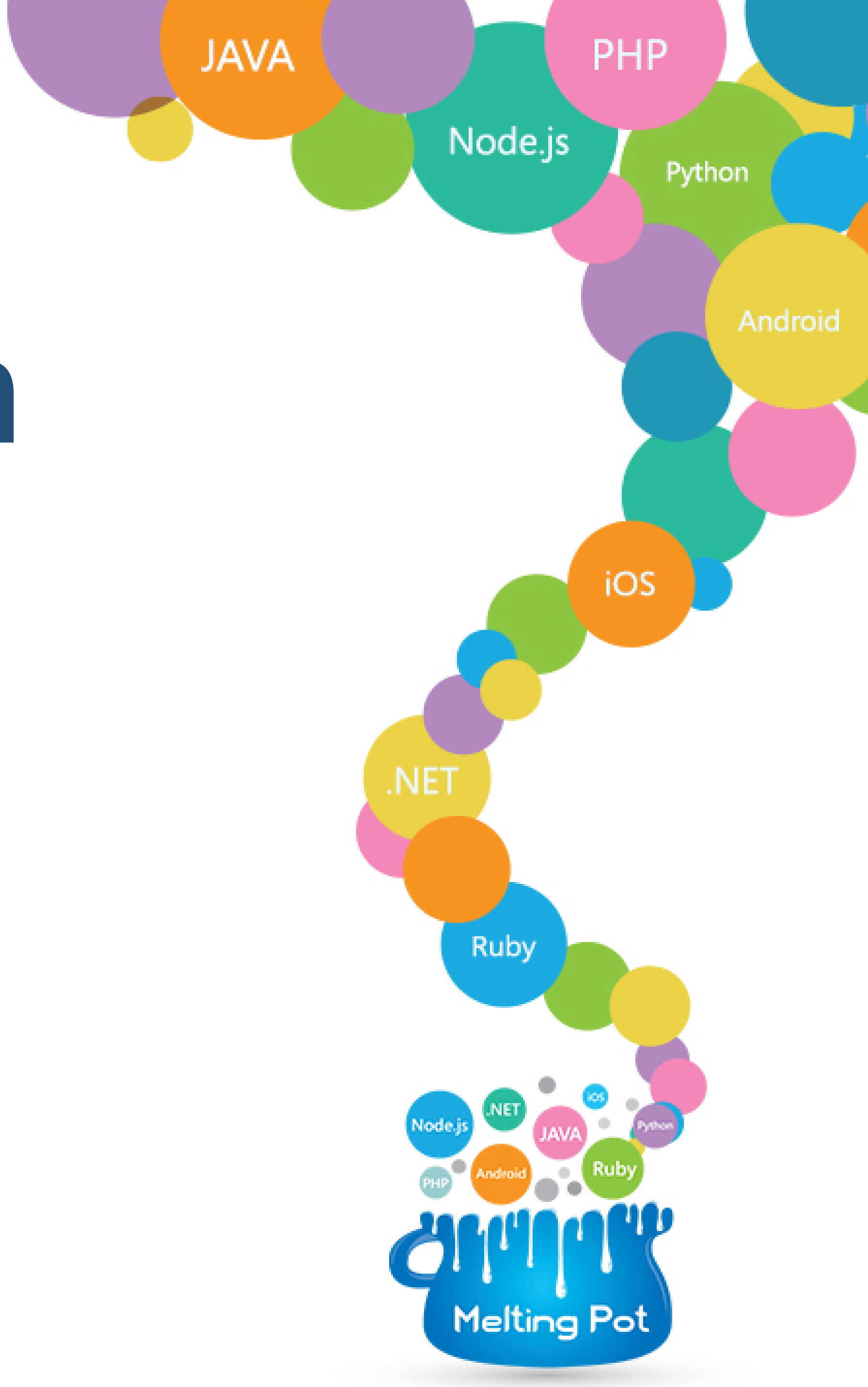
Lambda Expression & Concurrency API

김명신 부장

Principal Technical Evangelist



Agenda

완전 친절한

Lambda Expression

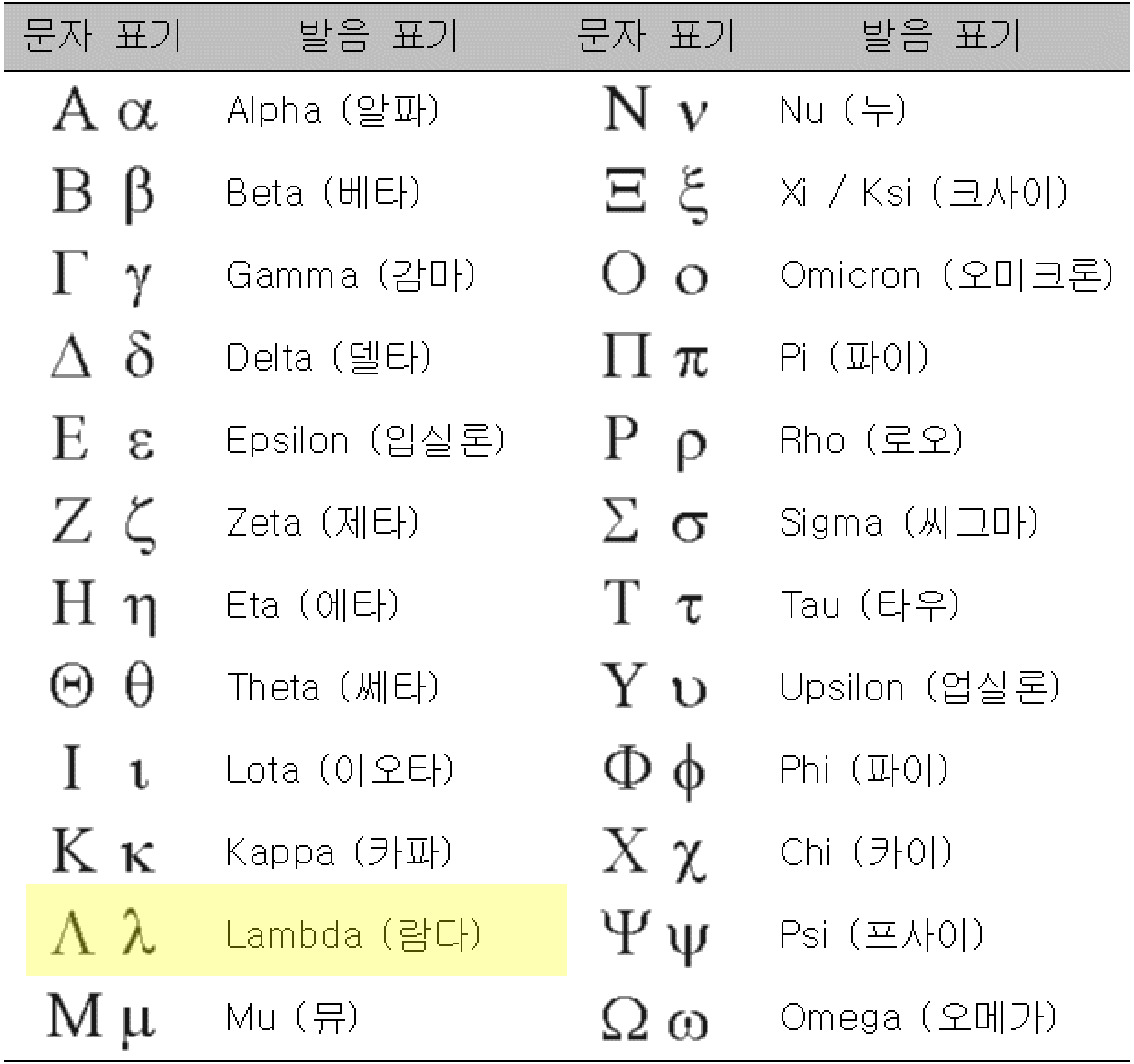
완전 불친절한

Concurrency API

완전 간단한

실천적 접근

Lambda Expression



문자 표기	발음 표기	문자 표기	발음 표기
Α α	Alpha (알파)	Ν ν	Nu (누)
Β β	Beta (베타)	Ξ ξ	Xi / Ksi (크사이)
Γ γ	Gamma (감마)	Ο ο	Omicron (오미크론)
Δ δ	Delta (델타)	Π π	Pi (파이)
Ε ε	Epsilon (입실론)	Ρ ρ	Rho (로오)
Ζ ζ	Zeta (제타)	Σ σ	Sigma (씨그마)
Η η	Eta (에타)	Τ τ	Tau (타우)
Θ θ	Theta (쎄타)	Υ υ	Upsilon (업실론)
Ι ι	Lota (이오타)	Φ φ	Phi (파이)
Κ κ	Kappa (카파)	Χ χ	Chi (카이)
Λ λ	Lambda (람다)	Ψ ψ	Psi (프사이)
Μ μ	Mu (뮤)	Ω ω	Omega (오메가)

Lambda Calculus

- Function abstraction &
  Application using variable binding and substitution
- Lambda expression
  - treats function “anonymously” $\mathrm{sqsum}(x,y) = x \times x + y \times y$ $(x,y) \mapsto x \times x + y \times y$
  - only uses functions of a single input $f{:}(X \times Y) \to Z$ $\mathrm{curry}(f){:}\ X \to (Y \to Z)$
- Higher order function
  - takes one or more functions as an input
  - outputs a function
- OOPL에도 도입 추세
  - C# 3.0(2007) / C++11(2011) / Java 8(2014)

Alonzo Church

Syntax

```
lambda-introducer:
        [ lambda-captureopt ]
lambda-capture:
        capture-default
        capture-list
        capture-default , capture-list
capture-default:
        &
        =
capture-list:
        capture ...opt
        capture-list , capture ...opt
capture:
        identifier
        & identifier
        this
lambda-declarator:
        ( parameter-declaration-clause ) mutableopt
                exception-specificationopt attribute-specifier-seqopt trailing-return-typeopt
```

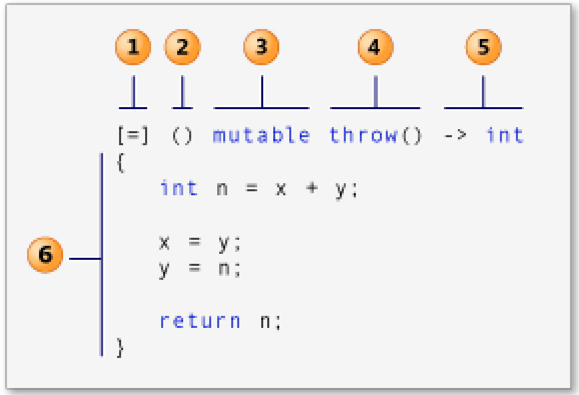
[*lambda-capture*] { *body* }

[*lambda-capture*] (*params*) { *body* }

[*lambda-capture*] (*params*) -> *ret* { *body* }

[*lambda-capture*] (*params*) *mutable exception attribute* -> *ret* { *body* }

```
[=, &x](int a1) mutable noexcept -> int
{ /* statements */ };
```



```
[=] () mutable throw() -> int
{
    int n = x + y;

    x = y;
    y = n;

    return n;
}
```

Capture clause

[lambda-capture] (params) mutable exception attribute -> ret { body }

```
int x = 10, y = 20;
[] {};              // capture 하지 않음
[x](int arg) { return x;};             // value(Copy) capture x
[=] { return x;};                      // value(Copy) capture all
[&] { return y;};                      // reference capture all
[&, x] { return y;};                   // reference capture all except x
[=, &y] { return x;};                  // value(Copy) capture all except y
[this] { return this->something;};     // this capture
[=, x] {};          // error
[&, &x] {};         // error
[=, this] {};       // error
[x, x] {};          // error
```

Capture default =, &를 하였더라도 body에서 사용하지 않았다면 capture는 일어나지 않음

Params/ret

[*lambda-capture*] (*params*) *mutable exception attribute* -> *ret* { *body* }

- 일반 함수와 다를 바 없음
- Return type deduction을 수행
  - return이 한번만 나타나거나, 혹은 없는 경우만 자동 타입 추론(C++11)
  - Body 내의 모든 반환 형이 동일할 경우 자동 타입 추론(C++ 14)

```
[](int &factor, int total) {
  if (factor == 0) return total;
  return factor;
};
```

```
[](float &factor, double total) -> double {
  if (factor == 0) return total;
  return factor;
};
```

mutable/exception

[lambda-capture] (params) mutable exception attribute -> ret { body }

- mutable
  - Lambda의 기본 call operator는 const-by-value
  - mutable을 사용하면 const를 제외하여 value(copy) capture 한 내용 수정 가능
- exception
  - throw() 혹은 noexcept와 같은 형태 가능
  - Exception throw시 terminate 수행
  - C++ 03의 throw(..) 는 사용하지 않도록

```
[x]() mutable { return ++x; };
```

```
[x]() throw() {
if (x == 0)
  throw std::bad_exception(); // warning
};
```

Review, again

```
[=, &x](int a1) mutable noexcept -> int
{ /* statements */ };
```

- Value(copy) capture all except x
- Reference capture x
- Pass by value, a1
- value capture 한 내용 수정 가능
- No exception occurred
- int return type

Lambda Expression

- constructor와 call operator를 가지고 있는 새로운 class를 생성
- Capture 구문에 따라 member variable이 추가됨
- Function object(functor)와 거의 유사함
- Capture clause가 없는 Lambda Expression
  - Stateless Lambda
  - 이 경우 calling convention을 사용하는 함수 포인터를 대체하여 사용할 수 있음 (stdcall, this call, cdecl) -> Win32 callback function과 호환

```
cout << typeid([] {}).name() << endl;
```

```
class <lambda_04197a50f746795ff56aab1f0f0bfa52>
```

QUIZ

Quiz

```
int x = 10, y = 20;

[x] { return ++x; };                           [x]() mutable { return ++x; };
[x, &] { return x + y; };                      [&, x] { return x + y; };
[=, x] { return x; };                          [=] { return ++x; };
[]() noexcept { throw bad_exception(); };      [] { throw bad_exception(); };
```

Quiz

```
int x = 1;
cout << x << endl;                  ←  1
[x]() mutable { ++x; }();
cout << x << endl;                  ←  1
[&x]() { ++x; }();
cout << x << endl;                  ←  2
```

Quiz

```
void fa(int x, function<void(void)> f) { ++x; f(); }
void fb(int x, function<void(int)> f) { ++x; f(x); }
void fc(int &x, function<void(void)> f) { ++x; f(); }

int x = 1;

fa(x, [x] { cout << x << endl; });                    1

fb(x, [](int x) { cout << x << endl; });              2

fc(x, [&x] { cout << x << endl; });                   2
```

활용 예(functor)

```
struct Less
{
  bool operator()(const int &left, const  int &right) const
  {
    return left < right;
  }
};

sort(begin(v), end(v), Less());
```

```
sort(begin(v), end(v), [](int x, int y) { return x < y; });
```

활용 예(functor)

```cpp
template<typename T>
class Less_than {
  const T val;
public:
  Less_than(const T& v) : val(v) {};
  bool operator()(const T& x) const { return x < val; };
};

int num = 5;
Less_than<int> less5{ num };
auto diff = count_if(begin(v), end(v), less5);
```

↓

```cpp
auto diff = count_if(begin(v), end(v), [num](int value) { return value < num; });
```

활용 예(Higher order Lambda Function)

```
// return lambda function
auto addtwointegers = [](int x) -> function<int(int)> {
  return [=](int y) { return x + y; };
};

// lambda function as parameter
auto higherorder = [](const function<int(int)>& f, int z) {
  return f(z) * 2;
};

auto answer = higherorder(addtwointegers(7), 8);
```

활용 예(callback function)

```
WNDCLASSEX wcex;
wcex.lpfnWndProc= [](HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) ->
LRESULT {
  switch (message) {
    case WM_COMMAND:

EnumWindows([](HWND hwnd, LPARAM lParam) -> BOOL {
  char szText[256];
  GetWindowTextA(hwnd, szText, 256);
  cout << szText << endl;
  return TRUE;
}, 0);
```

활용 예(callback function)

```
HANDLE hT = CreateThread(NULL, 0, [](LPVOID lpThreadParameter) -> DWORD {
  for (int i = 0; i < 1000; i++) {
    this_thread::sleep_for(milliseconds{ 10 });
    cout << i << endl;
  }
  return 0;
}, NULL, 0, NULL);
```

Demo

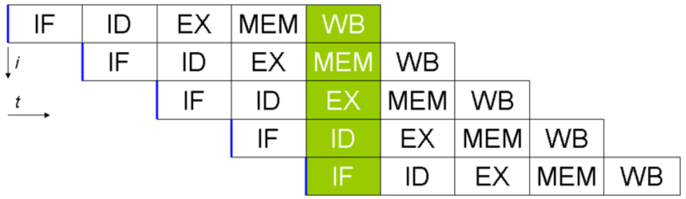
Concurrency API

Why Concurrency ?

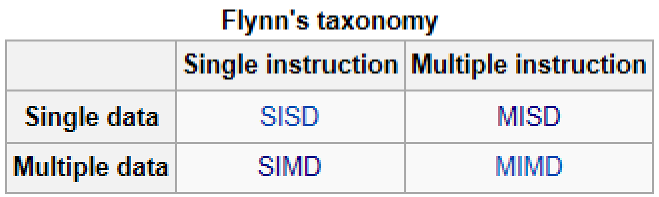
Hardware의 변화	Improve throughput	Improve responsiveness
Free lunch is over Multi Core Architecture Heterogeneous computing	Multi Core/ Multi Thread Programming Parallelization/Vectorization Heterogeneous programming	Isolate User Interface Thread Create new Working Thread

Parallelism

- Bit-level parallelism
  - 8bits→16bits→32bits→64bits→…
- Instruction-level parallelism
  - Multi-stage instruction pipelines
  - Intel’s Super scalar
- Data parallelism
  - SIMD, MIMD in Flynn’s taxonomy
- Task parallelism
  - Entirely difficult calculations can be performed on either the same of difficult s ets of data



IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB



Flynn's taxonomy

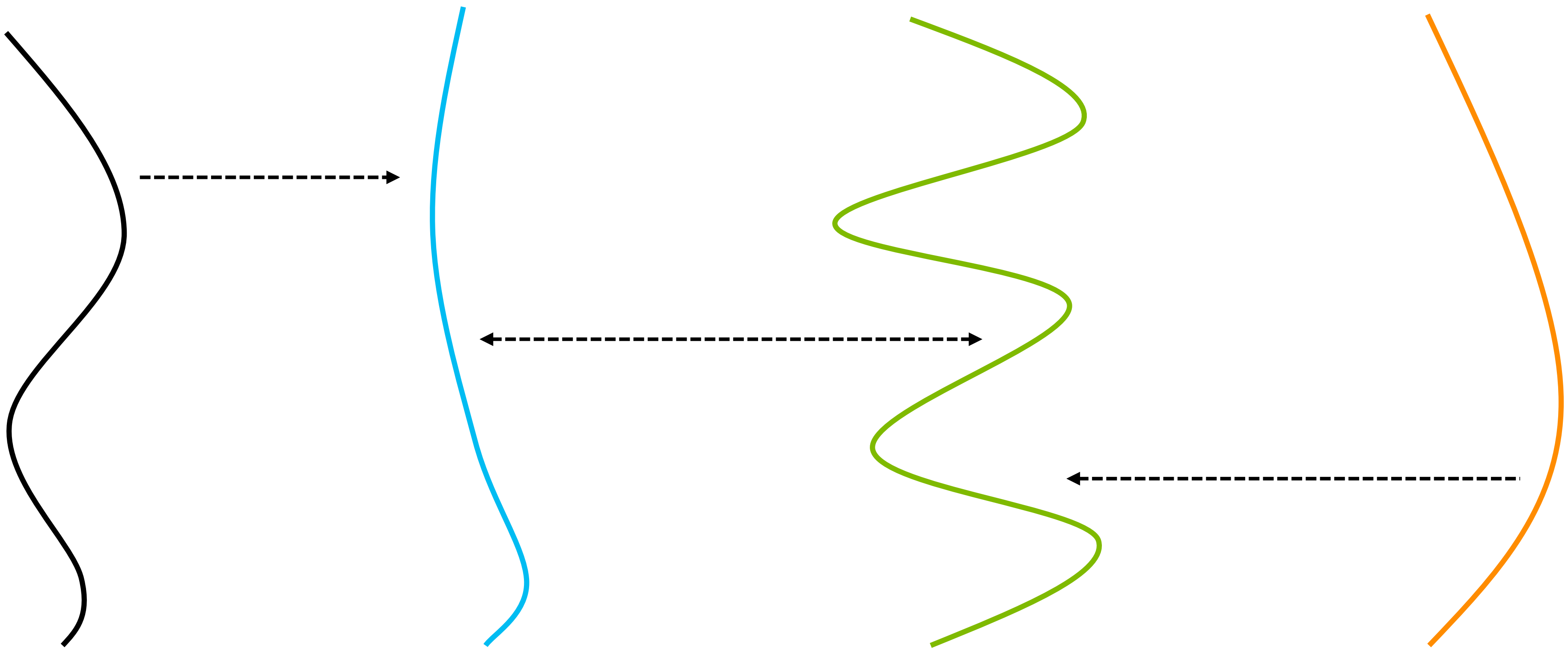
	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

이미 다 알고 있는 Concurrency

- The Execution of several tasks simultaneously
- 이미 Concurrency는 Programming에 있어 빼놓을 수 없는 도구
- 다양한 플랫폼에서 제공해 주는 Concurrency 관련 API/SDK
  - Windows API : Thread, Synchronization Object, Thread Pool, …
  - POSIX Thread : Pthread, Mutex, condition_variable, …
  - PPL(Parallel Patterns Library) : Task Parallelism, Parallel Algorithms, …
  - TBB(Intel Threading Build Blocks) : Parallelizing Loops, Atomic, Task Scheduler,…
  - 기타 등등등등, 등등등등, 등등등등, 등등등등 …
- 언어 차원에서 표준화된 도구를 제공하면 Portability가 향상 됨

한문장 요약

- Concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other



반드시 알아야 하는 경우~ 2가지 (혹은 3가지)

- 작업을 동시에 수행하는 방법
- 그들간에 통신하는 방법
- (작업을 만드는 방법)

돌발 Quiz

```
sizeof(int) = ?
```

C++의 Hardware friendly

- Hardware friendly 하지만 Hardware dependent 하지는 않음
- Some of the aspect of C++'s fundamental types are implementation defined

1=sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)<=sizeof(long long)

1<=sizeof(bool)<=sizeof(long)

sizeof(char)<=sizeof(wchar_t)<=sizeof(long)

sizeof(float)<=sizeof(double)<=sizeof(long double)

sizeof(N)=sizeof(signed N)=sizeof(unsigned N)

C++ 과 Concurrency

- Concurrency 기능은 Hardware dependent 한 구현이 필요

  → 지난 20여년간의 C++ 역사에 있어 혁신적인 변화가 필요

- Challenges
  - Memory Location, Instruction Reordering, Memory Order, Data Races, …
- 머신 아키텍트와 컴파일러 구현자가 컴퓨터 하드웨어를 최적으로 사용하기 위한 상호 협의의 결과

C++ Standard의 Concurrency

- ISO C++ 표준은 개발자들이 하드웨어의 세부적인 특성을 모르고도 프로그래밍 할 수 있도록 해주는 데 목적이 있음

- C++ Standard의 Concurrency
  - A Memory Model
  - Support for Programming without locks
  - A thread library

Modern C++의 Concurrency

A Memory Model	A Thread library	Support for Programming without locks
atomic memory_order CAS(Compare & Swap) operation fences volatile	thread thread_local mutex(timed_, recursive, ..) lock_guard / unique_lock call_once condition_varaible	packaged_task promise future shared_future async()

A Primitive type/function

- automic<T>
- automic_thread_fence(order)/automic_signal_fence(order)
- volatile
- mutex, recursive_mutex, timed_mutex, recursive_timed_mutex
- lock_guard<T>, unique_lock<T>
- call_once, condition_variable

작업을 동시에 수행하는 방법

thread, async( )

thread and async()

- thread
  - System(Platform) Level의 thread와 일대일 대응
  - 최적의 thread 개수는 여전히 미지수
- async()
  - 비동기로 수행 가능한 task를 생성하고, 이를 수행할 thread는 *thread launcher*에 위임
  - 최적의 thread 개수 등은 *thread launcher*에게 위임(보통 thread pool)

thread/join

```
void f2(const int arg) { cout << "f2(" << arg << ")" << endl; }
void f3(const int arg, int *pResult) {
  cout << "f3(" << arg << ")" << endl; *pResult = arg; }

int _tmain(int argc, _TCHAR* argv [])
{
  thread t1([] { cout << "f1()" << endl; });  // lambda expression
  thread t2(f2, 10);                          // passing argument
  int result;
  thread t3(f3, 10, &result);                 // how to get the result

  t1.join();t2.join(); t3.join();             // barrier
  cout << "Result = " << result << endl;
}
```

async()/future<T>.get()

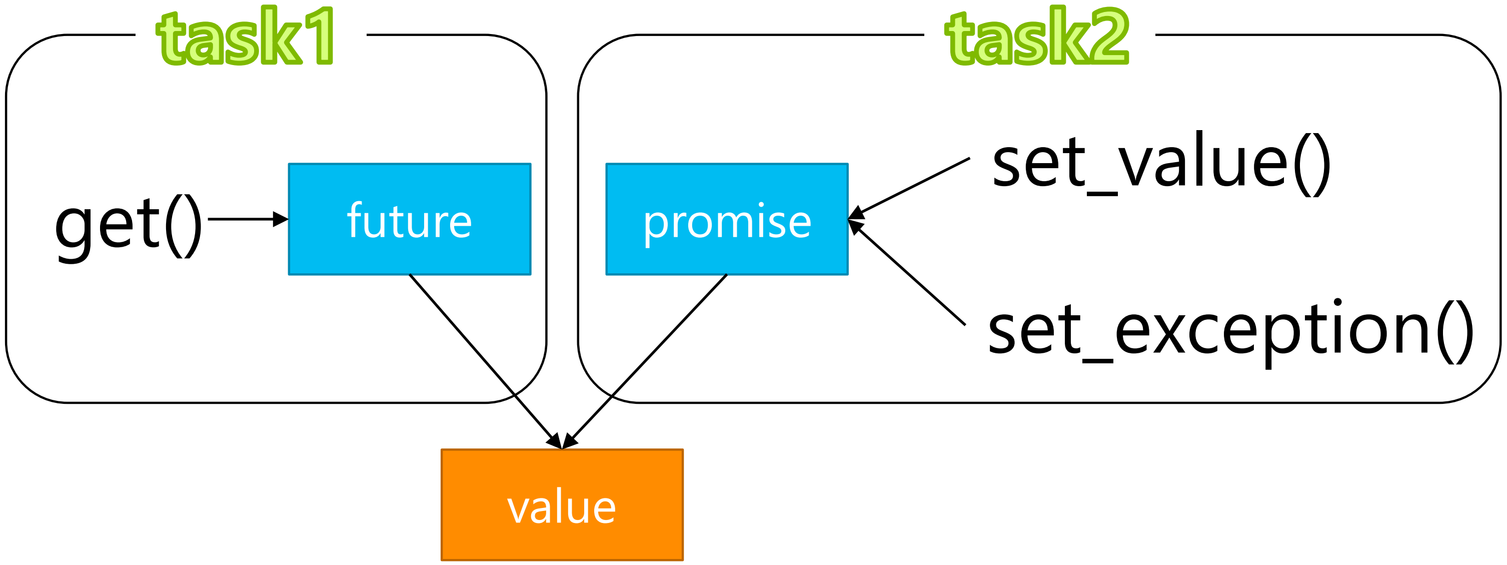
```
void f2(const int arg) { cout << "f2(" << arg << ")" << endl; }
void f3(const int arg, int *pResult) {
  cout << "f3(" << arg << ")" << endl; *pResult = arg;
}
int f4(const int arg) {
  cout << "f4(" << arg << ")" << endl; return arg;
}
int _tmain(int argc, _TCHAR* argv [])
{
  auto t1 = async([] { cout << "f1()" << endl; });   // lambda expression
  auto t2 = async(f2, 10);                    // passing argument
  int result;
  auto t3 = async(f3, 10, &result);           // how to get the result
  t1.get(); t2.get(); t3.get();
  auto t4 = async(f4, 10);                    // return value
  result = t4.get();
  cout << "Result = " << result << endl;
}
```

그들간에 통신하는 방법

future/promise

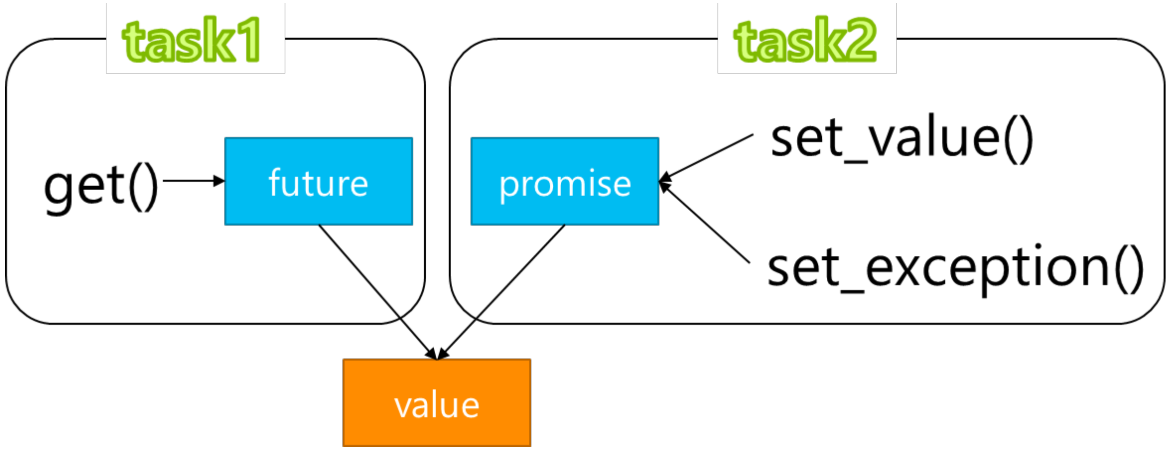
future/promise

- Communication between tasks is handled by a future/promise pair



future/promise

```
int sum(int n) { return n == 1 ? 1 : n + sum(n - 1); }
....
using value_type = int;
promise<value_type> pr;
future<value_type> fu = pr.get_future();
int num = 10;
pr.set_value(sum(num));
//pr.set_exception(make_exception_ptr(exception("error")));
try {
  value_type result = fu.get();
  cout << result;
} catch (exception &ex) {
  cout << ex.what() << endl;
}
```

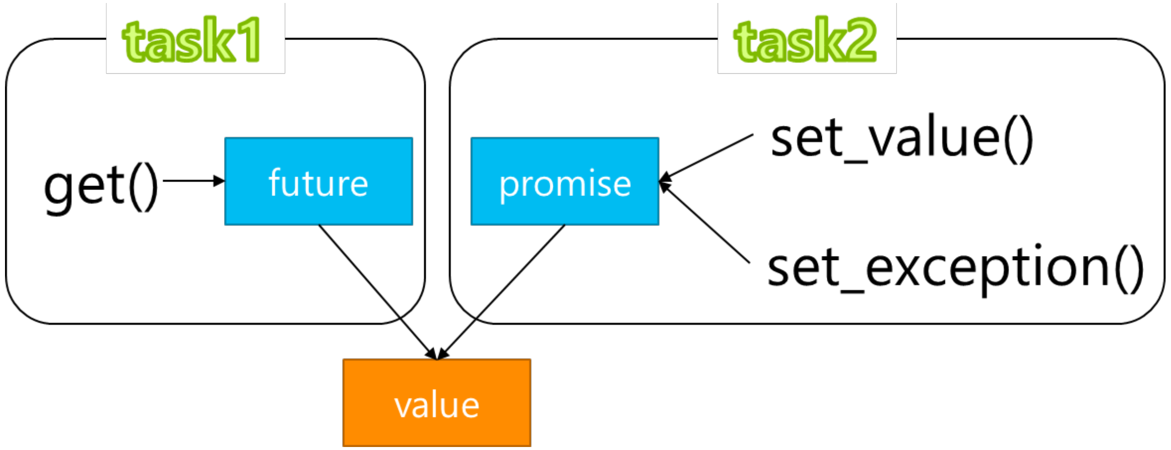


future/promise(new thread)

```
using value_type = int;
promise<value_type> pr;
future<value_type> fu = pr.get_future();

int num = 10;
thread t{ [&,num] {
  pr.set_value(sum(num));
  //pr.set_exception(make_exception_ptr(exception("error")));
}};

try {
  value_type result = fu.get();
  cout << result;
} catch (exception &ex) {
  cout << ex.what() << endl;
}
t.join();
```

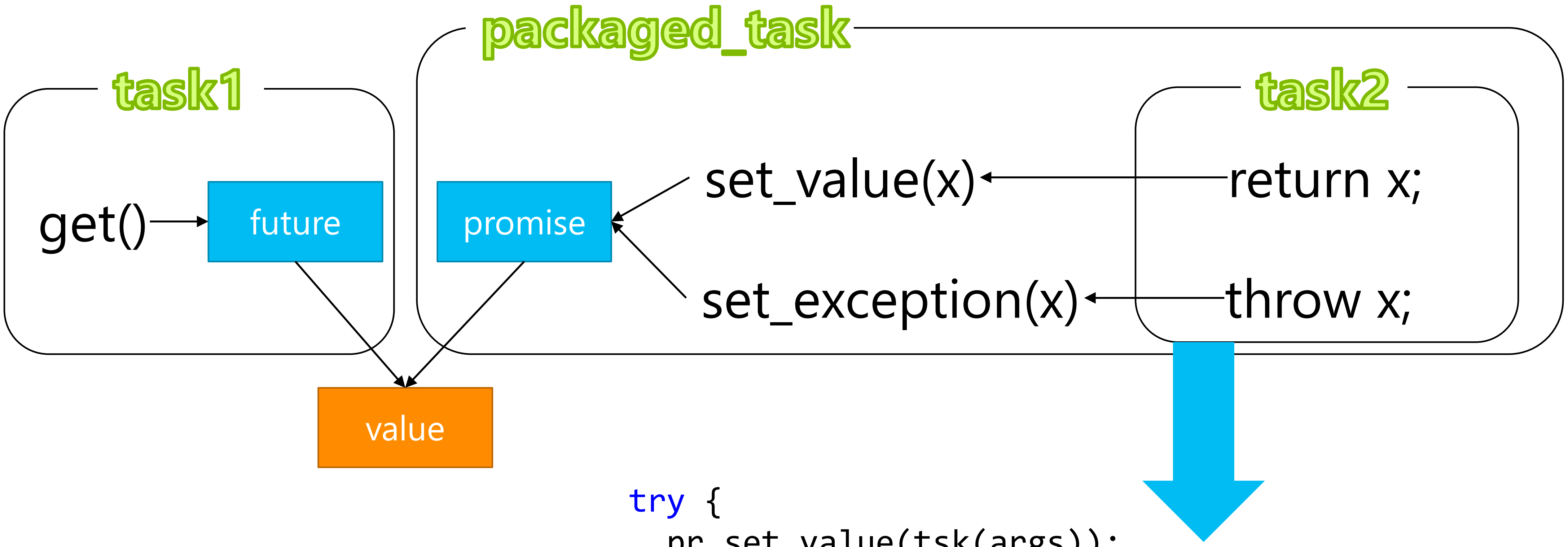


(작업을 만드는 방법)

packaged_task

packaged_task

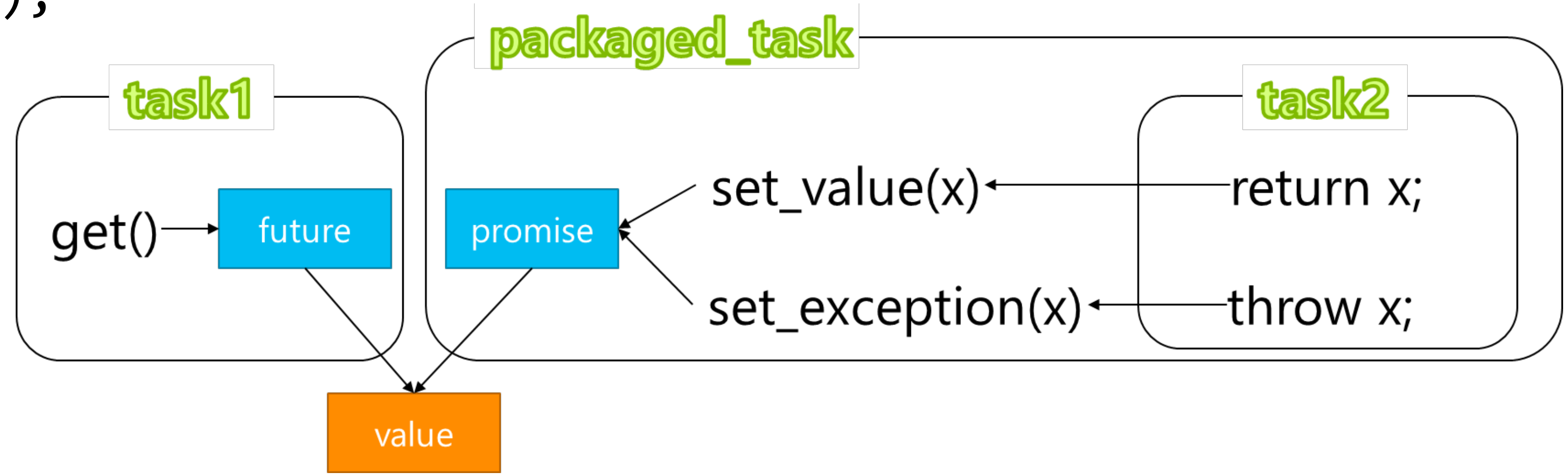
- Hold a task and a future/promise pair



```
try {
  pr.set_value(tsk(args));
} catch (...) {pr.set_exception(current_exception()); }
```

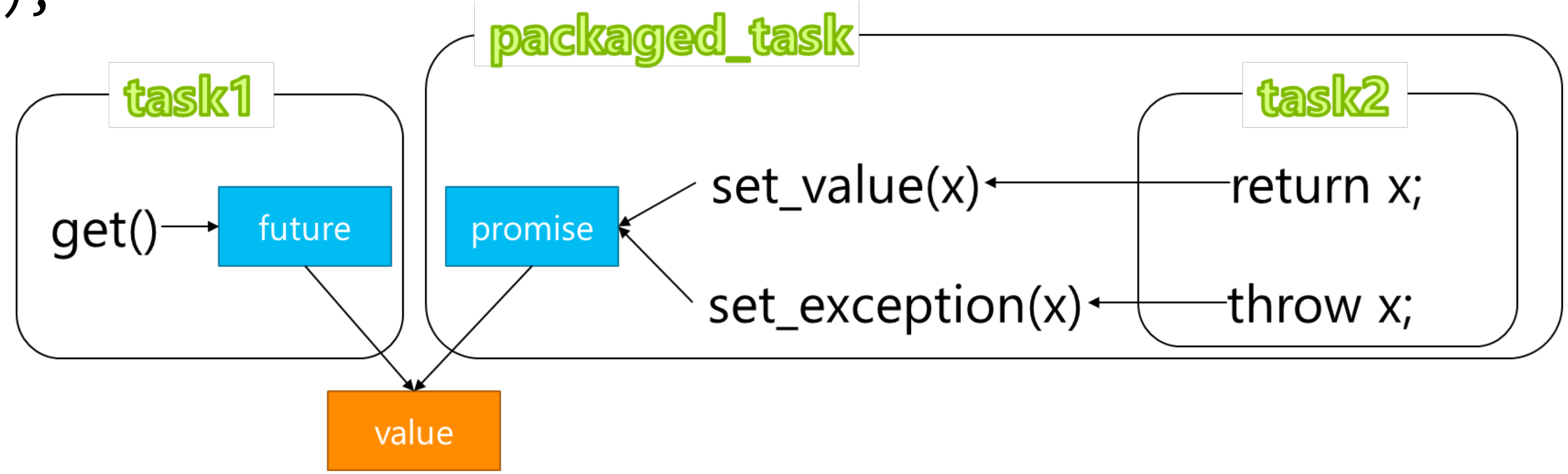
packaged_task

```
using value_type = int;
packaged_task<value_type(int)> pt{ sum };
future<value_type> fu = pt.get_future();
int num = 10;
pt(num);
try {
  value_type result = fu.get();
  cout << result;
} catch (exception &ex) {
  cout << ex.what() << endl;
}
```



packaged_task(new thread)

```
using value_type = int;
packaged_task<value_type(int)> pt{ sum };
future<value_type> fu = pt.get_future();
int num = 10;
thread t{ move(pt), 10 };
try {
  value_type result = fu.get();
  cout << result;
} catch (exception &ex) {
  cout << ex.what() << endl;
}
t.join();
```



producer/consumer

```
int sum(int n) { return n == 1 ? 1 : n + sum(n - 1); }
deque<packaged_task<int()> > q;
mutex mtx;
condition_variable cond;
```

```
void producer()
{
  int x = 1;
  while (x)
  {
    packaged_task<int()> pt{ bind(sum, x++) };
    {
      unique_lock<mutex> ul(mtx);
      q.push_back(move(pt));
    }
    cond.notify_one();
    this_thread::sleep_for(milliseconds{ 100 });
  }
}
```

```
void consumer()
{
  while (true)
  {
    packaged_task<int(void)> pt;
    {
      unique_lock<mutex> ul(mtx);
      cond.wait(ul, [] {return !q.empty(); });
      pt = move(q.front());
    }
    pt();
    cout << pt.get_future().get() << endl;
    q.pop_front();
  }
}
```

다시 살펴 보는 async( )

- 이미 promise/future 가 잘 구현되어 있음

```
using value_type = int;
future<value_type> fu = async(sum, 10);
try {
  value_type result = fu.get();
  cout << result;
} catch (exception &ex) {
  cout << ex.what() << endl;
}
```

Demo

실천적 접근

실천적 접근

- 반복적으로 사용되는 함수가 아니라면 Lambda Expression
- functor 를 써야하는 경우라면 lambda Expression 우선 고려
- fuctor를 반복해서 써야 한다면 named lambda expression 고려
- 추상화 수준이 높은 기법부터 우선 고려
- Portability : C/C++ 표준, 혹은 그 조합을 우선 고려하라.
- 마지막으로 대안이 없다면 범위를 제한하고

  Platform API를 사용하라.

Microsoft