OOP

Gavrilut Dragos

Course 6

Summary

- Casts
- Macros
- Macros vs Inline
- Literals
- Templates
- Function templates
- Class templates
- Template specialization
- Compile-time assertion checking

- Casts

Casts

- Assuming that a class A is derived from a class B, than it is possible for an object of type A to be converted to an object of type B.
- This is a normal behavior (obviously A contains every member defined in B).
- The conversion rules are as follows:
  - It will always be possible to convert a class to any of the classes that it inherits
  - It is not possible to convert from a base class to one of the classes that inherits it without an explicit cast
  - If the cast operator is overwritten, none of the above rules apply.

Casts

- Let's consider the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };
void main(void) {
      C* c = new C()
      B* b = &c;
}
```

- Assuming the pointer "c" points to the offset 300000, what do we need to do to obtain a pointer to an object of type B* ?

Offset	Field/Var
100000	c*
100004	b*
300000	c.a1
300004	c.a2
300008	c.a3
300012	c.b1
300016	c.b2
300020	c.c3
300024	c.c4

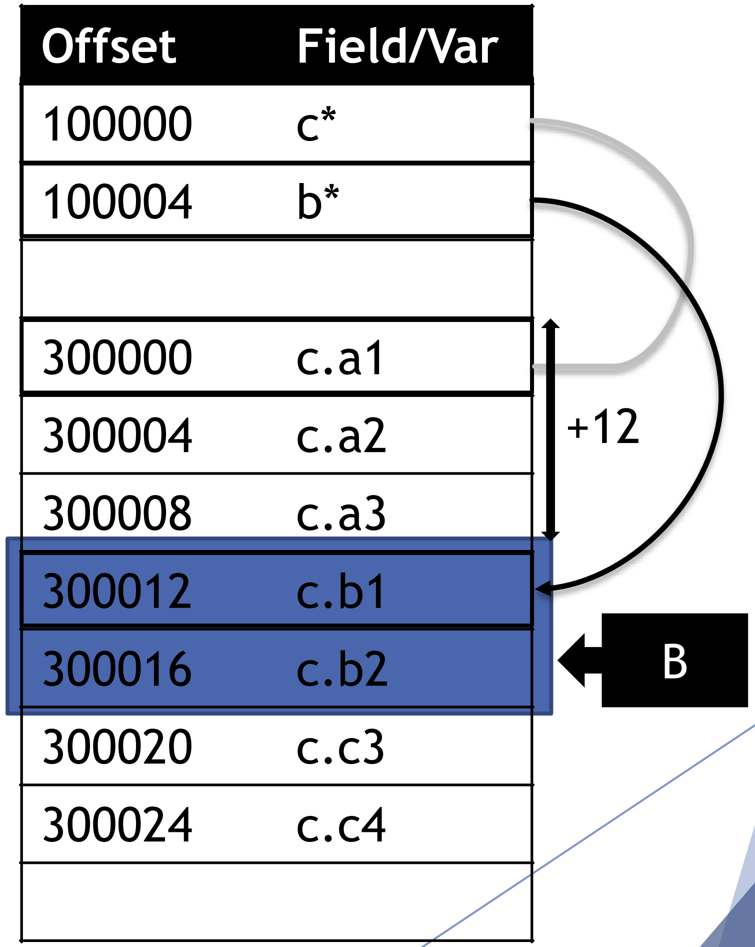
Casts

- Let’s consider the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };
void main(void) {
      C* c = new C()
      B* b = &c;
}
```

- Assuming the pointer “c” points to the offset 300000, what do we need to do to obtain a pointer to an object of type B* ?
- The simples way is to add (+12) to the offset where “c” variable points. This will position the new pointer to a location of a B type variable.



Offset	Field/Var
100000	c*
100004	b*
300000	c.a1
300004	c.a2
300008	c.a3
300012	c.b1
300016	c.b2
300020	c.c3
300024	c.c4

Casts

```
class A
{
public:
   int a1,a2,a3;
};
```

```
class B:public A
{
public:
   int b1,b2;
};
```

```
void main(void)
{
    B b;
    A* a = &b;
}
```

```
;B b;
lea       ecx,[b]
call      B::B
;A* a = &b;
lea       eax,[b]
mov       dword ptr [a],eax
```

Casts

```
class A
{
public:
   int a1,a2,a3;
};
```

```
class B
{
public:
   int b1,b2;
};
```

```
class C:public A,B
{
public:
   int c1,c2;
};
```

```
void main(void)
{
    C c;
    A* a;
    B* b;

    a = &c;
    b = &c;
}
```

Disasm

```
a = &c;
        lea     eax,[c]
        mov     dword ptr [a],eax
b = &c;
        lea     eax,[c]
        test    eax,eax
        je      NULL_CAST
        lea     ecx,[c]
        add     ecx,0Ch
        mov     dword ptr [ebp-104h],ecx
        jmp     GOOD_CAST
NULL_CAST:
        mov     dword ptr [ebp-104h],0
GOOD_CAST:
        mov     edx,dword ptr [ebp-104h]
        mov     dword ptr [b],edx
```

Casts

```
class A
{
public:
   int a1,a2,a3;
};
```

```
class B
{
public:
   int b1,b2;
};
```

```
class C:public A,B
{
public:
   int c1,c2;
};
```

```
void main(void)
{
    C c;
    A* a;
    B* b;

    a = &c;
    b = &c;
}
```

Disasm

```
a = &c;
        lea     eax,[c]
        mov     dword ptr [a],eax
b = &c;
        lea     eax,[c]
        test    eax,eax
        je      NULL_CAST
        lea     ecx,[c]
        add     ecx,0Ch (0Ch = 12)
        mov     dword ptr [ebp-104h],ecx
        jmp     GOOD_CAST
NULL_CAST:
        mov     dword ptr [ebp-104h],0
GOOD_CAST:
        mov     edx,dword ptr [ebp-104h]
        mov     dword ptr [b],edx
```

12 = sizeof(A)

Casts

```
class A
{
public:
   int a1,a2,a3;
};
```

```
class B
{
public:
   int b1,b2;
};
```

```
class C:public A,B
{
public:
   int c1,c2;
};
```

```
void main(void)
{
    C c;
    A* a;
    B* b;
    C* c2;
    a = &c;
    b = &c;
    c2 = (C*)b;
}
```

Disasm

```
c2 = (C*)b;
       cmp      dword ptr [b],0
       je       NULL_CAST
       mov      eax,dword ptr [b]
       sub      eax,0Ch (0Ch = 12)
       mov      dword ptr [ebp-110h],eax
       jmp      GOOD_CAST
NULL_CAST:
       mov      dword ptr [ebp-110h],0
GOOD_CAST:
       mov      ecx,dword ptr [ebp-110h]
       mov      dword ptr [c2],ecx
```

12 = sizeof(A)

Casts

- Let’s analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : private A, private B { public: int c1, c2; };

void main(void)
{
      C c;
      A* a = &c;
}
```

```
error C2243: 'type cast': conversion from
'C *' to 'A *' exists, but is inaccessible
```

- This type of cast can be performed only if the inheritance is public.
- In this example, there is a clear translation from “A*” to “C*”, but, as A is inherit using **private** access modifier, this cast is not allowed.

Upcast / downcast

- Whenever we discuss about conversion between classes that share an inheritance relationship, there are two notions that need to be mentioned: upcast & downcast
- Let's assume that we have the following classes:
- We have the following terminology:

App.cpp (C inherits B inherits A)

```
class A { public: int a1, a2, a3; };
class B : public A { public: int b1, b2; };
class C : public B { public: int c1, c2; };
```

App.cpp

```
void main() {
    C object;
    A * base = &object;
}
```

UPCAST: From child to parent ⇔ convert to base

- Always possible and safe

and

App.cpp

```
void main() {
    A object;
    C * child = (C*)(&object);
}
```

DOWNCAST: From parent to child

- Requires explicit cast or dynamic cast
- Unsafe !!!

Upcast / downcast

- Upcasting produces an interesting secondary effect called ***object slicing***

App.cpp

```
struct A {
    int a1, a2, a3;
    A(int value) : a1(value), a2(value), a3(value) {}
};
struct B : public A {
    int b1, b2;
    B(int value): A(value*value), b1(value), b2(value) {}
};
void DoSomething(A object) {
    printf("a1=%d, a2=%d, a3=%d\n", object.a1, object.a2, object.a3);
}
void main(void) {
    B b_object(5);
    DoSomething(b_object);
}
```

DoSomething has one parameter of type A
(NOT a reference to an A !!!)

DoSomething is called with a parameter of type B

- This code will compile and will print on the screen: a1=25,a2=25,a3=25. Since an instance of B can always be casted to its base class, you can ***slice*** from an object of type B values v1 and v2 and you will obtain an object of type A that can ne send to function *DoSomething*.

Upcast / downcast

- Upcasting produces an intere   *cing*

App.cpp

```
struct A {
    int a1, a2, a3;
    A(int value) : a1(value), a2(valu
};
struct B : public A {
    int b1, b2;
    B(int value): A(value*value), b1(
};
void DoSomething(A object) {
    printf("a1=%d, a2=%d, a3=%d\n"
}
void main(void) {
    B b_object(5);
    DoSomething(b_object);
}
```

```
B b_object(5);
push         5
lea          ecx,[b_object]
call         B::B (0C71181h)
DoSomething(b_object);
mov          eax,dword ptr [b_object]
mov          dword ptr [ebp-60h],eax
mov          ecx,dword ptr [ebp-10h]
mov          dword ptr [ebp-5Ch],ecx
mov          edx,dword ptr [ebp-0Ch]
mov          dword ptr [ebp-58h],edx
sub          esp,0Ch
mov          eax,esp
mov          ecx,dword ptr [ebp-60h]
mov          dword ptr [eax],ecx
mov          edx,dword ptr [ebp-5Ch]
mov          dword ptr [eax+4],edx
mov          ecx,dword ptr [ebp-58h]
mov          dword ptr [eax+8],ecx
call         DoSomething (0C712A3h)
add          esp,0Ch
```

Only the fields a1,a2 and a3 from b_object are copied on the stack

- This code will compile and w   25. Since an instance of B can always b   *e* from an object of type B values v1 and v2   of type A that can ne send to function *DoSomething*.

Upcast / downcast

- Upcasting produces an interesting secondary effect called ***object slicing***

App.cpp

```
struct A {
    int a1, a2, a3;
    A(int value) : a1(value), a2(value), a3(value
    A(const A& obj) : A(obj.a1) { }
};
struct B : public A {
    int b1, b2;
    B(int value): A(value*value), b1(value), b2(v
};
void DoSomething(A object) {
    printf("a1=%d, a2=%d, a3=%d\n", object.a1, o
}
void main(void) {
    B b_object(5);
    DoSomething(b_object);
}
```

```
B b_object(5);
push        5
lea         ecx,[b_object]
call        B::B (0C71181h)
DoSomething(b_object);
sub         esp,0Ch
mov         ecx,esp
lea         eax,[b_object]
push        eax
call        A::A (0A4137Fh)
call        DoSomething (0A412A3
add         esp,0Ch
```

Copy-ctor for class A

- However, if we add a copy-constructor to class A, it would be used.

Upcast / downcast

- Upcasting produces an interesting secondary effect called ***object slicing***

App.cpp

```
struct A {
    int a1, a2, a3;
    A(int value) : a1(value), a2(value), a3(value) {}
    A(const A& obj) : A(obj.a1) { }
};
struct B : public A {
    int b1, b2;
    B(int value): A(value*value), b1(value), b2(value
};
void main(void) {
    B b_object(5);
    A a_object = b_object;
}
```

```
B b_object(5);
push        5
lea         ecx,[b_object]
call        B::B (0C71181h)
A a_object = b_object;
lea         eax,[b_object]
push        eax
lea         ecx,[a_object]
call        A::A (081137Fh)
```

Copy-ctor for class A

- However, if we add a copy-constructor to class A, it would be used. This way of creating a object from another object that inherits it, is another usage of ***object slicing***.

Casts

- Let’s analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };

void main(void)
{
      C c;
      B* b = &c;
}
```

- What happens when we convert (cast) &c to B* ?

Casts

- Let’s analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };

void main(void)
{
      C c;
      B* b = &c;
}
```

```
  lea          eax,[c]
  test         eax,eax
  je           NULL_CAST
  lea          ecx,[c]
  add          ecx,0Ch
  mov          dword ptr [ebp-0F8h],ecx
  jmp          DONE
NULL_CAST:
  mov          dword ptr [ebp-0F8h],0
DONE:
  mov          edx,dword ptr [ebp-0F8h]
  mov          dword ptr [b],edx
```

- What happens whe

Casts

- Let’s analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };

void main(void)
{
      C c;
      B* b = &c;
}
```

- But – what if we **DO NOT WANT** to change the address “***b***” points to when the cast is performed ? What if we want “***b***” to point to the exact same address as “***&c***” ?

Casts

- Let's analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };

void main(void)
{
	C c;
	B* b = (B*)((void *)(&c));
}
```

```
lea             eax,[c]
mov             dword ptr [b],eax
```

- But – what if we **DO NOT WANT** to change the address "***b***" points to when the cast is performed ? What if we want "***b***" to point to the exact same address as "***&c***" ?
- One solution is to use a double cast (first cast "***&c***" to a ***void****, and then cast that ***void**** to a ***B****)

Casts

- Let's analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };

void main(void)
{
      C c;
      B* b = reinterpret_cast<B*> (&c);
}
```

- But – what if we **DO NOT WANT** to change the address "***b***" points to when the cast is performed ? What if we want "***b***" to point to the exact same address as "***&c***" ?
- Another solution is to use ***reinterpret_cast*** keyword from C++ to do this !

Casts

- C++ has several ways / keywords that can be use to specify how a cast should behave:
  - *static_cast*
  - *reinterpret_cast*
  - *dynamic_cast*
  - *const_cast*
- All of them have the following syntax:

Syntax

```
static_cast <type to cast to> (expression)
const_cast <type to cast to> (expression)
dynamic_cast <type to cast to> (expression)
reinterpret_cast <type to cast to> (expression)
```

reinterpret_cast

- ***reinterpret_cast*** is the simplest cast mechanism that translates in changing the type of a pointer, while maintaining the same address where it points to.

Syntax
reinterpret_cast <*tip**> (ptr) ⇔ ((tip*)((void*) ptr))

- This is the fastest cast possible (it guarantees very few assembly instructions that will be used for pointer type translation).
- However, it has a downside as it allows casts between pointers of incompatible types. The result may obtain a pointer with a data that has weird memory alignment, invalid pointers, etc.

reinterpret_cast

- Let’s analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : private A, private B { public: int c1, c2; };

void main(void)
{
      C c;
      A* a = &c;
}
```

```
error C2243: 'type cast': conversion from
'C *' to 'A *' exists, but is inaccessible
```

  - This code will not compile. While there is a valid conversion from a pointer of type ***C**** to a pointer of type ***A**** due to the inheritance, as class ***A*** is inherit in a private way, the conversion (cast) is not possible.

reinterpret_cast

- Let’s analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : private A, private B { public: int c1, c2; };

void main(void)
{
      C c;
      A* a = reinterpret_cast<A*> (&c);
}
```

- This code will compile. It is important to understand the ***reinterpret_cast*** is not bounded by class type or access modifier type.
- However, the only reason this code works is that “***A***” class fields (a1,a2 and a3) are the first fields in an instance of type ***C*** (if we want to obtain a pointer to B in this manner, we will only obtain a B* pointer that overlaps over the offset of local variable ***c***).

reinterpret_cast

- reinterpret_cast **can not** be used with constant values or with variable of basic type. The following expression will not compile.

C++
int x = reinterpret_cast <int>(1000);

int y = 100;
int x = reinterpret_cast <int>(y);

```
error C2440: 'reinterpret_cast': cannot convert from 'int' to 'int'
note: Conversion is a valid standard conversion, which can be
performed implicitly or by use of static_cast, C-style cast or
function-style cast
```

- reinterpret_cast can however, copy the value of a pointer to a variable (even if that variable is not large enough to hold the entire pointer value).

C++	ASM
int x = reinterpret_cast<int>(“test”);	mov dword ptr [x],205858h
char x = reinterpret_cast<char>(“test”);	mov eax, 205858h mov byte ptr [x],al

```
warning C4302:
'reinterpret_cast':
truncation from 'const
char *' to 'char'
```

- In this case **0x205858** is actually the address/offset where the string “test” is located in memory.

reinterpret_cast

- The same logic goes for float data types (float / double). Just like in the case of int, they can not be casted in this way.

Cod C++
float x = reinterpret_cast<float>(1.2f);
double x = reinterpret_cast<double>(1.2);

```
error C2440: 'reinterpret_cast': cannot convert from ‘float' to
‘float’
note: Conversion is a valid standard conversion, which can be
performed implicitly or by use of static_cast, C-style cast or
function-style cast
```

- reinterpret_cast can however be used with references to change their values. This includes references of a different type that the actual value.

Cod C++	ASM

int number = 10;
reinterpret_cast<int &>(number) = 20;
	mov dword ptr [number],20

int number = 10;
reinterpret_cast<**char** &>(number) = 20;
	mov **byte** ptr [number],20

reinterpret_cast

- reinterpret_cast can also be used for direct memory access to the actual code (in other words, it can be used to convert o pointer to a function, to a pointer to a data type).

App.cpp

```
int addition(int x, int y)
{
	return x + y;
}

void main(void)
{
	char* a = reinterpret_cast<char*> (addition);
}
```

- In this example, “a” pointes to the actual machine code that was generated for the ***addition*** instruction.
- This is widely used by obfuscator or pieces of code that modify the behavior of a program during the runtime.

static_cast

- **static_cast** implies a conversion where:
  - Values can be truncated upon assignment
  - In case of inheritance, the address where a pointer points to can be changed
- It can also be used with initialization lists
- It **does not** check in any way the type of the object (RTTI field from vfptr pointer)
- It is in particular useful if we want to identify a function with a specific set of parameters (especially if we have multiple instances of that functions/method and calling it might produce an ambiguity).

static_cast

- static_cast **can** be used with constants. In this case, the conversion and assignment are often done in the pre-compiling phase.

C++	ASM
int x = static_cast<int>(1000);	mov dword ptr [x],1000
char x = static_cast<char>(1000);	mov byte ptr [x],232 (***232=1000 % 256***)
char x = static_cast<char>(3.75);	mov byte ptr [x],3 (***3 = int(3.75)***)
char x = static_cast<char>("test");	error C2440: 'static_cast': cannot convert from 'const char [5]' to 'char'
const char * x = static_cast<const char *>(9);	error C2440: 'static_cast': cannot convert from 'int' to 'const char *'

static_cast

- Let’s analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };

void main(void)
{
      B b;
      A* a = static_cast<A *>(&b);
}
```

```
error C2440: 'static_cast': cannot convert from 'B *' to 'A *’
note: Types pointed to are unrelated; conversion requires
reinterpret_cast, C-style cast or function-style cast
```

- This code will not compile, as there is NO possible conversion from a pointer of type B to a pointer of type A

static_cast

- Let's analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
      public:
            int b1, b2;
            operator A* () { return new A(); }
};
void main(void)
{
      B b;
      A* a = static_cast<A *>(&b);
}
```

```
error C2440: 'static_cast': cannot convert from 'B *' to 'A *'
note: Types pointed to are unrelated; conversion requires
reinterpret_cast, C-style cast or function-style cast
```

- This code will not compile, even if we add a **cast operator**. The reason is the same, we try to convert a ***B**** to an ***A**** (the cast operator requires an object, not a pointer !!!).

static_cast

- Let's analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
      public:
            int b1, b2;
            operator A* () { return new A(); }
};
void main(void)
{
      B b;
      A* a = static_cast<A *>(b);
}
```

- Now the code works and a pointer of type A* is obtained by calling the ***cast operator overload*** from class B.

static_cast

- Let’s analyze the following program:

App.cpp

```
int Add(int x, char y)
{
      return x + y;
}
int Add(char x, int y)
{
      return x + y;
}
void main(void)
{
      int suma = Add(100, 200);
}
```

```
error C2666: 'Add': 2 overloads have similar conversions
note: could be 'int Add(char,int)'
note: or       'int Add(int,char)'
note: while trying to match the argument list '(int, int)'
```

- This code will not compile
  due to the ambiguity between two functions.

static_cast

- Let’s analyze the following program:

App.cpp

```
int Add(int x, char y)
{
	return x + y;
}
int Add(char x, int y)
{
	return x + y;
}
void main(void)
{
	int suma = static_cast<int(*) (int, char)> (Add)(100, 200);
}
```

- Now the code compiles. By using the ***static_cast*** we can tell the compiler witch one of the two ***Add*** function it should use !

dynamic_cast

- ***dynamic_cast*** works by **safely** convert a pointer/reference of some type into a pointer/reference of a different type, provided there is an inheritance relationship between those two types.
- In this case , the RTTI from the vfptr pointer is used and the evaluation is done at the runtime. This means that casting using ***dynamic_cast*** is generally slower than other type of casts. However, if the cast is done, we are certain that we have obtained a pointer to a valid pointer/reference
- This also implies that RTTI field must be present ➔ ***dynamic_cast*** works on classes that have at least one virtual method.
- ***dynamic_cast*** also works if there is a clear translation/cast between classes

dynamic_cast

- Let’s analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
      public:
            int b1, b2;
};
class C : public A, public B { public: int c1, c2; };
void main(void)
{
      C c;
      B *b = dynamic_cast<B*>(&c);
}
```

- This code will compile – but keep in mind that this is a safe (clear) translation as we will ***always*** be able to obtain an object of type B from an object of type C (that inherits B).

dynamic_cast

- Let's analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
      public:
            int b1, b2;
};
class C : public A, public B { public: int c1, c2; };
void main(void)
{
      B b;
      C *c = dynamic_cast<C*>(&b);
}
```

```
error C2683: 'dynamic_cast': 'B' is not a polymorphic type
note: see declaration of 'B'
```

- This code will not compile. While there is a possible conversion from ***B**** to ***C****, it's not a **safe one**. The only case such a conversion will be safe, is if that B* was obtained from a "C" object and then re-casted back to a "***C****". As this can be validated only for classes that have virtual methods (*polymorphic types*), the compiler will issue an error.

dynamic_cast

- Let’s analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
      public:
            int b1, b2;
            virtual void f() {};
};
class C : public A, public B { public: int c1, c2; };
void main(void)
{
      B b;
      C *c = dynamic_cast<C*>(&b);
}
```

- This code will compile.
- However, “c” will be ***nullptr*** (as we can not obtain a valid ***C*** object from a ***B*** object). This is very useful as it allows the programmer to check the result of the cast and have different actions based on it.

dynamic_cast

- Let’s analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
      public:
            int b1, b2;
            virtual void f() {};
};
class C : public A, public B { public: int c1, c2; };
void main(void)
{
      C c;
      B *b = (B*)&c;
      C *c2 = dynamic_cast<C*>(b);
}
```

- Now, the code compiles and the value of “c2” point to object “c”.
- In reality, ***b**** is a pointer to the B part of object “c” (this means that we can safely convert it to a ***C****)

dynamic_cast

- Let’s analyze the following code:

App.cpp

```
class A {
      public: int a1, a2, a3;
              virtual void f2() {};
};
class B {
      public: int b1, b2;
              virtual void f() {};
};
class C : public A, public B { public: int c1, c2; };
void main(void) {
      C c;
      A *a = (A*)&c;
      B *b = dynamic_cast<B*>(a);
}
```

- This code also compiles.
- While there is no relationship between ***A*** and ***B*** , since local variable “a” points to an object of type C , that contains a “B” component, this conversion is possible.

const_cast

- **const_cast** is used to change “const” characteristics for an object. It can be used only on data of the same type (it will not convert from a type to another), it will just remove the “const” characteristic of one object.
- It is also important to mention that this type of cast can produce ***undefined behavior*** depending on the way it is used !!!

const_cast

- Let's analyze the following code:

App.cpp

```
void main(void)
{
    int x = 100;
    const int * ptr = &x;

    int * non_const_pointer = const_cast <int *>(ptr);

    *non_const_pointer = 200;
    printf("%d", x);
}
```

- This code will compile and will convert ***ptr*** const pointer to a non-constant pointer that can be used to change the value of "x"
- As a result, the program will print "200" on the screen.
- It is important that the variable/memory zone where "ptr" pointer points to is NOT located on a READ-ONLY memory page (or simple put, it should be a constant pointer obtained from a non-constant variable).

const_cast

- Let’s analyze the following code:

App.cpp

```
void main(void)
{
    const char * txt = "C++ exam";

    char * non_const_pointer = const_cast <char *>(txt);

    non_const_pointer[0] = 'c';
    printf("%s", non_const_pointer);
}
```

- This code will compile but it will most likely produce a run-time error.
- In this case, “C++ exam” is located in a section of the executable that is constant (does not have the write permission for the memory page where it is located). As a result, event if we convert the const pointer to a non-const pointer, when we try to modify it, it will crash !

const_cast

- Let's analyze the following code:

App.cpp

```
void main(void)
{
        const int x = 100;
        x = 200;
}
```

- This code will not compile, as "x" is defined as ***const***, and ***const*** variables can not be changed !

const_cast

- Let’s analyze the following code:

App.cpp

```
void main(void)
{
        const int x = 100;
        *(const_cast<int*>(&x)) = 200;

}
```

- Now , this code will compile (using the ***const_cast*** we can remove the const attribute of “x”) allowing one to change “x” value.
- The behavior of the program is however , undefined → in this context **undefined** means that the actual value of “x” may not be 200 !!!
- A runtime error will not happen because “x” is actually located on the stack (this means that it is located on a memory page with the WRITE flag set).

const_cast

- Let’s analyze the following code:

App.cpp

```
void main(void)
{
      const int x = 100;
      *(const_cast<int*>(&x)) = 200;

      printf("%d", x);
}
```

```
    const int x = 100;
mov          dword ptr [x],64h
    *(const_cast<int*>(&x)) = 200;
mov          dword ptr [x],0C8h
    printf("%d", x);
mov          esi,esp
push         64h
push         2058A8h  ; "%d"
call         printf
add          esp,8
```

-

Caz 1: Debug mode, no optimizations

- Since “x” is defined as constant, the compiler assumes that whenever “x” is use it can replace its value with the constant. This mean, that the following instruction : “*printf("%d", x);*” is actually translated by the compiler into: “*printf("%d",100);*”

const_cast

- Let’s analyze the following code:

App.cpp

```
void main(void)
{
        const int x = 100;
        *(const_cast<int*>(&x)) = 200;

        printf("%d", *(const_cast<int*>(&x)));
}
```

```
    const int x = 100;
mov          dword ptr [x],64h
    *(const_cast<int*>(&x)) = 200;
mov          dword ptr [x],0C8h
    printf("%d", x);
mov          esi,esp
push         x
push         2058A8h   ; "%d"
call         printf
add          esp,8
```

- In this case, we force the compiler to use the value of “x” through a **const_cast** ➔ as a result value 200 is printed.
- However, keep in mind that this is not a normal usage of “x”

const_cast

- Let’s analyze the following code:

App.cpp

```
class Test
{
public:
	const int x;
	Test(int value) : x(value) { }
	void Set(int value) { *(const_cast<int*>(&x)) = value; }
};
void main(void)
{
	Test t(100);
	printf("%d ", t.x);
	t.Set(200);
	printf("%d ", t.x);
}
```

- This is however, a different case. Even if “x” is a constant, every instance of class Test can have a different value for “x”. This means, that the compiler will not use the value used to construct “x”, but rather the content of memory that corresponds to “x”. As a results, this code will print first 100 and then 200.

- Macros

Macros

- Macros are methods that can be used to modify (pre-process) the code in C/C++ before compiling
- They are defined using the following sintax:
  #define <macro> <value>
- Once defined a macro cand be removed from the definition list using the following sintax:
  #undef macro
- Macros work before the compile time in a *preprocessor* phase.
- One simple way of understanding them is to think that you are within an editor and you are applying a ***replace*** command (search for a text and replace it with another text).
- In reality macros a far more complex (in terms of how the replacement is done, and what king of things can be replaced by them).

Macros

- Examples (simple macros)

App.cpp

```
#define BUFFER_SIZE     1024

char Buffer[BUFFER_SIZE];
```

- After the *preprocessor* phase the code will look like this:

App.cpp

```

char Buffer[1024];
```

Macros

- Macros work sequentially (are applied immediately after they are defined). Also, during the ***preprocessor*** phase, there is no notion of identifiers, so there can be a variable and a macro that have the same name.

App.cpp

```
void main(void)
{
	int value = 100;
	int temp;
	temp = value;
#define value	200
	temp = value;
}
```

- After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
	int value = 100;
	int temp;
	temp = value;
	temp = 200;
}
```

Macros

- Macros can be written on more than on line. To do this, the special character '\' is used at the end of each line.
  Important → after this character there shouldn't be any other characters (except EOL).

App.cpp

```
#define PRINT\
    if (value > 100) printf("Greater!");  \
      else printf("Smaller!");

void main(void)
{
      int value = 100;
      PRINT;
}
```

- After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
      int value = 100;
      if (value > 100) printf("Greater!");
      else printf("Smaller!");;
}
```

Macros

- Macros can be defined using another macro. In this case the usage of **#undef** and **#define** can change the value of a macro during *preprocessor* phase .

App.cpp

```
#define BUFFER_SIZE          VALUE
#define VALUE                1024
char Temp[BUFFER_SIZE];

#undef VALUE

#define VALUE                2048
char Temp2[BUFFER_SIZE];
```

- After *preprocessor* phase the code will look as follows:

App.cpp

```
char Temp[1024];
char Temp2[2048];
```

Macros

- Macros can be defined to look like a function.

App.cpp

```
#define MAX(x,y) ((x)>(y)?(x) : (y))

void main(void)
{
      int v1, v2;
      v1 = 100;
      v2 = 200;
      int v3 = MAX(v1, v2);
}
```

- After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
      int v1, v2;
      v1 = 100;
      v2 = 200;
      int v3 = ((v1)>(v2) ? (v1) : (v2));
}
```

Macros

- Macros that are defined to look like a function can have a variable number of parameters if the specifier '…' is used.

App.cpp

```
#define PRINT(format,...) \
      { \
            printf("\nPrint values:"); \
            printf(format, __VA_ARGS__); \
      }
void main(void)
{
      int v1, v2;
      v1 = 100;
      v2 = 200;
      PRINT("%d,%d", v1, v2);
}
```

- After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
      int v1, v2;
      v1 = 100;
      v2 = 200;
      printf("\nPrint values:");
      printf("%d,%d", v1, v2);
}
```

Macros

- Macros can use the special character ‘#’ to change a parameter into its corresponding string:

App.cpp

```
#define CHECK(condition) { \
    if (!(condition)) { printf(“The condition '%s' wasn’t evaluated correctly", #condition);
};

void main(void)
{
      int v1, v2;
      v1 = 100;
      v2 = 200;
      CHECK(v1 > v2);
}
```

- After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
      int v1, v2;
      v1 = 100;
      v2 = 200;
      if (!(v1>v2)) { printf(“The condition '%s' wasn’t evaluated correctly", “v1 > v2”); };
}
```

Macros

- Macros can use the special character ‘#’ to change a parameter into its corresponding string:

App.cpp

```
#define CHECK(condition) \
    if (!(condition)) { printf(“The condition '%s' wasn’t evaluated correctly", #condition);
};

void main(void)
{
      int v1, v2;
      v1 = 100;
      v2 = 200;
      CHECK(v1 > v2);
}
```

- After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
      int v1, v2;
      v1 = 100;
      v2 = 200;
      if (!(v1>v2)) { printf(“The condition '%s' wasn’t evaluated correctly", “v1 > v2”; };
}
```

Macros

- Macros can use this sequence of characters “##” to concatenate parameters:

App.cpp

```
#define SUM(type) \
      type add_##type(type v1, type v2) { return v1 + v2; }

SUM(int);
SUM(double);
SUM(char);
void main(void)
{
      int x = add_int(10, 20);
}
```

- After *preprocessor* phase the code will look as follows:

App.cpp

```
int add_int(int v1, int v2) { return v1 + v2; }
double add_double(double  v1, double  v2) { return v1 + v2; }
char add_char(char  v1, char  v2) { return v1 + v2; }

void main(void)
{
      int x = add_int(10, 20);
}
```

Macros

- Macros don’t support overloading (if there are two macors with the same name, the last one will replace the other, a warning will be issued in the compiler and then the code will be compiled).
- The following code won’t compile because SUM needs 3 parameters (only the second macros is used)

App.cpp

```
#define SUM(a,b) a+b
#define SUM(a,b,c) a+b+c

void main(void)
{
      int x = SUM(1, 2);
}
```

```
warning C4005: 'SUM': macro redefinition
note: see previous definition of 'SUM'
```

```
warning C4003: not enough arguments for
function-like macro invocation 'SUM’
error C2059: syntax error: ';'
```

- The following code is compiled and works properly

App.cpp

```
void main(void)
{
      #define SUM(a,b) a+b
      int x = SUM(1, 2);
      #define SUM(a,b,c) a+b+c
      x = SUM(1, 2, 3);
}
```

Macros

- Use round brackets! Macros don’t analyse the expression; they just do a text replace → the results may be different than expected.

Incorrect	Correct
`#define DIV(x,y) x/y`	`#define DIV(x,y) ((x)/(y))`
`void main(void)`	`void main(void)`
`{`	`{`
`    int x = DIV(10 + 10, 5 + 5);`	`    int x = DIV(10 + 10, 5 + 5);`
`}`	`}`

- After precompilation the code will look as follows:

Incorrect	Correct
`void main(void)`	`void main(void)`
`{`	`{`
`    int x = 10 + 10 / 5 + 5;`	`    int x = ((10 + 10) / (5 + 5));`
`}`	`}`

Macros

- Be careful when they are used in a loop. Whenever possible use ‘{‘ and‘}’ to make a complex instruction easier to understand!

Incorrect

```c
#define PRINT(x,y) \
    printf("X=%d",x);printf("Y=%d",y);
void main(void)
{
      int x = 10, y = 20;
      if (x > y)
            PRINT (x, y);
}
```

Correct

```c
#define PRINT(x,y) \
    { printf("X=%d",x);printf("Y=%d",y); }
void main(void)
{
      int x = 10, y = 20;
      if (x > y)
            PRINT (x, y);
}
```

- After *preprocessor* phase the code will look as follows:

Incorrect

```c
void main(void)
{
      int x = 10, y = 20;
      if (x > y)
      printf("X=%d",x);printf("Y=%d",y);
}
```

Correct

```c
void main(void)
{
      int x = 10, y = 20;
      if (x > y)
      {printf("X=%d",x);printf("Y=%d",y);}
}
```

Macros

- Pay attention to operators and functions that modify the parameter of the macro. Because of the substitution some calls will be made more than one time.

Incorrect (res will be 3)

```
#define set_min(result,x,y) \
    result = ((x)>(y)?(x):(y));
void main(void)
{
      int x = 2, y = 1;
      int res;
      set_min(res, x++, y);
}
```

Correct

```
#define set_min(result,x,y) {\
    int t_1 = (x); \
    int t_2 = (y); \
    result = ((t_1)>(t_2)?(t_1):(t_2)); \
}
void main(void)
{
      int x = 2, y = 1;
      int res;
      set_min(res, x++, y);
}
```

- After *preprocessor* phase the code will look as follows:

Incorrect

```
void main(void)
{
      int x = 2, y = 1;
      int res;
      result = ((x++)>(y)?(x++):(y));
}
```

Correct

```
void main(void)
{
      int x = 2, y = 1;
      int res;
      { int t_1 = (x); int t_2 = (y);
      result = ((t_1)>(t_2)?(t_1):(t_2)); }
}
```

Macros

- There is a list of macros that are predefined for any C/C++ compiler:

Macro	Value
__FILE__	The current file in which the macro will be substituted
__LINE__	The line in the current file in which the macro will be substituted
__DATE__	The date when the code was compiled
__TIME__	The time when the code was compiled
__STDC_VERSION__	Compiler version(Cx98, Cx03,Cx13,…)
__cplusplus	It is defined if a C++ compiler is used
__COUNTER__	A unique identifier (0..n) used for indexing

- Besides these, each compiler defines it’s own specific macros(for compiler version, linkage options, etc)

Macros

- The usage of **__COUNTER__** is very helpful if we want to create unique indexes in our program:

App.cpp

```
void main(void)
{
        int x = __COUNTER__;
        int y = __COUNTER__;
        int z = __COUNTER__;
        int t = __COUNTER__;
}
```

- After precompilation the code will look as follows:

App.cpp

```
void main(void)
{
        int x = 0;
        int y = 1;
        int z = 2;
        int t = 3;
}
```

- Macros vs Inline

Macro vs Inline

- Let’s analyze the following code:

App.cpp

```
int Max(int x, int y)
{
	return x > y ? x : y;
}
int main()
{
	int x, y;
	x = rand();
	y = rand();
	printf("%d", Max(x, y));
	return 0;
}
```

- OBSERVATION: This code is compiled without any compiler optimizations !

Macro vs Inline

- Let's analyze the following code :

App.cpp

```
int Max(int x, int y)
{
	return x > y ? x : y;
}
int main()
{
	int x, y;
	x = rand();
	y = rand();
	printf("%d", Max(x, y));
	return 0;
}
```

```
mov         eax,dword ptr [y]
push        eax
mov         ecx,dword ptr [x]
push        ecx
call        Max
add         esp,8
push        eax
push        offset string "%d"
call        printf
```

- In this case , ***Max*** function will be called, with "x" and "y" being pushed on the stack.

Macro vs Inline

- So … how can we optimize this program:

App.cpp

```
int Max(int x, int y)
{
	return x > y ? x : y;
}
int main()
{
	int x, y;
	x = rand();
	y = rand();
	printf("%d", Max(x, y));
	return 0;
}
```

Macro vs Inline

- So … how can we optimize this program :

App.cpp

```
int Max(int x, int y)
{
      return x > y ? x : y;
}
int main()
{
      int x, y;
      x = rand();
      y = rand();
      printf("%d", Max(x, y));
      return 0;
}
```

App.cpp (with macros)

```
#define Max(x,y) x > y ? x : y

int main()
{
      int x, y;
      x = rand();
      y = rand();
      printf("%d", Max (x, y));
      return 0;
}
```

- If we replace ***Max*** function with ***Max*** macro, then the replacement is done in the pre-execution phase. As a result, the piece of code that pushes variable on the stack and ***Max*** function assembly stubs are no longer required.

Macro vs Inline

- So … how can we optimize this program :

App.cpp

```
int Max(int x, int y)
{
	return x > y ? x : y;
}
int main()
{
	int x, y;
	x = rand();
	y = rand();
	printf("%d", Max(x, y));
	return 0;
}
```

App.cpp

```
#define Max

int main()
{
	int x, y;
	x = rand();
	y = rand();
	printf("%d", Max (x, y));
	return 0;
}
```

```
        mov         eax,dword ptr [x]
        cmp         eax,dword ptr [y]
        jle         X_IS_SMALLER
        mov         ecx,dword ptr [x]
        mov         dword ptr [ebp-4Ch],ecx
        jmp         END_IF
X_IS_SMALLER:
        mov         edx,dword ptr [y]
        mov         dword ptr [ebp-4Ch],edx
END_IF:
        mov         eax,dword ptr [ebp-4Ch]
        push        eax
        push        offset string "%d"
        call        printf
```

- In this case, the replacement modifies the code from printf

Macro vs Inline

- So … how can we optimize this program :

App.cpp

```
int Max(int x, int y)
{
	return x > y ? x : y;
}
int main()
{
	int x, y;
	x = rand();
	y = rand();
	printf("%d", Max(x, y));
	return 0;
}
```

App.cpp (inline)

```
inline int Max(int x, int y)
{
	return x > y ? x : y;
}
int main()
{
	int x, y;
	x = rand();
	y = rand();
	printf("%d", Max (x, y));
	return 0;
}
```

- Another solution is to use ***inline*** specifier. This specifier tells the compiler that we ***recommend*** to insert the code of ***Max*** function directly in the code of caller (e.g. in this case **main** function).

Macro vs Inline

- So … how can we optimize this program :

App.cpp

```
int Max(int x, int y)
{
	return x > y ? x : y;
}
int main()
{
	int x, y;
	x = rand();
	y = rand();
	printf("%d", Max(x, y));
	return 0;
}
```

App.cpp (inline)

```
inline int Max(int x, int y)
{
	return x > y ? x : y;
}
int main()
{
	int x, y;
	x = rand();
	y = rand();
	printf("%d", Max (x, y));
	return 0;
}
```

Debug mode
(no optimizations)

```
mov          eax,dword ptr [y]
push         eax
mov          ecx,dword ptr [x]
push         ecx
call         Max
add          esp,8
push         eax
push         offset string "%d"
call         printf
```

- Another solution is to use ***inline*** specifier. This specifier tells the compiler that we recommend to insert the code of ***Max*** function directly in the code of caller (e.g. in this case **main** function).

Macro vs Inline

Release mode
(with optimizations)

```
cmp          edi,eax // edi = x
cmovg        eax,edi // eax = y
push         eax
push         offset string "%d"
call         printf
```

Debug mode
(no optimizations)

```
mov          eax,dword ptr [y]
push         eax
mov          ecx,dword ptr [x]
push         ecx
call         Max
add          esp,8
push         eax
push         offset string "%d"
call         printf
```

program :

```
      return x > y ? x
}
int main()
{
      int x, y;
      x = rand();
}
```

App.cpp (inline)

```
inline int Max(int x, int y)
{
      return x > y ? x : y;
}
int main()
{
      int x, y;
      x = rand();
      y = rand();
      printf("%d", Max (x, y));
      return 0;
}
```

cmovg = Compare and Move if Greater

- Another solution is to use ***inline*** specifier. This specifier tells the compiler that we recommend to insert the code of ***Max*** function directly in the code of caller (e.g. in this case **main** function).
- If we use ***optimizations*** “x” and “y” are used as registers and the whole process is optimized.

Macro vs Inline

- Not all functions can be used with *inline* specifier

App.cpp

```
inline int Iterativ(int x,int limit)
{
      int sum = 0;
      for (; limit <= x; limit++, sum += limit);
      return sum;
}

int main()
{
      int x;
      x = rand();
      printf("%d", Iterativ(x,x-5));
      return 0;
}
```

- If this code is compiled using optimization, the compiler will integrate ***Iterativ*** function in main (mainly because ***Iterativ*** function does not contain any recursive functionalities – in this case).

Macro vs Inline

- Not all functions can be used with *inline* specifier

App.cpp

```
inline int Recursiv(int x,int limit)
{
	if (x == limit)
		return limit;
	else
		return x + Recursiv(x-1,limit);}

int main()
{
	int x;
	x = rand();
	printf("%d", Recursiv(x,x-5));
	return 0;
}
```

- If we run this code with ***optimization*** the compiler is not able to ***inline*** the code from ***Recursiv*** function:
  - /permissive- /GS /GL /analyze- /W3 /Gy /Zc:wchar_t /Zi /Gm- /O2 /sdl /Fd"Release\vc141.pdb" /Zc:inline /fp:precise /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /errorReport:prompt /WX- /Zc:forScope /Gd /Oy- /Oi /MD /FC /Fa"Release\" /EHsc /nologo /Fo"Release\" /Fp"Release\TestCpp.pch" /diagnostics:classic
  - cl.exe → 19.16.27030.1

Macro vs Inline

- Using MACROS guarantees inline replacement. On the other hand, debugging is really tricky, especially if the MACRO is complex in nature (multiple lines).

- Using **inline** specifier DOES NOT GUARANTEES that inline replacement is going to happen (it should be seen more as a suggestion for the compiler). On the other hand, debugging is easier and without any bit issues.

- In case of ***inline*** , this is used on in release mode (or to be more exact, if the optimizations are enabled (-O1, -O2, …) at the compiler level.

Macro vs Inline

- Class methods (if defined in the definition of the class) are usually considered inline as well.

App.cpp

```
class Test
{
	int x; // privat
public:
	void SetX(int value)
	{
		x = value;
	}
	int  GetX() { return x; }
};
```

- There is one exception – when the class is exported (in this case the code from the header file is usually compiled and linked in the component that imports the library). In many cases, the optimization translates into direct access the data member.

Volatile specifier

- Let’s analyze the following code:

App.cpp

```
int main()
{
	int x = rand();
	printf("%d", x);
	return 0;
}
```

```
call        rand
push        eax
push        offset string "%d"
call        printf
```

- If we compile the code in the **release** mode (or more exactly, if optimizations are in place) the compiler might decide that local variable “x” is not necessary (e.g. we ca translate the entire code into “***printf(“%d”,rand())***”)
- The solution – if we want to avoid this behavior is to use ***volatile*** specifier.

App.cpp

```
int main()
{
	volatile int x = rand();
	printf("%d", x);
	return 0;
}
```

```
call        rand
mov         x, eax

mov         eax, x
push        eax
push        offset string "%d"
call        printf
```

- Literals

Literals

- Literals are a way of providing a special meaning to numbers by preceding them with a predefined string.
- Literals can be defined in the following way:

```
<return type> operator"" _<literal_name> ( parameter_type )
```

- Where:
  - <return-type> can be any kind of type
  - <literal_name> has to be a name (form out of characters, underline, …)
  - parameter_type has to be one of the following: const char *, unsigned long long int, long double, char, wchar_t, char8_t, char16_t, char32_t)
  - The underline is not required. However, compilers issue an warning if not used !

Literals

- Let's analyze the following code:

App.cpp

```
int operator"" _baza_3(const char * x)
{
    int value = 0;
    while ((*x) != 0)
    {
        value = value * 3 + (*x) - '0';
        x++;
    }
    return value;
}
void main(void)
{
    int x = 121102_baza_3;
}
```

- What we did in this case was to explain the compiler how to interpret if a number is follower by ***_baza_3***

Literals

- Let's analyze the following code:

App.cpp

```
int operator"" _baza_3(const char * x)
{
    int value = 0;
    while ((*x) != 0)
    {
        value = value * 3 + (*x) - '0';
        x++;
    }
    return value;
}
void main(void)
{
    int x = 121102_baza_3;
}
```

```
push        offset string "121102"
call        operator "" _baza_3
add         esp,4
mov         dword ptr [x],eax
```

- Since we define the literal ***_baza_3*** as one that works with strings, the compiler will translate 121102 (a number) into "121102" (a string) that can be send to the literal conversion function.

Literals

- Let’s analyze the following code:

App.cpp

```
int operator"" Mega(unsigned long long int x)
{
    return ((int)x)*1024*1024;
}
void main(void)
{
    int x = 3Mega;
}
```

- This code will compile and run just like the previous one.
- However, the compiler will throw an warning: “*warning C4455: 'operator ""Mega': literal suffix identifiers that do not start with an underscore are reserved*”

Literals

- Let’s analyze the following code:

App.cpp

```
int operator"" _Mega(int x)
{
    return ((int)x)*1024*1024;
}
void main(void)
{
    int x = 3_Mega;
}
```

```
error C3684: 'operator ""_Mega': declaration of literal
operator has an invalid parameter list
error C3688: invalid literal suffix '_Mega'; literal operator
or literal operator template 'operator ""_Mega' not found
note: Literal operator must have a parameter list of the form
'unsigned long long' or 'const char *'
```

- This code will not compile, as we did not use a valid parameter for the defined literal.

Literals

- Let’s analyze the following code:

App.cpp

```
int  operator"" _to_int(unsigned long long int x)
{
    return (int)x;
}

void main(void)
{
    unsigned long long int a;
    int sum = a_to_int;
}
```

```
error C2065: 'a_to_int': undeclared identifier
```

- This code will not compile, as the compiler will consider ***a_to_int*** a new identifier that was not defined yet.
- This translates, that we can use this with constant numbers or constant string (but not with variables).

Literals

- Let’s analyze the following code:

App.cpp

```
void operator"" _print_dec(unsigned long long int x)
{
    printf("Decimal: %d\n", (int)x);
}
void operator"" _print_hex(unsigned long long int x)
{
    printf("Hex: 0x%X\n", (unsigned int)x);
}
void main(void)
{
    150_print_dec;
    150_print_hex;
}
```

- A literal does not have to return a value. In this case we used literals to print the value of a number (as a decimal value and as a hexadecimal value).
- This code will print 150 and 0x96 on the screen.

Literals

- Let’s analyze the following code:

App.cpp

```
char*  operator"" _reverse(const char *sir, size_t sz)
{
    char * txt = new char[sz + 1];
    txt[sz] = 0;
    for (size_t poz = 0; poz < sz; poz++)
        txt[sz - poz - 1] = sir[poz];
    return txt;
}

void main(void)
{
    char * text = "c++ test today"_reverse;
    printf("%s\n", text);
    delete[]text;
}
```

```
push        0Eh ⇔ the length of “c++ test today”
push        offset string "c++ test today"
call        operator "" _reverse
add         esp,8
mov         dword ptr [text],eax
```

- In particular for literals that have a const char * parameter (a pointer), the size of the data the pointer points to can also be send if a secondary parameter is added. The secondary parameter **must be** a ***size_t***.

- Templates

Templates

- Macros bring great power to the code written in C/C++
- Templates can be considered to be derived from the notion of macros → adapted for functions and classes
- The goal is to define a model of a function or class in which the types of data we work with can be modified at the precompilation stage (similar to macros)
- For this there is a key word "***template***"
- Just like macros, the usage of templates generates extra code at compilation (code that is specific for the data types for which the templates are made for). However, the code is faster and efficient.

Templates

- The templates defined in C++ are of two types:
  - For classes
  - For functions
- Templates work just like a macro – through substitution
- The difference is that the template substitution for classes isn’t done where the first instance of that class is used, but separately, so that other instances that use the same type of data to be able to use the same substituted template
- **Important**: Because the substitution is made during precompilation, the templates must be stored in the files that are exported from a library (files.h) otherwise the compiler(in case it is tried the creation of a class from a template exported from a library) will not be able to do this.

Templates

- Templates for a function are defined as follows:

App.cpp

```
template <class T>
Return_type function_name(parameters)

or
template <typename T>
Return_type function_name(parameters)
```

- Atleast one of the “Return_type” or “parameters” must contain a member of type T.

App.cpp

```
template <class T>
T Sum(T value_1,T value_2)
{
	return value_1 + value_2;
}
```

- Function templates

Templates

- A simple example:

App.cpp

```
template <class T>
T Sum(T value_1, T value_2)
{
      return value_1 + value_2;
}

void main(void)
{
      double x = Sum(1.25, 2.5);
}
```

- The code compiles correctly → x receives the value 3.75

Templates

- A simple example:

App.cpp

```
template <class T>
T Sum(T value_1, T value_2)
{
	return value_1 + value_2;
}

void main(void)
{
	double x = Sum(1, 2.5);
}
```

```
error C2672: 'Sum': no matching overloaded function found
error C2782: 'T Sum(T,T)': template parameter 'T' is ambiguous
note: see declaration of 'Sum'
note: could be 'double'
note: or       'int'
error C2784: 'T Sum(T,T)': could not deduce template argument for
'T' from 'double'
```

- The code will not compile – 1 is converted to int, 2.5 to double, but the Sum function should receive two parameters of the same type.
- The compiler can’t decide which type to use → ambiguous code

Templates

- A simple example:

App.cpp

```
template <class T>
T Sum(T value_1, T value_2)
{
      return value_1 + value_2;
}

void main(void)
{
      int x = Sum(1, 2);
      double d = Sum(1.5, 2.4);
}
```

- In this case → x receives the value 3 and the called function will substitute class T with int, and d receives 3.9 (in this case the substitution will be made with double).
- In the compiled code there will be two Sum functions(one with parameters of type ***int*** and the other one with parameters of type ***double***)

Templates

- Be careful how you put the parameters!!!

App.cpp

```
template <class T>
T Sum(int x, int y)
{
	return (T)(x + y);
}

void main(void)
{
	int x = Sum(1, 2);
	double d = Sum(1.5, 2.4);
}
```

```
error C2672: 'Sum': no matching overloaded function found
error C2783: 'T Sum(int,int)': could not deduce template
argument for 'T'
note: see declaration of 'Sum'
error C2672: 'Sum': no matching overloaded function found
error C2783: 'T Sum(int,int)': could not deduce template
argument for 'T'
note: see declaration of 'Sum'
```

- The code above can't be compiled → not because the return value is different, but because the compiler can't deduce what type to use for class T(what return value is obtained)

Templates

- Be careful how you put the parameters!!!

App.cpp

```
template <class T>
T Sum(int x, int y)
{
	return (T)(x + y);
}

void main(void)
{
	int x = Sum<int>(1, 2);
	double d = Sum<double>(1.5, 2.4);
}
```

- In this case the code can be compiled (we use <> to specify the type we want to use in the template).
- X will be 3 and d will also be 3 (1.5 and 2.4 are converted to int)

Templates

- Templates can be made for more types

App.cpp

```
template <class T1, class T2, class T3>
T1 Sum(T2 x, T3 y)
{
      return (T1)(x + y);
}

void main(void)
{
      int x = Sum<int>(1, 2);
      double d = Sum<int,double,double>(1.5, 2.4);
}
```

- In the first case (Sum<int>) the compiler converts to Sum<int,int,int> (which will match the parameters)

Templates

- Templates can be made for more types

App.cpp

```
template <class T1, class T2, class T3>
T1 Sum(T2 x, T3 y)
{
	return (T1)(x + y);
}

void main(void)
{
	int x = Sum<int,char>(1, 10.5);
	double d = Sum<int,double,double>(1.5, 2.4);
}
```

- In the first case (Sum<int>) the compiler converts to Sum<int,int,int> (which will match the parameters)
- The compiler tries to deduce the type based on the parameters, if it is not specified. The return type must be specified because the compiler can't deduce it from the paramters.

Templates

- Function templates cand receive parameters with default values

App.cpp

```
template <class T1, class T2, class T3>
T1 Sum(T2 x, T3 y = T3(5))
{
	return (T1)(x + y);
}

void main(void)
{
	int x = Sum<int, char, int>(10);
}
```

- "***x***" receives the value 15 (10+5 → the default value for y).
- The use of default parameters requires the existance of a constructor for the type of the used class. Expressions like "T3 y = 10" are not valid unless type T3 accepts equality (or has an explicit assignment operator) with int.

- Class templates

Templates

- Class templates are defined as follows:

App.cpp

```
template <class T>
class MyClass {
...
};

or
template <typename T>
class MyClass {
...
};
```

- The ***T variable*** can be used to define class members, parameters within methods and local variables within methods.

Templates

- Class templates are defined as follows:

App.cpp

```
template <class T>
class Stack
{
	T List[100];
	int count;
public:
	Stack() : count(0) {}
	void Push(T value) { List[count++] = value; }
	T Pop() { return List[--count]; }
};
void main(void)
{
	Stack<int> s;
	s.Push(1); s.Push(2); s.Push(3);
	printf("%d", s.Pop());
}
```

- The code above prints “3”

Templates

- Be careful what types you use for functions when using templates in a class:

App.cpp

```
template <class T>
class Stack
{
	T List[100];
	int count;
public:
	Stack () : count(0) {}
	void Push(T &value) { List[count++] = value; }
	T& Pop() { return List[--count]; }
};
void main(void)
{
	Stack<int> s;
	s.Push(1); s.Push(2); s.Push(3);
	printf("%d", s.Pop());
}
```

```
error C2664: 'void Stack<int>::Push(T &)':
cannot convert argument 1 from 'int' to 'T &'
          with
          [
               T=int
          ]
```

- The code above can't be compiled. The Pop function is correct, but the Push function must receive a reference→ s.Push(1) doesn't give it a reference!!!

Templates

- Be careful what types you use for functions when using templates in a class:

App.cpp

```
template <class T>
class Stack
{
      T List[100];
      int count;
public:
      Stack() : count(0) {}
      void Push(T &value) { List[count++] = value; }
      T& Pop() { return List[--count]; }
};
void main(void)
{
      Stack<int> s;
      for (int tr = 1; tr <= 3;tr++)
            s.Push(tr);
      printf("%d", s.Pop());
}
```

- The code can be compiled and prints “3”.

Templates

- The same as function templates, class templates can use multiple types

App.cpp

```
template <class T1,class T2>
class Pair
{
	T1 Key;
	T2 Value;
public:
	Pair () : Key(T1()), Value(T2()) {}
	void SetKey(const T1 &v) { Key = v; }
	void SetValue(const T2 &v) { Value = v; }
};
void main(void) {
	Pair<const char*, int> p;
	p.SetKey(“exam grade");
	p.SetValue(10);
}
```

- “Pair() : Key(T1()), Value(T2())” → calls the default constructor for Key and Value (for “p” Key will be ***nullptr*** and Value will be ***0***).
- It is important that both T1 and T2 have a default constructor: T1() implies creating an object using its default constructor.

Templates

- The same as function templates, class templates can use more types

App.cpp

```
template <class T1,class T2>
class Pair
{
      T1 Key;
      T2 Value;
public:
      Pair() : Key(T1()), Value(T2()) {}
      Pair(const T1 &v1, const T2& v2) : Key(v1), Value(v2) {}
      void SetKey(const T1 &v) { Key = v; }
      void SetValue(const T2 &v) { Value = v; }
};
void main(void)
{
      Pair<const char*, int> p("exam_grade",10);
}
```

- An explicit constructor can also be used in a class that uses a template. In the example above the constructor receives a string (const char *) and a number.

Templates

- A more complex example:

App.cpp

```
template <class T>
class Stack
{
...
      void Push(T* value) { List[count++] = (*value); }
      T& Pop() { return List[--count]; }
};

template <class T1,class T2>
class Pair
{
...
};

void main(void)
{
      Stack<Pair<const char*, int>> s;

      s.Push(new Pair<const char*, int>(“asm_grade", 10));
      s.Push(new Pair<const char*, int>(“oop_grade", 9));
      s.Push(new Pair<const char*, int>("c#_grade", 8));
}
```

Templates

- Macros can be used togheter with templates:

App.cpp

```
#define P(k,v) new Pair<const char*, int>(k, v)
#define Stack MyStack<Pair<const char*, int>>

template <class T>
class MyStack
{
...
};
template <class T1,class T2>
class Pair
{
...
};

void main(void)
{
      Stack s;
      s.Push(P("asm_grade", 10));
      s.Push(P(“oop_grade", 9));
      s.Push(P("c#_grade", 8));
}
```

Templates

- Class templates can also accept parameters without a type(constant)

App.cpp

```
template <class T,int Size>
class Stack
{
	T List[Size];
	int count;
public:
	Stack() : count(0) {}
	void Push(const T& value) { List[count++] = (value); }
	T& Pop() { return List[--count]; }
};
void main(void)
{
	Stack<int, 10> s;
	Stack<int, 100> s2;
	for (int tr = 0; tr < 5; tr++)
		s.Push(tr);
}
```

- In this scenario, “s” will have 10 elements and “s2” 100 elements
- Two different classes will be created(one with 10 and one with 100 elements)

Templates

- Class templates can also accept parameters without a type(constant)

App.cpp

```
template <class T, int Size = 100>
class Stack
{
      T List[Size];
      int count;
public:
      Stack() : count(0) {}
      void Push(const T& value) { List[count++] = (value); }
      T& Pop() { return List[--count]; }
};
void main(void)
{
      Stack<int, 10> s;
      Stack<int> s2;
      for (int tr = 0; tr < 5; tr++)
            s.Push(tr);
}
```

- In this scenario, “s” will have 10 elements and “s2” 100 elements. In the case of s2 the default value for Size will be used.

Templates

- Class templates can accept default values for types

App.cpp

```
template <class T = int>
class Stack
{
	T List[100];
	int count;
public:
	Stack() : count(0) {}
	void Push(const T& value) { List[count++] = (value); }
	T& Pop() { return List[--count]; }
};
void main(void)
{
	Stack<double> s;
	Stack<> s2;
}
```

- In this scenario “s” will be a list of 100 doubles and s2 (because we didn’t specify the type) will be a list of ints.

Templates

- Class templates can accept default values for types

App.cpp

```
template <class T = int>
class Stack
{
	T List[100];
	int count;
public:
	Stack() : count(0) {}
	void Push(const T& value) { List[count++] = (value); }
	T& Pop() { return List[--count]; }
};
void main(void)
{
	Stack<double> s;
	Stack s2;
}
```

```
error C2955: 'Stack': use of class template
requires template argument list
note: see declaration of 'Stack'
error C2133: 's2': unknown size
error C2512: 'Stack': no appropriate default
constructor available
note: see declaration of 'Stack'
```

- The above code can’t be compiled - even if the type T is by default int
- It must be specified that when a variable of type Stack is made, that variable (“s2”) is in fact a template(***we must use <>***)

Templates

- Classes can use methods with templates

App.cpp

```
class Integer
{
	int value;
public:
	Integer() : value(0) {}
	template <class T>
	void SetValue(T v) { value = (int)v; }
};
void main(void)
{
	Integer i;
	i.SetValue<float>(0.5f);
	i.SetValue<double>(1.2);
	i.SetValue<char>('a');
}
```

- In the scenario above, Integer has 3 methods (with float, double and char parameter)

Templates

- Classes can use methods with templates

App.cpp

```
class Integer
{
      int value;
public:
      Integer() : value(0) {}
      template <class T>
      void SetValue(T v) { value = (int)v; }
};
void main(void) {
      Integer i;
      i.SetValue(0.5f);
      i.SetValue('a');
}
```

- Specifying the type in the template is not mandatory.
- In this scenario, Integer has two method:
  - one with a ***float*** parameter→ deduced by the compiler from ***0.5f***
  - one with a ***char*** parameter → deduced by the compiler from ***'a'***

Templates

- Classes that use templates cand also have static members:

App.cpp

```
template<class T>
class Number
{
	T Value;
public:
	static int Count;
};

int Number<int>::Count = 10;
int Number<char>::Count = 20;
int Number<double>::Count = 30;

void main(void)
{
	Number<int> n1;
	Number<char> n2;
	Number<double> n3;

	printf("%d,%d,%d", n1.Count, n2.Count, n3.Count);
}
```

Templates

- Classes that use templates cand also have static members:

App.cpp

```
template<class T>
class Number
{
	T Value;
public:
	static T Count;
};

int Number<int>::Count = 10;
char Number<char>::Count = ‘a’;
double Number<double>::Count = 30;

void main(void)
{
	Number<int> n1;
	Number<char> n2;
	Number<double> n3;

	printf("%d,%c,%lf", n1.Count, n2.Count, n3.Count);
}
```

Templates

- Classes that use templates can also have friend functions:

App.cpp

```
template<class T>
class Number
{
	T Value;
	int IntValue;
public:
	friend void Test(Number<T> &t);
};
void Test(Number<double> &t)
{
	t.Value = 1.23;
}
void Test(Number<char> &t)
{
	t.Value = 0;
}
void main(void)
{
	Number<char> n1;
	Number<double> n2;
	Test(n1);
	Test(n2);

}
```

Templates

- Classes that use templates can also have friend functions:

App.cpp

```
template<class T>
class Number
{
      T Value;
      int IntValue;
public:
      friend void Test(Number<T> &t)
      {
            t.Value = 5;
      }
};


void main(void)
{
      Number<char> n1;
      Number<double> n2;
      Test(n1);
      Test(n2);

}
```

- Template specialization

Templates

- Templates have a series of limitations:
  - The biggest limitation is that a method from a class that uses a template has exactly the same behaviour (the only thing that differs is the type of the parameters)
  - For example, if we define a function Sum with two parameters x and y, in which we return x+y, then we can't change this to another operation (e.g. if x si y are of type char return x*y instead x+y)
- This limitation can be overcome if we use specialized templates
- Specialized templates represent a way in which for a class we define a method that overwrites the initial code from the template with a more specific one for parameters of a certain type.

Templates

- An example:

App.cpp

```
template <class T>
class Number
{
	T value;
public:
	void Set(T t) { value = t; };
	int  Get() { return (int)value; }
};
void main(void)
{
	Number<int> n1;
	n1.Set(5);
	Number<char> n2;
	n2.Set('7');
	printf("n1=%d, n2=%d", n1.Get(), n2.Get());
}
```

- The code runs – but prints “n1=5, n2=55” even though we initialized n2 with ‘7’ → and we expected to print “n2 = 7”

Templates

- Another example:

App.cpp

```cpp
template <class T>
class Number
{
	T value;
public:
	void Set(T t) { value = t; };
	int  Get() { return (int)value; }
};
template <>
class Number <char>
{
	char value;
public:
	void Set(char t) { value = t-'0'; };
	int  Get() { return (int)value; }
};
void main(void)
{
	Number<int> n1;
	n1.Set(5);
	Number<char> n2;
	n2.Set('7');
	printf("n1=%d, n2=%d", n1.Get(), n2.Get());
}
```

Template specialized for **char**

- The code runs properly and prints n1=5, n2=7

Templates

- Specialized templates work for functions the same way:

App.cpp

```
template <class T>
int ConvertToInt(T value) { return (int)value; }

template <>
int ConvertToInt<char>(char value) { return (int)(value-'0'); }

void main(void)
{
      int x = ConvertToInt<double>(1.5);
      int y = ConvertToInt<char>('4');
}
```

Template specialized for **char**

- The code runs - x will be 1 and y will be 4

- Compile-time assertion checking

Compile-time assertion checking

- Templates are a really powerful tool. However, sometimes, one might need to add some constrains. Let’s analyze the following example:

App.cpp

```
template <typename T,int size>
class Stack
{
    T Elements[size];
public:
    Stack() {


    }
};
void main(void)
{
    Stack<float,200> a;
}
```

- How can we enforce some limits for template parameter **size ?** (e.g. we do not want ***size*** to be smaller than 2 or bigger then 100)

Compile-time assertion checking

- C++11 standard provides a keyword ***static_assert*** that can produce an evaluation of an expression during the compile phase and throw a compile error:
- Format:

```
static_assert ( condition, “<error message in case condition is false>” )
```

- Example:
  static_assert (a>10, “’a’ variable should be smaller or equal to 10”);

  This code will fail in the compile phase if “a” can be evaluated at that time (this usually means that “a” is a constant) and if its value is bigger than 10.
- Usually, ***static_assert*** is used within different class methods (e.g. constructor) to check the parameters or with templates to provide some constraints.

Compile-time assertion checking

- The previous code can be modify in the following way:

App.cpp

```
template <typename T,int size>
class Stack
{
    T Elements[size];
public:
    Stack() {
        static_assert (size > 1, "Size for Stack must be bigger than 1");
        static_assert (size < 100, "Size for Stack must be smaller than 100");
    }
};
void main(void)
{
    Stack<float,200> a;
}
```

- Now the code no longer compiles and shows the following error:

```
error C2338: Size for Stack must be smaller than 100
note: while compiling class template member function
'Stack<float,200>::Stack(void)'
note: see reference to function template instantiation
'Stack<float,200>::Stack(void)' being compiled
note: see reference to class template instantiation
'Stack<float,200>' being compiled
```

Compile-time assertion checking

- In particular for templates, ***static_assert*** can also be used to allow only a couple of types to be used in a template.
- This is however, a 2-step process. First we define a template that can tell us if a type is equal (the same) with another type.

App.cpp

```
template<typename T1, typename T2>
struct TypeCompare
{
    static const bool equal = false;
};

template<typename T>
struct TypeCompare<T,T>
{
    static const bool equal = true;
};
```

Specialized template for template ***TypeCompare*** that has both parameters of the ***same type***

- The trick here is to use a specialized template where the value of a static constant variable is changed.

Compile-time assertion checking

- The previous code can be modify in the following way:

App.cpp

```
template <typename T,int size>
class Stack
{
    T Elements[size];
public:
    Stack() {
        static_assert (TypeCompare<T,int>::equal, "Stack can only be used with type int");
}
};
void main(void)
{
    Stack<float,200> a;
}
```

```
error C2338: Stack can only be used with type int
note: while compiling class template member function
'Stack<float,200>::Stack(void)'
note: see reference to function template instantiation
'Stack<float,200>::Stack(void)' being compiled
note: see reference to class template instantiation
'Stack<float,200>' being compiled
```

- In this case the following happen: When TypeCompare<float,int> is called, the compiler choses the first (more generic form) of the template that has the value of ***equal*** **false**.
If we would have create a var using ***Stack<int,200>*** then TypeCompare<int,int> would have selected the specialized template that has the value of ***equal*** **true**

Compile-time assertion checking

- Let's analyze the following code:

App.cpp

```
void main(void)
{
    int x = rand();
    static_assert(x != 0, "X should not be 0 !");
}
```

```
error C2131: expression did not evaluate to a constant
note: failure was caused by a read of a variable outside its lifetime
note: see usage of 'x'
```

- Keep in mind that ***static_assert*** only works with constant value (e.g. values that can be deduce during the compile phase). In this case, the value of "x" is a random one (as a result, the compiler can not evaluate if "x" is 0 or not and an error will be reported).

Q & A