Dynamic allocation

Dynamic allocation in C++11

- Use smart pointers instead of raw (T *) pointers

```
#include <memory>
```

  - one owner (pointer cannot be copied)
    - negligible runtime cost (almost the same as T *)

```
void f() {
  std::unique_ptr< T> p = std::make_unique< T>();   // invokes new
  std::unique_ptr< T> q = std::move( p);  // pointer moved to q
  // p is nullptr now
}
```

  - shared ownership
    - runtime cost of reference counting

```
void f() {
  std::shared_ptr< T> p = std::make_shared< T>();   // invokes new
  std::shared_ptr< T> q = p; // pointer copied; object shared between q and p
}
```

- Memory is deallocated when the last owner disappears
  - Destructor of (or assignment to) the smart pointer invokes delete when required
  - Reference counting cannot deallocate cyclic structures

Dynamic allocation in C++11

- unique_ptr is uncopiable, shared_ptr is expensive to copy
  - avoid copying whenever possible

- When passing ownership, the parameter of the receiving function may be
  - passed by value

```
void store_pointer(std::shared_ptr<T> a) {
  storage_ = std::move(a);
}
```

  - passed by r-value reference

```
void store_pointer(std::shared_ptr<T> && a) {
```

    - the ownership transfer may be conditional

```
  if ( /*...*/ )
    storage_ = std::move(a);
}
```

- In **both** cases, pass the actual argument using **move**:

```
store_pointer(std::move(p));
```

  - if passed by value, the ownership is immediately moved to the argument a
    - and later moved again to the storage
  - if passed by reference, the ownership is moved directly to the storage
    - and may remain in the actual argument if not actually moved
    - if the calling function wants to use p after calling store_pointer(std::move(p)), there must be a mechanism informing it whether store_pointer actually moved or not

Dynamic allocation in C++11

- unique_ptr is uncopiable, shared_ptr is expensive to copy
  - avoid copying whenever possible

- If you don't need to pass ownership, do not pass smart pointers
  - Use a raw T * or const T * pointer
    - in this case, it is termed a (modifying) observer (to distinguish from old-style owning T *)
  - Raw pointers are always passed by value

```
void store_pointer(T * a) {
  storage_ = a;
}
```

  - If the actual argument is a smart pointer, it must be explicitly converted

```
std::shared_ptr<T> p = /*...*/
store_pointer(p.get());
store_pointer(&*p);
```

    - The &* version is preferred – it works also on iterators or raw pointers

- The observers are not considered co-owners
  - The object may be destructed by an owner with observers present
  - It is the programmers responsibility to avoid using observers after owners die

Using pointers in modern C++

- Owner of object
  - std::unique_ptr< T>, std::shared_ptr< T>
  - Use only if objects must be allocated one-by-one
    - Possible reasons: Inheritance, irregular life range, graph-like structure, singleton
    - For holding multiple objects of the same type, use std::vector< T>
  - std::weak_ptr< T>
    - To enable circular references with std::shared_ptr< T>, used rarely
- Modifying observer
  - T *
    - In modern C++, native (raw, T*) pointers shall not represent ownership
  - Save T * in another object which needs to modify the T object
    - Beware of lifetime: The observer must stop observing before the owner dies
    - If you are not able to prevent premature owner death, you need shared ownership
- Read-only observer
  - const T *
  - Save const T * in another object which needs to read the T object
- Besides pointers, C++ has references (T &, const T &, T &&)
  - Used (by convention) for temporary access during a function call etc.

Owners and observers

- Example – unique ownership

```
auto owner = std::make_unique< T>();      // std::unique_ptr< T>
```

  - Observer

```
auto modifying_observer = owner.get(); // T *
auto modifying_observer2 = &*owner;    // same effect as .get()
```

  - Read-only observer

```
const T * read_only_observer = owner.get();  // implicit conversion
auto read_only_observer2 = (const T *)owner.get();  // explicit conversion
const T * read_only_observer3 = modifying_observer; // implicit conversion
```

- Owner pointers can point only to a complete dynamically allocated block
- Observer pointers can point to any piece of data anywhere
  - Parts of objects

```
auto part_observer = & owner->member;
```

  - Static data

```
static T static_data[ 2];
auto observer_of_static = & static_data[ 0];
```

  - Local data (their lifetime is limited – avoid propagating observers outside of their scope)
  - Using references (T &) is preferred here to signalize the limited lifetime

```
void g( T * p);
void f() { T local_data; g( & local_data); }
```

Dynamic allocation

- Dynamic allocation is slow
  - compared to static/automatic storage
  - the reason is cache behavior, not only the allocation itself
- Use dynamic allocation only when necessary
  - variable-sized or large arrays
  - polymorphic containers (containing various objects using inheritance)
  - object lifetimes not corresponding to function invocations
- Avoid data structures with individually allocated items
  - linked lists, binary trees, ...
    - std::list, std::map, ...
  - prefer contiguous structures (vectors, hash tables, B-trees, etc.)
  - avoiding std::map is difficult - do it only if speed is really important
- This is how C++ programs may be made faster than C#/java
  - C#/java requires dynamic allocation of every class instance

Ukazatelé/reference - konvence

Ukazatele vs. reference

- C++ definuje několik druhů odkazů
  - Chytré ukazatele – unique_ptr<T>, shared_ptr<T>
  - Syrové ukazatele – T *, const T *
  - Reference – T &, const T &, T &&

- Technicky všechny formy umožňují téměř všechno
  - Přinejmenším za použití nedoporučovaných triků
  - Použití referencí je syntakticky odlišné od ukazatelů

- Použití určité formy odkazu signalizuje úmysl programátora
  - Konvence (a pravidla jazyka) omezují možnosti použití předaného odkazu
  - Konvence omezují nejasnosti týkající se extrémnějších situací:
    - Co když někdo zruší objekt, se kterým pracuji?
    - Co když někdo nečekaně modifikuje objekt?
    - …

Předávání ukazatelů a referencí – konvence pro C++11

	Co smí příjemce?	Jak dlouho?	Co mezitím smějí ostatní?
std::unique_ptr<T>	Změnit obsah, zrušit objekt	Libovolně	Nic (obvykle)
std::shared_ptr<T>	Změnit obsah	Libovolně	Číst/měnit obsah
T *	Změnit obsah	Až do dohodnutého okamžiku	Číst/měnit obsah
const T *	Číst obsah	Až do dohodnutého okamžiku	Číst/měnit obsah
T &	Změnit obsah	Po dobu běhu funkce/příkazu	Nic (obvykle)
T &&	Ukrást obsah		Nic
const T &	Číst obsah	Po dobu běhu funkce/příkazu	Nic (obvykle)

Ukládání hodnot vedle sebe

Pole a n-tice

	Homogenní (pole)	Polymorfní (n-tice)

Pevná velikost	// s kontejnerovým rozhraním
	static const std::size_t n = 3;
	std::array< T, n> a;
	
	a[0] = /*...*/;
	a[1].f();
		// struktura/třída
		struct S { T1 x; T2 y; T3 z; };
		S a;
		
		a.x = /*...*/;
		a.y.f();

	// syrové pole (nedoporučeno)
	static const std::size_t n = 3;
	T a[n];
	
	a[0] = /*...*/;
	a[1].f();
		// pro generické programování
		std::tuple< T1, T2, T3> a;
		
		std::get< 0>(a) = /*...*/;
		std::get< 1>(a).f();

Proměnlivá velikost	std::size_t n = /*...*/;
	std::vector< T> a(n);
	
	a[0] = /*...*/;
	a[1].f();
		std::vector< std::unique_ptr< Tbase>> a;
		a.push_back(std::make_unique< T1>());
		a.push_back(std::make_unique< T2>());
		a.push_back(std::make_unique< T3>());
		
		a[1]->f();

Pole a n-tice v paměti

`std::array< T, 3>`

nebo

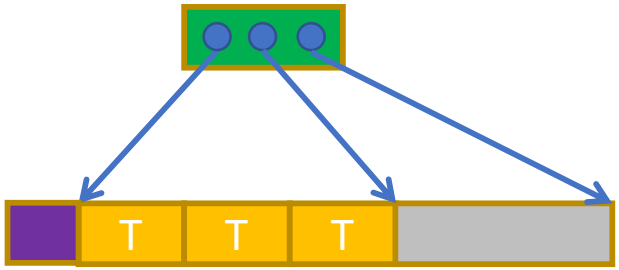
`T[ 3]`

`struct { T1 x; T2 y; T3 z;}`

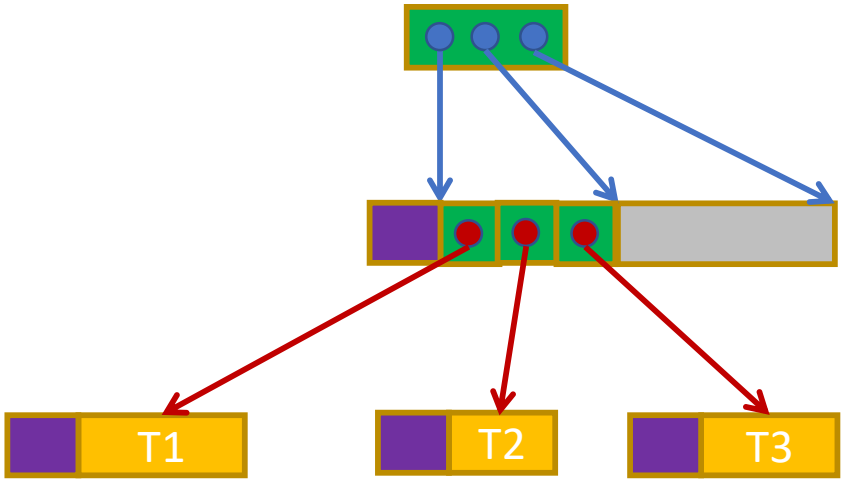
nebo

`std::tuple< T1, T2, T3>`

`std::vector< T>`



`std::vector< std::unique_ptr<Tbase>>`



Smart pointers and containers

	number of elements	storage	ownership	move	copy
array<T,N>	fixed N	inside	(unique)	by elements	by elements
optional<T>	0/1				
unique_ptr<T>		individually allocated	unique	transfer of ownership	N.A.
shared_ptr<T>			shared		sharing
unique_ptr<T[]>	any	contiguous block	unique		N.A.
shared_ptr<T[]>			shared		sharing
vector<T>			unique		by elements
deque<T>		several contiguous blocks			
other containers		individually allocated			

Smart pointers and containers

<table>
<tr><th></th><th>number of elements</th><th>storage</th><th>allocation (en masse)</th><th>insert/erase elements</th><th>random access</th></tr>
<tr><td>array<T,N></td><td>fixed, N</td><td rowspan="2">inside</td><td>(when constructed)</td><td>N.A.</td><td>[i]</td></tr>
<tr><td>optional<T></td><td rowspan="3">0/1</td><td>.emplace(...)</td><td rowspan="4">.reset()</td><td rowspan="3">N.A.</td></tr>
<tr><td>unique_ptr<T></td><td rowspan="2">individually allocated</td><td>= make_unique<T>(...)</td></tr>
<tr><td>shared_ptr<T></td><td>= make_shared<T>(...)</td></tr>
<tr><td>unique_ptr<T[]></td><td rowspan="5">any</td><td rowspan="3">contiguous block</td><td>= make_unique<T[]>(n)</td><td rowspan="4">[i]</td></tr>
<tr><td>shared_ptr<T[]></td><td>= make_shared<T[]>(n)</td></tr>
<tr><td>vector<T></td><td rowspan="3">vector<T>(n)
or
.resize(n)</td><td rowspan="2">may move elements</td></tr>
<tr><td>deque<T></td><td>several contiguous blocks</td></tr>
<tr><td>other containers</td><td>individually allocated</td><td>elements never move</td><td>no</td></tr>
</table>