Constant expressions

Keyword *constexpr* specifies that it is possible to evaluate the result of a function or the value of a variable at compile time. Example:

```
#include <numbers> // see https://en.cppreference.com/w/cpp/numeric
constexpr double CircleArea(double radius) {   return numbers::pi * radius * radius; }
// now function CircleArea() can be called from constant expressions
```

The following expression is an constant expression:

```
constexpr double a1 = CircleArea(10); // value of a1 will be calculated at compile time
a1 += 10; // error – a1 is constant
```

The following expressions are not constant expressions:

```
const double a2 = CircleArea(10); // value of a2 will be calculated at run time
double a3 = CircleArea(10); // value of a3 will be calculated at run time
double a4;
cin >> a4;
double a5 = CircleArea(a4); // value of a5 will be calculated at run time
```

Constant expressions may improve the application performance. See more at https://en.cppreference.com/w/cpp/language/constant_expression and https://en.cppreference.com/w/cpp/language/constexpr

if / else with initializing

Let us have code snippet:

```
int n = fun();
if (n > 0) {
    ….. // perform some operations with n
}
else {
   ..... // perform some other operations with n
}
```

Starting from C++ version 17 we may write this snippet as follows:

```
if (int n = fun();  n > 0) {
    ….. // perform some operations with n
}
else {
   ..... // perform some other operations with n
} // from this point variable "n" is out of scope
```

Variable defined and initialized in *if*-statement is visible and has memory:

- in conditional expression of *if*-statement as well as in the conditional expressions of the following *if-else*-statements;
- in the body of *if*-statement as well as in the body of the following *if-else*-statements and also in the body of final *else*-statement

switch with initializing

Similarly to *if / else*, in C++ version 17 the *switch*-statement may also include definition and initialization of variables. Example:

```
enum class colors { Red, Green, Blue };
colors GetColor() { …… }

switch (colors wall = GetColor(); wall)
{
  case colors::Red:
        …….  // do something with variable wall
        break;
  case colors::Blue:
         ……. // do something with variable wall
        break;
 case colors::Green:
        ……. // do something with variable wall
        break;
} // from this point variable "wall" is out of scope
```

Default constructors (1)

Default constructor has no arguments. Its body may be (but not must be) empty.

If the class declaration does not contain constructors, the compiler itself generates a default constructor having empty body. But sometimes you may need a class in which there are no constructors at all. In that case write:

```
class Test {
  public: Test() = delete; // explicitly deleted default constructor
  ……….
};
```

If the class declaration contains constructors (with or without arguments), the compiler does not generate its own constructor.

It is also possible to forbid the automatic generation of default copy constructor and default *operator*= for assignment overloading:

```
class Test {
  ……..
  public: Test(const Test &) = delete;
          Test& operator=(const Test &) = delete;
  ……….
};
```

Default constructors (2)

It may happen that the programmer does not see any need to include a default constructor into his / her class declaration. But for example the C++ standard containers operate only with objects from classes having the default constructor. In that case we need to add to the declaration of our class our own empty default constructor:

```
class Test {
  public: Test() = default; // explicitly defaulted constructor
                            // we may also write Test() { }
  ……….
};
```

Shorthand *return* (1)

Let us have:

```
struct Date {
  int day, month, year;
  Date(); // default constructor implemented in file Date.cpp calls the computer's clock
  Date(int d, int m, int y); // implemented in file Date.cpp
};
```

Then instead of

```
Date GetDate() {
    Date d; // default constructor is called
    return d;
}
```

we may write

```
Date GetDate() {
    return Date(); // default constructor is called
}
```

or

```
Date GetDate() {
    return { };
}
```

return { } means that the default constructor of the return value type is called.

Shorthand *return* (2)

Similarly instead of

```
Date GetDate() {
    Date d(26, 5, 2023);
    return d;
}
```

we may write

```
Date GetDate() {
    return Date(26, 5, 2023);
}
```

or

```
Date GetDate() {
    return { 26, 5, 2023 };
}
```

Conversion constructors (1)

Let us have class

```
class Test1
{
public:
  int value;
  Test1(int i) : value(i) { }
};
```

and function

```
void TestFun1(Test1 t)
{
      cout << t.value << endl;
}
```

Then

```
TestFun1(10);  // prints 10
```

is correct because the compiler handles the constructor as a casting method: it casts integer 10 to object *t* of class *Test1*. Of course, the equivalent expression

```
TestFun1(Test1(10));
```

is better to understand. From C++ version 11 any constructor with arguments may be interpreted as casting operator or in other words, is a conversion constructor. In the earlier versions a conversion constructor had to have default values for all except one of its arguments.

Conversion constructors (2)

Let us have class

```
class Test2
{
public:
  int value1, value2;
  Test2(int i, int j) : value1(i), value2(j) { }
};
```

and function

```
void TestFun2(Test2 t)
{
      cout << t.value1 << ' ' << t.value2 << endl;
}
```

Then

```
TestFun2( { 10, 20 } ); // prints 10 20
```

is equivalent with

```
TestFun2(Test2(10, 20));
```

To prevent interpreting a constructor as casting operator declare it with keyword *explicit*, for example:

```
explicit Test2(int i, int j) : value1(i), value2(j) { }
```

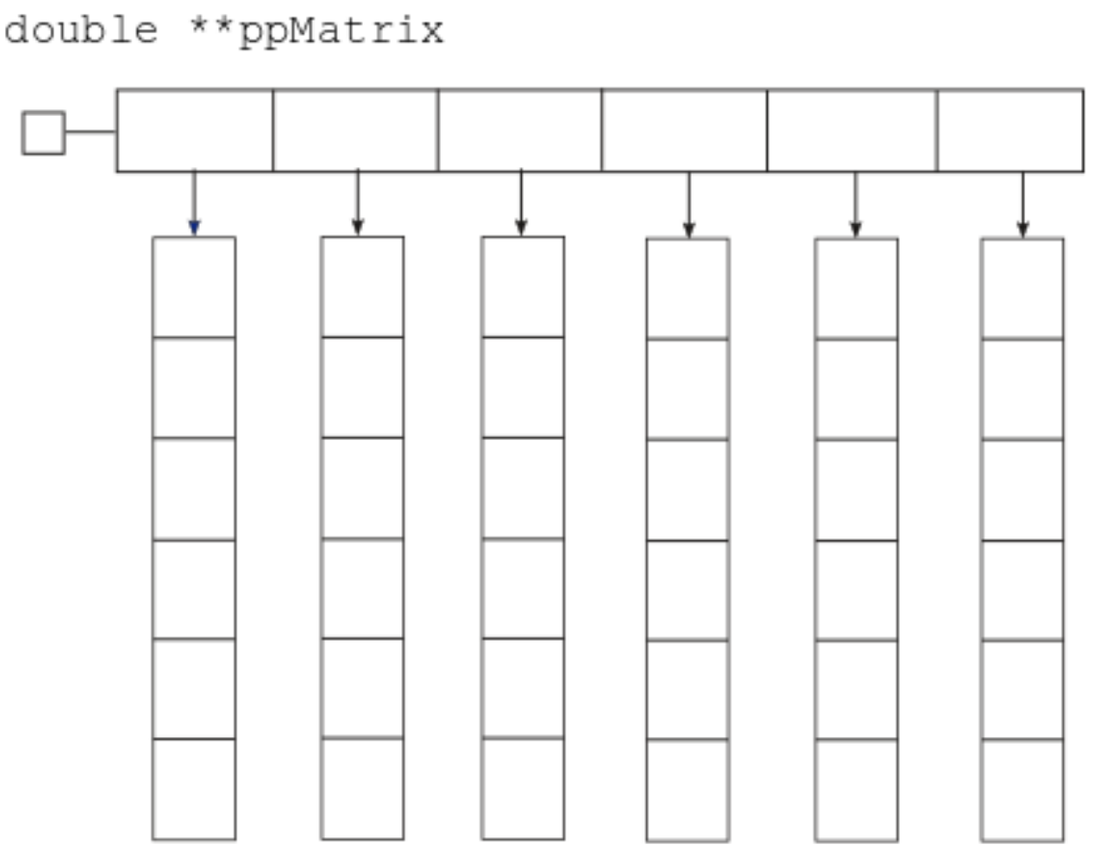
After that:

```
TestFun2( { 10, 20 } ); // compile error
```

Move semantics (1)

Suppose we have class Matrix:

```
class Matrix
{
private:
    int nRow = 0;
    int nColumn = 0;
    double **ppMatrix = nullptr;
public:
    Matrix() { }
    Matrix(int, int);
    Matrix(const Matrix &);
    ~Matrix();
    Matrix &operator=(const Matrix &);
    Matrix operator+(const Matrix &);
……..……..……..……..……..……..……..……..
};
```



Move semantics (2)

Suppose we have also a function with prototype:

```
Matrix Sum(Matrix &, Matrix &);
```

and code snippet

```
Matrix a, b;
Matrix c = Sum(a, b); // the same as c = a + b
```

As we use call by reference, *Sum* has access to *a* and *b,* therefore copying of arguments is not needed. But *Sum* has to create and return the temporary result matrix that is in turn the argument of copy constructor for *c*. The copy constructor copies all the values from result to *c*. At last the result as the local variable of *Sum* is removed (its destructor is called):

```
Matrix::Matrix(const Matrix &m)
{ // copy constructor, "this" is matrix c and m is the temporary return value of Sum
  this->nRow = m.nRow;
  this->nColumn = m.nColumn;
  this->ppMatrix = new double *[nRow];
  for (int i = 0; i < this->nRow; i++)
    *(this->ppMatrix + i) = new double[nColumn];
  for (int i = 0; i < this->nRow; i++)
     for (int j = 0; j < this->nColumn; j++)
         *(*(this->ppMatrix + i) + j) = *(*(m.ppMatrix + i) + j);
}
```

So we have 4 matrices: *a*, *b*, *c* and a temporary.

Move semantics (3)

But actually there is no need to allocate new vectors for matrix *c*, copy everything and at last destroy the temporary matrix from *Sum*. It is more reasonable to copy only the numbers of rows and columns and the pointer *ppMatrix,* i.e. simply capture the vectors from heap and use them in *c*. It is said that instead of making a copy we move heap data (actually we copy the pointers to them) from one object to another.

As at the end of *Sum* the destructor for its local temporary matrix is called anyway, during moving we must refuse to delete the data vectors. If the destructor is written in the following way:

```
Matrix::~Matrix()
{
  if (ppMatrix)
  { // if ppMatrix is set to 0, deletes nothing
      for (int i = 0; i < nRow; i++)
      {
          if (*(ppMatrix + i))
             delete *(ppMatrix + i);
      }
      delete ppMatrix;
  }
}
```

we must simply set the *ppMatrix* to *nullptr*.

Move semantics (4)

But the old copy constructor is still needed because the heap data moving is possible only when the original is a temporary matrix not needed afterwards. Consequently, we need two constructors: almost obligatory copy constructor and optional move constructor:

```
Matrix::Matrix(Matrix &&m)
{ // "this" is matrix c and m is the temporary return value of Sum
  this->nRow = m.nRow;
  this->nColumn = m.nColumn;
  this->ppMatrix = m.ppMatrix; // move data on heap
  m.ppMatrix = nullptr; // when the temporary matrix is removed, data on heap is kept
}
```

&& specifies a new data type: rvalue reference. The ordinary reference (*&*) or lvalue reference may refer only to lvalues located on a memory field that can be identified (by identifier, by array index, by pointer, etc.). The rvalue reference may refer to temporary objects we cannot identify. For example:

```
Matrix c = a + b;
```

Here a temporary matrix presenting the result of addition is created, but for us it has no name and cannot be handled. This matrix is an rvalue and it is wise to create *c* with the move constructor.

```
Matrix c = a;
```

Here we must create *c* with the copy constructor.

Move semantics (5)

The C++ compiler is able to detect whether to use copy constructor (argument is lvalue reference) or move constructor (argument is rvalue reference):

Matrix c = a + b; // if present, the move constructor is called; if not then the copy constructor

Matrix c = a; // the copy constructor is called

However,

Matrix c = Sum(a, b); // the copy constructor is called

The problem is that the compiler does not know what function *Sum* actually does and returns. For example, it may return not result of addition but one of the inputs. To force the call to move constructor, write:

Matrix c = move(Sum(a, b)); // std::move, if necessary, converts lvalue to rvalue

Unnecessary copying may also take place in *operator*= assignment overloading function. Therefore it may be wise to overload assignment twice: one with copying and the other with moving. The main ideas and the technique are the same as in case of constructors.

However, the move assignment operator function has an important difference: it must capture the heap data from temporary object standing right of the = sign, but it must also release its own heap data that has become outdated.

Matrix a(5, 5), b(5,5), c(5, 5);

…… // set values to elements of a

b = a; // actually b.operator=(a); copy assignment needed

c = a + b; // actually c.operator(a +b); move assignment may be used

Move semantics (6)

```
Matrix &Matrix::operator=(const Matrix &m)
{ // b = a; here "this" means matrix b and m means matrix a
  if (this == &m)
      return *this;
   if (!this->ppMatrix || !m.ppMatrix)
      throw new exception("Empty operand(s)");
   if (m.nRow != this->nRow || m.nColumn != this->nColumn)
      throw new exception("Dimensions do not match");
  for (int i = 0; i < this->nRow; i++)
     for (int j = 0; j < this->nColumn; j++)
           *(*(this->ppMatrix + i) + j) = *(*(m.ppMatrix + i) + j); // overwrites the old values
  return *this;
}
Matrix &Matrix::operator=(Matrix &&m)
{// c = a + b; here "this" means matrix c and m means temporary matrix got after addition
  …….. // the same as above written in brown font
   this->Destroy(); // the body is equivalent with the body of destructor
                          // removes the old values
   this->ppMatrix = m.ppMatrix;
   m.ppMatrix = nullptr;
   return *this;
}
```

Modules (1)

Header files (**.h* files) have several problems. For example, we need to avoid multiple including of the same file, sometimes we must watch on the order of *#include* directives, etc. The standard header files may consist of tens of thousands of lines of code – this is a problem for compilers. Modules are to solve those issues.
To work with modules in Visual Studio 2022 you need to install the modules for version build tools. To see the instruction type into Google "error C1011" (if the mentioned tools are not installed you get this fatal error).

Also, set the C++ language standard to *Preview – features from the latest C++ working draft (std:c++ latest)* and Enable Experimental C++ Standard Library Module to *Yes(/experimental:module)*.

Do not try to insert modules into a half ready project. Use modules as the main building components in your next project that you will start from scratch.

Caution: actually the support of modules in Visual Studio 2022 is experimental (although Microsoft claims that C++ v.20 standard modules are fully implemented). You may get strange error messages or even message E1504 (internal compiler error).

Some useful links:
https://learn.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-170
https://www.modernescpp.com/index.php/cpp20-a-first-module
https://www.modernescpp.com/index.php/c-20-module-interface-unit-and-module-implementation-unit

Modules (2)

A module consists of one (only one) module interface file and optional number of module implementation files. In Visual Studio 2022 the extension of module interface file is **.ixx*. Some other compilers use extension **.cppm*. The filename may be any. The module implementation files are ordinary *.cpp files.

To start add to your project a new module interface file (in Visual Studio it is similar to adding a new item or class). Example:

```
module; // starts to declare a module
// The next block is the global module fragment. Mostly it contains #include directives
// non-importable header files like *.h files from C
#include "stdio.h"
// Then we must set the module name. By default it is the same as the interface filename
export module Example1;
// In the next block we specify modules that our module must import. The C++ include files
// are importable. We may also import other modules from our project.
import <iostream>;
import <numbers>;
import <string>;
// Next the non-exported declarations must follow
using namespace std;
```

Modules (3)

```
// At last we must specify what our module will export. We may export classes, structs,
// functions, namespaces, enumeration classes, etc.
export class Circle
{
public:
      double radius = 0,
                  area = 0,
                  perimeter = 0;
      Circle(double r) : radius(r) {
            area = numbers::pi * radius * radius;
            perimeter = 2 * numbers::pi * radius;
      }
     string ToString()  {
          return "Radius: " + to_string(radius) + "  Area: " + to_string(area) +
                                 " Perimeter: " + to_string(perimeter);  }
      void PrintCircle() {
            printf("%s\n", ToString().c_str());  }
};
```

Here is the end of module intereface. In this simple example module implementation files are not needed.

Modules (4)

Here is file testing module Example1:

```
import <iostream>;
import Example1;
using namespace std;
int main()  {
   Circle c(10);
   cout << "Circle parameters ";
   c.PrintCircle();
   return 0;
}
```

Turn attention that although we imported *iostream* into module *Example1*, in the file for testing the module (this file does not belong to the module) we need to import it once more. *iostream* and any other module imported into our *Example1* module are not automatically exported into into files that in turn import *Example1*.

Instead of importing the C++ standard headers separately, you may import them with one sentence:

```
import std.core;
```

std.core does not include headers *<atomic>, <condition_variable>, <future>, <mutex>, <thread>*, they are joined into *std.threading*. See more from https://learn.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-170

21.06.23 – there were problems with std.core

Modules (5)

```
module;  // module interface file Example2.ixx
#include "time.h"
…… // other header files
export module Example2;
import <iostream>;
…….. // other imported modules
using namespace std;
export class Date {
private:
    int Day;
    char Month[4];
    int Year;
    …… // other attributes
public:
    Date();
    Date(int d, int m, int y);
    Date(const Date&);
    …….. // other methods
};
export Date CreateRandomDate(Date begin, Date end);   // exported function
```

Modules (6)

```
#include "time.h"  // needed because time() is a static function
module Example2; // module implementation file Date.cpp
#pragma warning(disable : 4996)
Date::Date() {  // constructor
   time(&Now);
   struct tm Tm;
   localtime_s(&Tm, &Now);
   Day = Tm.tm_mday; // 1...31
   strcpy(Month, MonthNames[Tm.tm_mon]); // 0...11
   iMonth = Tm.tm_mon + 1;
   Year = Tm.tm_year + 1900; // current year - 1900
}
…….. // other methods
Date CreateRandomDate(Date begin, Date end)
{  // implementation of function declared in module interface file
  ……..
}
```

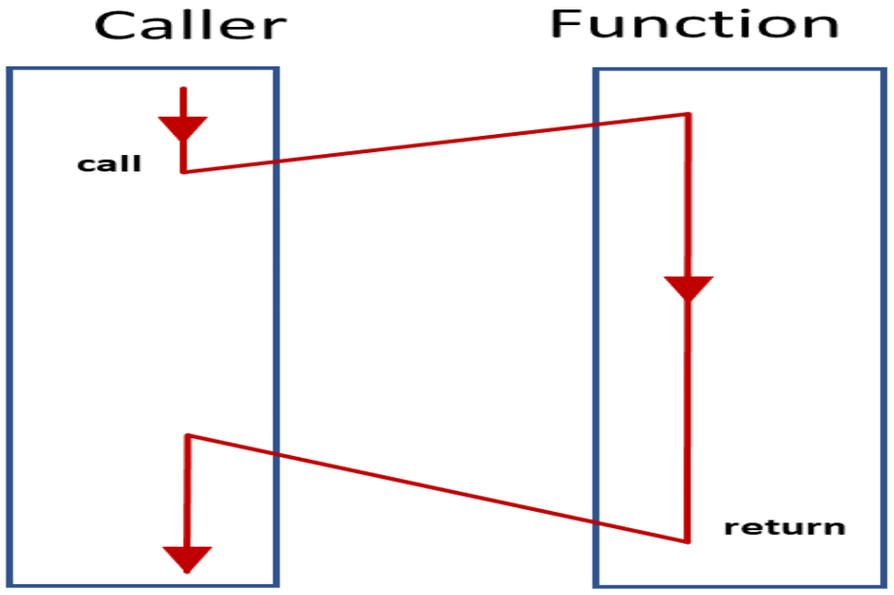
Modules (7)

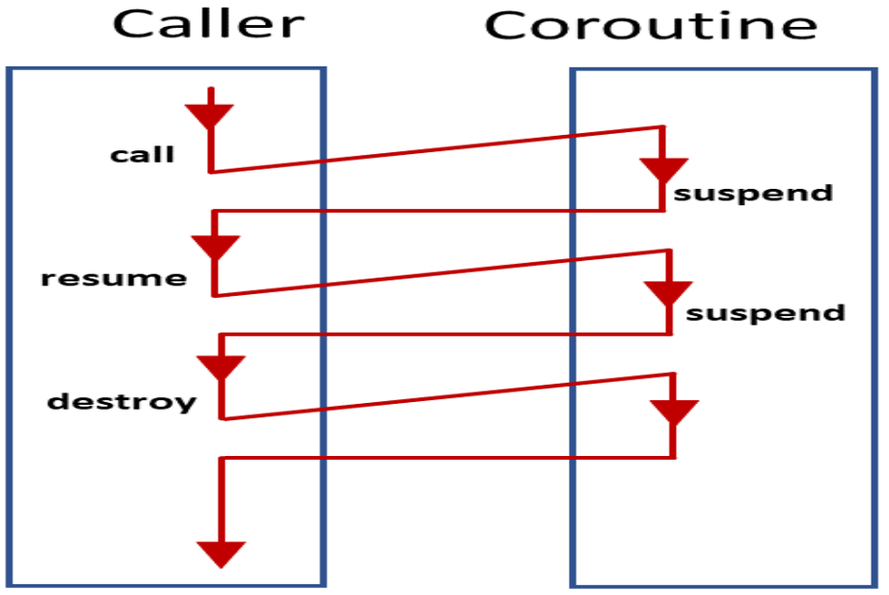
If you do not use modules and make changes in a *.*h* file, the system recompiles the complete project. Therefore, if you want to save time, put the code of class methods not into the class declaration but into a separate *.cpp file. In module files making changes in the code of class methods does not require the immediate recompilation (exception: it is true if you do not touch the function header). This is because a module interface file does not include any function implementations, even if the implementations are directly written into the interface file (as they are in *Example1*). In other words, methods implemented in *.h files are by default inline methods, methods implemented in modules are not.

Coroutines (1)

If a function (caller) calls another function (callee), then the callee runs until its end or until *return* statement or until an exception is thrown. After that the local variables of callee are removed and the caller continues to run.

If the callee is not an ordinary function but a coroutine, the callee may temporarily suspend its work and send some results to the caller. The control is switched back to the caller and the callee pauses its work. When the caller has decided that the callee must continue its work (for example, when the analysis of data sent by callee is completed), it forces the callee to resume. It is not multithreading – the caller and callee do not run concurrently. We may say that the coroutine simply runs in background and suspends and resumes its work time to time. The figure is from https://www.modernescpp.com/index.php/implementing-futures-with-coroutines:





Coroutines (2)

A coroutine behaves like an object: it has state. Coroutines do not use the stack: on suspending the state is stored in a heap-allocated separate object.

A coroutine must contain at least one of the following keywords: co_await, co_yield, co_return. A suspended coroutine may simply wait for the resume command (*co_await*). It may also exchange data with the caller (*co_yield*) and at the end send to the caller some result (*co_return* but not *return*).

Formally, a coroutine is defined as any other function:

return_value_type coroutine_name(parameter_list) { body }

The return value type cannot be *void*, C++ standard type like *int* or standard class like *string*. It must be a user's class built according to some strictly specified rules. When the coroutine starts to run, an object of this class is created (so actually we do not have a return value as such). That object called as the coroutine interface is responsible for creating, running (i.e. suspending, resuming, data exchanging) and destroying of the associated with it coroutine. Writing the code for a coroutine class is a serious work and unfortunately C++ v.20 has no any tools that would make it easier. C++ v.23 can help, but in specific simple cases only.

See the examples from *IAX0587 Examples.zip*.

The best source for studying coroutines is Nicolai M. Josuttis "C++ 20. The Complete Guide" 2022.