std::terminate()
__fastfail(FAST_FAIL_*) [Microsoft-specific]
sets
std::terminate_handler
if user-specified
std::abort()
if user-specified
std::signal(SIGABRT)
std::raise(SIGABRT)
sets
if user-specified
User-Specified Terminate Handler
User-specified Unexpected Handler
User-specified Signal Handler for SIGABRT
END

https://github.com/adishavit/Terminators/blob/master/README.md

Noexcept specifier in C++11

- Since C++11
  - Part of function type since C++17
  - But not part of of function signature (so no overload on noexcept)
- C++ functions are either non-throwing or potentially throwing
- C++ cannot execute full compile time check on possible exceptions
  - Partially due to possible call of non C++ functions
- Destructors, deallocation functions are non-throwing
- Implicitly declared or defaulted default-, copy- and move constructors, copy, move oper.
  - Unless base class or called operations throw
- Noexcept is important for optimizations and program safety

```
void f() noexcept(expr) { }
void f() noexcept(true) { }
void f() noexcept       { }  // noexcept(true)
```

Noexcept operator in C++11

- **bool noexcept(expr);**
- Can be used inside function template noexcept specifier
- Compile-time check: does not evaluate *expr* (like **sizeof**)
- False if
  - Expr throws
  - Expr has dynamic_cast or typeid
  - Has function which is no noexcept(true) and not constexpr
- Otherwise **true**

```
template <typename T>
void f() noexcept ( noexcept( T::g() ) )
{
    T::g();
}
```

Destructors

Destructors must not throw!

- Exception thrown during exception triggers **std::terminate()**
- Since C++11 every destructor is implicit **noexcept**
- It is possible to declare the destructor as **noexcept(false)**
- If exception is thrown during stack unwinding: **std::terminate** is called instead
- But the real situation is more complex:

https://akrzemi1.wordpress.com/2011/09/21/destructors-that-throw/

Exception safety

```
class T1 { ... };
class T2 { ... };

template <typename T1, typename T2>
void f( T1*, T2*);

void g()
{
    f( new T1(), new T2() );
    // ...
}
```

Scenario1 (before C++17)

Allocates memory for T1
Allocates memory for T2
Constructs T1
Constructs T2
Calls f

Scenario2

Allocates memory for T1
Constructs T1
Allocates memory for T2
Constructs T2
Calls f

Exception safety

- Usually solved by rearranging the source and use sequence points
- Since C++17, parameter evaluation order changed

undefined -> unspecified

```
class T1 { ... };
class T2 { ... };

template <typename T1, typename T2>
void f( std::unique_ptr<T1>, std::unique_ptr<T1>);



void g()
{
   std::unique_ptr<T1> ptr1(new T1());
   std::unique_ptr<T2> ptr2(new T2());

   f( ptr1, ptr2 );
   // ...
}
```

Exception safety in STL

- **Basic guarantee:** no memory leak or other resource issue
- **Strong guarantee:** the operation is atomic:
  it either succeeds or has no effect

  e.g. push_back() for vector, insert() for assoc. cont.
- **Nothrow guarantee:** the operation does not throw

  e.g. pop_back() for vector, erase() for assoc. cont., swap()

Exception safety in STL

	vector	deque	list	map
clear()	nothrow(copy)	nothrow(copy)	nothrow	nothrow
erase()	nothrow(copy)	nothrow(copy)	nothrow	nothrow
insert() one	strong(copy)	strong(copy)	strong	strong
insert() more	strong(copy)	strong(copy)	strong	strong
merge()			nothrow(comp)	
push_back()	strong	strong	strong	
push_front()		strong	strong	
pop_back()	nothrow	nothrow	nothrow	
pop_front()		nothrow	nothrow	
remove()			nothrow(comp)	
remove_if()			nothrow(pred)	
reverse()			nothrow	
splice()			nothrow	
swap()	nothrow	nothrow	nothrow	nothrow(cp,co)
unique()			nothrow(comp)	

Some new features in C++11

- **class exception_ptr** smart pointer type,default constructible, copyable, == if null or points to the same
- **make_exception_ptr(E e)** creates an exception_ptr pointing to the exception object **e**.
- **current_exception()** null ptr if called outside of exception handling or it returns an exception_ptr pointing to the current exception
- **rethrow_exception(std::exception_ptr p)** rethrow exception **p**
- **class nested_exception** polymorphic mixin class capture and store current exception
  has **rethrow_nested() const** member function
- **throw_with_nested(T&& t)**
  throw_if_nested(const E& e)

exception_ptr

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

void handle_eptr(std::exception_ptr eptr) // passing by value is ok
{
    try
    {
        if (eptr != std::exception_ptr())
        {
            std::rethrow_exception(eptr);
        }
    }
    catch(const std::exception& e)
    {
        std::cout << "Caught exception \"" << e.what() << "\"\n";
    }
}

int main()
{
    std::exception_ptr eptr;
    try
    {
        std::string().at(1); // this generates an std::out_of_range
    }
    catch(...)
    {
        eptr = std::current_exception(); // capture
    }
    handle_eptr(eptr);
} // destructor for std::out_of_range called here, when the eptr is destructed

// output: Caught exception "basic_string::at"
```

Optional (C++17)

- Maybe monad implementation
- Replaces return types like **std::pair<T,bool>**
- Optional contains value
  - Initialized/assigned with value of T
  - Initialized/assigned with **optional<T>** which contains value
- Optional does not contain value
  - Default initialized or initialized with value of **std::nullopt_t**
  - Initialized/assigned with **optional<T>** which does not contain value
- If **optional<T>** contains a value, than it is allocated as T
  - Not a pointer based heap storage
- Convertible to **bool**: true if contains value
- No optional reference

std::optional

```
std::optional<int> convert( const std::string& s)
try
{
    return std::stoi(s);  // C++11
}
catch (std::invalid_argument e)  // s is not an integer
{
    return {};   // std::optional<int>{std::nullopt}
}
catch (std::out_of_range e)   // result cannot be represented in int
{
    return {};   // std::optional<int>{std::nullopt};
}

int main()
{
    int i = convert("42").value_or(-1);
}
```

Use of optional

```
void f(bool b1)
{
  std::optional<int> opt1;  // default constr: std::nullopt
  std::cout << opt1.value_or(-1) << '\n';   // -1
  try
  {
    std::cout << opt1.value() << '\n';  // throw std::bad_optional_access
  }
  catch( std::bad_optional_access& e)
  {
    std::cerr << e.what() << '\n';
  }
  opt1 = b1 ? std::optional<int>(42) : std::nullopt;  // 42

  std::cout << opt1.value_or(-1) << '\n';  // 42
  if ( opt1 )  // true
  {
    std::cout << opt1.value() << '\n';  // 42
    *opt1 = 2;   // access contained data, also ->  exists
    int i = opt1.value();
    std::cout << i << '\n';   // 2
  }
}
```

```
-1
bad optional access
42
42
2
```

Use of pointers

```
void f(bool b1)
{
  std::optional<std::string> opt2;  // std::nullopt
  *opt2 = "Hello";   // undefined behavior if std::nullopt

  std::cout << *opt2 << '\n';
  std::cout << std::boolalpha << opt2.has_value() << '\n';   // false

  std::cout << opt2.value_or("no value") << '\n';   // “no value”
  std::string s = *std::move(opt2);

  std::cout << s << ", " << opt2->size() << '\n';
}
```

```
Hello
false
no value
Hello, 0
```

Expected (C++23)

Always holds either value_type or unexpected_type

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0323r12.html

https://youtu.be/PH4WBuE1BHI (Andrei Alexandrescu CppCon 2018)

```
#include <expected>  // since C++23

template <class T, class E>    // T can be void, but T must not be unexpected<>
class expected {
  // types
  using value_type = T;
  using error_type = E;
  using unexpected_type = std::unexpected<E>;
  template <class U> using rebind = expected<U, error_type>;

  // accessors
  bool has_value()
  operator bool()
  operator void()   // if T == void
  T*   operator->() // undefined behavior if not expected
  T&   operator*()  // undefined behavior if not expected
  void operator*()  // if T == void, undefined behavior if not expected
  T&     value()      // may throw std::bad_expected_value<E>
  E&     error()      // undefined behavior if expected
  T value_or(U def)
};
```