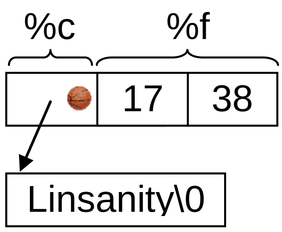
Lecture – C++ as a better C

Type-safe iostream library



- IO in C is type unsafe.

```
#include <stdio.h>
int main()
{
   printf("%c%f","Linsanity",17,38);
}
```

- C++ supports safe I/O.
  Data are printed according to their types.

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Linsanity" << 17;  cout << 38;
}          cout
                  cout
```

- `istream cin;` // represent standard input **stdin**
  `ostream cout;` // represent standard output **stdout**

  `cout << bingo` // insert data to output stream
  `cin >> bingo` // extract data from input stream

- Each iostream object has a format state.

```
                   manipulator
cout << hex << 1738;    //printf("%x",1738);
   cout
decimal    hexadecimal
```

- IO manipulator
  A manipulator modifies the internal state of a stream object.

```
int z=1738;
printf("%X%x%o%d",z,z,z,z);
cout << hex << uppercase << z;
cout << nouppercase* << z;
cout << oct << z << dec* << z;
```

```
#include <iomanip>                // for setw, setfill
printf("%-5d%05d\n",z,z);
cout << left << setw(5) << z; // setfill(' ')*
cout << right* << setfill('0') << setw(5) << z;
cout << endl;
```

Remarks
1 * indicates the default state. (DO NOT key it in.)
2 **setw** is a one-time manipulator.
3 **endl** inserts newline and then flushes ostream buffer.

- Iostream objects also have internal condition states.

good	Ready
fail	unsuccessful operation, e.g. invalid input format
bad	stream corrupted, e.g. read/write error
eof	encounter end-of-file

```
#include <iostream>
using namespace std;
void foo(int);           // left unspecified
int main()
{
   int n;
   cout << "Enter an integer: ";
   while (cin >> n) {    // convert cin to a truth value
      foo(n);
      cout << "Enter an integer: ";
   }
}
```

Where a Boolean value is needed, **cin** is converted to true, if it is in good state, and to false, otherwise.

Programming in the large

- C++ supports programming-in-the-large (whereas C supports programming-in-the-small).
- Namespaces make large programs easier to manage.
- Example

```
#include <iostream>
using namespace std;      // default namespace for library
namespace A
{
   int f() { return 1; }
   int g() { return 2; }
}
namespace B
{
   int f() { return 3; }
   int h() { return 4; }
}
int main()
{
   cout << A::f() << A::g() << B::f() << B::h();
   using namespace A;
   cout << f() << g();            // unqualified name
   cout << B::f() << B::h();      // qualified name
   using namespace B;
   cout << g() << h();
   cout << f();        // ambiguous! A::f() or B::f()?
}
```

- C-style vs C++-style header files

C-style header		C++-style header
`stdio.h`	→	`cstdio`
		`iostream`

C++-style headers enforce the concept of namespace.

```
#include <stdio.h>  →  #include <cstdio>
                       using namespace std;
```

- Using directive

  `using namespace std;`

  A using directive doesn't add any member to the declarative region in which it appears.

  During unqualified name look up, the names appear as if they were declared in the nearest enclosing namespace which contains both the using directive and the nominated namespace.

- Example

```
      #include <iostream>
      using namespace std;
      namespace A
      {
          int f() { return 1; }
      }
::f → int f() { return 2; }
      int main()
      {
A::f →    using namespace A;
          cout << f();      // ambiguous! ::f() or A::f()?
      }
```

It is as if `A::f()` were declared here in the global namespace, i.e. the global scope.

- Example

```
::x → int x=3;
      namespace A
      {
          namespace B { int x=2; }
          void p()
          {
B::x →        using namespace B;
              cout << x;     // A::B::x
          }
      }
```

It is as if `B::x` were declared here in namespace **A**.

- Using declaration

  ```
  using std::cout;
  ```

  A using declaration introduces a name into the declarative region in which it appears.

- Example

::f

A::f

```
#include <iostream>
using namespace std;
namespace A
{
    int f() { return 1; }
}
int f() { return 2; }
int main()
{
    using A::f;           // A::f() is added to main
    cout << f();          // ok, A::f()
    cout << ::f();        // ok, global f
}
```

- Example

std::cout

::cout

```
#include <iostream>
using namespace std;     // cout isn't added to global
int cout=1;
int main()
{
    cout << cout;        // ambiguous!
}                        // std::cout << ::cout;
```

Cf.

```
#include <iostream>
using std::cout;         // cout is added to global
int cout=1;              // error – redefinition of cout
int main()
{
    cout << cout;
}
```

Argument-dependent (name) lookup (ADL)

- ADL applies only to an unqualified function name that appears in the function position of a function call.
- The associated namespaces of the argument types, if any, are considered unless the usual unqualified name lookup finds a class member function declaration, or
  a block-scope function declaration that isn't a using declaration,
  or a declaration that isn't a function.
- Example on operator function

```
#include <iostream>
namespace A {
   struct complex { double r,i; };
   complex operator+(complex x,complex y)
   {
      complex z={x.r+y.r,x.i+y.i};
      return z;
   }
}
int main()
{
   A::complex x={2,3},y={4,5};
   A::complex z=A::operator+(x,y);
   std::cout << "ADL";
}
```

Comments

1. ADL allows us to unqualify the solid underlined call to
   `operator+(x,y)`   // *
   which in turn may be abbreviated to
   `x+y`   // operator overloading
2. The usual unqualified lookup for **`operator+`** in the starred line doesn't find anything; so the namespace **`A`** associated with the type **`complex`** of arguments **`x`** and **`y`** is considered.
3. This pattern is commonly used in C++ library. For example, the dotted underlined code is shorthand for
   `std::operator<<(std::cout,"ADL");`

- Example (Cont'd)

ADL is applied under certain conditions. For illustrative purpose, let's introduce a global **`operator+`**:

```
A::complex operator+(A::complex x,A::complex y)
{
   return A::operator+(x,y);
}
int main()
{
   A::complex x={2,3},y={4,5};
   A::complex z=x+y;          // ambiguous!
}
```

The usual unqualified lookup for **`operator+`** finds the global **`operator+`**, which does not meet the conditions; so the associated namespace **`A`** is also considered, making the call ambiguous.

Q: How to invoke **`::operator+`** without qualification?
A: Within **`main`**, declare VC++, x+y↓ operator+(x,y)↑, GNU C++, both ↓

```
A::complex operator+(A::complex,A::complex);
```

Comments

1 Since a block-scope declaration that isn't a using declaration is found, the associated namespace **`A`** isn't considered.
2 If the function signature is replaced by the using declaration
`using ::operator+;`
the call is still ambiguous, since the associated namespace **`A`** is also considered.

Q: How to invoke **`A::operator+`** without qualification?
A: Within **`main`**, declare

```
using A::operator+;
```

Comments

1 The associated namespace **`A`** is considered, too.
2 Both the usual unqualified lookup and ADL find **`A::operator+`**

Limited scope

- C++ supports limited scope.
- Declarations may interlace with statements
- Variables may be declared in each program point below. The boxed areas are the associated scopes.

```
for (int n=[  ];[  ];[  ])[  ]
for (;int n=[  ];[  ])[  ]
while (int n=[  ])[  ]              // declaration expression
if (int n=[  ]) [  ] else [  ]
switch (int n=[  ]) [  ]
```

- Example

Consider the task of limiting the scope of **n** to within the loop.

```
int n;
while (cin >> n) foo(n);
```

Version A – sacrifice one input case **n = 0**

```
while (int n=cin>>n? n: 0) foo(n);    //*
```

Version B

```
while (int n=cin>>n? foo(n),1: 0);
```

The starred loop may be written as:

```
for (;int n=cin>>n? n: 0;) foo(n);
```

```
loop:
if (int n=cin>>n? n: 0) {
   foo(n); goto loop;
}
```

```
loop:
switch (int n=cin>>n? n: 0) {
case 0: break;
default: foo(n); goto loop;
}
```

- Example (Cont'd)

  Note that variables cannot be declared in the condition part of **do**…**while** statement, e.g.

```
do
   foo(n);
while (int n=cin>>n?n:0);      // no
```

Lvalue and rvalue (address and value)

- Lvalue expressions
  1 Expressions that yield lvalues
  2 Where "addresses" or "values" (by lvalue-to-rvalue conversion) are needed, they may appear.
  3 e.g. variables, array subscripting, pointers

- Rvalue expressions
  1 Expressions that yield rvalues
  2 Only where "values" are needed can they appear.
  3 e.g. constants, arithmetic/logical/relational expressions

- Example

$$\overset{\text{lvalue}}{\overset{\downarrow}{\mathbf{x}}} = \overbrace{\underset{\underset{\text{lvalue-to-rvalue}}{\uparrow}}{\mathbf{x}} + \mathbf{1}}^{\text{rvalue}} \leftarrow \text{rvalue} \qquad \overbrace{\underset{\underset{\text{array-to-pointer}}{\uparrow}}{\mathbf{a}}\mathbf{[i]}}^{\text{lvalue}} = \overbrace{\underset{\underset{\text{lvalue-to-rvalue}}{\uparrow}}{\mathbf{*p}}}^{\text{lvalue-to-rvalue}}$$

  Note: The result of an array-to-pointer conversion is an rvalue.

- Prefix increment/decrement expressions yield rvalues in C and lvalues in C++.

```
x=5;
cout << ++x;
```

  In C, **++x** yields the value 6 stored in **x**.
  In C++, **++x** yields the address of **x**. In the preceding context, an lvalue-to-rvalue conversion then retrieves the value 6 stored in **x**.

- (Cont'd)

  On the other hand, there is no lvalue-to-rvalue conversion in this context:

  $$\texttt{++}\underbrace{\texttt{++x}}_{\texttt{x}}\texttt{;}$$

Remark:	`++++x;`	`x++++;`
C	×	×
C++	✓	×

- Postfix increment/decrement expressions yield rvalues in both C and C++. (Why?)

- Assignment expressions also yield rvalues in C and lvalues in C++.

C/C++	`x=y=10;`	i.e. `x=(y=10);`
C++	`(x*=10)+=y;`	i.e. `x=10*x+y;`

- A comma expression **`e1,e2`** yields an rvalue in C. But, it yields an lvalue in C++ iff **`e2`** yields an lvalue.

	`(x=2,3)=4`	`(x=2,y)=3`	
C	×	×	
C++	×	✓	// same as **`x=2,y=3`**

- A conditional expression **`e1?e2:e3`** yields an rvalue in C. But, it yields an lvalue in C++ iff both **`e2`** and **`e3`** yield lvalues and are of the same type.

  C/C++
  ```
  max=x>y?x:y;
  min=x>y?y:x;
  ```

  C++
  ```
  (x>y?max:min)=x;        // need parentheses
  (x>y?min:max)=y;
  ```

  The following two expressions yield rvalues in C/C++.

  1 **`x>=0?x:-x`**

  2
  ```
  int x;
  double y;
  x>y?x:y ≡ x>y?(double)x:y
  ```

Default arguments

- An expression specified in a parameter declaration is used as a default argument.
- Default arguments will be used in calls where trailing arguments are missing.
- Example

```
double len(double x=0.0,double y=0.0)
{
   return sqrt(x*x+y*y);
}
len()               // len(0.0,0.0)
len(1.0)            // len(1.0,0.0)
len(1.0,2.0)        // len(1.0,2.0)
```

- All parameters to the right of a parameter with a default argument shall also have default arguments.

```
double len(double=0.0,double);             // error
```

- Default arguments shall be specified before use and shan't be redefined (not even to the same values)

  Convention: Specify default arguments in declarations.

```
double len(double=0.0,double=0.0);
int main()
{
   cout << len();
}
// double len(double x=0.0,double y=0.0)   // error
double len(double x,double y)
{
   return sqrt(x*x+y*y);
}
```

- Declarations in different scopes may have different default arguments.

```
void p() { double len(double=1.0,double=1.0); }
void q() { double len(double=0.0,double=0.0); }
```

bool type

- Besides the C tradition of treating zero as false and non-zero as true, C++ also introduces a new Boolean type **`bool`** with two constants **`true`** and **`false`**.
- There are two standard conversions related to the **`bool`** type.

  Integral promotion

  Where a numeric value is needed, **`true`** is converted to 1 and **`false`** is converted to 0, e.g. **`true+false`**

  Boolean conversion

  Where a Boolean value is needed, zero is converted to **`false`** and any non-zero value is converted to **`true`**, e.g.
  `bool r="snoopy";`

  N.B. Standard conversions are implicit conversions defined for built-in types.

Safe cast

- In C, there is only one cast operator (*type*) for all kinds of type conversions, regardless of their safeness.
- C++ classifies type conversions into four categories and introduces four new cast operators for them, namely, **`static_cast`** **`const_cast`**, **`reinterpret_cast`**, and **`dynamic_cast`**.
- Example

```
double x=3.4;
int y=5;
int z=static_cast<int>(x)+y;          // for efficiency

const int x=2;
int* y=const_cast<int*>(&x);          // a must

double x=3.4;
int* y=reinterpret_cast<int*>(&x);    // a must
```

Function overloading

- C++ supports function overloading.

  In C, there are only **double** versions of math functions declared in **<math.h>**. This is problematic. For example, consider

```
float x=3.14f;
x=x+sqrt(x);
```

  In this context, it is reasonable to expect single precision floating point arithmetic.

  However, in C, there are three conversions involved:

  floating point conversion (double → float)

```
x = x + sqrt(x);
```

  floating point promotion   floating point promotion (float → double)

  In addition to the **double** versions of math functions declared in **<cmath>**, C++ adds **float** and **long double** versions of these functions, making them overloaded functions.

  For example, there are three overloaded **sqrt** functions:

```
float sqrt(float);
double sqrt(double);
long double sqrt(long double);
```

  Overload resolution then selects the best function to call.

  For the preceding call **sqrt(x)**, we have

```
float sqrt(float);              // identity
double sqrt(double);            // float → double
long double sqrt(long double);  // float → long double
```

  Clearly, the **float** version is the best and will be selected, as desired.

  N.B. C is monomorphic, whereas C++ is polymorphic.

- Example

```
bool prime(int n)
{
   for (int d=2;d<=sqrt(n);d++)  // d*d<=n
      if (n%d==0) return false;
   return true;
}
```

The call **`sqrt(n)`** is legal in C, but illegal in C++, because none of the three built-in **`sqrt`** is the best – they all need a floating-integral conversion – and the call is ambiguous. VC++↑, GNU C++↓

Thus, to invoke the **`double`** version, say, we have to write

```
d<=sqrt(static_cast<double>(n))
```

Alternatively, we may have our own integral square root function for computing $\lfloor\sqrt{n}\rfloor$, which may be defined as:

```
int sqrt(int n)
{
   return sqrt(static_cast<double>(n)); //*
}
```

or,

```
int sqrt(int n)
{
   int r=0;
   while(r*r<=n) r++;
   return r-1;
}
```

Now, for the call **`sqrt(n)`**, our own **`sqrt`** function is better than the three built-in **`sqrt`** functions.

Remarks

`::sqrt` and built-in **`std::sqrt`**'s are indeed not overloaded, as they are in different namespaces.
It is the global declaration
`using namespace std;` or **`using std::sqrt;`**
that makes them overloaded in the global namespace.

- Overloaded functions

Overloaded functions are functions in the same scope that have the same name but differ in the number or the types of the parameters.

Certain function declarations cannot be overloaded:

1 Function declarations that differ only in the return type can't be overloaded, e.g

```
int prime(int);
bool prime(int);
```

N.B. Due to expression statement, e.g. `prime(7);`

2 Parameter declarations that differ only in the use of typedef types are equivalent, e.g.

```
typedef int INT;
bool prime(int);
bool prime(INT);
```

N.B. typedef doesn't introduce new types

3 Parameter declarations that differ only in cv-qualification are equivalent, e.g.

```
bool prime(int);
bool prime(const int);
```

N.B. By definition, the **`const`** and **`volatile`** qualifiers are ignored, since the cv-qualification of argument has nothing to do that of parameter, e.g.

```
const int x=2;
prime(x);                    // neither is better
```

Digression

`volatile` is a hint to the compiler to avoid aggressive optimization involving a volatile object, as the value of the object might be changed undetectable by the compiler.
For example,

```
volatile bool available;  // is the device available?
while (!available) wait;
```

- Overloaded functions (Cont'd)

    3 (Cont'd) The qualifier **`volatile`** informs the compiler not to optimize the code by caching the variable **`available`** in a register.

    4 Parameter declarations that differ only in **`T[]`** vs. **`T*`** are equivalent, e.g.

    ```
    void sort(int[],int);
    void sort(int*,int);
    ```

    N.B. By definition, the array declaration **`T[]`** is adjusted to become the pointer declaration **`T*`**.

    5 Parameter declarations that differ only in **`T(T1,…,Tn)`** vs. **`T(*)(T1,…,Tn)`** are equivalent, e.g.

    ```
    void sort(int[],int,bool(int,int));
    void sort(int[],int,bool(*)(int,int));
    ```

    N.B. By definition, the function declaration is adjusted to become a pointer to function declaration.

    6 Parameter declarations that differ only in their default arguments are equivalent, e.g.

    ```
    double len(double,double);
    double len(double=0.0,double=0.0);
    ```

    N.B. The call **`len(1.0,2.0)`**, say, can't be resolved.

Overload resolution

- Candidate → Viable → Best viable

- Basically, candidate functions are functions that
    1 are visible in the context of the function call, and
    2 have the same name as the function being called.

- Viable functions are candidate functions that satisfy
    1 the number of parameters = the number of arguments, and
    2 there is an implicit conversion sequence for each pair of argument and parameter.

- The best viable function is the viable function that is better than all the other viable functions
  If such a function does not exist, the function call is ambiguous

- Better viable function

  Let $\mathrm{ICS}(F,k)$ be the implicit conversion sequence that converts the k^{th} argument to the type of the k^{th} parameter of function F.

  A viable function F is better than another viable function G, if
  1 $\forall$ argument k, $\mathrm{ICS}(F,k)$ is not worse than $\mathrm{ICS}(G,k)$, and
  2 $\exists$ argument k, $\mathrm{ICS}(F,k)$ is better than $\mathrm{ICS}(G,k)$

  What does it mean that an ICS is better will be defined soon.

- Ranks of standard conversions

<table>
<tr><th>Conversion</th><th>Category</th><th>Rank</th></tr>
<tr><td>No conversions required</td><td>Identity</td><td rowspan="5">Exact match</td></tr>
<tr><td>Lvalue-to-rvalue conversion</td><td rowspan="3">[1]Lvalue Transformation</td></tr>
<tr><td>Array-to-pointer conversion</td></tr>
<tr><td>Function-to-pointer conversion</td></tr>
<tr><td>Qualification conversions</td><td>[3]Qualification Adjustment</td></tr>
<tr><td>Integral promotions</td><td rowspan="2">[2]Promotion</td><td rowspan="2">Promotion</td></tr>
<tr><td>Floating point promotion</td></tr>
<tr><td>Integral conversions</td><td rowspan="6">[2]Conversion</td><td rowspan="6">Conversion</td></tr>
<tr><td>Floating point conversions</td></tr>
<tr><td>Floating-integral conversions</td></tr>
<tr><td>Pointer conversions</td></tr>
<tr><td>Pointer to member conversions</td></tr>
<tr><td>Boolean conversions</td></tr>
</table>

better > worse

Remark
Within an ICS, conversions are applied in the canonical order:
1 Lvalue transformation
2 Conversions or Promotions
3 Qualification Adjustment.

- Ranks of standard conversions (Cont'd)

  For example

```
void p(const void*);
int a[3]
p(a);
```

pointer conversion

int[3] → **int*** → **void*** → **const void***

Array-to-pointer conversion   qualification conversion

- Rank of an ICS = Rank of the worst conversion in the ICS

- Better implicit (standard) conversion sequence

  To determine if the standard conversion sequence S_1 is better than S_2, check the following conditions **in order**:

  Condition 1: S_1 is a proper subsequence of S_2, excluding any Lvalue Transformation.

  Condition 2: The rank of S_1 > the rank of S_2.

  Condition 3: S_1 and S_2 differ in their qualification conversion and S_1 yields a less cv-qualified type than that of S_2.

- Example

  ① **void p(int);**
  ② **void p(unsigned);**
  ③ **void p(double);**

Function call	**p(2);**	
Viable	①	identity conversion (no conversion)
	②	integral conversion
	③	floating-integral conversion
Best viable	①	by condition 1

Function call	**int x=2; p(x);**	
Viable	①	~~lvalue-to-rvalue~~ (becomes empty)
	②	~~lvalue-to-rvalue~~, integral conversion
	③	~~lvalue-to-rvalue~~, floating-integral conversion
Best viable	①	by condition 1

- Example (Cont'd)

Function call `p('\2');`
Viable ① integral promotion
② integral conversion
③ floating-integral conversion
Best viable ① by condition 2

Function call `p(2L);`
Viable ① integral conversion
② integral conversion
③ floating-integral conversion
Best viable Ambiguous!

To resolve this ambiguity, convert the argument to the desired type, e.g. `p(static_cast<double>(2L));`

- Example

① `void p(int*);`
② `void p(const int*);`

Function call `int a[3];`
`p(a);`
Viable ① ~~array-to-pointer~~ (becomes empty)
② ~~array-to-pointer~~, qualification
Best viable ① by condition 1

- Example

① `void p(const int*);`
② `void p(const volatile int*);`

Function call `int a;`
`p(&a);`
Viable ① qualification conversion
② qualification conversion
Best viable ① by condition 3

- Example

  ① `void p(int);` // call by value
  ② `void p(int&);` // call by reference

  Q: Consider `int x; p(x);`
  Which is better? call-by-value or call-by-reference?

  In technical terms:

  Function call `int x; p(x);`
  Viable ① ~~lvalue-2-rvalue~~ (becomes empty)
  ② identify conversion
  Best viable Ambiguous!

  This ambiguity can only be partially resolved – only the call-by-value version can be invoked.

  Function call `p(static_cast<int>(x));`
  Best viable ① identity conversion

  Note that ② isn't viable, since the argument is an rvalue.

  Comment
  Were Lvalue Transformation not excluded, call-by-reference would be better than call-by-value.

- Example – A similar example

  ① `void p(int*);`
  ② `void p(int(&)[3]);`

  Function call `int a[3]; p(a);`
  Viable ① ~~array-2-pointer~~ (becomes empty)
  ② identity conversion
  Best viable Ambiguous!

  Again, this ambiguity can only be partially resolved – only the version ① can be invoked.

  Function call `p(static_cast<int*>(a));`
  Best viable ① identity conversion

  Note that ② isn't viable, since the types don't match.

- Example

  ① `void p(int,double);`
  ② `void p(double,int);`

  Function call `p(2u,2);`
  Viable ① integral conversion / floating-integral conversion
  ② floating-integral conversion / identity
  Best viable ② by condition 1 (the 2nd argument)

  Function call `p(2,2);`
  Viable ① identity / floating-integral conversion
  ② floating-integral conversion / identity
  Best viable Ambiguous!

Inline functions

- The compiler generates code inline for calls to inline functions. Thus, inline function calls don't have the runtime overhead of non-inline function calls.
- The compiler may choose to ignore the inline request.
- Example

```
inline int square(int x) { return x*x; }
cout << square(2+3);          // square(2.3) ×
```

is compiled to

```
int x=2+3;                    // type check
cout << x*x;
```

C.f.

```
#define square(x) ((x)*(x))   // x may be of any type
cout << square(2+3);          // square(2.3) ✓
```

is preprocessed to

```
cout << ((2+3)*(2+3));        // 2+3 is evaluated twice
```

Function template

- A function template defines a generic function or a family of related functions.

template parameter

```
template<typename T>     // or, template<class T>
inline T square(T x)
{
   return x*x;
}
```

function parameter

- Function template instantiation
  – the act of instantiating a function from a function template

- Template argument deduction

  The compiler instantiates a function template by deducing the template arguments from the function arguments of a call.

  For the call

  `square(2)`

  the compiler deduces **`T = int`** and generates the instance

  ```
  inline int square(int x) { return x*x; }
  ```

  For the call

  `square(3.4)`

  the compiler deduces **`T = double`** and generates the instance

  ```
  inline double square(double x) { return x*x; }
  ```

  Comment – The signature of an instance actually includes the template argument, e.g.

  ```
  inline int square<int>(int x) { return x*x; }
  inline double square<double>(double x)
  {  return x*x; }
  ```

  The template argument may be omitted, if it can be deduced from the function parameter.

- Explicit template argument specification

  In some cases, it is necessary to explicitly specify the template arguments.

```
template<typename T>
T max(T x,T y) { return x<y? y: x; }

int x; double y;
max(x,y)
```

  This call is ambiguous, because the compiler cannot determine whether `T = int` or `T = double`.

  To make it work, either explicitly specify the template argument

```
max<double>(x,y)
```

  or explicitly convert the argument to parameter type

```
max(static_cast<double>(x),y)
```

- Template argument deduction (revisited)

  To deduce template arguments, the function parameter type and the function argument type needn't be the same.

  However, only conversions of the rank "Exact match" (i.e. lvalue transformation and qualification adjustment) are allowed.

```
template<typename T>
T sum(const T* a,int n)
{
   T s=T(0);          // static_cast<T>(0) or (T)0
   for (int i=0;i<n;i++) s+=a[i];
   return s;
}
int a[5]={1,2,3,4,5};
sum(a,5);
```

  Two conversions of the rank "Exact match" apply:

```
function argument type     int[5] → int* → const int*
function parameter type    const T*
                                       T=int
```

- Function template overloading

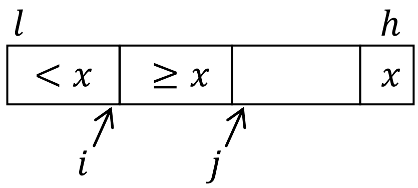
```
template<typename T>                // signature
int partition(T*,int,int);

template<typename T>                // quicksort
void sort(T* a,int l,int h)
{
   if (l<h) {
      int m=partition(a,l,h);
      sort(a,l,m-1);
      sort(a,m+1,h);
   }
}

template<typename T>
void sort(T* a,int n)
{
   sort(a,0,n-1);
}

int main()
{
   int a[9]={8,4,7,1,9,3,6,5,2};
   sort(a,9);                       // int[9]→int*
}

template<typename T>
int partition(T* a,int l,int h)
{
   T x=a[h];
   int i=l-1;
   for (int j=l;j<h;j++)
      if (a[j]<x) {
         i++; T z=a[i]; a[i]=a[j]; a[j]=z;
      }
   T z=a[i+1]; a[i+1]=a[h]; a[h]=z;
   return i+1;
}
```



Function template explicit specialization

- Function template explicit specialization lets function templates deal with special cases.
- A function template explicit specialization is a function, rather than a function template.
- Example

  In some cases, the general function template doesn't work for some types.

  ```
  // generic function template
  template<typename T>          // ∀T that supports <
  T max(T x,T y)
  {
     return x<y? y: x;
  }
  ```

  The call

  ```
  max("snoopy","pluto")         // T = const char*
  ```

  compares two pointers rather than two C-style strings.

  To compare two C-style strings, a function specialized for **`T = const char*`** must be provided:

  ```
  // function template explicit specialization for const char*
  template<>
  const char* max<const char*>
                     (const char* x,const char* y)
  {
     return strcmp(x,y)<0? y: x;
  }
  ```

  In this case, the explicit specification of the template argument (i.e. the underlined part) may be omitted from the template explicit specialization, because the template argument can be deduced from the function parameter.

  Now, the call

  ```
  max("snoopy","pluto")
  ```

  will invoke this specialization function.

- Example (Cont'd)

  // function template explicit specialization for **`char*`**

```
template<>
char* max(char* x,char* y)
{
   return strcmp(x,y)<0? y: x;
}
char s[]="snoopy",t[]="pluto";
```

  The call

```
max(s,t)
```

  will invoke this specialization function.

  Without this specialization, an instance of the general template, rather than the specialization for **`const char*`**, will be invoked.

  Indeed, this explicit specialization for **`char*`** is unnecessary, as we may invoke the explicit specialization for **`const char*`**:

```
max<const char*>(s,t)
max((const char*)s,(const char*)t)
```

- Example

  In some situations, the generic algorithm used in the general function template is inefficient for some specific types.

  // generic function template uses quicksort

```
template<typename T>
void sort(T* a,int n) { sort(a,0,n-1); }
```

  But an array of n 0's and 1's can easily be sorted in $O(n)$ time.

  // explicit specialization for **`bool`** by partition

```
template<>
void sort<bool>(bool* a,int n)
{
   bool x=a[n-1];
   a[n-1]=true;                  // use 1 as the pivot
   a[partition(a,0,n-1)]=x;
}
```

- A function template explicit specialization must be declared after the general function template and before the 1st use of that specialization.

```
template<typename T>
T max(T x,T y)
{
   return x<y? y: x;
}
int main()
{
   cout << max("snoopy","pluto");   // instantiation
}
template<>
const char* max(const char* x,const char* y)
{
   return strcmp(x,y)<0? y: x;
}
```

T max(T,T)

● T=const char*

This is illegal – a program can't have both an instantiation from the general template and an explicit specification with the same template arguments.

- The general function template needn't be defined, if it isn't to be instantiated.

```
template<typename T> T max(T,T);
template<>
const char* max(const char* x,const char* y)
{
   return strcmp(x,y)<0? y: x;
}
int main()
{
   cout << max("snoopy","pluto");
}
```

Overload resolution with instantiations

- Overload resolution may involve the following functions:
  1 function template explicit specialization
  2 function template instantiation
  3 ordinary function

- Candidate functions
  1 If template argument deduction succeeds, then
  1a If a function template explicit specialization exists for the template arguments deduced, it is a candidate function
  1b otherwise, the function template instantiation with deduced template arguments is a candidate function
  2 Ordinary functions of the name are candidate functions.

- Best viable functions
  If the best viable function exists, select it.
  Otherwise, perform overload resolution considering only those ordinary functions in the set of viable functions.

- Example ① $\boxed{}$ ② $\boxed{\bullet ③}$

```
① template<typename T> void sort(T*,int,int);
② template<typename T> void sort(T*,int);
③ template<> void sort<bool>(bool*,int);
```

	`int a[9]={8,4,7,1,9,3,6,5,2};`
Function call	`sort(a,0,9);`
Candidate	instantiation of ①
	`void sort<int>(int*,int,int)`
Function call	`sort(a,9);`
Candidate	instantiation of ②
	`void sort<int>(int*,int)`
	`bool a[9]={1,0,1,0,1,0,1,0,1}`
Function call	`sort(a,9);`
Candidate	③

Template argument deduction succeeds for ② with **T = bool** that agrees with the specialization.

- Example

① `template<typename T> T max(T,T);`
② `template<> double max<double>(double,double);`
③ `double max(double,double);`

`int x,y; double a,b;` ① ●② ●③

Function call	`max(x,y)`
Candidate	instantation of ① `int max<int>(int,int);` ③
Viable	instantation of ①,③
Best viable	instantation of ①

Function call	`max(a,b)`
Candidate	②,③
Viable	②,③
Best viable	③

The best viable function doesn’t exist in ② and ③; so, remove ②

Function call	`max(x,a)`
Candidate	③
Viable	③
Best viable	③

Template argument deduction fails for ①; so, ② isn’t a candidate

Remarks

1 All kinds of standard conversions can be applied to ordinary functions.
2 Only “exact match” standard conversions can be applied to function templates.

Function call	`max<double>(a,b)` `max<double>(x,a)`
Candidate	②
Viable	②
Best viable	②

Treat the explicitly specified template argument as deduced, except that ordinary functions aren’t considered.

- Example

  Why would it be useful to overload ordinary functions with function templates?

  Given

  `template<typename T> T max(T,T);`

  Suppose we frequently need to compute

  `max(x,y)`

  where **`x`** and **`y`** are of distinct signed integral type
  e.g. **`max(true,2)`**, **`max(2,'a')`**, etc.

  For each computation, we may explicitly specify the template argument,
  e.g. **`max<int>(true,2)`**, **`max<int>(2,'a')`**, etc.

  Alternatively, we may define an overloaded ordinary function:

  ```
  int max(int x,int y)
  {
     return max<int>(x,y);    // instantiation, not recursion
  }
  ```

  and leave the calls unchanged:
  e.g. **`max(true,2)`**, **`max(2,'a')`**, etc.

Partial ordering of function templates

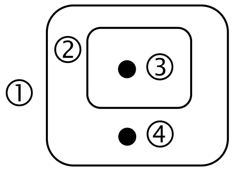
- With function template overloading, only the most specialized function template is chosen for instantiation.
  It is erroneous if the most specialized function template doesn't exist.

- A function template is more specialized than another if it can be instantiated to a more limited set of functions, or equivalently, to functions with a limited set of parameters.

- Example

① `template<typename T> T max(T,T);`

② `template<typename T>`
`T* max(T* x,T* y) { return *x<*y? y: x; }`



	`int *x,*y;`
Function call	`max(x,y)`
Candidate	instantation of ②
	`int* max<int>(int*,int*);`

Template ② is more specialized than template ①, and hence chosen for instantiation.

Next, add the following explicit specialization

③ `template<>`
`const char* max(const char*,const char*);`

As written, it is an explicit specialization of the more specialized template ②, rather than ①. It is equivalent to

```
template<>
const char* max<const char>(const char*,const char*);
```

Function call	`max("snoopy","pluto")`
Candidate	③

Template argument deduction succeeds for ② with `T = const char` that agrees with the specialization ③.

Function call	`max<const char*>("snoopy","pluto")`
Candidate	instantation of ①

Template argument deduction succeeds for ① with `T = const char*`

Now, if we add the following explicit specialization of template ①

④ `template<>`
`const char* max<const char*>(const char*,const char*);`

Function call	`max<const char*>("snoopy","pluto")`
Candidate	④

Template argument deduction succeeds for ① with `T = const char*` that agrees with the specialization ④.

- Example

```
template<class T1,class T2> void p(T1,T2);
template<class T1,class T2> void p(T1,T2*);
template<class T1,class T2> void p(T1*,T2);
template<class T1,class T2> void p(T1*,T2*);
template<class T> void p(T*,T*);
```

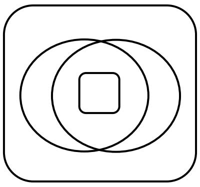
p(T*,T*) is more specialized than **p(T1*,T2*)**, bacause

{ **void p(T1*,T2*)** }
= { **void p(T*,T*)** } ∪ { **void p(T1*,T2*)** | **T1≠T2** }

p(T1*,T2*) is more specialized than **p(T1*,T2)**, bacuase

{ **void p(T1*,T2)** }
= { **void p(T1*,T2*)** } ∪
{ **void p(T1*,T2)** | **T2** isn't a pointer type **}**

In fact, these five function templates satisfy the partial order:



```
          p(T1,T2)
         /        \
  p(T1,T2*)   p(T1*,T2)
         \        /
         p(T1*,T2*)        //* if missing
              |
          p(T*,T*)
```

```
int *x,*y;
double* z;
p(x,y);         // p(T*,T*)
p(x,z);         // p(T1*,T2*)
p(x,*z);        // p(T1*,T2)
p(*x,z);        // p(T1,T2*)
p(*x,*z);       // p(T1,T2)
```

Remark
Were the starred function template missing, **p(x,z)** would be ambiguous, since neither **p(T1,T2*)** nor **p(T1*,T2)** is more specialized than the other.
This ambiguity can be resolved, e.g. **p<int*,double>(x,z)**.

Class template

- A class template defines a parameterized type or a family of related types.
- Example

```
template<typename T1,typename T2>
struct pair {
   T1 first;
   T2 second;
};
```

or, equivalently,

```
template<typename T1,typename T2>
class pair {
public:
   T1 first;
   T2 second;
};
```

```
pair<int,double> x;
pair<char,pair<int,double> > y;   // watch the space
```

These class template instantiations generate the instances:

```
struct pair<int,double> {
   int first;
   double second;
};
```

```
struct pair<char,pair<int,double> > {
   char first;
   pair<int,double> second;
};
```

The template arguments for class template instantiations must always be explicitly specified – they are never deduced, e.g.

```
pair r;        // error, what are the template arguments?
```

Class template explicit specialization

- Class template explicit specialization is analogous with function template explicit specialization.
  Function or class template explicit specialization allows one to provide different implementations of functions or classes (i.e. types), respectively, on special cases.

- A class template explicit specialization may have a different set of class memebers from the generic class template.

- Example

  As a trivial example, we may rename the two data members for a single special case: **T1 = int** and **T2 = unsigned**

```
// primary template
template<typename T1,typename T2>
struct pair {
   T1 first;
   T2 second;
};

// explicit specialization
template<>
struct pair<int,unsigned> {
   int numerator;
   unsigned denominator;
};

typedef pair<int,unsigned> rational;

void print(rational r)
{
   cout << r.numerator << "/" << r.denominator;
}
```

Class template partial specialization (for class only)

- Class template partial specialization allows one to specialize some template parameters while leaving the others generic.
  Put differently, class template partial specialization allows one to provide different implementations of classes on ***families*** of special cases.

- A class template partial specialization is a template.
  A class template explicit specialization is a class.

- A class template partial specialization may also have a different set of class memebers from the generic class template.

- Like function template explicit specializations, a class template explicit or partial specialization must be declared after the primary class template and before the 1st use of that specialization.

- Example

  As a trivial example, we may redeclare the two data members for a ***family*** of special cases, **T1 = T2**, as a 2-element array.

```
// partial specialization
template<typename T>
struct pair<T,T> {    // don't forget the underlined code
   T m[2];
};

template<typename T>
pair<T,T> minmax(T* a,int n)
{
   pair<T,T> r;
   r.m[0]=a[0];        // min
   r.m[1]=a[0];        // max
   for (int i=1;i<n;i++)
      if (a[i]<r.m[0]) r.m[0]=a[i];
      else if (a[i]>r.m[1]) r.m[1]=a[i];
   return r;
}
```

- Example (Cont'd)

  Alternatively, we may write

```
#include <limits>        // not <climits>

template<typename T>
pair<T,T> minmax(T* a,int n)
{
   pair<T,T> r;
   r.m[0]=numeric_limits<T>::max();    //*
   r.m[1]=numeric_limits<T>::min();    //*
   for (int i=0;i<n;i++)
      if (a[i]<r.m[0]) r.m[0]=a[i];
      else if (a[i]>r.m[1]) r.m[1]=a[i];
   return r;
}
```

  Or, we may replace the starred lines by:

```
if (numeric_limits<T>::is_integer) {
   r.m[0]=numeric_limits<T>::max();
   r.m[1]=numeric_limits<T>::min();
} else {
   r.m[0]=numeric_limits<T>::infinity();
   r.m[1]=-numeric_limits<T>::infinity();
}
```

  Digression: On **numeric_limits**

  This STL class template provides information about various properties of built-in numeric types.

  It consists of a primary class template and one explicit specialization for each built-in numeric type.

```
// primary class template
template<typename T>
class numeric_limits {
// all members have 0 or false values.
};
```

- Example (Cont'd)

One explicit specialization for each built-in numeric type, e.g.

```
template<>
class numeric_limits<int> {
// all members have values relative to int
};

template<>
class numeric_limits<double> {
// all members have values relative to double
};
```

and so on …

The following example illustrates some members of this class.

```
// use the explicit specialization for int
numeric_limits<int>::is_specialized;  // true
numeric_limits<int>::is_signed;       // true
numeric_limits<int>::is_integer;      // true
numeric_limits<int>::has_infinity;    // false
numeric_limits<int>::max();           // INT_MAX
numeric_limits<int>::min();           // INT_MIN
numeric_limits<int>::infinity();      // 0

// use an instantiation of the primary template
numeric_limits<int*>::is_specialized; // false
numeric_limits<int*>::is_signed;      // false
```

Remark

`infinity()` is meaningless unless **`has_infinity`** is true. Members (such as **`is_signed`**) of the primary class template are also meaningless.

Any value that is meaningless is set to 0 or false.

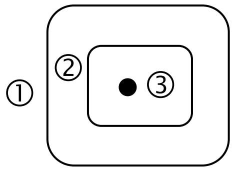
Matching of class template partial specifications

- Class template instantiation is tried in the following order:

  1 class template partial specification

  2 primary class template

  N.B. Class template explicit specifications are considered before class template instantiation.



- Example

```
① template<class T1,class T2> struct pair {};
② template<class T> struct pair<T,T> {};
③ template<> struct pair<int,int> {};
```

```
pair<int,int>           // ③
pair<double,double>     // ②, T = double
pair<int,double>        // ①, T1 = int, T2 = double
```

- Comment

  The function template counterpart of "matching of class template partial specifications" is "overload resolution with instantiations".

  Note that there are no "ordinary classes", because class names aren’t overloded.
  For example, **pair** can’t be used to name a non-tempalte class

```
struct pair {};         // error
```

- Comment

  The functiion template counterpart of "class template partial specializations" is "function template overloading".

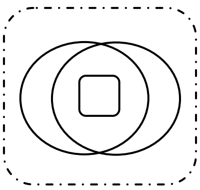
Partial ordering of class template partial specializations

- When considering class template partial specifications, only the most specialized one is chosen for instantiation.
  It is erroneous if the most specialized partial specialization does not exist.

- Example

```
template<class T1,class T2> struct s {};
template<class T1,class T2> struct s<T1,T2*> {};
template<class T1,class T2> struct s<T1*,T2> {};
template<class T1,class T2> struct s<T1*,T2*> {};
template<class T> struct s<T*,T*> {};
```

The class template partial specializations satisfy the following partial order.
Note: The primary template isn't considered in this diagram.



```
s(T1,T2*)    s(T1*,T2)
      \        /
     s(T1*,T2*)            // * if missing
          |
       s(T*,T*)
```

```
s<int*,int*>       // s(T*,T*)
s<int*,double*>    // s(T1*,T2*)
s<int*,double>     // s(T1*,T2)
s<int,double*>     // s(T1,T2*)
s<int,double>      // s(T1,T2)
```

Remark
Were the starred class template missing, `s<int*,double*>` would be ambiguous, since neither `s(T1,T2*)` nor `s(T1*,T2)` is more specialized than the other.
Moreover, this ambiguity can't be resolved – it is an error.

Template metaprogramming

- Metaprogram
  A metaprogram is a program that manipulates or generates programs.

- Metaprogramming
  – the act of programming in metaprograms

- C++ template metaprogramming
  1 C++ metaprograms are executed during compile time
  2 C++ metaprograms are recursive – template specializations are the boundaries of the recursive instantiations.

- Example (background)

```
struct A {
   enum color { red,green,blue };    // type member
};
```

  Comments

  1 The type member and enumerators must be accessed by qualified names, e.g.
  A::color x=A::red;
  2 The enumerators may also be accessed through objects:
  A a;
  x=a.green; // **x=1**
  3 The enumerators are constants and occupy no storage. Although struct **A** has no data members, it is required that sizeof(A)≠0. Usually, sizeof(A)=1.

- Example – A classic example

```
template<int n>       // non-type template parameter
struct f {
   enum { v=n*f<n-1>::v };
};
template<>
struct f<0> { enum { v=1 }; };
int main() { cout << f<3>::v; }
```

- Example (Cont'd)

  Since enumerators are constants whose values are computed at compile time, **`f<3>::v`** causes the primary template to be recursively instantiated by the compiler:

```
struct f<3> {
   enum { v=6 };      // v=3*f<2>::v
};
struct f<2> {
   enum { v=2 };      // v=2*f<1>::v
};
struct f<1> {
   enum { v=1 };      // v=1*f<0>::v
};
```

  The recursion terminates at the explicit specializaion.

  This template metafunction can only be used to compute the factorial of an integral constant at compile time. It cannot be used to compute the factorial of an integral variable:

```
int n=3;
cout << f<n>::v;      // error – not a constant
```

  Q: Why cannot the metafunction be written as

```
template<int n>       // *
struct f {
   enum { v=n==0?1:n*f<n-1>::v };
};
```

  A:
  Eager approach (C++)
  Instantiations are performed before evaluation
  With this approach, the starred template doesn't work.
  E.g. **`f<3>::v`** will result in infinite instantiations.

  Lazy approach
  Instantiations are performed only when needed by evaluation
  With this approach, the starred template works.

- Example

```
template<int n>
int f() { return n*f<n-1>(); }
template<>
int f<0>() { return 1; }
```

The call `f<3>()` instantiates the following instances:

```
int f<3>() { return 3*f<2>(); }
int f<2>() { return 2*f<1>(); }
int f<1>() { return 1*f<0>(); }
```

Again, the recursion terminates at the explicit specializaion.

Unlike the preceding example in which the value of `f<3>::v` is computed at compile time, the value of `f<3>()` isn't computed at compile time.
However, were the functions declared inline, `f<3>()` would be unfolded to 3*2*1*1, which can be computed by the compiler.

- Example

```
template<int m,int n>
struct c {
   enum { v=c<m-1,n>::v+c<m-1,n-1>::v };
};
template<int m>
struct c<m,m> { enum { v=1 }; };
template<int m>
struct c<m,0> { enum { v=1 }; };
template<>
struct c<0,0> { enum { v=1 }; };
```

Here, the recursion terminates at the two partial specializaions.

The explicit specialization is necessary to handle the special case `c<0,0>`. Without it, `c<0,0>` would be ambiguous, as neither of the two partial specializations is the most specialized.

Reference types

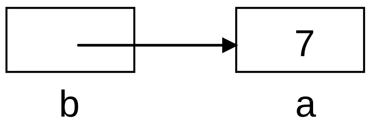
- References are implicit pointers.

```
int a=7;
int& b=a;
cout << b;
```

cf.

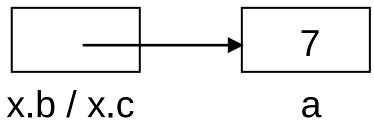
```
int a=7;
int* b=&a;
cout << *b;
```

Both give rise to this diagram:



Example – View the implicit pointer

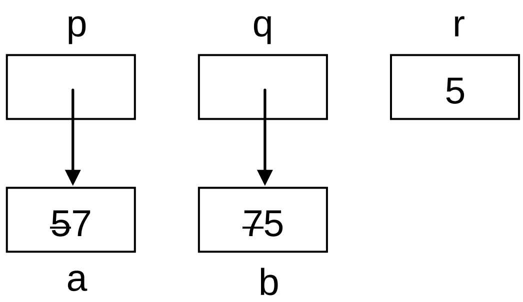
```
int a=7;
union X {
   int& b;
   int* c;
};
int main()
{
   X x={a};
   cout << x.b;            // 7
   cout << x.c;            // address of variable a
}
```



- References as parameters

Example – call by reference

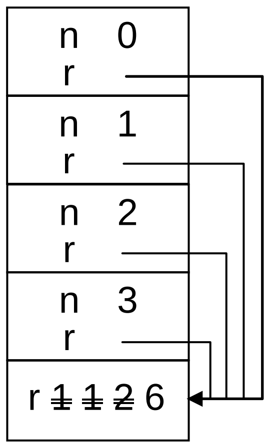
```
void swap(int& p,int& q)   // p,q: inout
{
   int r=p; p=q; q=r;
}
int main()
{
   int a=5,b=7;
   swap(a,b);
   cout << a << b << endl;
}
```



- References as parameters (Cont'd)

Example – call by reference

```
void f(unsigned n,unsigned& r)  // n: in; r: out
{
   if (n==0) r=1;
   else {
      f(n-1,r); r*=n;
   }
}
int main()
{
   unsigned r;
   f(3,r);
   cout << r;
}
```



Example – call by const reference **(efficiency + safety)**

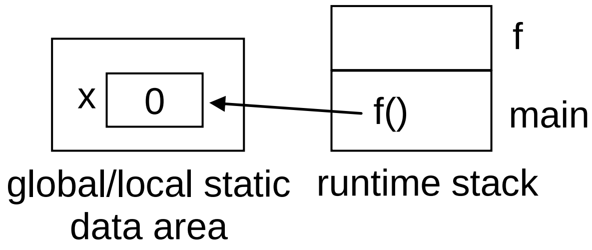
```
struct foo { double x[1000],y[1000]; } bar;
inline double sq(double x) { return x*x; }
double distance(const foo& p)   // p: in, large object
{
   double d=0.0;
   for (int i=0;i<1000;i++) {
      double e=sqrt(sq(p.x[i])+sq(p.y[i]));
      if (d<e) d=e;
   }
   return d;
}
int main() { cout << distance(bar); }
```

Remark

In STL (Standard Template Library), an in-functionality template type parameter **`T`** is usually, but not always, passed by const reference: **`const T&`**.

- References as function values (return by reference)

```
int& f()
{
   static int x=0;
   return x;
}
int main()
{
   for (int i=1;i<=10;i++) f()+=i;
   cout << f();
}
```



Remarks

1 Never return references to local auto variables.
2 A function returning a reference yields an lvalue; otherwise, it yields an rvalue.

- References to const objects

```
int a=7;
const int& b=a;              // ok, identity conversion

const int a=7;
int& b=a;                    // error! no implicit conversion

int& b=const_cast<int&>(a);
```

Remarks

1 For `const_cast`, the target must be a reference or pointer type, e.g.

```
int* b=const_cast<int*>(&a);
```

2 A cast to a reference type yields an lvalue; otherwise, it yields an rvalue,e.g.

```
const_cast<int&>(a)++;
```

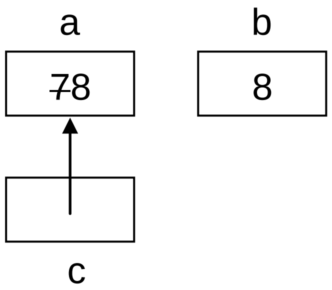
cf.

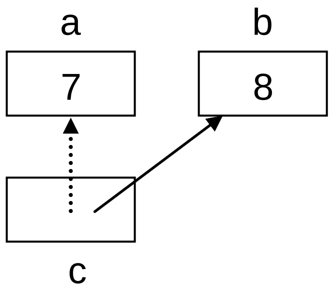
```
const_cast<int*>(&a)++;        // no
(*const_cast<int*>(&a))++;     // ok
```

References vs Pointers

- Reference variables must be initialized, and are always const.

```
int a=7,b=8;                 cf    int a=7,b=8;
int& c=a;                          int* c;  c=&a;
c=b;                               c=&b;
```






```
int&const c=a;
```

Warning: Qualifiers on references are ignored.

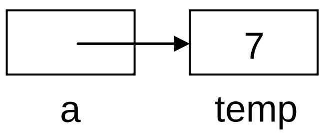
- References are for call-by-reference and return-by-reference. Pointers are for dynamic data structures.

- To facilitate call- and return-by-reference, references may refer to const rvalues.

```
const int& a=7;       cf    const int* a=&7;  ×
```

This is compiled to something like

```
const int temp=7;
const int& a=temp;
```

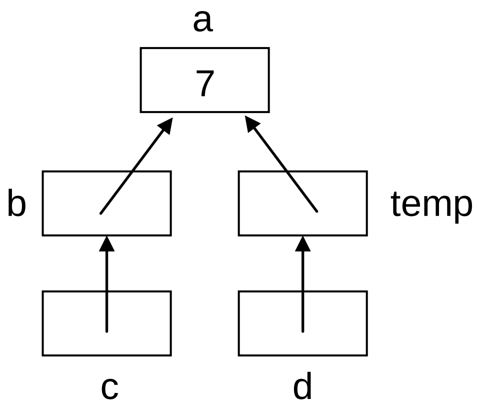


Motivation: Like call- and return-by-value, the actual parameter or return value may be an lvalue or rvalue.

```
void p(int);   ⇒  void p(const int&);
int x=7;
p(x);             p(x);    // ok
p(7);             p(7);    // if NOT, one has to write
                           // const int temp=7;
                           // p(temp);
```

- References may refer to const rvalues (Cont'd)

```
int a=7;
int* b=&a;
int*& c=b;               ①
int*const& d=&a;         ②
```

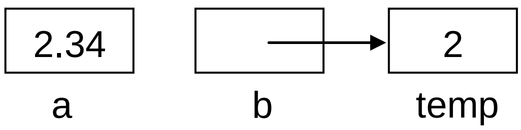


① `b` is an lvalue.
② `&a` is an rvalue.
It is compiled to:

```
int*const temp=&a;
int*const& d=temp;
```

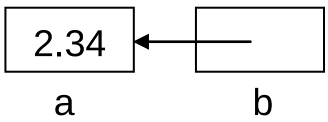
- To facilitate call- and return-by-reference, references may refer to const objects (treated as rvalues) of different types, provided that the required conversions exist.

```
double a=2.34;                    double a=2.34;
const int& b=a;   compiled to     const int temp=a;
                                  const int& b=temp
```



cf

```
double a=2.34;
int* b=reinterpret_cast<int*>(&a);
```



Motivation: Like call- and return-by-value, standard conversions apply to the actual parameter or return value.

```
void p(int);   ⇒   void p(const int&);
double a=2.34;
p(a);              p(a);       // if NOT, one has to write
p(2.34)            p(2.34)     // const int temp=a;
                               // p(temp);
```

- The following are absent from reference types.

  1 Conversion from a reference type to another type

  Q:
  ```
  int a=7;
  int& b=a;
  double c=b;
  ```
  What is the conversion in the last declaration?
  Is it **`int&`** → **`double`**?

  2 Generic references (cf. generic pointer **`void*`**)

  3 Null references

  Q:
  ```
  const int& a=0;
  ```
  Is 0 a null reference here?

  4 Pointers/references to references

  Q:
  ```
  int a=7;
  int& b=a;
  int& & c=b;          // watch the space
  int&* d=&b;
  ```
  What should be the types of **`c`** and **`d`**?

  5 Arrays of references

  Q:
  ```
  int& a[7];
  ```
  What is the type of **`a`** as a pointer?

  A: **`int&[7]`** → **`int&*`**

References to arrays

- Example – One-dimensional array

  Version A – Array as pointer (pointer to array)

```
template<typename T>
void print(T* a,int sz)
{
   for (int i=0;i<sz;i++) cout << a[i];
   cout << endl;
}
int a[2]={1,2};
int b[3]={1,2,3};
print(a,2);              // array-to-pointer conversion
print(b,3);
```

  These two calls generate a single instance:

```
void print<int>(int* a,int sz);
```

  Version B – Array as array (reference to array)

```
template<typename T,int sz>
void print(T (&a)[sz])
{
   for (int i=0;i<sz;i++) cout << a[i];
   cout << endl;
}
int a[2]={1,2};
int b[3]={1,2,3};
print(a);                // no array-to-pointer conversion
print(b);
```

  These two calls generate two distinct instances:

```
void print<int,2>(int (&a)[2]);
void print<int,3>(int (&a)[3]);
```

- Example (Cont'd)

Version A1 // **void print(T* a,int sz)**

```
template<typename T>
void print(T a,int sz)  // remove * from version A
{
   for (int i=0;i<sz;i++) cout << a[i];
   cout << endl;
}
print(a,2);             // array-to-pointer conversion
print(b,3);
```

These two calls generate a single instance:

```
void print<int*>(int* a,int sz);
```

Comment

Version A is more specialized than version A1, because **T*** is more specialized than **T**.

Put differently, version A1 allows the parameter **a** to be of any type, including pointer type, that supports the indexing operator **[]**; whereas, version A requires that the parameter **a** be of pointer type.

Thus, if both versions coexist,

```
print(a,2);          // A
print<int*>(a,2);    // A1
```

When passing a one-dimensional array as a pointer, **T*** is more often used than **T**, because the element type is known in **T***, but not in **T** alone.
For example, to sum up the elements of an array, we may write

```
template<typename T>
T sum(T* a,int sz)
{
   T s=T(0);
   for (int i=0;i<sz;i++) s+=a[i];
   return s;
}
```

- Example (Cont'd)

On the other hand, we cannot write

```
template<typename T>
U sum(T a,int sz)     // T=U*
{
   U s=U(0); …        // no, U isn't a template parameter
}
```

However, we may let **U** be a template parameter:

```
template<typename U,typename T>    // watch the order
U sum(T a,int sz)
{
   U s=U(0);
   for (int i=0;i<sz;i++) s+=a[i];
   return s;
}
```

Since **U** doesn't appear in function parameter declarations, it cannot be deduced from function arguments and must always be explicitly specified:

```
int a[2]={1,2};
sum<int>(a,2)          // T needn't be explicitly specified
```

Note that only the trailing template arguments may be omitted.

Can we do it without the template parameter **U**?
Yes, we can.

```
template<typename T>
typename iterator_traits<T>::value_type
sum(T a,int sz)
{
   typedef typename iterator_traits<T>::value_type U;
   U s=U(0);
   for (int i=0;i<sz;i++) s+=a[i];
   return s;
}
```

- Example (Cont'd)

Digression: On **`iterator_traits`**

This STL class template provides information about various properties of pointer types and pointer-like classes.

It consists of a primary class template for pointer-like classes and a partial specialization for pointer types.

```
// Primary class template
template<class Iterator>
struct iterator_traits {
    // five typedef members, including value_type
};

// Partial specialization for pointer types
template<class T>
struct iterator_traits<T*> {
    typedef T value_type;
    // four more typedef members
};
```

End of digression

```
// Version B1                    // void print(T (&a)[sz])

template<typename T,int sz>
void print(T a[sz])             // remove & from version B
{
    for (int i=0;i<sz;i++) cout << a[i];
    cout << endl;
}
int a[2]={1,2};
print(a);          // array-to-pointer conversion
                   // cannot deduce sz
```

Comment

B1, A, A1	All pass an array as a pointer.
B1, B	Both don't pass the array size as a function parameter.

- Example (Cont'd)

Version B1 is equivalent to

```
template<typename T,int sz>
void print(T* a)      // or, T a[]
{
   for (int i=0;i<sz;i++) cout << a[i];
   cout << endl;
}
```

Since **sz** doesn't appear in function parameter declarations, it cannot be deduced from function arguments and must always be explicitly specified:

```
int a[2]={1,2};
int b[3]={1,2,3};
print<int,2>(a);      // 1st instance
print<int,3>(b);      // 2nd instance
```

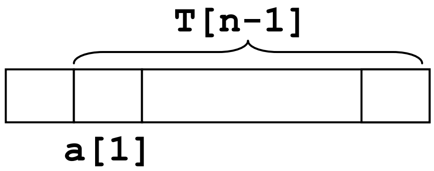
- Example – One-dimensional array (Recursion)

```
// Version A1
template<typename T>
T sum(T* a,int n)
{
   return n==1? a[0]: a[0]+sum(a+1,n-1);
}

int a[5]={1,2,3,4,5};
cout << sum(a,5);
```

One instance **int sum<int>(int*,int);**

// Version A2 – Metaprogram   VC++↑, GNU C++↓




```
template<typename T,int n>
T sum(T (&a)[n])
{
   return
   a[0]+sum(reinterpret_cast<T(&)[n-1]>(a[1]));
}
```

- Example (Cont'd)

```
template<typename T>
T sum(T (&a)[1])
{
   return a[0];
}
```

```
int a[5]={1,2,3,4,5};
cout << sum(a);
```

Four instances `int sum<int,k>(int(&)[k]);` k = 5,4,3,2
One instance `int sum<int>(int(&)[1]);`

However, were they declared **`inline`**, the recursion would be unfolded at compile time. That is,

```
cout << sum(a);
```

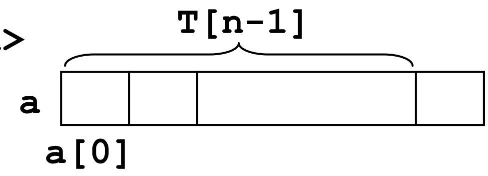
would be compiled to

```
cout << a[0]+a[1]+a[2]+a[3]+a[4];
```

// Version B1

```
template<typename T>
T sum(T* a,int n)
{
   return n==1? a[0]: a[n-1]+sum(a,n-1);
}
```

// Version B2 VC++↑, GNU C++↓




```
template<typename T,int n>
T sum(T (&a)[n])
{
   return
   a[n-1]+sum(reinterpret_cast<T(&)[n-1]>(a));
}
template<typename T>
T sum(T (&a)[1])
{
   return a[0];
}
```

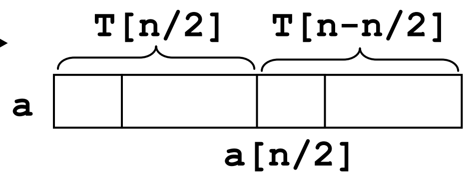
// or, `a[0]`

- Example (Cont'd)

// Version C1

```
template<typename T>
T sum(T* a,int n)
{
   return n==1? a[0]:sum(a,n/2)+sum(a+n/2,n-n/2);
}
```

// Version C2 VC++↑, GNU C++↓




```
template<typename T,int n>
T sum(T (&a)[n])
{
   return
   sum(reinterpret_cast<T(&)[n/2]>(a)) +
   sum(reinterpret_cast<T(&)[n-n/2]>(a[n/2]));
}
template<typename T>
T sum(T (&a)[1])
{
   return a[0];
}
```

- Example (Two-dimensional array)

Version A – Array as pointer (pointer to array)

```
template<typename T,int sz2>
void print(T (*a)[sz2],int sz1)  // or, T a[][sz2]
{
   for (int i=0;i<sz1;i++) {
      for (int j=0;j<sz2;j++) cout << a[i][j];
      cout << endl;
   }
}
int a[2][3]={{1,2,3},{4,5,6}};
int b[3][3]={{1,2,3},{4,5,6},{7,8,9}};
print(a,2);        // int[2][3] → int(*)[3]
print(b,3);        // int[3][3] → int(*)[3]
```

These two calls yield a single instance.

- Example (Cont'd)

  Version B – Array as array (reference to array)

```
template<typename T,int sz1,int sz2>
void print(T (&a)[sz1][sz2])
{
   same as Version A
}
```

```
int a[2][3]={{1,2,3},{4,5,6}};
int b[3][3]={{1,2,3},{4,5,6},{7,8,9}};
print(a);   ①
print(b);   ②
```

  These two calls yield two instances that differ in the value of **sz1**.

  Version C – Overloaded function templates (Metaprogram)

```
template<typename T>
void print(T& a)
{
   cout << a;
}
```

```
template<typename T,int sz>
void print(T (&a)[sz])
{
   for (int i=0;i<sz;i++) print(a[i]);
   cout << endl;
}
```

  The call ① causes the recursive instantiations of the overloaded function templates, yielding

```
void print(int[3](&a)[2])  // int(&a)[2][3]
{
   for (int i=0;i<2;i++) print(a[i]);
   cout << endl;
}
```

- Example (Cont'd)

```
void print(int(&a)[3])
{
   for (int i=0;i<3;i++) print(a[i]);
   cout << endl;
}
void print(int& a) { cout << a; }
```

The call ② causes the instantiation of one more instance:

```
void print(int[3](&)[3]); // int(&a)[3][3]
{
   for (int i=0;i<3;i++) print(a[i]);
   cout << endl;
}
```

Comment – With inline function templates, e.g.

```
template<typename T>
inline void print(T& a) { cout << a; }
```

the code generated for

```
   print(a);
```

is simply a nested loop

```
   for (int i=0;i<2;i++)
      for (int j=0;j<3;j++)
         cout << a[i][j];
```

Version D (with Version C)

// Version C goes here

```
template<typename T>
void print(T *a,int sz)
{
   for (int i=0;i<sz;i++) print(a[i]);
   cout << endl;
}
int a[2][3]={{1,2,3},{4,5,6}};
print(a,2);
```

- Example (Cont'd)

  This call causes the instantiation of the following instance.

```
void print(int[3] *a,int sz) // int(*a)[3]
{
   for (int i=0;i<sz;i++) print(a[i]);
   cout << endl;
}
```

  Q: Why cannot we use this function template alone?
  A: There is no information on the size of the 2nd dimension.

  Comments on function template overloading

```
template<typename T>          ①
void print(T&);
```

  This template can be instantiated for a call with a non-array argument or a k-dimensional array argument, $k \geq 1$, e.g.

```
 void print(int&);
*void print(int[2]&);          // int(&)[2]
 void print(int[2][3]&);       // int(&)[2][3]
```

```
template<typename T,int sz>   ②
void print(T(&)[sz]);
```

  This template can be instantiated for a call with a k-dimensional array argument, $k \geq 1$, e.g.

```
*void print(int(&)[2]);
 void print(int[3](&)[2]);     // int(&)[2][3]
```

  Thus, template ② is more specialized than template ①.

```
int a[2][3];
print(a);                      // instantiate ②
print(a[i]);                   // instantiate ②
print(a[i][j]);                // instantiate ①
```

  N.B. The two starred instances are distinct, as their signatures are different:

```
void print<int[2]>(int(&)[2]);
void print<int,2>(int(&)[2]);
```

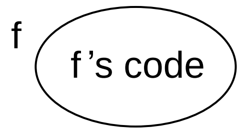
Function types

- Functions and pointers/references to functions

`int(int)`	Functions from `int` to `int`
`int(*)(int)`	Pointers to functions of type `int(int)`
`int(&)(int)`	References to functions of type `int(int)`

- Types of functions

Given a function declared by

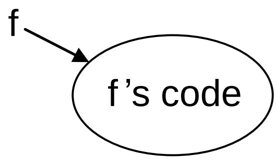



```
int f(int);
```

The function `f` is of the function type `int(int)`.

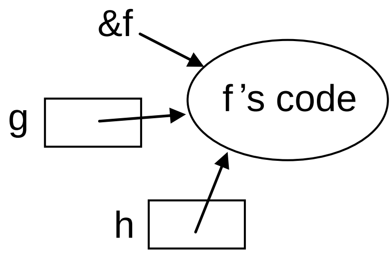
But in certain cases, it is implicitly converted to `int(*)(int)` – the function name `f` is treated as a pointer pointing to the code of the function. This is called function-to-pointer conversion.

- Functions as functions




```
f(3)                   int(*)(int)
&f                        &f
int (&g)(int)=f;       int(int)
```

`sizeof(f)` ✗ Illegal



- Functions as pointers

```
int (*h)(int)=f;
```

Remark

`f(3)`, `g(3)`, `h(3)` and `(*h)(3)` are all equivalent.

For a function call, the function position shall be of
1 (reference to) function type
(In this case, function-to-pointer conversion is suppressed.)
or
2 pointer to function type

- Example

`f(3)`	function type
`(&f)(3)`	pointer to function type
`(*f)(3)`	function type; 1 function-to-pointer conversion
`(*&f)(3)`	function type
`(&*f)(3)`	pointer to function type; 1 function-to-pointer conversion
`(**f)(3)`	function type; 2 function-to-pointer conversions
`(& &f)(3)`	error

- Functions as parameters

$$\sum a[i] \qquad \prod a[i] \qquad \max a[i] \qquad \min a[i]$$

Version A – Functions as pointers **const T&**

```
template<typename T>                    [first,last)
T accumulate(T* first,T* last,T init,
               T (*f)(const T&,const T&)) // or, f
{
   T result=init;
   for (T* it=first;it!=last;++it)
      result=f(result,*it);      // watch the order
   return result;
}
template<typename T>
T plus(const T& x,const T& y)
{
   return x+y;
}
template<typename T>
T multiplies(const T& x,const T& y)
{
   return x*y;
}
int a[7]={1,2,3,4,5,6,7};
accumulate(a,a+7,0,plus<int>)    // or, &plus<int>
accumulate(a,a+7,1,multiplies<int>)
```

- Functions as parameters (Cont'd)

Version B – Functions as functions

```
template<typename T>                        // same code
T accumulate(T* first,T* last,T init,
                    T (&f)(const T&,const T&));

int a[7]={1,2,3,4,5,6,7};
accumulate(a,a+7,0,plus<int>)         // &plus<int> ×
accumulate(a,a+7,1,multiplies<int>)
```

Comment   VC++↓. GNU C++↑

It is unlikely to have both versions. However, If both exist,

```
accumulate(a,a+7,0,plus<int>)         // Ambiguous!
accumulate(a,a+7,0,&plus<int>)        // A
```

Next, STL functions `max` and `min` are defined in `<algorithm>` and `<iostream>` as:

```
template<class T>
const T& max(const T& x,const T& y)
{
    return x<y? y: x;           // LessThanComparable
}                               // return x when equal

template<class T>
const T& min(const T& x,const T& y)
{
    return y<x? y: x;           // LessThanComparable
}                               // return x when equal
```

Q: Can they return `T` objects?

A: They can. But returning by value is inefficient.

Q: Can `plus` and `multiplies` return `const T&` objects?

A: They can't. Otherwise, they would return references to local temporary objects!

```
template<typename T>
const T& plus(const T& x,const T& y)
{
    return x+y;                 // const T temp=x+y;
}                               // return temp;
```

- Functions as parameters (Cont'd)

Approach 1
Albeit inefficient, we may use version A or B to compute maximum and minimum with our own **max** and **min**:

```
template<typename T>
T max(const T& x,const T& y)
{
   return std::max(x,y);
}
template<typename T>
T min(const T& x,const T& y)
{
   return std::min(x,y);
}
accumulate(a,a+7,INT_MIN,::max<int>)
accumulate(a,a+7,INT_MAX,::min<int>)
```

Recall that **::max** and **std::max** are not overloaded function templates, as they are in different namespaces.

Approach 2
Define an overloaded function template specialized to functions returning **const T&**.

Version A′

```
template<typename T>                              // same code
T accumulate(T* first,T* last,T init,
             const T& (*f)(const T&,const T&));

accumulate(a,a+7,0,plus<int>)                     // A
accumulate(a,a+7,INT_MIN,std::max<int>)           // A′
```

Version B′

```
template<typename T>
T accumulate(T* first,T* last,T init,
             const T& (&f)(const T&,const T&));

accumulate(a,a+7,0,plus<int>)                     // B
accumulate(a,a+7,INT_MIN,std::max<int>)           // B′
```

- Functions as parameters (Cont'd)

Version C – Automate and generalize approach 2

```
template<typename T,typename Bfn>     // same code
T accumulate(T* first,T* last,T init,Bfn f);
```

```
accumulate(a,a+7,0,plus<int>)          ①
accumulate(a,a+7,INT_MIN,max<int>)
```

Function-to-pointer conversions apply.
① `Bfn = int(*)(const int&,const int&))`

Alternatively, declare `Bfn& f`
Then, no function-to-pointer conversions apply.
① `Bfn = int(const int&,const int&))`

On function objects (or functors)

Version C can also be instantiated with function objects.

A function object is an object that supports `operator()`.

```
#include <functional>                 // for plus
#include <numeric>                    // for accumulate
accumulate(a,a+7,0,plus<int>())       // Bfn = plus<int>
accumulate(a,a+7,1,multiplies<int>())
```

Below is the abridged class template `plus`:

```
template<class T>
struct plus
{
   T operator()(const T& x,const T& y) const
   {
      return x+y;
   }
};
plus<int> a;              // call the default constructor, too
a.operator()(2,3) ≡ a(2,3)
plus<int>()(2,3)
```

// call the default constructor to create an anonymous object

- Arrays of pointers to functions

  `int(*[3])()`   array of 3 pointers to functions of type `int()`

  Comments

  1 The following types are all illegal, as there are no "arrays of functions" and "arrays of references".

  ```
                  int(&[3])()
  int*[3]()       int&[3]()
  int(*)[3]()     int(&)[3]()
  ```

  2 High precedence `[] ()` left associativity
  Low precedence `* &` right associativity

- Example

  ```
  template<int n> int c() { return n; }
  int (*f[3])()={c<0>,c<1>,c<2>};
  cout << f[2]() << (*f[2])();
  ```

  For readability, give the pointer to function type a name:

  ```
  typedef int (*pf)();
  pf f[3]={c<0>,c<1>,c<2>};
  ```

  Alternatively, give the function type a name:

  ```
  typedef int F();
  F* f[3]={c<0>,c<1>,c<2>};
  ```

- Example (Cont'd)

  ```
  int main()
  {
     int n;
     cin >> n;
     cout << f[n]();        // trade space for speed
  }
  Cf. switch (n) {
     case 0: cout << c<0>(); break;
     case 1: cout << c<1>(); break;
     case 2: cout << c<2>(); break;
     }
  ```

- Functions returning pointers/references to functions

	Functions that take no parameters and
`int(*())()`	return a pointer to a function of type `int()`
`int(&())()`	return a reference to a function of type `int()`

N.B. The following types are all illegal, as there are no "functions returning functions".

```
int*()()        int&()()
int(*)()()      int(&)()()
```

- Example

```
void msg() { cout << "hello\n"; }
void (*mkmsg())() { return msg; }       // * → &, ok
int main() {  mkmsg()(); (*mkmsg())(); }
```

- Example – Function composition

Version A – Hypothetical code

```
int (*c(int (*f)(int),int (*g)(int)))(int)
{
   int h(int x) { return f(g(x)); }
   return h;
}
```

Version B

```
int f(int x) { return x+x; }
int g(int x) { return x*x; }

int (*_f)(int);                          // * → &, no
int (*_g)(int);                          // * → &, no
int _h(int x) { return _f(_g(x)); }

int (*c(int (*f)(int),int (*g)(int)))(int)
{                                        // * → &, ok
   _f=f; _g=g; return _h;
}
int main() { cout << c(f,g)(5); }
```

Q: Can we do better?

Dynamic storage management (Part I)

- In this lecture, we consider only dynamic storage allocation for objects of POD (Plain Old Data) types.

- A POD type is a C++ type that has an equivalent in C.

- POD types include
  1. scalar types, i.e. arithmetic, pointer, etc
  2. POD class types
     i.e. class without user-defined constructor and destructor, private nonstatic data members, etc.

Dynamic (de)allocation of single object

- Allocators

  operator new

  `void* operator new(size_t sz);` // C's **`malloc`**

  1 allocate a block of raw, uninitialized storage of size **`sz`**
  >2 or, throw a **`bad_alloc`** exception, if the heap overflows.

  new operator

  `new T(`_argument, if any_**`)`** assume that **`T`** is a POD type

  1 call **`operator new`** to obtain a block of raw storage of size **`sizeof(T)`**
  2 the storage may or may not be initialized:
  `new T` // uninitialized
  `new T()` // zero-initialized
  `new T(`_single argument_**`)`** // initialized with the argument
  // **`T`** must be a scalar type
  3 yield a **`T*`** pointer pointing to the object created

- Deallocators

operator delete

```
void operator delete(void* p);    // C's free
```

1 do nothing, if **p=0**
2 undefined, if **p** wasn't obtained by an earlier call to **operator new**
3 otherwise, free the storage pointed to by **p**

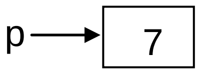
delete operator

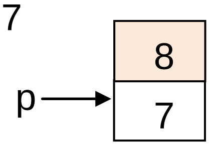
delete p where **p** is **T*** (assume that **T** is a POD type)

1 do nothing, if **p=0**
2 undefined, if **p** wasn't obtained by a previous single-object **new** expression
3 call **operator delete** to free the storage pointed to by **p**
4 yield no value, i.e. the type of **delete p** is **void**.

- Example

```
int* p=new int;                // uninitialized
```
or
```
int* p=new int();              // zero initialized
```
or
```
int* p=new int(7);             // initialized with 7
delete p;
```





Alternative code

```
int* p=static_cast<int*>(operator new(sizeof(int)));
*p=7;                          // assignment, not initialization
operator delete(p);
```

```
void* buf=operator new(sizeof(int));
int* p=new (buf) int(7);       // placement new, initialization
```

or, simply

```
int* p=new (operator new(sizeof(int))) int(7);
```

Principle – Always use the same form of new and delete

Dynamic (de)allocation of array objects

- Allocators

operator new [ ]

```
void* operator new[](size_t sz);
```

1 This is the array-allocation equivalent of **`operator new`**.
2 The storage is uninitialized.

new operator

`new T[`*n*`]` assume that **`T`** is a POD type

1 This is similar to single-object allocation, except that it calls **`operator new[]`** to obtain a block of raw storage of size *n***`*sizeof(T)`**.
2 The storage is uninitialized. (∵ **`T`** is a POD type.)
3 It yields a **`T*`** pointer pointing to the 0^{th} element of the array

- Deallocators

operator delete [ ]

```
void operator delete[](void* p);
```

1 This is the array-deallocation equivalent of **`operator delete`**.
2 The pointer **`p`** must be resulted from a previous call to **`operator new[].`**

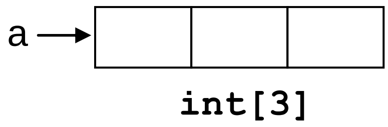
delete operator

`delete [] p` where **`p`** is **`T*`** (assume that **`T`** is a POD type)

1 This is similar to single-object deallocation, except that it calls **`operator delete[]`** to free the storage.
2 The pointer **`p`** must be obtained by an earlier array-object **`new`** expression.

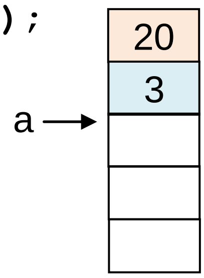
- Example

```
int* a=new int[3];
delete [] a;
```



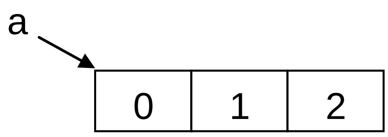
Alternative code

```
void* buf=operator new[](3*sizeof(int));
int* a=static_cast<int*>(buf);
operator delete[](a);
```



- Example – Heap array initialization for POD type

```
int* a=new int[3];
for (int i=0;i<3;i++)
   new (a+i) int(i);        // a[i]=i; assignment
delete [] a;
```



or

```
int* a
=static_cast<int*>(operator new[](3*sizeof(int)));
for (int i=0;i<3;i++)
   new (a+i) int(i);
operator delete[](a);
```

- Example – Dynamically-allocated two-dimensional arrays

The size of the 1st dimension may be determined at run time, but the size of the 2nd dimension must be constant known at compile time.

```
int (*a)[3]=new int[2][3];    // ok
int m=2;
int (*b)[3]=new int[m][3];    // ok
int n=3;
int (*c)[n]=new int[2][n];    // no
int (*d)[n]=new int[m][n];    // no
```

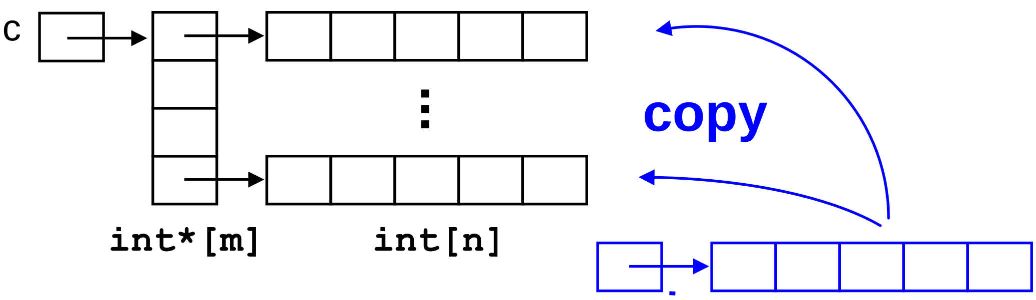
To dynamically allocate an $m \times n$ integer matrix, initialize each element with the integer val, and then destroy it, do this:

- Example (Cont'd)

```
void p(int m,int n,int val)
{
   int** c=new int*[m];
   for (int i=0;i<m;i++) c[i]=new int[n];     ①
   for (int i=0;i<m;i++)
      for (int j=0;j<n;j++) c[i][j]=val;     ②
   for (int i=m-1;i>=0;i--)
      delete [] c[i];        // in reverse order
   delete [] c;
}
```

① or, `new (c+i) int*(new int[n])`

② or, `new (c[i]+j) int(val)`



Comment

```
#include <vector>
void p(int m,int n,int val)
{
   vector<vector<int> > c(m,vector<int>(n,val));
}  // automatically destroyed
```

This piece of code uses STL vector to construct a vector object **c** whose internal structure is similar to the preceding diagram.

Pro – easy to code

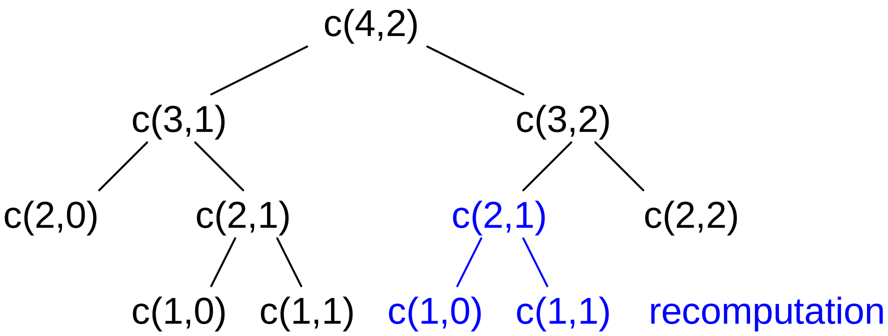
Con – time and space inefficient
The code

```
vector<int>(n,val)
```

constructs a temprorary vector object which is destroyed after being copied to each of the **m** elements of vector **c**.

- Example – combinations (dynamic programming)

$$
\begin{aligned}
c(m,n) &= 1 && n = 0 \text{ or } m = n \\
&= c(m-1,n-1) + c(m-1,n) && \text{otherwise}
\end{aligned}
$$



Algorithm A – Recursion + Tabulation (Top-down)

m \ n	0	1	2	3
0	~~1~~			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4
5		5	10	10
6			15	20
7				35

Instead of using an $(m+1) \times (n+1)$ table, we may use a smaller $(m-n+1) \times (n+1)$ table.

Version A1

```
int c(int m,int n,int** cache)
{
   if (cache[m-n][n]==0)
      if (m==n||n==0) cache[m-n][n]=1;
      else cache[m-n][n]=
                  c(m-1,n,cache)+c(m-1,n-1,cache);
   return cache[m-n][n];
}
```

- Example (Cont'd)

```
int c(int m,int n)
{
   int**cache=new int*[m-n+1];
   for (int i=0;i<=m-n;i++) {
      cache[i]=new int[n+1];
      for (int j=0;j<=n;j++) cache[i][j]=0;
   }
   int ans=c(m,n,cache);
   for (int i=m-n;i>=0;i--) delete [] cache[i];
   delete [] cache;
   return ans;
}
```

Version A2 – Named vector

```
int c(int m,int n,vector<vector<int> >& cache)
{
   // same as Version A1
}

int c(int m,int n)          // may be omitted; default = 0
{
   vector<vector<int> > cache(m-n+1,
                              vector<int>(n+1,0));
   return c(m,n,cache);
}
```

Version A3 – Anonymous vector VC++↓,GNU C++↑

```
int c(int m,int n,
            const vector<vector<int> >& cache)
{
   if (cache[m-n][n]==0)
      const_cast<vector<vector<int> >&>(cache)
      [m-n][n] = m==n||n==0? 1:
                  c(m-1,n,cache)+c(m-1,n-1,cache);
   return cache[m-n][n];
}
```

- Example (Cont'd)

```
int c(int m,int n)
{
   return c(m,n,vector<vector<int> >(m-n+1,
                              vector<int>(n+1,0)));
}
```

Algorithm B – Iteration + Tabulation (Bottom-Up)

					n
		0	1	2	3
	0	1	1	1	1
	1	1	2	3	4
	2	1	3	6	10
	3	1	4	10	20
$m-n$	4	1	5	15	35

Version B1

```
int c(int m,int n)
{
   int**cache=new int*[m-n+1];
   for (int i=0;i<=m-n;i++) cache[i]=new int[n+1];
   for (int j=0;j<=n;j++) cache[0][j]=1;
   for (int i=0;i<=m-n;i++) cache[i][0]=1;
   for (int i=1;i<=m-n;i++)
      for (int j=1;j<=n;j++)
         cache[i][j]=cache[i][j-1]+cache[i-1][j];
   int ans=cache[m-n][n];
   for (int i=m-n;i>=0;i--) delete [] cache[i];
   delete [] cache;
   return ans;
}
```

- Example (Cont'd)

Version B2 – Vector

```
int c(int m,int n)
{
   vector<vector<int> > cache(m-n+1,
                              vector<int>(n+1,1)); //*
   for (int i=1;i<=m-n;i++)
      for (int j=1;j<=n;j++)
         cache[i][j]=cache[i][j-1]+cache[i-1][j];
   return cache[m-n][n];
}
```

Comment
The starred line unnecessarily initializes the entire vector.

Version B3 – Keep one row

```
int c(int m,int n)
{
   int* cache=new int[n+1];
   for (int i=0;i<=n;i++) cache[i]=1;
   for (int i=1;i<=m-n;i++)
      for (int j=1;j<=n;j++)
         cache[j]=cache[j-1]+cache[j];
   int ans=cache[n];
   delete [] cache;
   return ans;
}
```

Version B4 – Keep one row, Vector

```
int c(int m,int n)
{
   vector<int> cache(n+1,1);
   for (int i=1;i<=m-n;i++)
      for (int j=1;j<=n;j++)
         cache[j]=cache[j-1]+cache[j];
   return cache[n];
}
```

- Example – Matrix chain multiplication (Optimization problem)

Given n matrices $M_1, M_2, \dots, M_n$, where M_i has dimension $d_{i-1} \times d_i$, compute the matrix product $M_1 M_2 \cdots M_n$ in a way that minimizes the number of scalar multiplications.

For example,

M_1	2×3		
M_2	3×4	$(M_1 M_2) M_3$	$2 \times 3 \times 4 + 2 \times 4 \times 5 = 64$
M_3	4×5	$M_1 (M_2 M_3)$	$3 \times 4 \times 5 + 2 \times 3 \times 5 = 90$

The brute-force approach solves many subproblems again.
For example, consider $n = 4$:

$M_1(M_2(M_3M_4))$ $\quad$ $M_1((M_2M_3)M_4)$

$(M_1M_2)(M_3M_4)$

$(M_1(M_2M_3))M_4$ $\quad$ $((M_1M_2)M_3)M_4$

Dynamic programming solution

Let $m_{ij} =$ the optimal cost for computing $\underbrace{M_i \cdots M_k}_{d_{i-1} \times d_k} \underbrace{M_{k+1} \cdots M_j}_{d_k \times d_j}$

Then,

$m_{ii} = 0$

$$m_{ij} = \min_{i \le k < j} \left(m_{ik} + m_{k+1,j} + d_{i-1} d_k d_j \right), \quad i < j$$

Wanted: m_{1n}

Comments

1 Try all the possible ways of dividing into 2 subproblems and find the best way.

$M_i | M_{i+1} M_{i+2} \cdots M_{j-1} M_j \qquad k = i$

$M_i M_{i+1} | M_{i+2} \cdots M_{j-1} M_j \qquad k = i + 1$

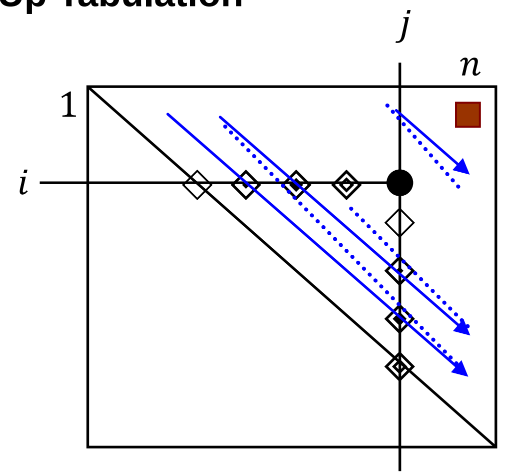
$\vdots$

$M_i M_{i+1} M_{i+2} \cdots M_{j-1} | M_j \qquad k = j - 1$

2 Use tabulation to avoid recomputation.

- Example (Cont'd)

Bottom-Up Tabulation



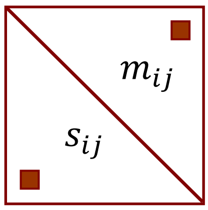
$m_{ij}, 1 \leq i \leq j \leq n$

Construct an optimal solution

$s_{ij} =$ the value of k such that $(M_i \cdots M_k)(M_{k+1} \cdots M_j)$ results in an optimal solution for $M_i \cdots M_j$, $i < j$

Note that s_{ii} is undefined.

Also, observe that s_{ij} and m_{ij} may share an $n \times n$ table.



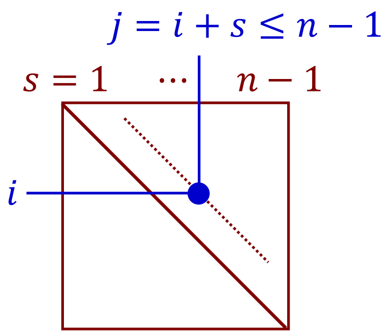

```
int mcm(int* d,int n);

int main()
{
   int d[7]={30,35,15,5,10,20,25};
   cout << mcm(d+1,6);
}
```

Compute $M_0 \cdots M_{n-1}$
Note that the matrices are numbered from 0. So, the dimensions are $d_{-1} d_0 d_1 \cdots d_{n-1}$.

- Example (Cont'd)

```
void optsol(int i,int j,int** tbl)
{
   if (i==j) cout << "M" << i;
   else {                          // (M0 (M1M2) ) ( (M3M4) M5)
      int k=tbl[j][i];
      if (k!=i)   cout << "("; optsol(i,k,tbl);
      if (k!=i)   cout << ")";
      if (k!=j-1) cout << "("; optsol(k+1,j,tbl);
      if (k!=j-1) cout << ")";
   }
}
```




```
int mcm(int* d,int n)
{
   int** tbl=new int*[n];
   for (int i=0;i<n;i++) {
      tbl[i]=new int[n];  tbl[i][i]=0;
   }
   for (int s=1;s<n;s++)
      for (int i=0;i<n-s;i++) {
         int j=i+s,mij=INT_MAX,sij;
         for (int k=i;k<j;k++) {
            int next=tbl[i][k]+tbl[k+1][j]+
                                       d[i-1]*d[k]*d[j];
            if (next<mij) { mij=next; sij=k; }
         }
         tbl[i][j]=mij;
         tbl[j][i]=sij;
      }
   int ans=tbl[0][n-1];
   optsol(0,n-1,tbl);  cout << endl;
   for (int i=0;i<n;i++) delete [] tbl[i];
   delete [] tbl;
   return ans;
}
```

Finally, the vector version is left to you.

Placement new

- Single objects

  operator new // placement operator new

  ```
  void* operator new(size_t,void* buf)
  {
     return buf;
  }
  ```

  new operator

  new (buf) T(*arguments*, *if any***)** // placement new

  This is similar to single-object allocation, except that it calls

  operator new(sizeof(T),buf)

  to obtain storage.

- Array objects

  operator new [ ] // placement operator new[]

  ```
  void* operator new[](size_t,void* buf)
  {
     return buf;
  }
  ```

  new operator

  new (buf) T[n] // placement new

  This is similar to array-object allocation, except that it calls

  operate new[](sizeof(T)*n,buf)

  to obtain storage.

- There are many uses of placement new.
  The simplest use is to place an object in a particular memory location.
  Another use is nothrow new.

- Nothrow new

  Recall that

```
void* operator new(size_t sz);
void* operator new[](size_t sz);
```

  throw a **`bad_alloc`** exception, if the heap overflows.

  The following overloaded functions return a null point, if the heap overflows.

```
void* operator new(size_t,const std::nothrow_t&);
void* operator new[](size_t,const std::nothrow_t&);
```

  The type **`nothrow_t`** and the object **`nothrow`** are defined in **`<new>`** as

```
struct nothrow_t {};
const nothrow_t nothrow;
```

  The nothrow new

  `new (nothrow) T(`*arguments,if any***`)`**
  `new (nothrow) T[n]`

  call

```
operator new(sizeof(T),nothrow)
operator new[](n*sizeof(T),nothrow)
```

  respectively, to obtain storage.

- Comment

  In general, the placement new expression

  `new (A,B,C,…) T(`*arguments*, *if any***`)`**
  `new (A,B,C,…) T[n]`

  calls

```
operator new(sizeof(T),A,B,C,…)
operator new[](n*sizeof(T),A,B,C,…)
```

  respectively, to obtain storage.

- Comment

```
void* operator new(size_t,void* buf)   // built-in
{
   return buf;
}
```

// heap storage

```
new (operator new(sizeof(int))) int(7);
```

// static or stack storage

```
int x;
new (&x) int(7);          //*
```

Both will invoke the built-in **operator new**.

Next, let's introduce the following overloaded function for fun:

```
void* operator new(size_t,int* buf)   // user-defined
{
   cout << "Bingo!";      // for testing purpose
   return buf;
}
```

Now, the call in the starred line will invoke our own **operator new**.

In real world, we may define

```
void* operator new(size_t,char* buf)   // user-defined
{
   // manage the storage pointed to by buf
   return  a pointer to the allocated space;
}
```

```
char pool[1000000];
new (pool) int(7);
new (pool) double(3.14);
```