W.P.Hwang
CS-CCS-NCTU

Lecture – C++ as a better C

CDT and ADT

- A data type consists of a set of values of the same kind and a set of operations acting on the values.
- Data abstraction
  One needs only know *what* operations on the data are available, and needs not know *how* the data are represented.
- Procedural abstraction
  One needs only know *what* a procedure does, and needs not know *how* it does, i.e. how it is implemented.

  Syntactic abstraction
  One needs only know *what* a macro does, and needs not know *how* it does, i.e. how it is implemented.
- Concrete data type (CDT)
  The data representation is visible to the user of the data type.
- Abstract data type (ADT)
  The data representation is hidden (data encapsulation) and can be replaced by another representation without changing the external behavior of the operations.
- Example – Stack as concrete data type

  Stack implementation – Sequential array representation

```
struct stack {                 class stack {
   int top;             ≡      public:
   int stk[80];                   int top,stk[80];
};                             };
```

```
inline void push(stack* s,int n)
{
   s->stk[++s->top]=n;      // top==-1 for empty stack
}
```

- Example (Cont'd)

```
inline void pop(stack* s) { s->top--; }

inline int* top(stack* s)
{
    return &s->stk[s->top];
}

inline const int* top(const stack* s)
{
    return &s->stk[s->top];
}

inline bool empty(const stack* s)
{
    return s->top==-1;
}
```

Comment on STL (Standard Template Library)

Instead of declaring

```
int top(const stack*);          // return by value
```

we declares two STL-style overloaded functions

```
int* top(stack*);               // version A
const int* top(const stack*); // version B
```

1 STL uses call- and return-by-reference. At this moment, we use by-value to simulate by-reference.

2 The parameter of version A is of in-functionality, but it can't be declared as
`int* top(const stack*)` // `const int*` → `int*`
because this cannot coexist with version B.

3 Version A is useful in case we need to modify the stack top. Version B is useful when the stack is read-only, e.g.

```
void print_stack(const stack* s)
{
    printf("%d",*top(s));   // can't use version A
}
```

- Example

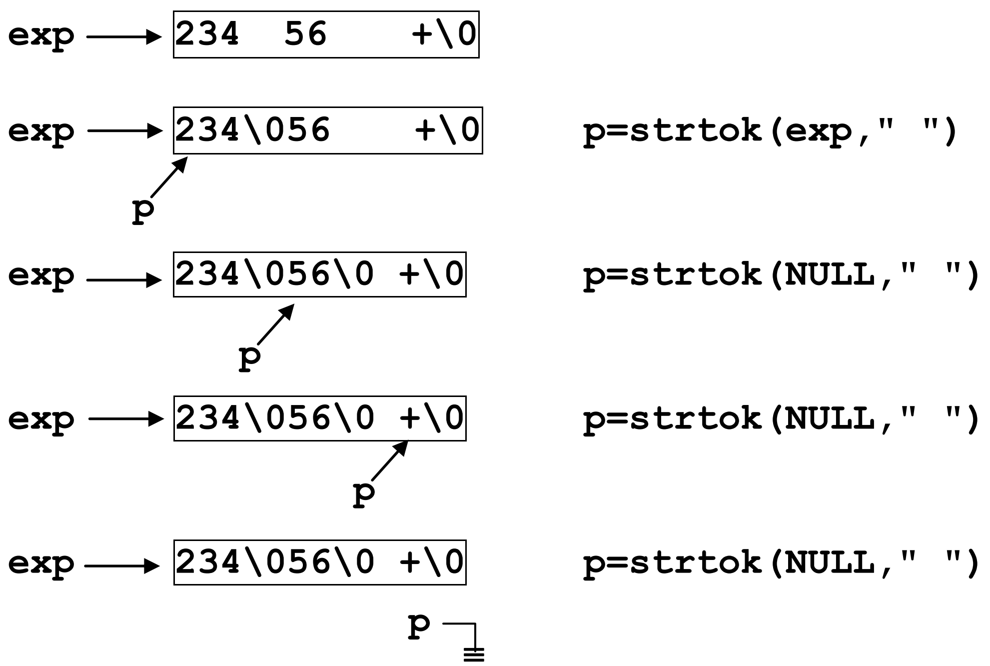
Stack application: Evaluation of postfix expressions

```
int main()
{
   const int sz=80;
   char exp[sz];
   printf("Enter a postfix expression: ");
   while (gets(exp)!=NULL) {
      printf("Value = %d\n",eval(exp));
      printf("Enter a postfix expression: ");
   }
}
```

We assume that each line contains a syntactically correct integral postfix expression in which tokens are separated by spaces. We shall use

```
char* strtok(char* exp,const char* delimiters)
```

to extract successively the tokens in **exp** that are separated by characters in **delimiters**.



After a first call to **strtok**, the function may be called with **NULL** as the first argument to extract the next token following by where the last call to **strtok** found a delimiter.

- Example (Cont'd)

A postfix expression may be evaluated with the help of a stack.

postfix expression			stack		
2 3 4 * +			4		
i.e. **2+3*4**		3	3	12	
	2	2	2	2	14
2 3 + 4 *		3		4	
i.e. **(2+3)*4**	2	2	5	5	20

```
int eval(char* exp)
{
   stack s={-1};
   char* p=strtok(exp," ");
   while (p!=NULL) {
      if (strstr("+-*/",p)==NULL)        ②
         push(&s,atoi(p));                ①
      else {
         int v=*top(&s);
         pop(&s);
         switch (*p) {
         case '+': *top(&s)+=v; break;
         case '-': *top(&s)-=v; break;
         case '*': *top(&s)*=v; break;
         case '/': *top(&s)/=v; break;
         }
      }
      p=strtok(NULL," ");
   }
   return *top(&s);
}
```

① **atoi**, **atol**, **atoll** and **atof** convert strings to **int**, **long**, **long long** and **double**, respectively.

```
atoi("      777")       ⇒ 777
atoi("777bingo")        ⇒ 777
atoi("bingo777")        ⇒ 0
```

- Example (Cont'd)

  ② `const char* strstr(const char* s,const char* t);`
  `char* strstr(char* s,const char* t);` // C++ only

  These two functions check if `t` is a substring of `s`.

```
strstr("+-*/","*")   ⇒ "*/"
strstr("+-*/","%")   ⇒ NULL
char s[]="+-*/";
strstr(s,"*")[0]='%'
```

Disadvantages of CDT

1. On the application side, the user might mistakenly manipulate the stack, say, by `s.top*=5`, but the compiler cannot detect such an error.
2. The application and implementation are not independent – the application code is mixed up with part of the implementation code.

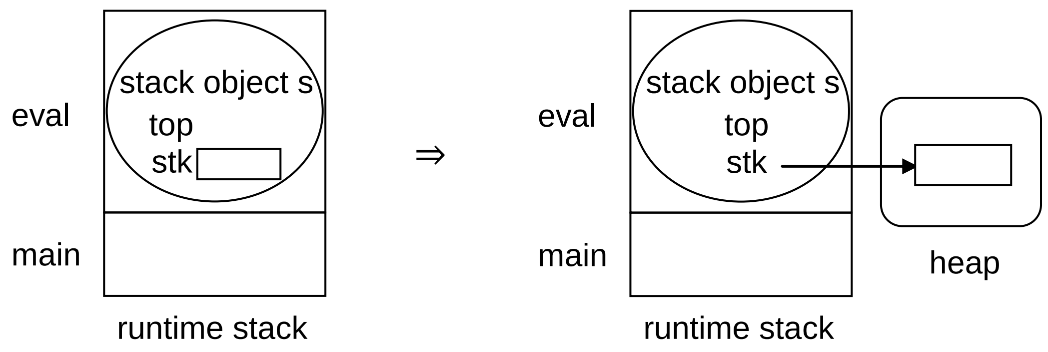
For the latter, assume that the implementer decides to allocate the array dynamically and declares the stack type as follows:

```
struct stack {
    int top,*stk;
};
```

The implementation of the stack operations remains unchanged. But, the application side has to be modified:

```
int eval(char* exp)
{
    stack s={-1};
    stack s={-1,(int*)calloc(80,sizeof(int))};
            ⋮
    return *top(&s);
    int r=*top(&s);
    free(s.stk);
    return r;
}
```

- Example (Cont'd)



- Example – Stack as abstract data type

  Stack implementation – Sequential array representation

```
class stack {
public:                                   Interface
   stack();
   void push(int);
   void pop();
   int* top();                            // stack s;
   const int* top() const;                // s.top(); vs top(&s);
   bool empty() const;
private:                                  .............................
   int _top;                              Implementation
   int stk[80];
};

inline stack::stack() : _top(-1) {}

inline void stack::push(int n) { stk[++_top]=n; }

inline void stack::pop() { _top--; }

inline int* stack::top() { return &stk[_top]; }

inline const int* stack::top() const
{
   return &stk[_top];
}

inline bool stack::empty() const { return _top==-1; }
```

- Example (Cont'd)

  Remarks

  1 Member functions defined inside the class definition are inline functions; member functions defined outside are non-inline functions, unless they are declared so.
  The **`inline`** specifier may appear in the declaration or the definition or both – it isn't a part of the function's type.

  2 The **`const`** qualifier must appear in both the declaration and the definition, as it is a part of the function's type.

  Stack application – Evaluation of postfix expressions

```
int eval(char* exp)
{
   stack s;                      // *
   char* p=strtok(exp," ");
   while (p!=NULL) {
      if (strstr("+-*/",p)==NULL)
         s.push(atoi(p));
      else {
         int v=*s.top();
         s.pop();
         switch (*p) {
         case '+': *s.top()+=v; break;
         case '-': *s.top()-=v; break;
         case '*': *s.top()*=v; break;
         case '/': *s.top()/=v; break;
         }
      }
      p=strtok(NULL," ");
   }
   return *s.top();
}              (cf. evaluate an expression; execute a statement)
```

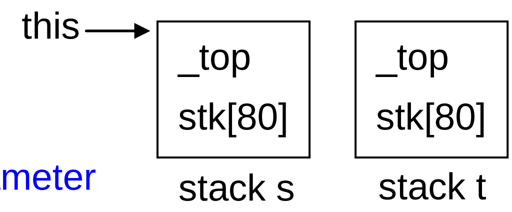
The elaboration of the declaration in the starred line will call the default ctor (i.e. a ctor that can be called without an argument) tacitly to initialize the stack.

- Example (Cont'd)

  Q: Why are ctors (i.e. constructors) needed?
  A: Since the data representation is hidden, one can't initialize the private data of a stack from the application side.

  CDT and compiled ADT

```
inline void stack::push(int n) // object-dependent code
{
   stk[++_top]=n;              // which stk? which _top?
}
s.push(atoi(p));
```



are compiled to

implicit parameter

```
inline void push(stack* this,int n)
{
   this->stk[++this->_top]=n;
}
push(&s,atoi(p));
```

respectively. Except for the parameter name, they are exactly the same as to our earlier CDT operations:

```
inline void push(stack* s,int n)
{
   s->stk[++s->top]=n;
}
push(&s,atoi(p));
```

The object pointed to by **this** may be cv-qualified, e.g.

```
inline bool stack::empty() const
{
   return _top==-1;
}
```

is compiled to

- Example (Cont'd)

```
inline bool empty(const stack* this)
{
   return this->_top==-1;
}
```

which again is the same as to our earlier CDT operation:

```
inline bool empty(const stack* s)
{
   return s->top==-1;
}
```

Advantages of ADT

1 Data encapsulation
2 The application and implementation are independent.
3 Code reusability

For the 2nd point, consider again allocating the array dynamically. The only changes that have to be made are given below.

```
class stack {
public:
   stack();
   ~stack();
   void push(int);
   void pop();
   int* top();
   const int* top() const;
   bool empty() const;
private:
   int _top,*stk;
};

inline stack::stack()
: _top(-1),stk((int*)calloc(80,sizeof(int)))
{}

inline stack::~stack() { free(stk); }
```

- Example (Cont'd)

In particular, the application side remains the same.

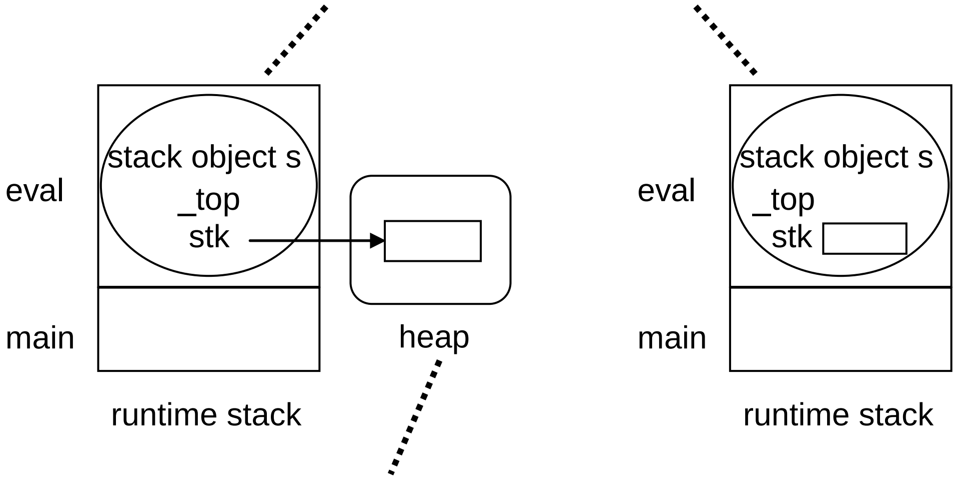
```
int eval(char* exp)
{
    stack s;        // call the ctor tacitly to construct the object
        ⋮
    return s.top();
}                   // call the dtor tacitly to destruct the object
```

Q: Why are dtors (destructors) needed?
A: Again, it is because the data representation is hidden.

A ctor is invoked when the lifetime of an object begins; the dtor is invoked when the object's lifetime ends.

Principle
Define a dtor for classes with dynamically allocated memory.

The runtime stack is managed by the code generated by the compiler.



The heap has to be managed by user provided code (in ctors and dtors, when considering ADT's).

Implicitly generated dtor

The fact that the dtor is invoked automatically when an object's lifetime ends implies that every class must have a dtor.

- Example (Cont'd)

  Instead of burdening the programmer with the task of defining a dtor for a class without dynamically allocated memory, the compiler will implicitly generate one.

  For example, the implicitly generated dtor for the **`stack`** class implemented by statically-allocated array reads as

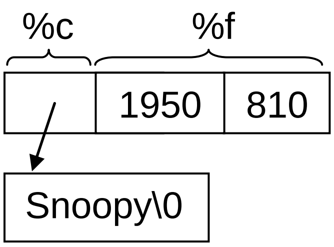
```
inline statck::~stack() {}
```

  Observe that this is an inline function without any code in its body. Therefore, a call to it has no compiled code at all. Put another way, all of the semantics, pragmatics and efficiency issues are well looked after.

Type-safe iostream library

- IO in C is type unsafe.

```
#include <stdio.h>
int main()
{
   printf("%c%f","Snoopy",1950,810);
}
```



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

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Snoopy" << 1950;  cout << 810;
}          cout
                cout
```

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

cout << bingo       // inserts data to output stream
cin >> bingo        // extracts data from input stream
```

- Each iostream object has a format state.

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

cin >> hex >> bingo;    // scanf("%x",&bingo);
   cin
           hexadecimal
```

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

```
int z=1950; printf("%X%x%o%d%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.

- Each iostream object also has internal condition states.

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

- Example

Version A

```
#include <iostream>
using namespace std;
void foo(int);            // unspecified
int main()
{
   int n;
   while (cout << "Enter:",cin >> n)
      foo(n);                     cin
}
```

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

- Example (Cont'd)

Version B In good state, unless **cout** corrupts

```
int main()
{                cout
   while (cout << "Enter:") {
      int n; cin >> n;
      if (cin.eof()) break;
      foo(n);
   }
}
```

Q: Can we write

```
   while (cout << "Enter:",!cin.eof()) {
      int n; cin >> n; foo(n);
   }
```

A: No, it makes no sense to interrogate the state before an input operation.

Version C – Handling invalid inputs

```
int main()
{
   while (cout << "Enter:") {
      int n; cin >> n;
      if (cin.eof()) return;
      if (cin.fail()) {
         cout << "Try again.\n";
         cin.clear();                // reset to good state
         cin.ignore(INT_MAX,'\n');
         continue;
      }                        // eat at most INT_MAX characters
      foo(n);                  // or up to character '\n'
   }
}
```

Q: Can the order of the two **if** statements be reversed?

A: No. **cin.fail()** returns true, if **cin** is in bad, eof, or fail state, i.e. **cin.fail()==!cin.good()**. In fact, reaching end-of-file enters both eof and fail states.

- Implementation of condition states

  Below is a simplified, abridged implementation

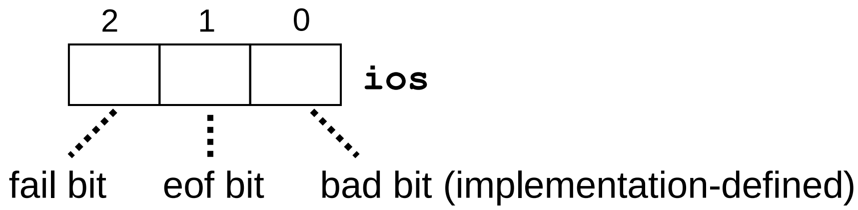
```
class istream {
public:
   typedef enum { goodbit=0,badbit=1<<0,
                  eofbit=1<<1,failbit=1<<2 }
          iostate;
   istream() : ios(goodbit) {}
   bool good() { return ios==goodbit; }
   bool eof();
   bool fail();
   void clear(iostate =goodbit);
   iostate rdstate() { return ios; }
private:
   iostate ios;
};

inline bool istream::eof()
{
   return (ios&eofbit)!=0;      //1
}

inline bool iostream::fail()
{
   return (ios&badbit)!=0||(ios&failbit)!=0; //2
}
```

Comments
As mentioned above, reaching end-of-file enters both **eofbit** and **failbit** iostates. Therefore,
1 line 1 can't be written as **ios==eofbit**
2 line 2 needn't check **eofbit**

```
inline void istream::clear(iostate s) { ios=s; }
```

Comments
1 The default argument may appear in the declaration or the definition of the member function, but not both.
2 **cin.clear()** = **cin.clear(gootbit)**

3 **`cin.clear(failbit)`**
doesn't mean to clear the "fail bit". What it means is to set **`cin`** to the **`failbit`** iostate



- Example

```
int main()
{
   cout << istream::goodbit;          // 0
   cout << istream::failbit;          // 4 (usually)
   cout << boolalpha;
   cout << cin.good();                // true
   cout << cin.fail();                // false
   cin.clear(istream::failbit);
   cout << noboolalpha;
   cout << cin.fail();                // 1
}
```

- Example

```
int main()
{
   cout << "Enter: ";
   int n;
   cin >> n;
   istream::iostate ios=cin.rdstate();
   cout << ios << endl;
}
```

Smaple runs

```
Enter: 7
0
Enter: Snoopy
4
```

```
Enter: ^Z     (or, ^D in Unix)
6
```

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

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

- Example

```
std::cout -> using namespace std;   // cout isn't added to global
::cout -> 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 to an unqualified function call – the associated namespaces of the argument types, if any, are considered.
- Example on operator function

```
namespace C {
   class complex {
   public:
      complex(double r,double i) : r(r),i(i) {}
      complex operator+(complex rhs)
      {
         return complex(r+rhs.r,i+rhs.i);
      }
   private:
      double r,i;
   };
}
```

// Version A – with using

```
using namespace C;            // as if class complex were
int main()                    // declared globally
{
   complex x(2,3),y(4,5);     // call the ctor tacitly
   complex z=x.operator+(y);
}
```

Comments

1 The underlined call to **operator+** may be abbreviated to
  x+y // operator overloading

2 Given
  x.operator+(y)
  the compiler will look up **operator+** in class **complex**, due to the type of object **x**. It is in fact equivalent to
  x.complex::operator+(y)

3 To qualify with the namespace **C**, we have to write
  x.C::complex::operator+(y)
  rather than
  x.C::operator+(y)

- Example (Cont'd)

  // Version B – without using

```
int main()
{
   C::complex x(2,3),y(4,5);
   C::complex z=x.C::complex::operator+(y);
}
```

  Comments

  1 ADL allows us to unqualify the underlined call to
  x.complex::operator+(y) //*
  which in turn may be abbreviated to
  x.operator+(y) //*
  which in turn may be abbreviated to
  x+y //*

  2 The usual unqualified name lookup for **operator+** in the starred lines doesn't find anything, since class **complex** is invisible. So, the namespace **C** associated with the type of argument **y** is considered, making class **complex** visible.

  3 This pattern is commonly used in C++ library.
  For example,

```
#include <iostream>
int main()
{
   std::cout << "ADL";
}
```

  The dotted underlined code is shorthand for
  std::operator<<(std::cout,"ADL");

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

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

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

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

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(,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);            // ×
```

- 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)    // ×
{
   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 (`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, here are overloaded `sqrt` functions:

  ```
  float sqrt(float);                // 1
  double sqrt(double);              // 2
  long double sqrt(long double);    // 3
  template<typename T>              // 4, C++11
  double sqrt(T);
  ```

  Overload resolution then selects the best function to call.

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

  Version 1: identity
  Version 2: `float` → `double`
  Version 3: `float` → `long double`
  Version 4: at this moment, let's ignore this function template

  Clearly, version 1 is the best and will be selected, as desired.

- C++ is polymorphic, whereas C is monomorphic.
- Example

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

Prior to C++11, the call **`sqrt(n)`** is ambiguous, because none of the first three versions is the best:

$$\left.\begin{array}{l}\text{Version 1: } \texttt{int} \to \texttt{float} \\ \text{Version 2: } \texttt{int} \to \texttt{double} \\ \text{Version 3: } \texttt{int} \to \texttt{long double}\end{array}\right\}\ \text{floating-integral conversion}$$

Thanks to version 4, now in C++11, it will invoke the instance
`double sqrt<int>(int);`
of the function template. (More to say later on.)

Therefore, the test
`d<=sqrt(n)`
is evaluated as
`static_cast<double>(d)<=sqrt(n).`
But, logically specking, the condition ought to be
`d<=static_cast<int>(sqrt(n))`

Instead of using version 4, we may introduce our own integral square root function.

```
using std::sqrt;       // or, using namespace std;
int sqrt(int n)        // for ⌊√n⌋
{
    return sqrt(double(n));    // not a recursive call
}
```

This definition resorts to **`std::sqrt(double)`** and needs two type conversions.

- Example (Cont'd)

  Alternatively, we may define

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

  Comments

  1 Now, for the test **`d<=sqrt(n)`**, our own **`sqrt`** function will be selected by the compiler.

  2 Our own **`::sqrt`** and the 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.

  3 Alternatively, we may make them overloaded in the **`std`** namespace by enlarging the **`std`** namespace:

```
namespace std {
    int sqrt(int n)      // 5
    {
        int r=0;
        while(r*r<=n) r++;
        return r-1;
    }
}
```

- 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.

- Overloaded functions (Cont'd)

  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;
  ```

  The **`volatile`** informs the compiler not to optimize the code by caching the variable **`available`** in a register.

- Overloaded functions (Cont'd)

  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 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, ICS(F, k) is not worse than ICS(G, k), and
  2. $\exists$ argument k, ICS(F, k) is better than 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
V
worse

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

- Example

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

pointer conversion Rank: Conversion

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

array-to-pointer conversion qualification conversion

- Comment

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

- Better implicit (standard) conversion sequence

Let S_1 and S_2 be two ICS's, to determine if 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 is better than 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

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

- Example (Cont'd)

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

Comment

```
① void p(int*);
② void p(int const*);
③ void p(int*const);
④ void p(int const*const);
```

① and ③ can't be overloaded; ② and ④ can't be overloaded.

- 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 `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!

nullptr (C++11)

- Motivation

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

  Function call `p(0);` // prefer ①
  `p(NULL);` // prefer ②
  Viable ① identity
  ② null pointer conversion
  Best viable ①

  To invoke ②, an explicit cast is needed:

  Function call `p((int*)NULL);`
  Viable ② null pointer conversion
  Best viable ②

  C++11 introduces **`nullptr`** to make the call simple and more readable.

  Function call `p(nullptr);`
  Viable ② null pointer conversion
  Best viable ②

- **`nullptr`** a reserved word that denotes a constant of the type **`std::nullptr_t`**.

- **`nullptr`** can be converted to
  1 the null pointer value of any pointer type
  2 **`false`** of **`bool`** type
  3 nothing else

  Comment
  `NULL` is a macro that expands to 0 of some integral type.
  In addition to 1 and 2 above, **`NULL`** may also be converted to **`nullptr`** of **`nullptr_t`** type.

- Example

```
struct node {};
node* p=NULL;              // null pointer conversion
node* q=nullptr;           // null pointer conversion
if (p==nullptr) …          // null pointer conversion
if (q==NULL) …             // null pointer conversion
if (NULL==nullptr) …       // null pointer conversion

if (NULL) …                // Boolean conversion
if (nullptr) …             // Boolean conversion

NULL+2.3                   // floating-integral conversion
nullptr+2.3                // error
```

- The null pointer conversion
  `nullptr/NULL` → *cv* **`T*`**
  is a single pointer conversion, not the sequence of a pointer conversion followed by a qualification conversion.

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

p(nullptr);                // ambiguous
p((int*)nullptr);          // ①
p((const int*)nullptr)     // ②
```

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);
```

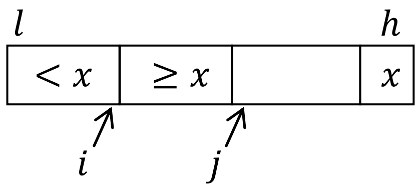
  Two conversions of the rank "Exact match" apply:

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;
      }
   a[h]=a[i+1]; a[i+1]=x;
   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); }
```

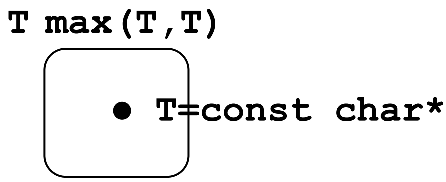
  But an array of n 0's and 1's can easily be sorted in $O(n)$ time.

  // 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;
}
```



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

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

② ● ③

```
int a[5]={3,5,1,4,2};
```

Function call `sort(a,0,4);`
Candidate instantiation of ①

```
void sort<int>(int*,int,int)
```

Function call `sort(a,5);`
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. **`short x=3; max(x,2),max(2,'a')`**, etc.

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

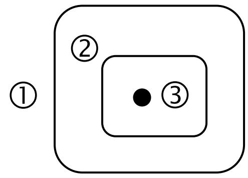
Partial ordering of function templates

- 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 x,T y)
  {
     return x<y? y: x;
  }
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* x,const char* y)
  {
     return strcmp(x,y)<0? y: x;
  }
```

As written, it is an explicit specialization of the more specialized template ② (for **`T = const char`**), rather than ① (for **`T = 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.

Comment
It is unreasonable to treat ③ as an explicit specialization of ①. Nonetheless, if you really want to do so, the template argument **`T = const char*`** must be explicitly specified.

- 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*)**

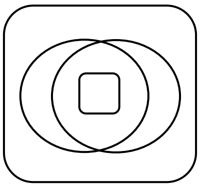
$\because$ { **p(T1*,T2*)** } = { **p(T*,T*)** } $\cup$ { **p(T1*,T2*)** | **T1**$\neq$**T2** }

which in turn is more specialized than **p(T1*,T2)**

$\because$ { **p(T1*,T2)** } = { **p(T1*,T2*)** } $\cup$
{ **p(T1*,T2)** | **T2** isn't a pointer type }

N.B. The return type doesn't matter.

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



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

//* if missing

```
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)
```

Comment

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.
To resolve this ambiguity, write, say, **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;
}
```

- Comment

A class template explicit specialization may be quite different from the generic class template.
But, a function template explicit specialization must be obtainable from the generic function template by substituting template arguments for template parameters.

Example

Given
```
template<typename T>
T p(T x) { return x; }
```
Which is a legal explicit specialization?

a)
```
template<>
int p(int x) { return x+1; }
```
b)
```
template<>
void p(int x) { cout << x; }
```
A: a) is legal, but b) isn't.

Given
```
template<typename T>
struct X {
   T p(T x) { return x; }
};
```
Which is a legal explicit specialization?

a)
```
template<>
struct X<int> {
   int p(int x) { return x+1; }
};
```
b)
```
template<>
struct X<int> {
   void p(int x) { cout << x; }
};
```
c)
```
template<>
struct X<int> {
   double x;
};
```
A: All of them are legal.

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

To guarantee correct compilation, all classes here have the same members. However, any value that is meaningless to a class is set to 0 or false.

For example

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

- Example

Consider the following program

```
#include <cmath>
int main()
{
   cout << sqrt(9.0);              // 1
   cout << sqrt<double>(9.0);      // 2 ×
   cout << sqrt(9);                // 3
   cout << sqrt<int>(9);           // 4
}
```

Under GNU C++ and Clang C++, the 2nd call to **`sqrt`** causes a compilation error. The remaining three calls invoke **`double sqrt(double)`**.

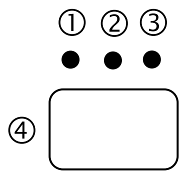
Attempt 1

```
template<typename T>
double sqrt(T x)
{
   return sqrt((double)x);
}
template<> float sqrt(float) {…}              //*
template<> double sqrt(double) {…}
template<> long double sqrt(long double) {…}  //*
```

This attempt fails, because the explicit specializations in the two starred lines are illegal.

Attempt 2

```
float sqrt(float) {…}                   ①
double sqrt(double) {…}                 ②
long double sqrt(long double) {…}       ③
template<typename T>                    ④
double sqrt(T x)
{
   return sqrt((double)x);
}
```



- Example (Cont'd)

  This attempt admits all four calls to **`sqrt`**.
  The 1st call invokes ② directly.
  The remaining three calls invoke an instance of ④, which in turn invokes ②.

  Solution

```
template<typename T>          // for non-integral types
struct is_integer
{};                           // no typedef for type

template<>
struct is_integer<int>        // one for each integral type
{
   typedef double type;
};

float sqrt(float) {…}
double sqrt(double) {…}
long double sqrt(long double) {…}

template<typename T>
typename is_integer<T>::type sqrt(T n)
{
   return sqrt((double)n);
}
```

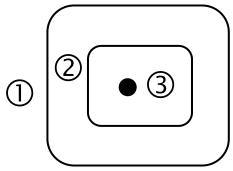
  Case 1: **`T = int`**
  `is_integer<int>::type` is defined to be **`double`**.
  Thus, the 3rd and 4th calls correctly compile, as in Attempt 2.

  Case 2: **`T = double`**
  `is_integer<dound>::type` is undefined.
  Thus, the 2nd call **`sqrt<double>(9.0)`** can't compile, as it causes a template argument deduction/substitution failure.

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".

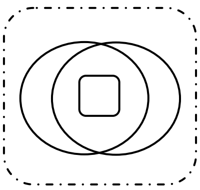
Partial ordering of class template partial specializations

- 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 – 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; }
```

Since enumerators are constants whose values are computed at compile time, `f<3>::v` causes the primary class template to be recurisvely 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.

- Example (Cont'd)

  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 expression
```

  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.

- Example (Cont'd)

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.

Again, this function template cannot be used to compute the factorial of an integral variable:

```
int n=3;
cout << f<n>();          // error – not a constant expression
```

In order to compute factorials at compile and run time, we need a metafunction for compile-time computation and an ordinary function for run-time computation.

C++11 offers a better solution.

```
constexpr int f(int n)      // constexpr function
{
   return n==0? 1: n*f(n-1);
}
```

Case 1: Run-time computation

```
int n=2;
cout << f(n);            // not a constant expression
```

Case 2: Compile-time computation

```
const int n=2;
int a[f(n)];             // a constant expression
```

Comment

Many C++ compilers, e.g. GNU C++, Clang C++, support semi-dynamic arrays. Under these compilers, it is not at all clear whether array **`a`** is a static or semidynamic array.

To be sure that **`f(n)`** is indeed a constant expression, test this instead:

```
template<int n> void p() { cout << n; }
p<f(n)>();
```

constexpr (generalized constant expression)

- **`const`** means unmodifiable.
  However, a **`const`** variable is also a constant expression if it is initialized from a constant expression.

- **`constexpr`** means constant expression. (since C++11)
  `constexpr` implies **`const`**.

- Example (Cont'd)

```
int n=2;
const int x=n;           // non-constant expression
const int y=2+3;         // constant expression
const int z=f(y);        // constant expression
constexpr int a=n;       // error
constexpr int b=2+3;     // constant expression
constexpr int c=f(b);    // constant expression
```

`constexpr` variable

- A **`constexpr`** variable must satisfy the following constraints:
  1 it shall have literal type.
  2 it shall be initialized with a constant expression.

- **`constexpr`** variables are implicitly **`const`**.

- Example

```
void p(const int);           // ok
void p(constexpr int);       // error; not for parameter
```

- Example

```
constexpr int a[3]={1,2,3};  // const int[3]
int* b=a;                    // error
const int* b=a;              // ok
constexpr int* b=a;          // error; int*const
constexpr const int* b=a;    // ok; int const*const
```

- Example

```
int a[3]={1,2,3};
int main()
{
    constexpr int* p=a;      // constant expression
    int*const q=a;           // constant expression
    int b[3]={1,2,3};
    constexpr int* r=b;      // error
    int*const s=b;           // non-constant expression
}
```

A pointer is a constant expression iff it points to an object in the global/static data area.

Digression

- A scalar type is an arithmetic type, an enumeration type, a pointer type, or the **`nullptr_t`** type.

- A literal type is
  1. a scalar type
  2. an array of literal type, or
  3. a class whose data members are of literal type (among other conditions. See later for an example).

`constexpr` function

- A **`constexpr`** function must satisfy the following constraints:
  1. its return type and the type of each parameter shall be a (reference to) literal type.
  2. its body shall contain exactly one **`return`** statement: **`return`** *expression*;
  3. In addition, its body may contain null statements, **`typedef`** and **`using`** declarations that generate no actions at runtime

- If no actual parameters exist such that the function invocation substitution (i.e. substituting actuals for formals) would produce a constant expression, the function is ill-formed.

- **`constexpr`** functions are implicitly **`inline`**.

- Example

```
constexpr int sq(int x) { return x*x; }
int n=3;
cout << sq(n);       ⇒ int x=n; cout << x*x;
constexpr int n=3;
const int n=3;
cout << sq(n);       ⇒ cout << 9;
```

- Example

```
constexpr int sum(const int* a,int n)
{
   return n==1? a[0]: a[0]+sum(a+1,n-1);
}
constexpr int a[5]={1,2,3,4,5};
int main()
{
   static constexpr int b[5]={1,2,3,4,5};
   constexpr int c[5]={1,2,3,4,5};
   static const int d[5]={1,2,3,4,5};
   p<sum(a,5)>();     // ok
   p<sum(b,5)>();     // ok
   p<sum(c,5)>();     // error, not in global/static data area
   p<sum(d,5)>();     // error, unmodifiable only
   int x=sum(c,5);    // most compilers ignore inline
}
```

- Example

```
constexpr int quad(int x)
{
//  return sq(x)*sq(x);         // inefficient
//  constexpr int y=sq(x);  // ill-formed ∵ declaration
//  return y*y;
    return sq(sq(x));         // better
}
```

- Example (Cont'd)

```
constexpr int inc(int x)
{
   return x+1;              // ok
// return x++,x;            // ill-formed ∵ assignment
}
```

Due to the assignment, no function invocation substitution will produce a constant expression. (∵ **x++** is erroneous, if **x** is replaced by a constant expression.)

constexpr constructor

- A **constexpr** constructor must satisfy the following conditions:
  1. the type of each parameter shall be a (reference to) literal type
  2. its body shall be empty
  3. other conditions omitted at this moment

- **constexpr** constructors are implicitly **inline**.

- Example

```
class rectangle {
public:
   constexpr rectangle(int l,int w)       // 1
   : l(l),w(w) {}
   constexpr int area() { return l*w; }   // 2
private:
   int l,w;
};

constexpr rectangle r(2,3);               // 3
p<r.area()>();
const rectangle s(2,3);         // unmodifiable only
p<s.area()>();                  // error
cout << s.area();               // ok
```

- Example (Cont'd)

  Q: Why is the ctor in line 1 declared **`constexpr`**?
  A: Were it not, the class would not be a literal type and the code would be ill-formed, because
  1 If a class isn't a literal type, it can't contain **`constexpr`** member functions. Thus, line 2 is erroneous.
  2 If a class isn't a literal type, it can't be used to initialize **`constexpr`** objects. Thus, line 3 is erroneous.

  Q: Why is the object in line 3 declared **`constexpr`**?
  A: Were it not, **`r.area()`** would not be evaluated at compile time.

  Q: Why is the member function in line 2 declared **`constexpr`**?
  A: Were it not, **`r.area()`** would be ill-formed, since a **`const`** object can't invoke a non-**`const`** member function. (See the next point)

- **`constexpr`** member functions are implicitly **`const`**.

  Example

```
constexpr int rectangle::area() { return l*w; }
int rectangle::area() { return l*w; }
```

  These two member functions can be overloaded, because the former is implicitly **`const`**, i.e. as if

```
int rectangle::area() const { return l*w; }
```

  which is different from the latter.

  On the other hand, the following two non-member functions

```
constexpr int sq(int x) { return x*x; }
int sq(int x) { return x*x; }    // redefine sq
```

  can't be overloaded, because they have the same function type **`int(int)`**.

Reference type

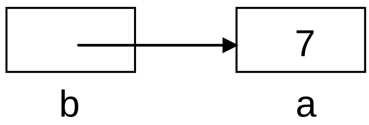
- References are implicit pointers.

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

cf.

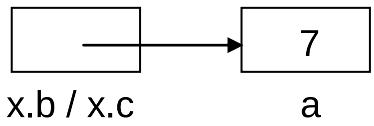
```
int a=7;
int* b=&a;
cout << *b;
```

Both give rise to this diagram:



Example – View the implicit pointer

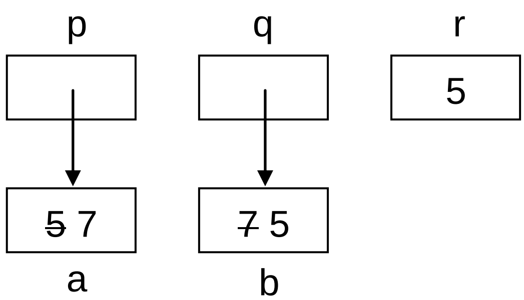
```
int a=7;
union X {
   int& b;
   int* c;
};
int main()
{
   X x={a};
   cout << x.b;          // 7
   cout << x.c;          // address of variable a
}
```



- References as parameters

Example – call by reference

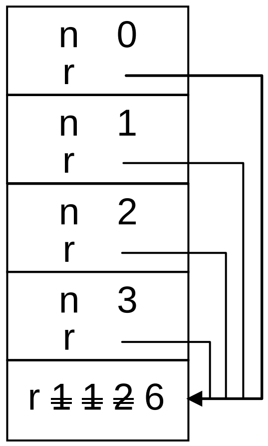
```
void swap(int& p,int& q)   // p,q: inout
{
   int r=p; p=q; q=r;
}
int main()
{
   int a=5,b=7;
   swap(a,b);
   cout << a << b;
}
```



- 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)**

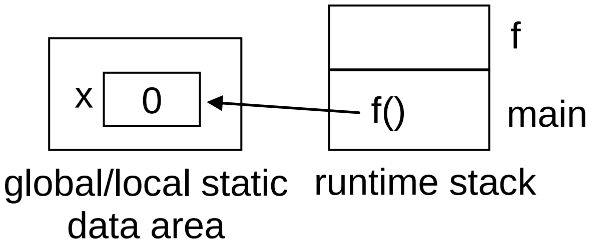
```
struct foo { double x[1000],y[1000]; } bar;
constexpr double sq(double x) { return x*x; }
double distance(const foo& p)   // p: in, large object
{
   double d=0.0;
   for (int i=0;i<1000;i++) {
      double e=sqrt(sq(p.x[i])+sq(p.y[i]));
      if (d<e) d=e;
   }
   return d;
}
int main() { cout 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

Remark

In STL (Standard Template Library), an in-functionality template type parameter `T` is almost 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 non-static 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)++;          // ok
(*const_cast<int*>(&a))++;      // ok
const_cast<int*>(&a)++;         // no
```

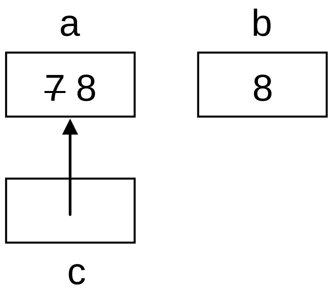
Reference vs Pointer

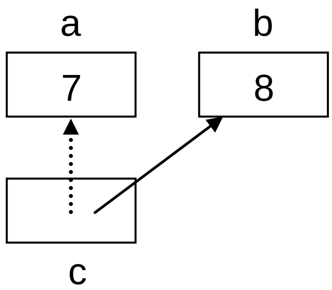
- Reference variables must be initialized, and are always const.

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

cf

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

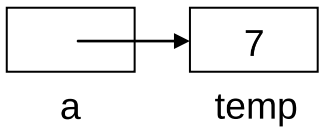
cf

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

×

This is compiled to something like

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

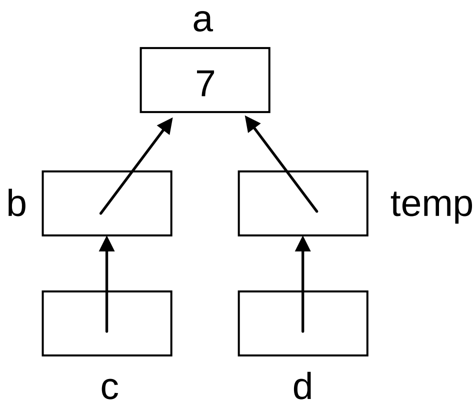


Motivation: Like call- and return-by-value, the actual parameter or return value may be an lvalue or rvalue.

```
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
```



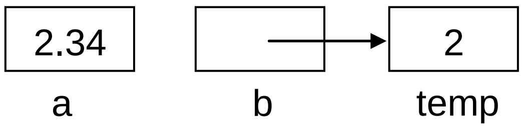
① **b** is an lvalue.

- **&a** is an rvalue.
  It is compiled to:

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

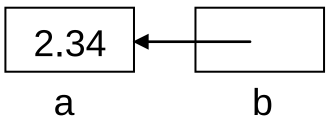
- To facilitate call- and return-by-reference, references may refer to const objects (treated as rvalues) of different types, provided that the required conversions exist.

```
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&*`**

Lvalue reference and rvalue reference

- Syntax and semantics

  cv **`T&`** lvalue reference
  cv **`T&&`** rvalue reference (C++11)

  Except where explicitly noted, they are semantically equivalent (e.g. they must be initialized, the binding can't be altered, etc) and commonly referred to as references.

- Rvalue references are useful for resource stealing and move semantics.

- Example

```
int x=2;
int& y=x;           // int& y=2;     ✗
const int& z=2;
int&& z=2;          // int&& z=x;    ✗
```

Reference to array

- Example – One-dimensional array

  Version A – 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*,int);
```

  Version B – 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;
}
print(a);           // no array-to-pointer conversion
print(b);
```

  These two calls generate two distinct instances:

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

- Example (Cont'd)

Version A1 – Pointer to array // **void print(T* a,int sz)**

```
template<typename T>      // remove * from version A
void print(T a,int sz)
{
   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*,int);

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);               // version A
print<int*>(a,2);         // version 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.

  In fact, the extra template parameter **U** isn't really needed. It may be obtained by **iterator_traits**.

```
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)

  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<typename Iterator>
struct iterator_traits {
    // five typedef members, including value_type
};
// Partial specialization for pointer types
template<typename T>
struct iterator_traits<T*> {
    typedef T value_type;
    // four more typedef members
};
```

Version B1 – Pointer to array // **`void print(T(&a)[sz])`**
// remove **`&`** from version B

```
template<typename T,int sz>
void print(T a[sz])      // or, 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 (Two-dimensional array)

  Version A – 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:

```
void print<int,3>(int (*)[3],int);
```

  Version B – 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);
print(b);
```

  These two calls yield two instances:

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

- Example (Cont'd)

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 code

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

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]);  // int[3]
   cout << endl;
}

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

void print(int& a)
{
    cout << a;
}
```

- Example (Cont'd)

  Comments

  1
```
template<typename T> void print(T&); ①
template<typename T,int sz>
void print(T(&)[sz]);
```

  Both can be instantiated for k-dimensional arrays, $k \geq 1$. Template ① can also be instantiated for non-arrays

  Unlike previous template metaprograms, the more general template ① is used to terminate the recursion.

  2 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];
```

  3 What this template metaprogram does can't be done by **constexpr**.

  4 What this template metaprogram does can't be done by pointers to arrays.

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

Type specifier

auto type specifier

- **`auto`** is no longer is storage class specifier, e.g.

```
void p()
{
    auto int x;        // error in C++11
    static int y;
}
```

- **`auto`** is now a type specifier, signifying that
  1. the type of a variable being declared shall be deduced from its initializer using template argument deduction, or
  2. a function declarator shall include a *trailing-return-type*.

Type inference

- **`auto x=`***exp***`;`**
  The type and value of *exp* are used to determine the type and initial value of variable **`x`**.

- **`auto`** ignores top-level **`const`**s, as normal initialization does.

- Example

```
const int a=2;
int b=a;              // ignore top-level const
auto b=a;             // int

int a=2;
const int b=a;        // ignore top-level const
const auto b=a;       // const int

const int a=2;
const int& b=a;       // keep low-level const
auto& b=a;            // const int&
auto c=b;             // int   dereferencing, as usual
```

- Example (Cont'd)

```
const int* a=nullptr;
auto b=a;              // const int*
const auto z=c;        // const int*const
```

Trailing return type

- Trailing-return-types are convenient when the return type of a function is complex.
- Example

```
auto sq(int x) -> int { return x*x; }
auto p() -> void {}
```

- Example

```
int a[7]={1,2,3,4,5,6,7};
int (&f())[7] { return a; }
int (*g())[7] { return &a; }
auto f() -> int(&)[7] { return a; }
auto g() -> int(*)[7] { return &a; }
cout << f()[0];
cout << (*g())[0];
```

decltype type specifier

- **decltype(***exp***) x;**
  The type of *exp* is used to determine the type of variable **x**. The *exp* isn't evaluated.
- Since there is no initialization, the top-level **const** isn't ignored.
- Example

```
const int a=2;
const int& b=a;

decltype(a) x=a;   // const int
decltype(b) y=a;   // const int&    no dereferencing
```

- Let **`T`** be the type of *exp*, then

  1 if *exp* is an unparenthesized identifier, **`decltype(`***exp***`) = T`**
  2 otherwise, if *exp* is a function call, **`decltype(`***exp***`) = T`**
  3 otherwise, if *exp* is an lvalue, **`decltype(`***exp***`) = T&`**
  4 otherwise, **`decltype(`***exp***`) = T`**

- Example

```
int a=5;
int* p=&a;
int& f() { return a; }

decltype (p) x;         // int*    1
decltype (*p) y=a;      // int&    3
decltype ((p)) z=p;     // int*&   3
decltype ((*p)) r=a;    // int&    3

decltype (f()) s=a;     // int&    2
decltype ((f())) u=a;   // int&    2
decltype (f()+0) v;     // int     4   dereferencing
```

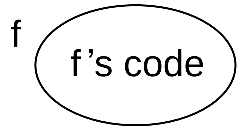
Function type

- 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

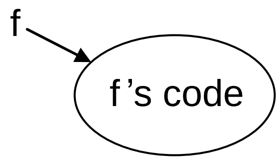



```
int f(int);
```

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.

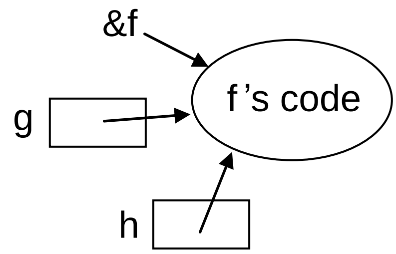
- Functions as functions




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

- 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 – Pointer to function

```
                                             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 – Reference to function

```
template<typename T>
T accumulate(T* first,T* last,T init,
                     T (&f)(const T&,const T&))
{
   same as Version A
}
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>)
```

Comments

1 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>)` // version A

2 Since function-to-pointer conversion doesn't lose any information on the function type, pointers to functions are most often used, following the C tradition.

Next, consider STL function templates **max** and **min**, defined in **<algorithm>** and **<iostream>**:

```
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;
}
```

Q: Can **max** and **min** return **T** objects?
A: They can. But returning by value is inefficient.

- Functions as parameters (Cont'd)

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;
```

Now that these functions have two different function types, we shall generalize the type of the function parameter, rather than provide two overloaded **accumulate**'s, one for each type.

Version C

```
template<typename T,typename Bfn>
T accumulate(T* first,T* last,T init,Bfn f)
{
   T result=init;
   for (T* it=first;it!=last;++it)
      result=f(result,*it);
   return result;
}
int a[7]={1,2,3,4,5,6,7};
accumulate(a,a+7,0,PLUS<int>)          ①
accumulate(a,a+7,INT_MIN,max<int>)     ②
```

Function-to-pointer conversions apply.
① **Bfn = int(*)(const int&,const int&)**
② **Bfn = const int&(*)(const int&,const int&)**

In fact, **Bfn** could be any type that supports the binary function call operator:

```
f(result,*it)
```

In other words, **f** could be a function or a function object.

- Functions as parameters (Cont'd)

On function objects (or functors)

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
{
   plus() {}            // default constructor
   ~plus() {}           // default destructor
   T operator()(const T& x,const T& y) const
   {
      return x+y;
   }
};
```

```
plus<int> a;            // call the default constructor
a.operator()(2,3) ≡ a(2,3)
plus<int>()(2,3)
```

call the default constructor to create an anonymous object

Comments

1 **sizeof(plus<int>)** = 1

2 In this case, the default constructor and destructor needn't be defined by the user, as they will be generated by the compiler automatically.

- 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)();
using pf=int(*)();        // C++11, type alias declaration
pf f[3]={c<0>,c<1>,c<2>};
```

  Alternatively, use `decltype`

```
decltype(c<0>)* f[3]={c<0>,c<1>,c<2>};
decltype(&c<0>) 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
```
or
```
auto mkmsg() -> void(*)() { return msg; }
int main() {  mkmsg()(); (*mkmsg())(); }
```

- Example – Function composition

Hypothetical code

```
int (*c(int (*f)(int),int (*g)(int)))(int)
{
   int h(int x) { return f(g(x)); }
   return h;
}
```
or
```
auto c(int f(int),int g(int)) -> int(*)(int)
{
   int h(int x) { return f(g(x)); }
   return h;
}
```

Lambda expression

- A lambda expression denotes an anonymous function.
- Basic syntax

  [*capture***] (***parameters***) ->** *return-type* **{** *body* **}**

  Example

  ```
  [](int x) -> int { return x*x; }
  ```

  In case the trailing return type is omitted, the type of **x*x** is the return type.

- A lambda expression creates a function object of a unique class type, called the *closure type*, that supports **operator()**.
- Example

  ```
  int main()
  {
      cout << [](int x){ return x*x; }(3);
      cout << [](int x){ return x*x; }.operator()(3);
  }
  ```

- The name of the closure type of each lambda expression is uniquely generated by the compiler.
  E.g. the two lambda expressions in preceding example are of distinct type.
- To give a lambda expression a name, the name of its closure type must be known. To this end, we may resort to **auto**, **decltype**, template argument deduction, etc.

- Example

```
int main()
{
   auto f=[](int x){ return x*x; }
   decltype(f) g=f;
   cout << f(3) << g(3);
   int a[7]={1,2,3,4,5,6,7};
   cout << accumulate(a,a+7,0,
                  [](int x,int y){ return x+y; });
}
```

Use template argument deduction to deduce its type.

- A lambda expression with free variables is meaningless.
For example, what is the meaning of this lambda expression?

```
[](int x){ return x+y; }
```

y is a free variable

- Free variables must be captured by value (copy) or reference.
- Example

```
int main()        may be omitted
{
   int x=2,y=3;
   auto f = [x,y](){ return x+y; };     // value
   auto g = [&x,&y](){ return x+y; }; // reference
   x=4; y=5;
   cout << f() << g(); // 59
}
```

Comments

```
[=]{ return x+y; };      // default capture by value
[&]{ return x+y; };      // default capture by reference
```

Both capture **x** by value and **y** by reference:

```
[=,&y]{ return x+y; }
[&,x]{ return x+y; }
```

- Example

```
#include <algorithm>        // for for_each
int main()
{
   int a[7]={1,2,3,4,5,6,7};
   int sum=0;
   for_each(a,a+7,[&sum](int x)->void{ sum+=x; });
   cout << sum;
}                                 // may be omitted
```

Note that the call to `for_each` essentially executes the loop:

```
for (int* it=a;it!=a+7;++it)
   [&sum](int x){ sum+=x; }(*it);
```

- Example

```
int main()
{
   int x=2,y=3;
   auto f = [x,&y]{ return x+y; };
   x=4; y=5;
   cout << f();
}
```

It is compiled to something like

```
int main()
{
   int x=2,y=3;
   class I_have_no_name {
   public:
      I_have_no_name(int a,int& b) : x(a),y(b) {}
      int operator()() const { return x+y; }
   private:
      int x,&y;
   };
   auto f = I_have_no_name(x,y);
   x=4; y=5;
   cout << f();
}
```

Polymorphic function wrapper

- The **`function`** class template provides polymorphic wrappers that encapsulate arbitrary callable objects.

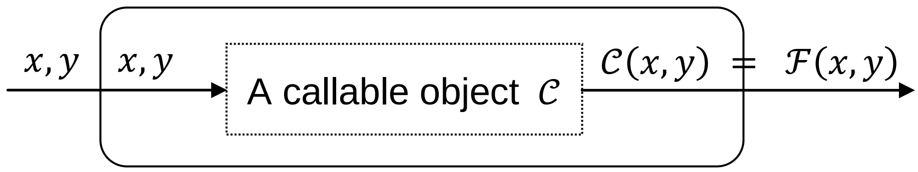
- Example

  The type
  `std::function<int(int,int)>`
  encapsulates all callable objects that have the call signature **`int(int,int)`**.

```
#include <functional>
int add(int x,int y) { return x+y; }
int main()
{
   typedef function<int(int,int)> bfn;
   bfn f = [](int x,int y){ return x+y; };
   bfn g[2] = {plus<int>(),add};
   cout << f(2,3) << g[0](2,3) << g[1](2,3);
}
```

  Comment

  A **`bfn`** object holds a callable object and supports a call operation that forwards to that object.



`bfn` object $\mathcal{F}$

```
int bfn::operator()(int x,int y) const
{
   return 𝒞(x,y);          // ℱ forwards x and y to 𝒞
}
```

  Notice that , for $\mathcal{C} =$ **`plus<int>()`**, $\mathcal{F}$ forwards **`x`** and **`y`** to $\mathcal{C}$ by reference. (This is OK.)
  For the other two cases, $\mathcal{F}$ forwards **`x`** and **`y`** to $\mathcal{C}$ by value.

- Example – Function composition

// Version A

```
function<int(int)> c(int f(int),int g(int))
{
   return [f,g](int x){ return f(g(x)); };
}
int f(int x) { return x+x; }
int g(int x) { return x*x; }
int main()
{
   cout << c(f,g)(3) << endl;
}
```

// Version B – File comp.cpp

```
#include <iostream>
#include <functional>
using namespace std;
using ufn=function<int(int)>;
ufn c(ufn f,ufn g)
{
   return [f,g](int x){ return f(g(x)); };
}
int main()
{
   cout << c([](int x){ return x+x; },
                   [](int x){ return x*x; })(3);
   cout << endl;
}
```

Note: Use GNU C++ compiler to compile the file comp.cpp.

```
bsd2> g++47 -std=c++11
          -rpath=/usr/local/lib/gcc47 comp.cpp
bsd2> ./a.out
18
```

for GLIBCXX_3.4.14

List-initialization

- List-initialization is the initialization of an object from a braced initializer list.
- Narrowing conversions are not allowed in list- initializations.
- Example

```
int a[3]={1,2,3};          // ok, as usual
int b[3]={1,2,3.4};        // error in C++11, narrowing
int c[3]{1,2,3};           // new in C++11
int d[3]{};                // 000
int e[3]{1,2,3,4};         // error, as usual

struct X { int x,y; };
X a={1,2};                 // ok, as usual
X b{1,2};                  // new in C++11
```

- Example – C++11 extension on list-initialization

```
// variable
int a=3.4;                 // ok, as usual
int b(3.4);                // ok, as usual
int c={3.4};               // new in C++11, error
int d{3.4};                // new in C++11, error
int e{};                   // 0

// assignment
int a=0;
a=3.4;                     // ok, as usual
a={3.4};                   // new in C++11, error

// return
X f() { X a={1,2}; return a; }
X f() { return {1,2}; }    // new in C++11
```

initializer_list

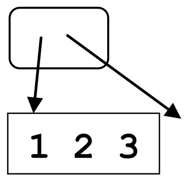
- Below is the abridged class template **initializer_list**:

```
template<class T>
class initializer_list {
public:
   initializer_list(braced-initializer-list);
   const T* begin() const { return _begin; }
   const T* end() const { return _end; }
private:
   const T* _begin;      // first element
   const T* _end;        // one past the last element
};

template<class T>
const T* begin(initializer_list<T> a)
{
   return a.begin();
}

template<class T>
const T* end(initializer_list<T> a)
{
   return a.end();
}
```

- A braced initializer list may be used to construct an **initializer_list** object.

- Example

object **a**




```
#include <initializer_list>
initializer_list<int> a={1,2,3};
initializer_list<int> a{1,2,3};

for (const int* it=begin(a);it!=end(a);++it)
   cout << *it;
```

Range-based `for` statement

- Syntax

  for **(** *for-range-declaration* **:** *expression* **)** *statement*
  for **(** *for-range-declaration* **:** *braced-init-list* **)** *statement*

- Example

```
initializer_list<int> a{1,2,3};
for (int x : {1,2,3}) cout << x;
for (int x : a) cout << x;
for (const int& x : a) cout << x;
```

  The last one must be **const** qualified and is compiled to

```
auto& c=a;
for (auto it=begin(c),finish=end(c);
                            it!=finish;++it) {
   const int& x=*it;
   cout << x;
}
```

- Example

```
int a[3]{1,2,3};
for (int& x : a) x++;
```

  The compiled code is similar, except that it uses the predefined **begin** and **end** specialized for arrays:

```
template<typename T,size_t N>
T* begin(T (&array)[N])
{
   return array;
}

template<typename T,size_t N>
T* end(T (&array)[N])
{
   return array+N;
}
```

- Example

```
void print(int (&a)[5])
{
   for (int& x : a) cout << x;  // & is optional here
}
void print(int* a,int sz)
{
   for (int& x : reinterpret_cast<int(&)[sz]>(*a))
      cout << x;
}
int a[5]{1,2,3,4,5};
print(a);
print(a,5);
```

- Example

```
void print(int (&a)[2][3])
{
   for (int (&x)[3] : a) {    // & is necessary here
      for (int &y : x)        // & is optional here
         cout << y;
      cout << endl;
   }
}
void print(int (*a)[3],int sz1)
{
   for (int (&x)[3] :
        reinterpret_cast<int(&)[sz1][3]>(*a)) {
      for (int &y : x) cout << y;
      cout << endl;
   }
}
int a[2][3]{1,2,3,4,5,6};
print(a);
print(a,2);
```

Dynamic storage management

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`

1 call `operator new` to obtain a block of raw storage of size `sizeof(T)`
2 initialize the storage

```
new T                    // default-initialized
new T()                  // value-initialized
new T{}                  // value-initialized
new T(arguments)         // direct-initialized
new T{arguments}         // list-initialized
```

3 yield a `T*` pointer pointing to the object created

On initialization: A brief introduction

1 Default initialization
For class type, initialize by its default ctor
For non-class type, uninitialized
2 Value initialization
For class type, initialize by its default ctor
For non-class type, zero-initialized
3 Direct initialization
For class type, initialize by the best viable ctor
For non-class type, initialize with the given arguments
4 List initialization (C++11)
For class type, initialize by the best viable ctor or list ctor
For non-class type, initialize with the given arguments

- 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*`

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

- Example

```
int* p=new int;
operator delete(p);          // undefined
```

Principle – Always use the same form of new and delete

- Example

```
int a;                       // uninitialized, if local
int a();                     // function prototype
int a{};                     // 0
int a(7);                    // 7
int a{7};                    // 7

int* p=new int;              // uninitialized
int* p=new int();            // 0
int* p=new int{};            // 0
int* p=new int(7);           // 7
int* p=new int{7};           // 7
delete p;
```

- Example (Cont'd)

Comment

```
cout << int() << int{};        // 00
cout << int(7) << int{7};      // 77
```

Alternatively, we may use the pair of functions.

Method A: Assignment

```
int* p=static_cast<int*>(operator new(sizeof(int)));
*p=7;
operator delete(p);
```

Method B: Initialization by placement new

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

- Example

```
class X {
public:
   X(int n=0)                          // ctor + default ctor
   : x(new int(n)){}
   X(initializer_list<int> a)          // initializer-list ctor
   : x(new int(*begin(a))) {}
   ~X() { delete x; }
private:
   int* x;
};

X a;                 // default ctor
X a();               // function prototype
X a{};               // default ctor
X a(7);              // ctor
X a{7};              // initializer-list ctor

X a({7});            // ok, for class type
int a({7});          // no, for non-class type
```

- Example (Cont'd)

With the initializer-list ctor

$\mathbf{X\{}a_1, \ldots, a_n\mathbf{\}} = \mathbf{X(\{}a_1, \ldots, a_n\mathbf{\})}$

```
X a{7.8};                  // error, narrowing
X a{7,8};                  // initializer-list ctor; 8 isn't used
```

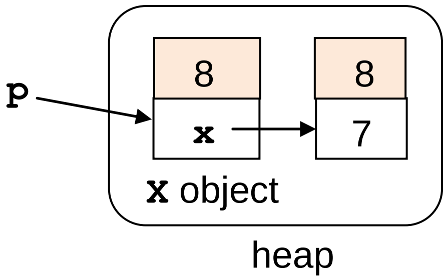
Without the initializer-list ctor

$\mathbf{X\{}a_1, \ldots, a_n\mathbf{\}} = \mathbf{X(\{}a_1, \ldots, a_n\mathbf{\})} \Rightarrow \mathbf{X(}a_1, \ldots, a_n\mathbf{)}$ + narrowing?

```
X a{7};                    // ctor
X a{7.8};                  // error, narrowing
X a{7,8};                  // error, no viable ctor

X* p=new X;                // default ctor
X* p=new X();              // default ctor
X* p=new X{};              // default ctor
X* p=new X(7);             // ctor
X* p=new X{7};             // initializer-list ctor
delete p;
```

The code in the last two lines may be rewritten as

```
X* p=new (operator new(sizeof(X))) X{7};
p->~X();
operator delete(p);
```



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.

- Allocators

new operator

`new T[`*n*`]`

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 initialized.

`new T[`*n*`]`	// default-initialized
`new T[`*n*`]()`	// value-initialized
`new T[`*n*`]{}`	// value-initialized
`new T[`*n*`](`*arguments***`)`**	// error
`new T[`*n*`]{`*arguments***`}`**	// direct- or list-initialized

3 It yields a **`T*`** pointer pointing to the 0th 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*`**

1 This is similar to single-object deallocation, except that it calls **`operator delete[]`** to free the storage.
2 For class type **`T`**, its destructor is first called as many times as there are objects in the array to destroy them, in the reverse order of their construction. The storage pointed to by **`p`** is then freed.
3 The pointer **`p`** must be obtained by an earlier array-object **`new`** expression.

- Example

```
int a[3];                   // uninitialized, if local
int a[3]();                 // error, array of functions
int a[3]{};                 // 000
int a[3](7,8,9);            // error, bad array initializer
int a[3]{7,8,9};            // 789

int* p=new int[3];          // uninitialized
int* p=new int[3]();        // 000
int* p=new int[3]{};        // 000
int* p=new int[3](7,8,9);   // error
int* p=new int[3]{7,8,9};   // 789
delete [] p;
```

The code in the last two lines may be rewritten as

```
int* p=new (operator new[](3*sizeof(int)))
           int[3]{7,8,9};
operator delete(p);
```

- Example

```
X a[3];                     // 3 default ctors
X a[3]();                   // error, array of functions
X a[3]{};                   // 3 default ctors
X a[3](7,8,9);              // error, bad array initializer
X a[3]{7,8,9};              // 3 ctors
X a[3]{{7},{8},{9}};        // 3 initializer-list ctors
X a[3]{{7},8,9};            // initializer-list ctor, ctor, ctor

X* p=new X[3];              // uninitialized
X* p=new X[3]();            // 3 default ctors
X* p=new X[3]{};            // 3 default ctors
X* p=new X[3](7,8,9);       // error
X* p=new X[3]{7,8,9};       // 3 ctors
X* p=new X[3]{{7},{8},{9}};    // 3 initializer-list ctors
X* p=new X[3]{{7},8,9};// initializer-list ctor, ctor, ctor
delete [] p;
```

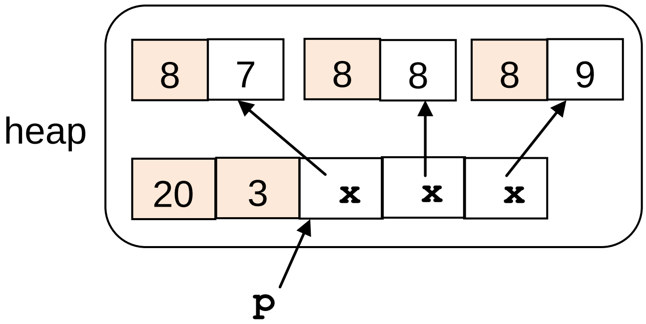
- Example (Cont'd)

  Consider

```
X* p=new X[3]{7,8,9};
delete [] p;
```

  The code in these two lines is in effect equivalent to

```
X* p=(X*)operator new[](3*sizeof(X));
for (int i=0;i<3;i++) new (p+i) X(7+i);
for (int i=2;i>=0;i--) p[i].~X();
operator delete[](p);
```



- Example – Non-constant array size

```
int n;
X* p=new X[n]{};          // n default ctors
X* p=new X[n]{7,8,9};     // undefined if n < 3
```

  The code in the last line may cause a runtime error if **n** < 3. It has to be written as

```
X* p=(X*)operator new[](n*sizeof(X));
for (int i=0;i<n;i++) new (p+i) X(7+n);
for (int i=n-1;i>=0;i--) p[i].~X();
operator delete[](p);
```

- Example – Dynamically-allocated two-dimensional arrays

```
int (*a)[3]=new int[2][3]{{1,2,3},{4,5,6}};
int (*a)[3]=new int[2][3]{1,2,3,4,5,6};   ✘
```

  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.

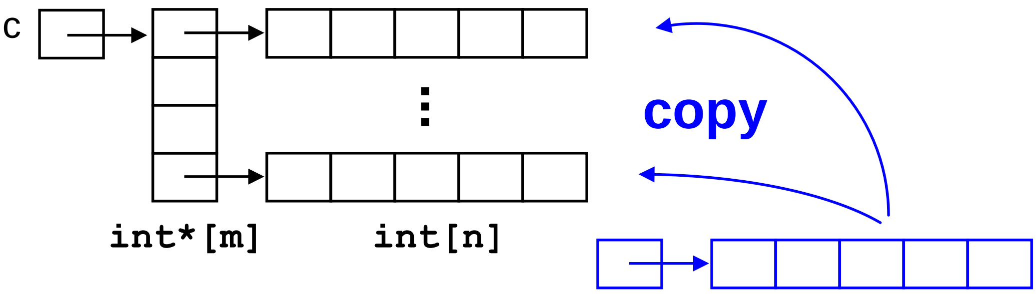
- Example (Cont'd)

```
int m=2,n=3;
int (*a)[3]=new int[m][3]int      // ok
int (*a)[n]=new int[2][n];        // no
int (*a)[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:

```
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.

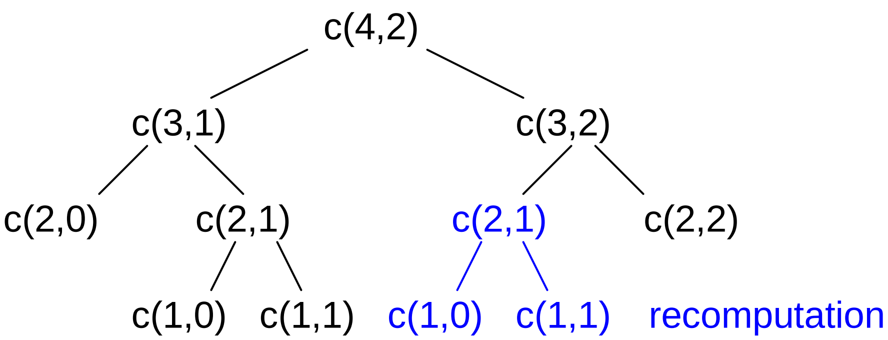
- Example (Cont'd)

  Pro – easy to code

  Con – time and space inefficient
  The ctor call
  `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)

$$c(m,n) = 1 \qquad\qquad n = 0 \text{ or } m = n$$
$$\quad = c(m-1,n-1) + c(m-1,n) \quad \text{otherwise}$$



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.

- Example (Cont'd)

Version A1

```
int c(int m,int n,int** cache)
{
   if (cache[m-n][n]==0)
      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];
}

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]{};  // zero-initialized
   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);
}
```

- Example (Cont'd)

Version A3 – Anonymous vector; const lvalue reference

```
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];
}

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

Version A4 – Anonymous vector; rvalue reference

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

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

- Example (Cont'd)

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;
}
```

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];
}
```

Note: The starred line unnecessarily initializes the entire vector.

- Example (Cont’d)

  Version B3 – Keep one row or one column, whichever is smaller

```
int c(int m,int n)
{
   if (n>m-n) n=m-n;          // row size > column size
   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 or one column, Vector

```
int c(int m,int n)
{
   if (n>m-n) n=m-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.

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

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

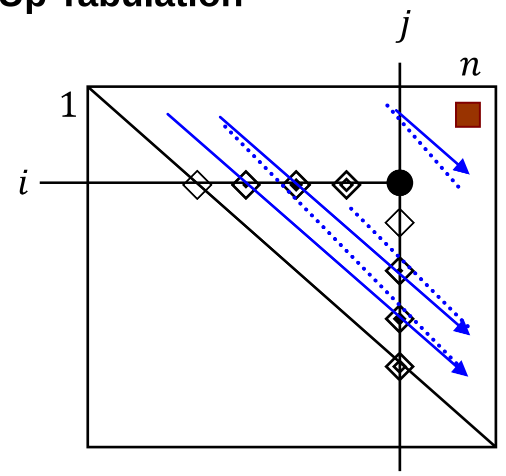
$\vdots$

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

2 Use tabulation to avoid recomputation.

- Example (Cont'd)

Bottom-Up Tabulation



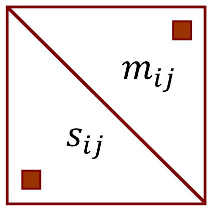
$m_{ij}, 1 \le i \le j \le n$

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.

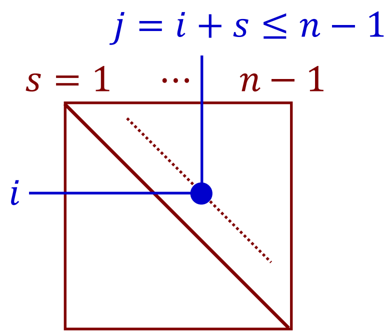



```
int mcm(int* d,int n);

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 $d_{-1}d_0d_1 \cdots d_{n-1}$.

- 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;
}
```

N.B. 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
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
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);
```

Smart pointer

- Smart pointers are pointer-like objects that are equipped with additional features such as automatically deletion of the pointee

  Comment – The ordinary pointers are "raw pointers".

```
int main()
{
   int *p=new int{7};                  // raw pointer
   if (p!=nullptr) cout << *p;
   if (p) cout << *p;
   cout << p;
}                   // the pointee is not reclaimed automatically
```

- There are several kinds of smart pointers.
  1 **auto_ptr** deprecated
  2 **shared_ptr** shared ownership semantics
  3 **unique_ptr** strict ownership semantics

shared_ptr

- **shared_ptr** implements shared ownership semantics based on reference counting.

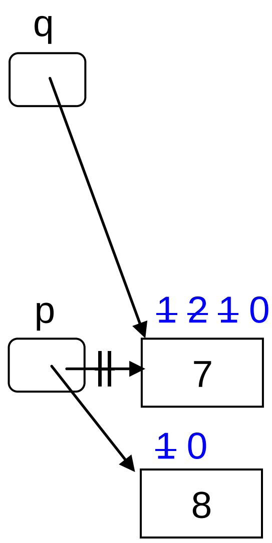
- Reference counting
  - count the number of owners of a pointee
  - increment (or decrement) the counter each time an owner is added (or removed)
  - reclaim the pointee when the count reaches 0.

- Example

```
#include <memory>
int main()
{
   shared_ptr<int> p(new int{7});   // smart pointer
   if (p!=nullptr) cout << *p;
   if (p) cout << *p;
}                       // the pointee is reclaimed automatically
```

- Example

```
void X(shared_ptr<int> q)
{
   cout << q.use_count();        // 2
}
int main()
{
   shared_ptr<int> p;            // empty
   cout << p.use_count();        // 0
   p=make_shared<int>(7);
   cout << p.use_count();        // 1
   X(p);
   cout << p.use_count();        // 1
   p=make_shared<int>(8);
   cout << p.use_count();        // 1
}
```



- Reference-counting-based shared pointers solves the memory leak (or garbage) and dangling pointer problems, but not the sharing problem.

- Example – memory leak

```
int* p=new int{7};
p=nullptr;                  // memory leak

shared_ptr<int> p(new int{7});
p=nullptr;                  // storage reclaimed
```

- Example – dangling pointer

```
int* p=new int{7};
int* q=p;
delete q;
cout << *p;                 // dangling pointer

shared_ptr<int> p(new int{7});
shared_ptr<int> q(p);
q.~shared_ptr<int>();
cout << *p;                 // storage still pointed to by p
```

- Example – sharing

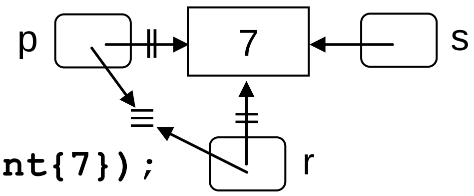
```
int* p=new int{7};
int* q=p;
*q=8;
cout << *p;      // sharing causes *p to be altered silentcly
```

```
shared_ptr<int> p(new int{7});
shared_ptr<int> q(p);
*q=9;
cout << *p;      // sharing causes *p to be altered silentcly
```

- `shared_ptr` doesn't support arrays directly. But, see below.

unique_ptr

- `unique_ptr` implements strict ownership semantics: a pointee has a unique owner.
- A `unique_ptr` object can't be copied or assigned. However, it can be moved or transfer its ownership.
- Example




```
int main()
{
   unique_ptr<int> p(new int{7});
   if (p!=nullptr) cout << *p;
   if (p) cout << *p;
   p=p;                                 // no, assign
   unique_ptr<int> q(p);                // no, copy
   unique_ptr<int> r(p.release());      // ok, transfer
   unique_ptr<int> s(std::move(r));     // ok, move
}
```

- Example – unique_ptr for array

```
int main()
{
   unique_ptr<int[]> p(new int[3]{1,2,3});
   cout << p[0] << p[1] << p[2];
}
```

- Example (Cont'd)

  Comments

  1 **unique_ptr<T[]>** is a specialization for **T[]**, rather than **T***.

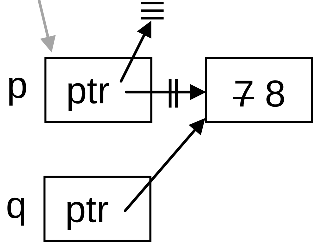
  2 **unique_ptr** uses **delete** as a default deleter and supports **operator***, but not **operator[]**.

  3 **unique_ptr<T[]>** uses **delete[]** as a default deleter, and supports **operator[]**, but not **operator***.

- Here is a stripped-down version of **unique_ptr**:

```
template<typename T>
class unique_ptr {
public:
// ctor
   constexpr unique_ptr() : ptr(nullptr) {}
   unique_ptr(T* p) : ptr(p) {}
// move ctor
   unique_ptr(unique_ptr&& rhs)
   : ptr(rhs.ptr) { rhs.ptr=nullptr; }
// copy ctor
   unique_ptr(const unique_ptr&)=delete;
// dtor
   ~unique_ptr() { delete ptr; }
// copy assignment operator
   unique_ptr& operator=(const unique_ptr&)=delete;
// observer
   T& operator*() const { if (ptr) return *ptr; }
   T* get() const { return ptr; }
// modifier
   T* release() { T* p=ptr; ptr=nullptr; return p; }
private:
   T* ptr;
};
```

```
unique_ptr<int> p(new int(7));
(*p)++ ≡ p.operator*()++;
unique_ptr<int> q(std::move(p));
```



- Here is a stripped-down version of **unique_ptr<T[]>**:

```
template<typename T>
class unique_ptr<T[]> {                          // 1
public:
// ctor
   constexpr unique_ptr() : ptr(nullptr) {}
   unique_ptr(T* p) : ptr(p) {}
// dtor
   ~unique_ptr() { delete [] ptr; }
// observer
   T& operator[](size_t n) const
   { if (ptr) return ptr[n]; }
// other members omitted
private:
   T* ptr;
};

unique_ptr<int[]> p(new int[3]{1,2,3});    // 2
p[0]++ ≡ p.operator[](0)++
```

Comment
T[] in line 1 is a reminder reminding that this specialization is for arrays. Although it makes no sense, **T[]** could be replaced by other legal type, e.g. **T***, **T(T)**.

A contrived example

```
template<typename T> struct X;
template<typename T>
struct X<T[]> {                          // T[] ≠ T*
   void p(T[]) { cout << "T[]"; }        // T[] = T*
};
template<typename T>
struct X<T*> {
   void p(T*) { cout << "T*"; }
};

int a[3]{1,2,3};
X<int[]> x; x.p(a);      // T[]
X<int*> y;  y.p(a);      // T*
```

deleter

- **default_delete** and **default_delete<T[]>** serve as the default deleters for **unique_ptr** and **unique_ptr<T[]>**, respectively.

```
template<class T>
struct default_delete {
   void operator()(T* p) const { delete p; }
};
```

```
template<class T>
struct default_delete<T[]> {
   void operator()(T* p) const { delete [] p; }
};
```

- Example

  unique_ptr<int> p(new int(7));
  deleter = **default_delete<int>()**

  unique_ptr<int[]> p(new int[3]{1,2,3});
  deleter = **default_delete<int[]>()**

- For unique pointers, one may specify the desired deleter and its type.

- Example

```
unique_ptr<int,void(*)(void*)>
             p((int*)malloc(sizeof(int)),free);
```

  Each type below can be used to replace the underlined type.

```
void(&)(void*)     ≡  decltype(free)&
decltype(free)*    ≡  decltype(&free)
function<void(void*)>
```

- Recall that **shared_ptr** supports only single objects and so the default deleter is the single-object **delete**.

  However, for shared pointers, one may also specify the desired deleter (but not its type).

- Example

```
shared_ptr<int> p((int*)malloc(sizeof(int)),free);
```

- Example – share_ptr for array

```
shared_ptr<int>
     a(new int[3]{1,2,3},default_delete<int[]>());
for (int i=0;i<3;i++)
   cout << a.get()[i];
```

where **get()** returns the stored pointer.
N.B. **shared_ptr** supports **operator***, but not **operator[]**

Final remark

- Recall the principle

  Define a dtor for classes with dynamically allocated memory

  By this we mean raw pointers are used to point to the allocated memory. In case smart pointers are used, there is no need to define a dtor by ourselves. (See next lecture for explanation.)

Deleted function

- Example

  Prior to C++11, to prevent a **`unique_ptr`** object from being copied and assigned, we would declare the copy ctor and copy assignment operator private and provide no definitions for them

```
template<typename T>
class unique_ptr {
public:
//  unique_ptr(const unique_ptr&)=delete;
//  unique_ptr& operator=(const unique_ptr&)=delete;
private:
   T* ptr;
   unique_ptr(const unique_ptr&);
   unique_ptr& operator=(const unique_ptr&);
};
```

  A member function copies or assigns ⇒ link-time error
  A non-member function copies or assigns ⇒ compile-time error

- The accessibility of a deleted member function is immaterial.
- A function must be deleted on its first declaration.

```
struct X { X(); };
X::X() = delete;            // error, not first declaration
int main() {}
```

- A deleted function also participates in overload resolution.
- Example

```
void p(int) {}              // 1
void p(double) = delete;    // 2, enforce int invocation

p(2);           // case 1: select 1
p('2');         // case 1
p(2.3);         // case 2: select 2, but error
p(2.3f);        // case 2
p(2u);          // case 3: ambiguous
```

- Example (Cont'd)

  Case 1: $T \rightarrow$ int identity or integral promotion
  Case 2: $T \rightarrow$ double identity or floating promotion
  Case 3: $T \rightarrow$ int $\wedge$ $T \rightarrow$ double both are conversions

- Comment
  Overload resolution has a higher priority than availability and accessibility, i.e.
  Overload resolution $\rightarrow$ Deleted? Accessible?

- Example (Cont'd)

```
class X {
public:
   void p(int) {}          // 1
private:
   void p(double) {}       // 2
};
X().p(2);       // case 1: select 1
X().p('2');     // case 1
X().p(2.3);     // case 2: select 2, but error
X().p(2.3f);    // case2
X().p(2u);      // case 3: ambiguous
```