Run time type information (1)

Sometimes it is difficult or even impossible to specify the type of pointers. In that case we may declare the type as *auto*. But later (especially during debugging but for other reasons too) we may need to know what is the actual type.

Operator typeid(expression) returns an object of standard class *type_info*. Function *name()* of this class returns a string specifying the type of result of the expression.

Examples:

```
int i;
Date d, *pd = new Date;
cout << typeid(i).name() << endl; // prints "int"
cout << typeid(d).name() << endl; // prints "class Date"
cout << typeid(pd).name() << endl; // prints "class Date *"
cout << typeid(*pd).name() << endl; // prints "class Date"
```

The *type_info* objects can be compared. Example:

```
Date *pd1 = new Date, *pd2 = new Date;
cout << boolalpha << (typeid(*pd1)) == typeid(*pd2)) << endl; // prints "true"
```

If you have a chain of inherited classes then the *typeid* operator works correctly only if the base class has at least one virtual function (for example, the destructor).

Run time type information (2)

It is always better to apply the *typeid* operator not to the pointer but (using dereference operator) to the object itself.

Example:

```
class Base {…..};
class Derived : public Base { …..};
Derived *pd = new Derived;
Base *pb = pd;
cout << typeid(pb).name() << " " << typeid(pd).name() << endl;
 // Prints "class Base * class Derived *"
 // Formally correct, but actually pb points to an object of class derived
cout << typeid(*pb).name() << " " << typeid(*pd).name() << endl;
 // Prints " class Derived class Derived"
 // Here we have got the actual situation in memory
```

decltype specifier

Keyword *decltype* specifies the type from the result of expression:

```
decltype (expression) variable_name = initial_value;
```

The initial value is optional. Examples:

```
Date d;
decltype (d) d1; // d1 is also of type Date
decltype (d.GetYear()) i; // i is of type int
decltype (d.GetYear()) i = 2018; // i is of type int and gets initial value 2018
```

As *auto, decltype* also simplifies the work of code writers. But mostly it is used in templates.

Example:

```
template <typename T1, typename T2> void Fun(T1 a, T2 b)
{
    typedef decltype(a + b) T;
    T x, y;
    ……..
}
```

Remark that this slide presents only a simplified definition of *decltype*. A detailed discussion may be found on https://en.cppreference.com/w/cpp/language/decltype.html

Byte

Type *std::byte* was introduced in C++ version 17. Earlier, if we wanted to work with memory bytes we had to use types *signed char* or *unsigned char*. Their difference with *byte* is that a *byte* cannot have a numeric or character interpretation: it is just a sequence of 8 bits and nothing more. The *byte* supports comparing, bitwise operations and shifting, but not arithmetic operations. Explicitly it can be casted to integers and vice versa. Examples:

```
// include <cstddef> // see more on https://en.cppreference.com/w/cpp/types/byte
byte b1 { 0xFF }, b2 { 255 }, b3 = static_cast<byte>(0xFF); // but b3 = 0xFF is an error
byte  b4 { 0b11110000 };
byte b5 { 0 }; // all the bits are 0
byte b6 { 1 } ; // all the bits are 1
cout << hex << static_cast<int>(b1) << endl; // prints ff
cout << hex << to_integer<unsiged int>(b1) << endl; // prints ff
byte b7 = b1 << 1;
byte b8 = b2 | b4;
if (b5 == byte { 0 } ) // but not if (!b5)
{ ……….. }
```

To print a byte in binary format you need to use bitsets (see more in chapter "Containers"):

```
byte by { 0b10101010 };
unsigned long int lu = to_integer<unsigned long>(by);
bitset<8> bits(lu);
cout << bits << endl;  // prints 10101010
```

Any (1)

An instance of class *any* can hold a value of any type or no value at all. This feature was first introduced in C++ version 17. Examples:

```
#include <any> // see https://en.cppreference.com/w/cpp/utility/any
any a1; // no value
any a2 = 10; // has value 10, type is int
any a3 = string("Hello"); // has value "Hello", type is string
```

To know the type of value stored in *any* use method *type* and operator *typeid*. To retrieve the value stored in *any* use *any_cast*. Example:

```
if (a3.type() == typeid(string)) {
      string s = any_cast<string>(a3); // copies the contents of a3 into s
      …….. // do something with s
}
```

If you do not check the type, you may get an exception:

```
try {
    int i = any_cast<int>(a3);
}
catch (bad_any_cast &e) {
   cout << e.what() << endl;
}
```

Any (2)

You can change the value stored in *any* to another value of the same type or some other type. Example:

```
any a = string("Hello");
cout << any_cast<string>(a) << endl; // prints "Hello"
a = string("Goodbye");
cout << any_cast<string>(a) << endl; // prints "Goodbye"
a = 10;
cout << any_cast<int>(a) << endl; // prints 10
```

To access the value stored into *any* directly, cast to pointer or reference. Example:

```
any a = string("Hello");
string* p = any_cast<string>(&a); // not "any_cast<string *>"
p->insert(5, " world");
cout << any_cast<string>(a) << endl; // prints "Hello world"
string& r = any_cast<string&>(a);
r.insert(11, " champion");
cout << any_cast<string>(a) << endl; // prints "Hello world champion"
```

Turn attention that

```
any a = "Hello";
cout << a.type().name() << endl; // prints "const char *" and not "string"
```

Any (3)

To remove the contents of *any* use method *reset()*:

```
a.reset();
cout << a.type().name() << endl; // prints "void"
```

To check whether there is a value in *any* use method *has_value()*:

```
cout << boolalpha << a.has_value() << endl; // prints "false"
auto p = any_cast<string>(&a); // p is nullptr
```

Usage example: suppose we need to write function that needs an integer as input value. But this integer may be presented as variable of type *int* or as an object of class *string*. Due to *any* we may instead of two functions

```
bool fun(int);
bool fun(string);
```

write only one:

```
bool fun(any);
```

and call it like:

```
fun(200);
```

or

```
fun(string("100"));
```

The implementation is on the following slide.

Any (4)

```
bool fun(any a) {
 int i;
 if (a.type() == typeid(string)) {
     try {
              i = stoi(any_cast<string>(a));
     }
      catch (exception &e) {
              cout << e.what() << endl;
              return false; // string does not present an integer
     }
 }
 else if (a.type() != typeid(int)) {
      return false; // input value is neither integer nor string
 }
 else {
      i = any_cast<int>(a);
 }
  …… // do something with variable i
 return true;
}
```

Optional (1)

Object specified by template *optional<T>* holds an object of class *T* or nothing at all. This template was first introduced in C++ version 17.

If a function must return the pointer to result but fails, it returns *nullptr*. If a function must return the resulting object itself but fails, it may return value *nullopt*.

Example (see also https://en.cppreference.com/w/cpp/utility/optional):

```
#include <optional>
optional<int> convert(string s) {
   try  {
        return stoi(s);
   }
   catch (exception)  {
        return nullopt;
    }
}
```

Usage:

```
optional<int> oi = convert("xxx");
if (!oi)
    cout << "Failed" << endl;
else
    cout << *oi << endl;
```

Optional (2)

Alternative solution:

```
optional<int> convert(string s)
{
  optional<int> result; // automatically initializes to nullopt
  try {
       result = stoi(s);
   }
   catch (exception) { }
   return result;
}
```

Alternative usage:

```
optional<int> oi = convert("xxx");
if (!oi.has_value())
    cout << "No result" << endl;
else
    cout << oi.value() << endl;
```

If we call method *value()* but the value is not present, *bad_optional_accesss* expression is thrown. If we use deferencing to retrieve the non-existing value, the result is unpredictable.

Due to template *optional* we do not need to use tricks for expressing the failure (for example returning values like -*1, ""*, etc. symbolizing the absence of result).

Optional (3)

Class attributes or function parameters may be also optional. Example:

```
void PrintName(string first, optional<string> middle, string last) {
  cout << first << ' ';
  if (middle.has_value()) {
      cout << middle.value() << ' ';
  }
   cout << last << endl;
}
```

Usage:

```
PrintName("John", "Edward", "Smith");
PrintName("James", nullopt, "Sailor");
```

In a class:

```
class Name {
   string First;
   optional<string> Middle;
   string Last;
   Name(string s1, optional<string>s2, string s3) : First(s1), Middle(s2), Last(s3) { }
……..
};
```

Optional (4)

Examples about defining and initializing of optional values:

```
optional<int> oi; // nullopt
optional<string> os1("Hello"), os2 = "Hello";
optional<int> oi1(10), oi2 = 10, oi3 = make_optional(10), oi4 = oi3;
optional<Date> od1(Date(1, 1, 2021)), od2 = Date(1, 1, 2021), od3 { Date { 1, 1, 2021 } };
```

It is possible to compare optional values (actually to compare values wrapped into template):

```
if (o4 == o3)
    cout << "Equal" << endl;
```

Read also: https://www.bfilipek.com/2018/05/using-optional.html

Variant (1)

Variants introduced in C++ version 17 are to replace unions from classical C.

```
typedef union { double d;  int i; } UN;
UN un1 = { 10.0 }; // union contains a double value
UN un2 = { 20 };   // union contains an integer value
```

But

```
typedef union { string s;  int i; } UN;
UN un1 = { "Hello" }; // compile error, unions with non-trivial members do not work
```

The corresponding variant:

```
#include <variant> // see https://en.cppreference.com/w/cpp/utility/variant
variant<string, int> vr;
```

The variant is not empty: if the initial value is not specified, the default constructor of the first type is called. Here *vr* contains empty string.

Examples about defining and initializing of variants:

```
variant<string, int > vr1 = "Hello", vr2{ "Hello" }, vr3 = 20, vr4{ 20 };
variant<vector<int>, vector<double>> vr5 = vector<int> { 1, 2, 3 };
```

Later we can reset the value, for example:

```
vr3 = "Goodbye";
vr2 = 20;
```

Variant (2)

To know what is the type of value stored in variant use method *index()*:

```
variant<string, int > vr1 = "Hello", vr2 { "Hello" }, vr3 = { "Hello" }, vr4{ 20 };
cout << vr1.index() << endl; // prints 0
cout << vr4.index() << endl; // prints 1
```

There are also several function templates:

```
if (holds_alternative<string>(vr1))
    vr1 = "Good morning";
if (get_if<string>(&vr2))
    vr2 = "Good evening";
if (get_if<0>(&vr3))
    vr3 = "Good day";
```

To retrieve the value of type *T* use function template *get<T>()*:

```
cout << get<string>(vr1) << endl; // prints "Good morning"
cout << get<0>(vr2) << endl; // prints "Good evening"
```

But:

```
cout << get<int>(vr1) << endl; // throws exception bad_variant_access
cout << get<1>(vr1) << endl;   // throws exception bad_variant_access
```

Variant (3)

Let us have

```
variant<Date, Time> vr;
vr = Date(25, 10, 2021);
cout << get<Date>(vr).GetDate() << endl; // prints 25
```

```
Date d = get<Date>(vr);
```

Now d is the copy of Date stored in variant.

```
d.SetDay (27);
cout << get<Date>(vr).GetDate() << endl; // still prints 25
```

```
get<Date>(vr).SetDay(26);
cout << get<Date>(vr).GetDate() << endl; // prints 26
```

```
vr = d; // replace the value in variant
cout << get<Date>(vr).GetDate() << endl; // prints 27
```

An alternative is to use method emplace:

```
vr.emplace<Date>(28, 10, 2021);
cout << get<Date>(vr).GetDate() << endl; // prints 28
```

Variant (4)

Sometimes the initialization is ambiguous, for example:

```
variant<unsigned char, char> vr {100}; // which of the alternatives?, error
```

To help the compiler, use templates *in_place_type* or *in_place_index*, for example

```
#include <utility>
variant<unsigned char, char> vr1 { in_place_type<char>, 100 };
variant<unsigned char, char> vr2 { in_place_index<0>, 100 };
```

If two variants have the same alternatives in the same order, their comparison is possible:

```
variant<string, int > vr1 { "Hello" }, vr2 { "Hello" };
if (vr1 == vr2)
    cout << "Identical" << endl;
```

Variants are excellent for creating heterogeneous collctions. Example:

```
class Book { …. };
class Article { …. };
class Link { …. };
vector<variant<Book, Article, Link>> Entries;
Entries.push_back(Book ("Nicolai Josuttis", "Complete C++ 17", "978-3-96730-017-8",
2020)); // insert an object of class Book into vector
Entries.push_back(Link("Bartolomiej Filipek", "Everything you need to know about
std::variant from C++ 17", " https://www.bfilipek.com/2018/06/variant.html "));
                // insert an object of class Link into the same vector
```

Variant (5)

The values in a variant may be processed using visitors and standard function std::visit(). A visitor is a callable object that is able to accept arguments of any type defined in the current variant. The simplest visitor is a functor, for example:

```
class Visitor
{
  public:
      void operator() (Book b) { ….. } // process somehow an object of class Book
      void operator()  (Article a) { ….. }
      void operator()  (Link l) { ….. }
};
vector<variant<Book, Article, Link>> Entries;
for (variant<Book, Article, Link> v : Entries)
{
    visit(Visitor(), v); // for each element in vector the appropriate processing function is called
}
```

Tuples (1)

Tuples are generalizations of pairs: they can store any number of values. In most cases those values are of different types:

tuple <type_1, type_2, …., type_n> tuple_name(value_1, value_2, ……,value_n);

Initial values are optional:

tuple <type_1, type_2, …., type_n> tuple_name;

Example:

```
#include <tuple> // see https://en.cppreference.com/w/cpp/utility/tuple.html
tuple<long long int, string, string, double> student(123456789LL, "John", "Smith", 4.25);
```

Tuples are similar to *structs* from classical C. Difference: the elements of a tuple have indeces starting from 0 but no names. To retrieve a member of tuple use standard template *get*:

get<index>(tuple_name);

Examples:

```
cout << get<0>(student) << ' ' << get<1>(student) << ' ' << get<2>(student) << ' ' <<
get<3>(student) << endl;
cout << typeid(get<0>(student)).name() << endl; // prints __int64
```

Initial values of tuple elements may be presented by other variables. Example:

```
string string1 = "Jeans", string2 = "Wrangler";
double price = 49.99;
tuple<string, string, double> item(string1, string2, price);
cout << get<0>(item) << ' ' << get<1>(item) << ' ' << get<2>(item) << endl;
```

Tuples (2)

However:

```
string string1 = "Jeans", string2 = "Wrangler";
double price = 49.99;
tuple<string, string, double> item(string1, string2, price);
price = 59.99;
cout << get<2>(item) << endl; // still 49.99
```

To modify the values in a tuple we must use references or pointers:

```
tuple<string, string &, double &> item(string1, string2, price);
price = 59.99;
string2 = "Lee";
cout << get<0>(item) << ' ' << get<1>(item) << ' ' << get<2>(item) << endl;
                                                    // now Jeans Lee 59.99
```

or

```
string *pCompany = new string("Wrangler");
tuple<string, string *, double *> item(string1, pCompany, &price);
cout << get<0>(item) << ' ' << *get<1>(item) << ' ' << *get<2>(item) << endl;
                                                    // Jeans Wrangler 49.99
price = 45.99;
pCompany = new string("Arizona");
cout << get<0>(item) << ' ' << *get<1>(item) << ' ' << *get<2>(item) << endl;
                                                    // now Jeans Arizona 45.99
```

Tuples (3)

There is another way to construct a tuple – use method *make_tuple*:

tuple <type_1, type_2, …., type_n> tuple_name = make_tuple(value_1, value_2, ……,value_n);

It is more convinient, because we may use *auto*. Example:

```
auto item = make_tuple(10, 20, 30); // item is tuple<int, int, int>
```

or

```
tuple<string, string, double> item;
item = make_tuple("Jeans", "Wrangle", 49.99);
```

but

```
auto item = make_tuple("Jeans", "Wrangle", 49.99); // error, not able to guess the type
```

However,

```
string *pCompany = new string("Wrangler");
double price = 49.99;
tuple<string, string *, double *> item;
item = make_tuple("Jeans", pCompany, &price); // correct, works
```

but we cannot use references:

```
tuple<string &, double &> item; // error
string string2 = "Wrangler";
double price = 49.99;
auto item = make_tuple(string2, price); // correct, but we cannot tell that the
                                        // parameters must be references
```

Tuples (4)

The solution is to use utility functions *ref* and / or *cref*:

```
#include <functional>
string string1 = "Jeans", string2 = "Wrangler";
double price = 49.99;
auto item = make_tuple(ref(string1), ref(string2), ref(price));
cout << get<0>(item) << ' ' << get<1>(item) << ' ' << get<2>(item) << endl;
                                                    // prints "Jeans Wrangler 49.99"
string2 = "Lee";
price = 59.99;
cout << get<0>(item) << ' ' << get<1>(item) << ' ' << get<2>(item) << endl;
                                                    // prints "Jeans Lee 59.99"
```

reference_wrapper is a class template thar wraps a reference into an object. This object can be copied and assigned. Function *ref* returns such an object from the proper class. *cref* is for constant references.

Tuples include operator functions for relational operations (*operator*==, *operator*<, etc.), for example:

```
tuple<string, string, double> item1 = make_tuple("Jeans", "Wrangler", 49.99),
                                  item2 = make_tuple("Jeans", "Lee", 59.99);
cout << boolalpha << (item1 == item2) << endl; // prints false
```

Tuples (5)

Tuples include also *operator*= for assignment, for example:

```
tuple<int, int, int> t1(1, 1, 1);
tuple<double, double, double> t2(0, 0, 0);
t2 = t1; // Assignes the values of t1 to t2. If the assignment of one or more values is not
         // possible or the number of members is different, we get compile error
cout << get<0>(t2) << ' ' << get<1>(t2) << ' ' << get<2>(t2) << endl; // 1 1 1
cout << typeid(get<0>(t2)).name() << endl; // still double
```

Two different tuples can be concatenated into one with standard function tuple_cat. Example:

```
tuple<string> item1("Jeans");
tuple<string, double> item2("Wrangler", 49.99);
auto item3 = tuple_cat(item1, item2);
cout << get<0>(item3) << ' ' << get<1>(item3) << ' ' << get<2>(item3) << endl;
                                                        // Jeans Wrangler 49.99
```

Smart pointers (1)

Objects of smart pointer class (i.e. the smart pointers) automatically deallocate the memory to which they point. In the simplest cases it happens when the smart pointer goes out of its scope:

unique_ptr<item_type> pointer_name (memory_allocation_with_new_operator);

Example: // see also https://en.cppreference.com/book/intro/smart_pointers

```
void fun()
{
 ……..
 unique_ptr<Date> pDate(new Date); // local variable pDate points to an object of class Date
 cout << pDate->GetYear() << endl; // operator -> is supported
 Date date = *pDate; // dereference is supported
 if (date == Date(1, 1, 2019)
     throw new exception("Not working day"); // pDate memory automatically released
 ……
} // pDate memory automatically released
```

But there is no smart pointer arithmetics:

```
unique_ptr<double> pd(new double[10]); // allowed
for (int i = 0; i < 10; i++)
   *(pd + i) = 10; // compiler error, operations like pd++, pd[i], etc. not allowed
```

Smart pointers (2)

Copying of *unique_ptr* smart pointers is not allowed. Example:

```
unique_ptr<Date> pDate(new Date(29, 11, 2018));
unique_ptr<Date> pDate1 = pDate; // compile error
```

If you need several smart pointers to point to the same memory field, use *shared_ptr*:

```
shared_ptr<Date> pDate(new Date(29, 11, 2018));
shared_ptr<Date> pDate1 = pDate; // allowed
```

Example:

```
void fun(shared_ptr<Date>pd) {……} // usage of unique_ptr not possible
int main()
{
  shared_ptr<Date> pDate(new Date(29, 11, 2018));
  fun(pDate);
    // formal parameter pd of function fun is now out of scope but the memory of pDate
    // is not released. shared_ptr has a counter incremented each time when a new pointer
    // points to the resource and decremented when destructor of object is called. If the
    // counter becomes 0, the memory is released. Here when function fun is running, this
    // couter is 2
   return 0;
}
```

Older C++ versions define *auto_ptr* smart pointer. It is now deprecated.

Enumerations (1)

Let us have:

```
#define ENGINEER 1
#define TEACHER 2
#define PENSIONER 3
#define ADMINISTRATOR 4
class Person
{
  ……..
  int Occupation;
  ……..
};
Person John;
John.SetOccupation(ENGINEER);
```

In this way we can avoid usage of strings, i.e. the memory allocation and releasing for them.

Enumerations (2)

The alternative is to use enumeration classes:

```
enum class enum_name { list_of_constants };
```

Example:

```
enum class Occupation { Engineer, Teacher, Pensioner, Administrator };
```

Now:

```
class Person
{
  ……..
  Occupation profession;
  ……..
  void SetProfession(Occupation oc) { profession = oc; }
  Occupation GetProfession() { return profession; }
};
Person John;
John.SetProfession(Ocupation::Administrator);
cout << John.GetProfession() << endl; // prints 3
Occupation oc = Occupation::Teacher; // variable of type Occupation
cout << oc << endl; // prints 1
```

Enumerations (3)

Each constant in enumeration has an associated with it integer. By default the first value is associated with 0, the second with 1, etc. But we can set our own values, for example:

```
enum class Occupation { Engineer = 10, Teacher, Pensioner, Administrator };
        // Teacher is now associated with 11, Pensioner with 12, etc.
enum class Occupation { Engineer, Teacher, Pensioner = 10, Administrator };
        // Engineer is now associated with 0, Teacher with 1,
        // Pensioner with 10, Administrator with 11
```

Remark: enumeration classes are defined in C++ version 11 and higher. C has simple enumerations like

```
enum Occupation { Engineer, Teacher, Pensioner, Administrator };
```

They work well in C but their usage in C++, however, may lead to problems.