# Pytania do egzaminu z przedmiotu “Metody i języki programowania”

1. Operatorem pobrania adresu jest:
2. Poinformowanie kompilatora, że typ i identyfikator obiektu są znane oraz przydzielenie obiektowi związanego z nim obszaru pamięci to
3. Rozmiar zmiennej typu `double` wynosi
4. Instrukcja wielowariantowego wyboru to
5. Lokalizacja wartości, adres obszaru pamięci przydzielonego obiektowi
6. Rozmiar zmiennej typu `int` wynosi
7. Wartość jaką odczytuje się z obszaru przydzielonego obiektowi to
8. Instrukcja warunkowa to
9. Rozmiar zmiennej typu `long` wynosi
10. Pojawiająca się w tekście programu stała dosłowna to
11. Literał zmiennopozycyjny domyślnie jest typu
12. Ustawienie obiektu jako niemodyfikowalnego, tak by mógł być używany jedynie jako p-wartość wymaga użycia modyfikatora
13. Rozmiar zmiennej typu `float` wynosi
14. Zmienna typu `signed char` może przyjmować wartości
15. Ustawienie obiektu lokalnego tak by pozostawał w pamięci przez cały czas wykonywania programu wymaga użycia modyfikatora
16. Literał całkowity domyślnie jest typu
17. Poinformowanie kompilatora, że typ i identyfikator obiektu są znane ale bez przydzielenia obiektowi związanego z nim obszaru pamięci to
18. Słowem kluczowym w języku C++ nie jest
19. Ochrona obiektu (wartość tego obiektu może być modyfikowana poza programem) przed zbyt intensywną optymalizacją kodu operującego na tym obiekcie wymaga użycia modyfikatora
20. Rozmiar zmiennej typu `long double` wynosi
21. Umieszczenie obiektu w specjalnym obszarze pamięci procesora, by skrócić czas dostępu do niego wymaga użycia modyfikatora
22. Tablice (wbudowane) w języku C++ są indeksowane od
23. Rozmiar zmiennej typu `short` wynosi
24. Określenie rozmiaru i rozmieszczenia danych w pamięci, określenie zakresu dopuszczalnych danych oraz określenie operacji jakie można wykonać na tych danych to
25. W wyniku posłużenia się dyrektywą preprocesora `#include <iostream>`
26. Ile razy wykona się poniższa pętla `for( , , );`
27. Ile razy wykona się poniższa pętla `for( ; ; );`
28. Operatorem wyłuskania jest:
29. `int *tab=new int[20](10);` Obok utworzono:
30. `int a, *wsk=&a;` Dostęp do obiektu wskazywanego przez wskaźnik `wsk` jest realizowany przez
31. Instrukcja `long *nnu(long, long);` to
32. Instrukcja `double *nnu(double, int);` to
33. Instrukcja `double *nun(double, long);` to
34. Instrukcja `void nnu(double, *int);` to
35. Instrukcja `int *unn(int, int);` to
36. Instrukcja `int (*unu)(int, int);` to
37. Poprawna definicja wskaźnika `b` to
38. Poprawna definicja wskaźnika `b` to
39. Poprawna definicja wskaźnika `b` to
40. Poprawna definicja wskaźnika `b` to
41. Poprawna definicja wskaźnika `b` to
42. Poprawna definicja wskaźnika `b` to
43. Poprawna definicja wskaźnika `b` to
44. Poprawna definicja wskaźnika `b` to
45. Poprawna definicja wskaźnika `b` to
46. Która z poniższych definicji tablicy trzech wskaźników do funkcji jest błędna
```
double tg(double x)
{ return sin(x)/cos(x); }
double ctg(double x)
{ return cos(x)/sin(x); }
double szesc(double x)
{ return x*x*x; }
```
47. Jeśli wskaźnik `wsk` wskazuje na obiekt typu strukturalnego
`struct Dane{ double a,b; };`
to dostęp do pola `a` tego obiektu jest realizowany poprzez
48. Która z poniższych definicji tablicy trzech wskaźników do funkcji jest błędna
```
double suma(double x, double y)
{ return x+y; }
double roznica(double x, double y)
{ return x-y; }
double dziel(double x, double y)
{ return x/y; }
```
49. Który fragment zawiera błędy
50. Poprawna definicja wskaźnika `b` to
51. Która z podanych niżej definicji jest niepoprawna
52. Poniżej zdefiniowano
`struct { double a, b; } A;`
53. Które z linii poniżej to definicje
A `extern float EPS;`
B `int max(int, int);`
C `double p;`
D `double bezwzgl(double x) { return x > 0 ? x : -x; }`
54. Które z linii poniżej to definicje
A `extern int n;`
B `double sinus(double);`
C `int licznik;`
D `double tangens(double x) { return sin(x)/cos(x);}`

55. Poniżej zdefiniowano
```
struct { double a, b; } A;
```
56. Który fragment programu zawiera błędy
57. Który fragment zawiera błędy
58. Który fragment zawiera błędy
59. Który fragment zawiera błędy
60. Który fragment zawiera błędy
61. Która z definicji jest błędna
62. Która z podanych niżej definicji jest poprawna
63. Który fragment zawiera błędy
64. Która z definicji jest błędna
65. Który z wymienionych niżej literałów jest niepoprawny
66. `int *i[10];` Obok zadeklarowano
67. Deklaracja tablicy dziesięciu wskaźników do liczb całkowitych to
68. `int (*i)[10];` Obok zadeklarowano
69. Deklaracja wskaźnika do tablicy dziesięciu liczb całkowitych to
70. `extern int (&i)[10];` Obok zadeklarowano
71. Deklaracja referencji do tablicy dziesięciu liczb całkowitych to
72. `int *(*i)[10];` Obok zadeklarowano
73. Deklaracja wskaźnika do tablicy dziesięciu wskaźników do liczb całkowitych to
74. `extern int* (&i)[10];` Obok zadeklarowano
75. Deklaracja referencji do tablicy dziesięciu wskaźników do liczb całkowitych to
76. `const double* eps;` Obok zadeklarowano
77. Deklaracja niestałego wskaźnika do stałej liczby zmiennopozycyjnej podwójnej precyzji to
78. `extern const double* const eps;` Obok zadeklarowano
79. Deklaracja stałego wskaźnika do stałej liczby zmiennopozycyjnej podwójnej precyzji to
80. `extern double* const eps;` Obok zadeklarowano
81. Deklaracja stałego wskaźnika do liczby zmiennopozycyjnej podwójnej precyzji to
82. `extern const double& eps;` Obok zadeklarowano
83. Deklaracja referencji do stałej liczby zmiennopozycyjnej podwójnej precyzji to
84. `extern double& const eps;` Obok zadeklarowano
85. Pamięć dla dynamicznej tablicy dziesięciu liczb całkowitych można przydzielić za pomocą
86. Jeśli `wsk` wskazuje na tablicę dynamiczną to zwolnienie pamięci dla tej tablicy realizuje się za pomocą
87. W linii `char* wsk = new char[100];`
88. Dostęp do obiektu wskazywanego przez wskaźnik `wsk` w celu przypisania mu wartości 10, jest możliwy poprzez
```
int b=4; int* wsk=&b;
```
89. W linii `int* wsk = new int(10);`
90. Co wyświetli poniższy fragment programu
```
int i = 0;
do {
  cout << i << ' ';
  i++;
} while (i < 5);
```
91. Jaki będzie wynik wywołania `lit('B')` i `lit('b')` dla poniższej funkcji
```
char lit(char c)
{ return 'A' <= c && c <= 'Z' ?
    c : 'A' + c - 'a'; }
```
92. Jaka będzie wartość zwrócona przez funkcję w wywołaniu `wiekszy("bcde", "azds")` dla funkcji zdefiniowanej poniżej
```
bool wiekszy(char c1, char c2)
{ return c1 > c2; }
int wiekszy(const char* s1, const char* s2)
{ for (int i = 0; i < strlen(s1) && i <
strlen(s2); i++)
  if (wiekszy(s1[i], s2[i]))
    return 1;
  return 0;
}
```
93. Jaka będzie wartość zmiennej `odl` po wykonaniu instrukcji
```
int a[4] = {4, 3, 2, 1};
int* p = &a[2];
int odl = p - a;
```
94. Co wyświetli poniższy fragment programu
```
int i = 0;
do {
  cout << i << ' ';
  i++;
} while (i > 5);
```
95. Jakie wartości zawierać będzie tablica `t` po wykonaniu instrukcji
```
int t[5] = {1, 2, 3, 4, 5};
int* p = t;
p = p+2;
*p = 10;
```
96. Co wyświetli poniższy fragment programu
```
int nr = 2;
switch (nr) {
  case 0: cout << 'a';
  case 1: cout << 'b';
  case 2: cout << 'c';
  case 3: cout << 'd';
  default: cout << 'e';
}
```
97. Jaki będzie wynik wywołania `lit('B')` i `lit('b')` dla poniższej funkcji
```
char lit(char c)
{
  return 'a' <= c && c <= 'z' ?
     c : 'a' + c - 'A';
}
```
98. Wartością zmiennej `x` po wykonaniu następującego kodu
```
double a=4, b=2; double x=2*(a+b)/3;
```
będzie

99. Korzystając z poniższych struktur opisujących ogród zoologiczny
```
struct Data
{ int d, m, rok; };
struct Zwierze
{ char gatunek[30]; char imie[30];
int klatka; Data* urodz; };
struct OgrodZool
{ int l_zwierzat; Zwierze zwierzeta[50]; };
OgrodZool* zoo = new OgrodZool;
```
jak można odwołać się do roku urodzenia pierwszego zwierzęcia

100. Jaki będzie wynik działania programu
```
int fun()
{
  static int a = 12;
  double x = 10;
  cout << a << ' ' << x << ',';
  a++;
  x *= 10;
}
int main()
{ fun(); fun(); fun(); return 0; }
```

101. Jaka będzie wartość zmiennej `odl` po wykonaniu instrukcji
```
double b[5] = {11, 22, 33, 44};
double* p = &b[3]; int odl = p - b;
```

102. Korzystając z poniższych struktur opisujących ogród zoologiczny
```
struct Data
{ int d, m, rok; };
struct Zwierze
{ char gatunek[30]; char imie[30];
  int klatka; Data urodz; };
struct OgrodZool
{ int l_zwierzat; Zwierze* zwierzeta[500]; };
OgrodZool* zoo = new OgrodZool;
```
jak można odwołać się do roku urodzenia pierwszego zwierzęcia

103. Jakie wartości zawierać będzie tablica `a` po wykonaniu instrukcji
```
float a[5] = {1, 2, 3, 4, 5};
float* p = a; p = p+4; *p = 11;
```

104. Jakie wartości będą miały zmienne `i` oraz `j` po wykonaniu przypisań
```
int i = 5;
int &r = i;
int j = 10;
r = j;
r = 11;
```

105. Jaka będzie wartość zwrócona przez funkcję w wywołaniu `mniejszy("bcde", "azds")` dla funkcji zdefiniowanej poniżej
```
bool mniejszy(char c1, char c2)
{ return c1 < c2; }
int mniejszy(const char* s1, const char* s2)
{
  for (int i = 0; i < strlen(s1)
        && i < strlen(s2); i++)
    if (mniejszy(s1[i], s2[i])) return 1;
  return 0;
}
```

106. Jakie wartości będą miały zmienne `i` oraz `j` po wykonaniu przypisań
```
int i = 5;
int* r = &i;
int j = 10;
r = &j;
*r = 11;
```

107. Jaki będzie wynik działania programu
```
int g()
{
  int a = 12;
  static double x = 10;
  cout << a << ' ' << x << ',';
  a++;
  x *= 10;
}
int main()
{ g(); g(); g(); return 0; }
```

108. Jaka będzie wartość zmiennej `wyr`
```
double x = 4, y = 5, z = 2;
int i = 3;
double wyr = (x+y)*(1+z)/i+6;
```

109. Co wyświetli poniższy fragment programu
```
int opcja = 1;
switch (opcja) {
case -1: cout << 'A';
case 0: cout << 'B';
case 1: cout << 'C';
case 2: cout << 'D';
default: cout << 'E';
}
```

110. Wartością zmiennej `x` po wykonaniu następującego kodu `double a=4, b=2; double x=2/3*(a+b);` będzie

111. Wartością zmiennej x po wykonaniu następującego kodu `double a=4, b=2, c=3, d=2;`
`double x=d/c*(a+b);`
będzie

112. Jaka będzie wartość zmiennej `wyr`
```
double x = 2; double y = 10.0;
int i = 3; double wyr = 2 * y/x+i;
```

113. Jaki będzie wynik wywołania funkcji
```
int f(signed int a)
{ return (a>0)?f(a-1)+1:2; }
```
z argumentem 3;

114. Jeżeli zdeklarowano obiekty `float a; bool b;` a następnie użyto ich jako parametry aktualne funkcji `fun(a, b);`, to która z wersji tej funkcji przeciążonej zostanie wywołana

115. Jeżeli zdeklarowano obiekty `float a; int b;` a następnie użyto ich jako parametry aktualne funkcji `fun(a, b);`, to która z wersji tej funkcji przeciążonej zostanie wywołana

116. Jeżeli zdeklarowano obiekty `char a; bool b;` a następnie użyto ich jako parametry aktualne funkcji `fun(a, b);`, to która z wersji tej funkcji przeciążonej zostanie wywołana

117. Jeżeli zdeklarowano obiekty `int a; int b;` a następnie użyto ich jako parametry aktualne funkcji `fun(a, b);`, to która z wersji tej funkcji przeciążonej zostanie wywołana

118. Jeżeli zdeklarowano obiekty `int a[2]; int b;` a następnie użyto ich jako parametry aktualne funkcji `fun(a, b);`, to która z wersji tej funkcji przeciążonej zostanie wywołana

119. Jeżeli zdeklarowano obiekty `int a[2]; short b;` a następnie użyto ich jako parametry aktualne funkcji `fun(a, b);`, to która z wersji tej funkcji przeciążonej zostanie wywołana

120. Jeżeli zdeklarowano obiekty `const int a; int b[2];` a następnie użyto ich jako parametry aktualne funkcji `fun(a, b);`, to która z wersji tej funkcji przeciążonej zostanie wywołana

121. Jeżeli zdeklarowano obiekty `char a; int b;` a następnie użyto ich jako parametry aktualne funkcji `fun(a, b);`, to która z wersji tej funkcji przeciążonej zostanie wywołana

122. Jeżeli zdeklarowano obiekty `float a; float b;` a następnie użyto ich jako parametry aktualne funkcji `fun(a, b);`, to która z wersji tej funkcji przeciążonej zostanie wywołana

123. Obsługa sytuacji wyjątkowych jest mechanizmem

124. Procesowi zwijania stosu towarzyszy

125. Ponowne zgłoszenie sytuacji wyjątkowej

126. Specyfikacja sytuacji wyjątkowych w deklaracji funkcji gwarantuje

127. Wynik działania kodu
```
int main()
try {
  try {
    if (false) throw bool(0);
    else throw int(3);
  }
  catch (bool b) {cerr << b <<" "; throw;}
  return 0;
}
catch (...) {cerr << "blad";}
```
będzie następujący

128. Wynik działania kodu
```
int main ()
try {
  try {
    if (true) throw bool(0);
    else throw int(3);
  }
  catch (bool b) {cerr << b << " "; throw;}
  return 0;
}
catch (...) {cerr << "blad";}
```
będzie następujący

129. Po zgłoszeniu wyjątku we fragmencie poniżej
```
string s("Ala ma kota");
double *p = new double[100];
throw 1;
p[10] = 100;
```
automatycznie wywołany będzie/będą

130. Po zgłoszeniu wyjątku we fragmencie poniżej
```
void f() {
  auto_ptr<A> ap(new A);
  A* p = new A;
  throw 1;
  delete p;
};
```
zostanie zwolniona pamięć wskazywana przez

131. Wyjątek, który nie został przechwycony w programie na żadnym poziomie, powoduje

132. Próba skompilowania wywołania funkcji `void f() throw() { throw 1; }`

133. Prawidłowa deklaracja funkcji, gwarantująca że nie zostanie zgłoszona żadna sytuacja wyjątkowa to

134. Do obsługi sytuacji wyjątkowej dowolnego typu służy klauzula

135. Jeśli próba przydzielenia pamięci za pomocą operatora `new` nie powiodła sie, zgłaszana jest sytuacja wyjątkowa typu

136. Typ zdefiniowany przez użytkownika służący do definiowania pojęć, których nie można bezpośrednio wyrazić za pomocą typów wbudowanych to

137. Funkcja składowa klasy to

138. Dana składowa klasy to

139. Nazwa klasy poprzedzona słowem kluczowym `class` wraz z listą składowych ujętych w parę nawiasów klamrowych zakończona średnikiem to

140. Nazwa klasy poprzedzona słowem kluczowym `class` i zakończona średnikiem to

141. Wyodrębnianie i ukrywanie prywatnej, wewnętrznej reprezentacji i implementacji klasy to

142. Składowa obiektu dostępna w każdym miejscu bieżącego zasięgu oraz w odpowiednich zasięgach zagnieżdżonych to

143. Składowa obiektu dostępna jedynie w zasięgu klasy to

144. Składowa obiektu dostępna w tylko zasięgu klasy oraz w zasięgach klas potomnych

145. Zezwolenie pewnym funkcjom, metodom innych klas lub całym klasom na dostęp do składowych niepublicznych to

146. Metoda niewirtualna definiowana w definicji klasy jest automatycznie kwalifikowana jako

147. Metoda klasy gwarantująca niezmienność wartości pól obiektu to metoda zadeklarowana jako

148. Zezwolenie na modyfikowanie wartości pola obiektu metodami stałymi klasy wymaga modyfikatora

149. Wskaźnik na obiekt, na którym wykonywana jest funkcja składowa to

150. Wspólne pole wszystkich obiektów klasy to pole

151. Poprawna deklaracja metody statycznej klasy `Punkt` to

152. Konstruktor klasy to

153. Konstruktor niewymagający podawania argumentów aktualnych to

154. Inicjowanie jednego obiektu klasy wartością innego obiektu tej samej klasy realizuje się przez:

155. Dla obiektu klasy, w której nie zdefiniowano konstruktora kopiującego, podczas inicjowania wartością innego obiektu
156. Mechanizm przypisania składowa po składowej to mechanizm wykorzystywany
157. Destruktor klasy to metoda wywoływana
158. Przeciążenie operatora to definiowanie
159. Operatory, które muszą być definiowane jako metody klasy to
160. Kolejność wywoływania konstruktorów klas składowych zależy od
161. Prawidłowa deklaracja destruktora klasy Auto wygląda następująco:
162. Prawidłowa deklaracja destruktora klasy `B` wygląda następująco:
163. Prawidłową deklaracją konstruktora domyślnego dla klasy `Dom` jest:
164. Składowe do których maja dostęp tylko metody danej klasy i funkcje zaprzyjaźnione to składowe:
165. Między strukturą a klasą w C++ są następujące różnice:
166. Składowe klasy zadeklarowane bez specyfikatora dostępu domyślnie są:
167. Składowe struktury zadeklarowane bez specyfikatora dostępu domyślnie są:
168. Składowe do których maja dostęp wszystkie funkcje i metody to składowe:
169. Zadeklarowano obiekt klasy `A obj;` (zmiennna globalna). Prawidłowy dostęp do jej składowej statycznej `b` jest realizowany przez:
170. Zadeklarowano obiekt klasy `A obj;` (zmiennna globalna). Prawidłowy dostęp do jej składowej publicznej `b` jest realizowany przez:
171. Zadeklarowano obiekt klasy `A obj;` (zmiennna globalna) Prawidłowy dostęp do jej składowej prywatnej `b` jest realizowany przez:
172. Zdefiniowano wskaźnik na obiekt klasy (zmiennna globalna)
```
A* obj=new A;
```
Prawidłowy dostęp do jej składowej publicznej `b` jest realizowany przez:
173. Zdefiniowano wskaźnik na obiekt klasy (zmiennna globalna)
```
A* obj=new A;
```
Prawidłowy dostęp do jej publicznej składowej statycznej `b` jest realizowany przez:
174. Jaki jest błąd w poniższym kodzie?
```
class X{
  int a;
  static int b;
public:
  static void f(){cout<<a<<b;}
};
```
175. Jaki jest błąd w poniższym kodzie
```
class A {int a; public: void g() const
{a++;}};
```
176. W języku C++ nie da się przeładować operatora:
177. W języku C++ da się przeładować operator:
178. Jaki jest błąd w poniższym kodzie
```
class A {mutable int a; public: void g()
const {a++;}};
```
179. Jaki jest błąd w poniższym kodzie
```
class A{
private:
  int a;
public:
  A(int aa) {a=aa;}
};
A a;
```
180. Dla którego kodu zostanie wywołany konstruktor kopiujący?
181. Po utworzeniu obiektu klasy `A`
```
class A {
  int b;
public:
  A() : b(0) {}
  void f() {int b=10; b+=10;}
};
```
i wywołaniu metody `f()` pole `b` tego obiektu ma wartość:
182. Jaki błąd występuje w następującym kodzie
```
class A {public: A* a() {return this;} };
```
183. Definiowanie wersji operatora przeznaczonej dla argumentów będących obiektami klas to:
184. Jaki błąd występuje w następującym kodzie:
```
class A {public: static A* a(){return
this;}};
```
185. Prawidłowa deklaracja operatora postinkrementacji w treści klasy `A` wygląda następująco:
186. Które z poniższych stwierdzeń jest nieprawdziwe:
A) można definiować operatory których wszystkie argumenty są typami podstawowymi
B) nie można deklarować operatorów jako prywatnych metod klasy
C) operator nie będący metodą klasy musi być funkcją zaprzyjaźnioną
187. Które z poniższych stwierdzeń jest nieprawdziwe:
188. Dana jest klasa:
```
class A { public: A() {} };
```
Instrukcja
```
A a();
```
jest:
189. Zdefiniowano wskaźnik na obiekt klasy
```
A* a = new A;
```
Które z wywołań destruktora jest prawidłowe:
190. Co jest błędem w poniższym kodzie
```
class A { const int a; A() { a = 4; } };
```
191. Dana jest definicja klasy
```
class A {
public:
  A(int) { }
  A operator+(A) { return A(0); }
};
```
oraz definicja obiektów `A a1(1), a2(2);` Które z poniższych wywołań operatora `+` jest poprawne:
192. W definicji klasy, można inicjować pola:

193. Modyfikator, który powoduje że pole klasy jest wspólne dla wszystkich obiektów tej klasy to:

194. Fragment
```
class Punkt;
```
jest

195. Fragment
```
class K {
  int i;
  double x;
};
```
jest

196. Fragment
```
class Punkt { float x, y, z; };
Punkt p; p.x = 10;
```
jest

197. Które stwierdzenie jest nieprawdziwe?

198. W języku C++ `this` to

199. Mając klasy `Punkt` i `Linia` oraz deklarację funkcji globalnej
```
Punkt przeciecie(Linia l1, Linia l2);
```
określić, które stwierdzenie jest nieprawdziwe

200. Poniższa definicja zmiennych `p` i `r` typu `Punkt`
```
class Punkt {
public:
  Punkt(float x, float y);
  ~Punkt();
};
Punkt p, r(1, 4);
```

201. Poniższa definicja zmiennych `p` i `r` typu `Punkt`
```
class Punkt {
public:
  Punkt(float x = 0, float y = 0);
  ~Punkt();
};
Punkt p, r(1, 4);
```

202. Prawidłowa definicja funkcji składowej `odleglosc` w klasie
```
class Punkt
{ double odleglosc(Punkt inny) const; };
```
umieszczona poza definicją klasy, to

203. Mając klasę `Punkt`, w linii
```
Punkt* wierzcholki[4];
```

204. Zmienna `sam` w poniższym fragmencie
```
struct Auto { /*...*/ };
Auto* sam[4];
```

205. Prawidłowe wywołanie metody `odleglosc` na obiekcie `p1` z argumentem `p2` klasy
```
class Punkt {
  double odleglosc(Punkt inny);
};
```
to

206. Dla obiektu `p` klasy
```
class Punkt {
public:
  float x, y, z;
  static float x0, y0, z0;
};
```
które przypisanie zmiennej statycznej do niestatycznej jest prawidłowe

207. Modyfikator `const` na końcu nagłówka funkcji składowej
```
double A::f(int a, string s) const
{ /*...*/ }
```
oznacza, że

208. Które odwołanie do zmiennej składowej `x` obiektu typu
```
class Punkt {
  float x, y;
public:
  double odleglosc(Punkt inny);
};
```
w definicji metody `odleglosc` jest nieprawidłowe

209. Aby można było dodawać obiekty klasy `Wek`
```
Wek w1, w2, suma;
suma = w1 + w2;
```
przeładowany operator dodawania powinien być zadeklarowany jako

210. Mając klasę `Punkt`, w drugiej z poniższych linii
```
Punkt p;
Punkt p2 = p;
```
wywoływany jest

211. Mając zmienne `p` i `p2` klasy `LiczbaWym`, w linii
```
p2 = p;
```

212. Jeśli w definicji klasy nie zadeklarowano operatora `=`, to podczas przypisania `a = b` odbywa się kopiowanie

213. Jeśli w definicji klasy `K` nie zadeklarowano konstruktora kopiującego, to w linii
```
K obiekt(inny_obiekt);
```
odbywa się kopiowanie

214. Konstruktor klasy `K` nie jest automatycznie wywoływany

215. Konstruktor klasy `K` nie jest automatycznie wywoływany

216. Dla poniższych definicji
```
class A { /*...*/ };
class B {
  friend class A;
};
```

217. Definicja statycznej zmiennej składowej `s` klasy
```
class K { static double s; };
```
to np.

218. Wskaźnik `this` dostępny jest

219. Słowo kluczowe `friend` we fragmencie poniżej
```
class K { friend int f(double); };
```
oznacza, że

220. Prawidłowa deklaracja operatora wyjścia dla klasy `K` to

221. Prawidłowa deklaracja operatora wejścia dla klasy `K` to

222. W poniższym fragmencie, gdzie `LWym` to klasa liczb wymiernych
```
class LWym
{
public:
  LWym& operator += (LWym& l2);
  LWym operator + (LWym& l2);
};
LWym operator- (LWym& l1, LWym& l2);
LWym operator/ (int licznik, int mianownik);
```
nieprawidłowo zadeklarowano operator

223. Konstruktor `Punkt::Punkt(Punkt inny)`

224. W linii
```
K* wskaznik = new K(1, "abcd");
```

225. Obiekt klasy `K`, dla którego przydzielono dynamicznie pamięć w linii
```
K* wskaznik = new K;
```

226. Fragment
```
class A {
  int i;
  static int j;
public:
  static void f() { j = i; }
};
```
jest

227. Fragment
```
class A {
public:
  static double f(double x);
};
cout << A::f(2) << endl;
```
jest

228. Fragment
```
class A {
  int i;
public:
  void f() const { cout << i << endl; }
};
A obj; obj.f();
```
jest

229. Fragment
```
class A {
  int i;
public:
  void f() { i++; }
};
A obj; const A& ca = obj; ca.f();
```
jest

230. Fragment
```
class A {
public:
  void f();
};
void global(const A& a) { a.f(); }
```
jest

231. Fragment
```
class Tab {
public:
  double operator[] (int i) const;
  double& operator[] (int i);
};
Tab t; t[2] = 5;
const Tab* w = &t;
cout << (*w)[2] << endl;
```
jest

232. Które funkcje składowe klasy `A` można wywołać w funkcji `g`?
```
class A {
public:
  int c(int j) const;
  void m();
  static void s();
};
void g(A& a);
```

233. Które funkcje składowe klasy `A` można wywołać w funkcji `g`?
```
class A {
public:
  int c(int j) const;
  void m();
  static void s();
};
void g(const A& a);
```

234. W linii `C tab[3];`
gdzie `C` to
```
class C {
public:
  C(): i(1) {}
  C(int j): i(j) {}
protected:
  int i;
};
```

235. W linii `C* tab = new C[3];`
gdzie `C` to
```
class C {
public:
  C(): i(1) {}
  C(int j): i(j) {}
protected:
  int i;
};
```

236. W linii `C** tab = new C* [3];`
gdzie `C` to
```
class C {
public:
  C(): i(1) {}
  C(int j): i(j) {}
protected:
  int i;
};
```

237. W linii `C* c = new C(3) ;`
gdzie `C` to
```
class C {
public:
  C(): i(1) {}
  C(int j): i(j) {}
protected:
  int i;
};
```

238. Fragment
```
class A {
public:
  void operator = (const A& inny);
};
A a, b, c;
a = b; b = c;
```
jest

239. Fragment
```
class A {
public:
  void operator = (const A& inny);
};
A a, b, c;
a = b = c;
```
jest

240. Fragment
```
class A {
public:
  A& operator = (const A& inny);
};
A a, b, c;
a = b = c;
```
jest

241. Konstruktor kopiujący klasy `A`:
```
class A {
  int i;
public:
  A();
  A(const A& a);
};
class B {
  A zmienna;
public:
  B();
};
```
w linii `B b, b2(b);`

242. Która z poniższych deklaracji konstruktora klasy `K` jest prawidłowa:

243. Która z poniższych deklaracji konstruktora klasy `K` jest prawidłowa:

244. Która z poniższych definicji konstruktora klasy `K` jest prawidłowa:

245. Która z poniższych deklaracji destruktora klasy `K` jest prawidłowa:

246. Konstruktor to:

247. Destruktor jest:

248. Konstruktor kopiujący to:

249. Dana jest klasa `K` zawierająca pola `double x, y`. Która z poniższych definicji konstruktorów tej klasy jest nieprawidłowa:

250. Dana jest klasa `K` zawierająca pola `double x, y`. Która z poniższych definicji konstruktorów tej klasy jest prawidłowa:

251. Dana jest klasa:
```
class K {
  double x, y;
public:
  void metoda(void);
};
```
Który z poniższych zapisów jest prawidłowy:

252. Dana jest klasa:
```
class K {
  double x, y;
public:
  void metoda(void);
};
```
Który z poniższych zapisów wywołania metody jest prawidłowy:

253. Dana jest klasa:
```
class K {
  double x, y;
  void metoda(void);
};
```
Który z poniższych zapisów jest prawidłowy:

254. Dana jest klasa:
```
class K {
  double x, y;
public:
  static int a;
};
```
Który z poniższych zapisów jest nieprawidłowy:

255. Dana jest klasa:
```
class K {
  double x, y;
public:
  void metoda(void);
};
```
Który z poniższych zapisów jest nieprawidłowy:

256. Która z poniższych deklaracji jest nieprawidłowa:

257. Która z poniższych deklaracji jest prawidłowa:

258. Definicja
`struct K { private: int a; };`
odpowiada:

259. Definicja
`class K { public: int a; };`
odpowiada:

260. Który z poniższych zapisów jest nieprawidłowy:

261. Który z poniższych zapisów jest prawidłowy:

262. Który z poniższych zapisów jest prawidłowy:

263. Która z poniższych deklaracji operatora `<<` odpowiada następującemu jego wywołaniu:
`K obj; cout << obj << endl;`

264. Dana jest klasa:
```
class K {
  double x;
  const double y;
};
```
Który z poniższych zapisów jest prawidłowy:

265. Obiekty klasy posiadającej tylko jeden konstruktor, będący składową prywatną, mogą być definiowane:

266. Która z poniższych deklaracji konstruktora klasy `K` jest prawidłowa:

267. Która z poniższych deklaracji konstruktora klasy `K` jest prawidłowa:

268. Która z poniższych definicji konstruktora klasy `K` jest prawidłowa:

269. Konstruktor to:

270. Dana jest klasa `K` zawierająca pola `double x, y`. Który z poniższych zapisów jest nieprawidłowy:

271. Dana jest klasa:
```
class K{
  double x, y;
public:
  void metoda(void);
};
```
Który z poniższych zapisów wywołania metody jest prawidłowy:

272. Dana jest klasa:
```
class K {
  double x, y;
public:
  K metoda(void);
};
```
Który z poniższych zapisów jest prawidłowy:

273. Dana jest klasa:
```
class K {
  double x, y;
public:
  void metoda(K, double);
};
```
Który z poniższych zapisów jest prawidłowy:

274. Dana jest klasa:
```
class K {
  double x, y;
public:
  int metoda(K, double);
};
```
Który z poniższych zapisów jest prawidłowy:

275. Dana jest klasa:
```
class K {
  double x, y;
public:
  K metoda();
};
```
Który z poniższych zapisów jest prawidłowy:

276. Dana jest klasa:
```
class K {
  double x, y;
public:
  K& operator!(void);
  K& metoda(void);
};
```
Który z poniższych zapisów jest prawidłowy:

277. Która z poniższych deklaracji jest nieprawidłowa:

278. Która z poniższych deklaracji jest prawidłowa:

279. Który z poniższych zapisów jest nieprawidłowy:

280. Który z poniższych zapisów jest prawidłowy:

281. Która z poniższych deklaracji jest prawidłowa:

282. Dana jest klasa:
```
class K{double x; const double y;};
```
Który z poniższych zapisów jest prawidłowy:

283. Obiekty klasy posiadającej tylko jeden konstruktor, będący składową prywatną, mogą być definiowane:

284. Dla wyrażenia:
```
K obj; obj++;
```
zostanie wywołany:

285. Dla wyrażenia:
```
K obj; ++obj;
```
zostanie wywołany:

286. Dana jest klasa:
```
class K{
  double a;
  double b;
  double metoda(void);
};
```
Który z poniższych zapisów jest prawidłowy:

287. Dana jest klasa:
```
class K{
  double a;
  double b;
  double metoda(void);
};
```
Który z poniższych zapisów jest prawidłowy:

288. Dana jest klasa:
```
class K {
  double x;
  double y;
  K& metoda(void);
};
```
Który z poniższych zapisów jest prawidłowy:

289. Słowo kluczowe `static` może być wykorzystane wobec:

290. Dana jest klasa:
```
class K {
  double x;
  static double y;
};
```
Zmienna statyczna `y` zostanie zdefiniowana:

291. Dana jest klasa:
```
class K {
  double x;
  static double y;
};
```
Zmienna statyczna `y` zostanie zdefiniowana:

292. Dana jest klasa:
```
class K {
  double x;
  const static double y;
};
```
Który z poniższych zapisów w zasięgu globanym jest prawidłowy:

293. Wywołanie operatora
`double& K::operator[](double)`
może być

294. Wywołanie operatora
`double K::operator[](double)`
może być

295. Dana jest klasa:
```
class K {
  double x;
public:
  K(int x); // pierwszy
  K(double a = 0.0); // drugi
  K(const K& wzor); // trzeci
};
```
Który z zadeklarowanych konstruktorów jest domyślny:

296. Fragment
```
class G { public: void g(); };
G g[10];
```
jest

297. Fragment
```
class G {
public:
  G(int);
};
G g[10];
```
jest

298. Fragment
```
class G {
public:
  G(int);
};
G* g = new G[10];
```
jest

299. Fragment
```
class G {
public:
  G(int);
};
G g[] = {3, 4, 6, 8};
```
jest

300. Fragment
```
class G {
public:
  G();
};
G g[] = {3, 4, 6, 8};
```
jest

301. Fragment
```
class G {
public:
  G();
  G(double, int);
  G(int, const char*);
};
G g[3] = {G(2.2, 3), G(4, "ala"), G()};
```
jest

302. Specyfikatorem dostępu do składowych w języku C++ nie jest słowo kluczowe:

303. Składowe do których maja dostęp metody danej klasy i klas po niej dziedziczących oraz funkcje zaprzyjaźnione to składowe:

304. Jaki jest błąd w poniższym kodzie?
```
class Class {
public:
  virtual Class() {}
  virtual ~Class() {}
};
```

305. Jaki jest błąd w poniższym kodzie
```
class A {public: virtual void f()=0;};
class B:public A {public: void f(){}};
A a; B b;
```

306. Jaki jest błąd w poniższym kodzie
```
class A {public: virtual void f()=0;};
class B:public A {public: void f(int a){}};
B b;
```

307. Słowo kluczowe `dynamic` w języku C++ oznacza

308. Słowo kluczowe `explicit` jest używane do

309. Jaki jest błąd w poniższym kodzie
```
class B {int a[20]; public: virtual int&
operator[](int indeks)=0; };
```

310. Jaki jest błąd w poniższym kodzie
```
class A {
  int a[20];
public:
  virtual int& operator[](int indeks)
  {return a[indeks];}
};
```

311. W jakiej kolejności będą wywoływane konstruktory
```
class A {public: A(){} virtual void f()=0;};
class B: public A {public:B(){} void f(){}};
class C: public B {public:C(){} void f(){}};
```
dla obiektu klasy `C c;`

312. W jakiej kolejności będą wywoływane destruktory
```
class A {public:~A(){} virtual void f()=0;};
class B: public A {public:  B(){} void
f(){}};
class C: public B {public:  C(){} void
f(){}};
```
dla obiektu klasy `C c;`

313. Jaki jest błąd w poniższym kodzie
```
class A {public: virtual ~A()=0;};
```

314. Jaki błąd występuje w poniższym kodzie
```
class A {int n; A() {}};
class B : public A {public: B() : A() {} };
B b;
```

315. Jaki błąd występuje w poniższym kodzie
```
class A {int n; public: A() : n(0) {}};
class B : public A {public: B() : A(){}};
B b;
```

316. Jaki błąd występuje w poniższym kodzie
```
class A {int n; protected: A() : n(0) {}};
class B : public A {public: B() : A() {}};
B b;
```

317. Jaki błąd występuje w poniższym kodzie
```
class A {int n; protected: A() : n(0) {}};
class B : public A {public: B() : A() {}};
B b; A a;
```

318. W jakiej kolejności będą wywoływane konstruktory dla klasy `D`
```
class A {public: A() { /*...*/ }};
class B {public: B() { /*...*/ }};
class C {public: C() { /*...*/ }};
class D : public B, public C, public A
{public: D(): C(), B(), A() { /*...*/ }};
```

319. W jakiej kolejności będą wywoływane konstruktory dla klasy `D`
```
class A {public: A() { /*...*/ }};
class B {public: B() { /*...*/ }};
class C {public: C() { /*...*/ }};
class D: public A, public B, public C
{public: D() : B(), C(), A() { /*...*/ }};
```

320. W jakiej kolejności będą wywoływane konstruktory dla klasy `D`
```
class A {public: A() { /*...*/ }};
class B {public: B() { /*...*/ }};
class C {public: C() { /*...*/ }};
class D : public C, public B, public A
{public: D() : B(), A(), C() { /*...*/ }};
```

321. Co jest błędem w poniższym kodzie:
```
class A {
  int a;
public:
  A(int a) { this->a = a; }
};
class B : public A {
  int a;
public:
  B(int) { }
};
```

322. Co wyświetli poniższy program:
```
class A {
public:
  void print() { std::cout << "a "; }
};
class B : public A {
public:
  virtual void print() { std::cout << "b "; }
};
int main() {
  A *tab[] = { new A(), new B() };
  tab[0]->print(); tab[1]->print();
  return 0;
}
```

323. Która z poniższych konwersji jest niejawna:

324. Która z poniższych konwersji jest niejawna:

325. W której linii zaznaczono, że klasa `Poch` dziedziczy po klasie `Baz`

326. Dla poniższych definicji
```
struct Baz
{
  int p;
  char* s;
};
class Poch: public Baz
{ /*...*/ };
```
składowa `Poch::p` jest zmienną

327. Dla poniższych klas
```
class Baz
{
public:
  void f();
protected:
  int g();
private:
  double u(double x, int t);
};
class Poch: public Baz
{ /*...*/ };
```
które zdanie jest nieprawdziwe

328. Podczas przykrywania (overriding) funkcji wirtualnej `virtual void f(double x);`
klasy `Baz` w klasie pochodnej `Poch` prawidłowe wywołanie wersji odziedziczonej z klasy `Baz` to

329. Co wyświetli poniższy program?
```
class Baz
{
public:
  virtual void f() { cout << "Bf ";}
  void g() { cout << "Bg ";}
  static void h() { cout << "Bh ";}
};
class Poch: public Baz
{
public:
  void f() { cout << "Pf ";}
  void g() { cout << "Pg ";}
};
int main()
{
  Baz b;  Poch p;  Baz* wb = &p;
  b.f(); b.g();
  p.f(); p.g();
  wb->f(); wb->g();
}
```

330. Fragment
```
class B { protected: int bi; };
class D: public B { void f(B& obiekt); };
void D::f(B& obiekt)
{
  cout << obiekt.bi << endl;
  cout << bi << endl;
}
```
jest

331. Fragment
```
class B { /*...*/ };
class D: public B { /*...*/ };
void global(const B& b) { /*...*/ }
D obiekt; global(obiekt);
```
jest

332. Fragment
```
class B { /*...*/ };
class D: public B { /*...*/ };
void global(const D& d) { /*...*/ }
B obiekt; global(obiekt);
```
jest

333. Dana jest klasa `K` zawierająca pola `double x, y`. Który z poniższych zapisów jest prawidłowy:

334. Mechanizm pozwalający definiować nową klasę w oparciu o już istniejącą to

335. Relacja między typem a podtypem, dzięki której bez interwencji programisty wskaźnik lub referencja do obiektu klasy podstawowej mogą odnosić się do obiektu dowolnego podtypu klasy to

336. Klasa, której obiektów nie można tworzyć to

337. Deklarowanie w klasie pochodnej składowej o identyfikatorze użytym w klasie podstawowej to

338. Klasa pochodna nie ma dostępu do składowych

339. Klasa pochodna ma dostęp do składowych

340. Jeżeli w klasie pochodnej jest deklarowana metoda niewirtualna o nazwie, która nie występuje w klasie podstawowej, to za pomocą wskaźnika klasy podstawowej ustawionego na obiekt klasy pochodnej

341. Jeżeli w klasie pochodnej i podstawowej jest deklarowana metoda niewirtualna mająca taką samą nazwę to za pomocą wskaźnika klasy podstawowej ustawionego na obiekt klasy pochodnej

342. Które z poniższych zdań jest prawdziwe

343. Kolejność wywoływania konstruktorów klas podstawowych zależy od

344. Kolejność wywoływania destruktorów klas podstawowych zależy od

345. Fragment
```
class G {
public:
  explicit G(int);
};
G g[] = {3, 4, 6, 8};
```
jest

346. Gdy w funkcji
```
int f() throw (A, B);
```
zgłaszany jest wyjątek `E` niezgodny ze specyfikacją

347. Przyjmując, że klasa `B` jest klasą pochodną od klasy `A`, po wykonaniu następującego kodu
```
void foo() { throw B(); }
// ...
try { foo(); }
catch (const A& e) { cerr << "exA"; }
catch (const B& e) { cerr << "exB"; }
catch (...) { cerr << "exO"; }
```
na standardowym wyjściu diagnostycznym pojawi się napis

348. Przyjmując, że klasa `B` jest klasą pochodną od klasy `A`, po wykonaniu następującego kodu
```
void foo() { throw B(); }
// ...
try { foo(); }
catch (...) { cerr << "exO"; }
catch (const A& e) { cerr << "exA"; }
catch (const B& e) { cerr << "exB"; }
```
na standardowym wyjściu diagnostycznym pojawi się napis

349. Przyjmując, że klasy `B` i `C` są klasami pochodnymi od klasy `A`, po wykonaniu następującego kodu
```
void foo() { throw B(); }
// ...
try { foo(); }
catch (const C& e) { cerr << "exC"; }
catch (const A& e) { cerr << "exA"; }
catch (const B& e) { cerr << "exB"; }
catch (...) { cerr << "exO"; }
```
na standardowym wyjściu diagnostycznym pojawi się napis

350. Przyjmując, że klasa `B` jest klasą pochodną od klasy `A`, po wykonaniu następującego kodu
```
void foo() { throw B(); }
// ...
try { foo(); }
catch (const B& e) { cerr << "exB"; }
catch (const A& e) { cerr << "exA"; }
catch (...) { cerr << "exO"; }
```
na standardowym wyjściu diagnostycznym pojawi się napis

351. Przyjmując, że klasa `B` jest klasą pochodną od klasy `A`, po wykonaniu następującego kodu
```
void foo() { B b; A* a = &b; throw *a; }
// ...
try { foo(); }
catch (const B& e) { cerr << "exB"; }
catch (const A& e) { cerr << "exA"; }
catch (...) { cerr << "exO"; }
```
na standardowym wyjściu diagnostycznym pojawi się napis

352. Przyjmując, że klasa `B` jest klasą pochodną od klasy `A`, po wykonaniu następującego kodu
```
void foo() { B b; A* a = &b; throw *a; }
// ...
try { foo(); }
catch (const A& e) { cerr << "exA"; }
catch (const B& e) { cerr << "exB"; }
catch (...) { cerr << "exO"; }
```
na standardowym wyjściu diagnostycznym pojawi się napis

353. Przyjmując, że klasa `B` jest klasą pochodną od klasy `A`, po wykonaniu następującego kodu
```
void foo() { B b; A* a = &b; throw b; }
// ...
try { foo(); }
catch (const B& e) { cerr << "exB"; }
catch (const A& e) { cerr << "exA"; }
catch (...) { cerr << "exO"; }
```
na standardowym wyjściu diagnostycznym pojawi się napis

354. Przyjmując, że klasa `B` jest klasą pochodną od klasy `A`, po wykonaniu następującego kodu
```
void foo() { throw B(); }
// ...
void bar()
{
  try { foo(); }
  catch (const A& a)
     { cerr << "exA"; throw; }
}
// ...
try { bar(); }
catch (const B& e) { cerr << "exB"; }
catch (const A& e) { cerr << "exA"; }
```
na standardowym wyjściu diagnostycznym pojawi się napis

355. Przyjmując, że klasa `B` jest klasą pochodną od klasy `A`, po wykonaniu następującego kodu
```
void foo() { throw B(); }
// ...
void bar()
{
  try { foo(); }
  catch (const A& e)
     { cerr << "exA"; throw; }
}
// ...
try { bar(); }
catch (const A& e) { cerr << "exA"; }
catch (const B& e) { cerr << "exB"; }
catch (...) { cerr << "exO"; }
```
na standardowym wyjściu diagnostycznym pojawi się napis

356. Przyjmując, że klasy `B` i `C` są klasami pochodnymi od klasy `A`, po wykonaniu następującego kodu
```
void foo() { throw B(); }
// ...
void bar()
{
  try { foo(); }
  catch (const C& e)
    { cerr << "exC"; throw; }
}
// ...
try { bar(); }
catch (const A& e) { cerr << "exA"; }
catch (const B& e) { cerr << "exB"; }
catch (...) { cerr << "exO"; }
```
na standardowym wyjściu diagnostycznym pojawi się napis

357. Przyjmując, że klasy `B` i `C` są klasami pochodnymi od klasy `A`, po wykonaniu następującego kodu
```
void foo() { throw B(); }
// ...
void bar()
{
  try { foo(); }
  catch (const B& e)
    { cerr << "exB"; throw; }
}
// ...
try { bar(); }
catch (const C& e) { cerr << "exC"; }
catch (const B& e) { cerr << "exB"; }
catch (const A& e) { cerr << "exA"; }
```
na standardowym wyjściu diagnostycznym pojawi się napis

358. Przyjmując, że znana jest defincja klasy `A` poniższy kod
```
int foo() { throw A(); return 1; }
// ...
class B {
public:
  B() try : b(1) { foo(); }
  catch (const A& a) { cerr << "exA";}
protected:
  int b;
};
// ...
try { B b; }
catch (...) { cerr << "exO"; }
```

359. Przyjmując, że znana jest defincja klasy `A` poniższy kod
```
int foo() { throw A(); return 1; }
// ...
class B {
public:
  B() try : b(foo()) { A a; }
  catch (const A& a) { cerr << "exA";}
protected:
  int b;
};
// ...
try { B b; }
catch (...) { cerr << "exO"; }
```

360. Przyjmując, że znana jest defincja klasy `A` poniższy kod
```
int foo() { throw A(); return 1; }
// ...
class B {
public:
  B() : b(foo())
  {
    try { A a; }
    catch (const A& a) { cerr << "exA"; }
  }
protected:
  int b;
};
// ...
try { B b; }
catch (...) { cerr << "exO"; }
```

361. Przyjmując, że znana jest defincja klasy `A` poniższy kod
```
int foo() { throw A(); return 1; }
// ...
class B {
public:
  B() : b(1)
  {
    try { foo(); }
    catch (const A& a) { cerr << "exA"; }
  }
protected:
  int b;
};
// ...
try { B b; }
catch (...) { cerr << "exO"; }
```

362. W wyniku wykonania następującego kodu
```
class A {
public:
  char fun() {return 'a';}
};
class B : public A {
public:
  char fun() {return 'b';}
};
// ...
try { throw B(); }
catch (A a) { cout << a.fun() << endl; };
```

363. W wyniku wykonania następującego kodu
```
class A {
public:
  char fun() {return 'a';}
};
class B : public A {
public:
  char fun() {return 'b';}
};
// ...
try { throw B(); }
catch (A &a) { cout << a.fun() << endl; };
```

364. W wyniku wykonania następującego kodu
```
class A {
public:
  virtual char fun() {return 'a';}
};
class B : public A {
public:
  char fun() {return 'b';}
};
// ...
try { throw B(); }
catch (A a) { cout << a.fun() << endl; };
```

365. W wyniku wykonania następującego kodu
```
class A {
public:
  virtual char fun() {return 'a';}
};
class B : public A {
public:
  char fun() {return 'b';}
};
// ...
try { throw B(); }
catch (A &a) { cout << a.fun() << endl; };
```

366. Deklaracja funkcji
```
int fun() throw ();
```

367. Aby było możliwe zgłoszenie wyjątku będącego obiektem klasy

368. W której sekcji `catch` zostanie obsłużony zgłoszony wyjątek?
```
class E: public std::exception { };
// ...
try {
  // ...
  throw E();
  // ...
} catch (std::exception&) {
  //(1)
} catch (E&) {
  //(2)
} catch (...) {
  //(3)
}
```

369. W której sekcji `catch` zostanie obsłużony zgłoszony wyjątek?
```
class E: public std::exception { };
// ...
try {
  // ...
  throw E();
  // ...
} catch (E&) {
  //(1)
} catch (std::exception&) {
  //(2)
} catch (...) {
  //(3)
}
```

370. Zgłoszenie wyjątku z destruktora jest

371. Po zgłoszeniu wyjątku we fragmencie poniżej
```
void f() {
  vector<string> vs(2);
  vs[0] = "Ala";
  vs[1] = "ma kota";
  throw 1;
  vs[2] = "i psa";
};
```

372. Po zgłoszeniu wyjątku we fragmencie poniżej
```
void f() {
  vector<string*> vs(2);
  vs[0] = new string("Ala");
  vs[1] = new string("ma kota");
  throw 1;
  vs[2] = new string("i psa");
};
```

373. Przyjmując, że `sout` jest poprawnie zdefiniowanym obiektem formatowanego strumienia wyjściowego, poniższy kod
```
int w = 10; sout << noshowbase << hex << w;
```

374. Przyjmując, że `sout` jest poprawnie zdefiniowanym obiektem formatowanego strumienia wyjściowego, poniższy kod
```
int w = 16; sout << showbase << oct << w;
```

375. Przyjmując, że `xin` jest poprawnie zdefiniowanym obiektem formatowanego strumienia wejściowego, poniższy kod
```
float w = 8; xin << showpos << w;
```

376. Przyjmując, że `sout` jest poprawnie zdefiniowanym obiektem formatowanego strumienia wyjściowego, poniższy kod
```
float w = 8; bool f = false;
sout << scientific << w << boolalpha << f;
```

377. Przyjmując, że `sout` jest poprawnie zdefiniowanym obiektem formatowanego strumienia wyjściowego, poniższy kod
```
float w = -10; char t[] = "ala";
sout << uppercase << t << scientific << w ;
```

378. Przyjmując, że `sout` jest poprawnie zdefiniowanym obiektem formatowanego strumienia wyjściowego, poniższy kod
```
int w = -10;
sout.fill('#');
sout << setw(5) << internal << w ;
```

379. Przyjmując, że `xin` jest poprawnie zdefiniowanym obiektem formatowanego strumienia wejściowego, a w strumieniu tym znajduje się następujący ciąg znaków `alamakota`, poniższy kod
```
string s; xin >> setw(6) >> s;
```

380. Przyjmując, że `xin` jest poprawnie zdefiniowanym obiektem formatowanego strumienia wejściowego, a w strumieniu tym znajduje się następujący ciąg znaków `alamakota`, poniższy kod
```
string s; xin << setw(6) << s;
```

381. Przyjmując, że `xin` jest poprawnie zdefiniowanym obiektem formatowanego strumienia wejściowego, a w strumieniu tym znajduje się następujący ciąg znaków `10 20 30`, poniższy kod
```
int a = 0, b = 0, c = 0;
xin >> hex >> a >> dec >> b >> oct >> c;
```

382. Przyjmując, że `xin` jest poprawnie zdefiniowanym obiektem strumienia wejściowego, odczytanie bieżącej pozycji znacznika pobierania realizuje instrukcja

383. Przyjmując, że `xin` jest poprawnie zdefiniowanym obiektem strumienia wejściowego, ustawienie na końcu strumienia pozycji znacznika wstawiania realizuje instrukcja

384. Przyjmując, że `sout` jest poprawnie zdefiniowanym obiektem strumienia wyjściowego, odczytanie jego stanu realizowane jest instrukcją

385. Przyjmując, że `sout` jest poprawnie zdefiniowanym obiektem strumienia wyjściowego, przywrócenie domyślnego stanu strumienia realizowane jest instrukcją

386. Po wykonaniu następującego kodu
```
char a[10] = "Alicja";
int b = 0;
istringstream c("4 koty");
c >> a >> b;
cout << a << b;
```
do strumienia standardowego wyjścia zostanie wstawiony napis

387. Po wykonaniu następującego kodu
```
char a[10] = "Alicja";
int b = 0;
istringstream c("4 koty");
c >> b >> a;
cout << a << b;
```
do strumienia standardowego wyjścia zostanie wstawiony napis

388. Po wykonaniu następującego kodu
```
char a[10] = "Alicja";
int b = 0;
stringstream c("4 koty", ios::out);
c << a << b;
cout << c.str();
```
do strumienia standardowego wyjścia zostanie wstawiony napis

389. Po wykonaniu następującego kodu
```
char a[10] = "Alicja";
int b = 0;
stringstream c("4 koty", ios::out|ios::ate);
c << a << b;
cout << c.str();
```
do strumienia standardowego wyjścia zostanie wstawiony napis

390. Zmienna `line` przechowuje numer bieżącej linii pliku z danymi wejściowymi. W przypadku błędnych danych należy zgłosić sytuację wyjątkową (jedną z dostępnych w bibliotece standardowej) z informacją o numerze błędej linii

391. Obiekt
```
fstream plik("plik.txt", ios::in);
```
jest

392. Obiekt klasy `ostringstream` pozwala na

393. Po wykonaniu instrukcji
```
cout << setfill('#') << setw(5)
     << right << -12;
```
na standardowym wyjściu zostanie wypisane

394. Po wykonaniu instrukcji
```
cout << setfill('#') << setw(5)
     << left << -12;
```
na standardowym wyjściu zostanie wypisane

395. Po wykonaniu instrukcji
```
cout << setfill('#') << setw(5)
     << internal << -12;
```
na standardowym wyjściu zostanie wypisane

396. Po wykonaniu kodu
```
char tekst[ ] = "-10.5e2$";
istringstream in(tekst);
int d; char z;
in >> d >> z;
```
zmienne `d` i `z` uzyskają wartość

397. Po wykonaniu kodu
```
char tekst[ ] = "-10.5e2$";
istringstream in(tekst);
float f; char z;
in >> f >> z;
```
zmienne `f` i `z` uzyskają wartość

398. Jeśli na strumieniu `cin` wywołano metodę `cin.exceptions(ios::failbit | ios::badbit)`, to aby wykryć błąd strumienia, należy

399. Instrukcje
```
cout << "Tekst" << endl;
cout << "Tekst" << '\n';
```

400. Instrukcje
```
cout << "Tekst" << endl;
cout << "Tekst" << '\n' << flush;
```

401. Funkcję
```
void f(ostream& os);
```
można wywołać np. z parametrem

402. Prawidłowa postać deklaracji bezargumentowego manipulatora

403. Bezargumentowy manipulator `endl` jest w rzeczywistości

404. Jakie są minimalne wymagania dla typu `T` będącego parametrem następującego szablonu funkcji
```
template<typename T>
T xam(T a, T b)
{ return a > b ? a : b; }
```

405. Jakie są minimalne wymagania dla typu `T` będącego parametrem następującego szablonu funkcji
```
template<typename T>
const T& xam(const T& a, const T& b)
{ return a > b ? a : b; }
```

406. Niepoprawna deklaracja wzorca funkcji to

407. Niepoprawna deklaracja wzorca funkcji to

408. Konkretyzacja szablonu funkcji zadeklarowanego następująco
```
template<typename T>
const T& xam(const T& a, const T& b);
```
zakończy się niepowodzeniem dla wywołania

409. Która z konkretyzacji szablonu klasy
```
template<typename T = int>
class A {
public:
  A(const T& a, const T& b)
    : c(a < b ? a : b) { }
private:
  T c;
};
```
zakończy się niepowodzeniem

410. Dla szablonu klasy
```
template<typename T>
class C {
public:
  C(const T& a) { c = a; }
private:
  static T c;
};
```
poprawna definicja jej pola statycznego wygląda następująco

411. Dla szablonu klasy
```
template<typename T>
class C {
public:
  C(const T& a) { c = a; }
private:
  static double c;
};
```
poprawna definicja jej pola statycznego wygląda następująco

412. Dla następującego szablonu klasy
```
template<typename T> class A { };
```
poprawnym dziedziczeniem jest

413. Dla następujących szablonów klas
```
template<typename T> class A { };
class B: public A<int> { };
```
poprawną instrukcją jest

414. Dla następujących szablonów klas
```
template<typename T> class A { };
template<typename T>
class B: public A<T> { };
```
poprawną instrukcją jest

415. Dla szablonu klasy
```
template<typename T, int U>
class A { };
```
poprawną specjalizacją jest

A:
```
template<>
  class A<int, 2> { };
```

B:
```
template<typename T>
  class A<T, 3> { };
```

416. Dla szablonu klasy
```
template<typename T, int U>
class A { };
```
poprawną specjalizacją jest

A:
```
template<typename T, int U>
  class A<T*, U> { };
```

B:
```
template<typename T>
  class A<T, 3> { };
```

417. Dla wzorca funkcji
```
template<typename T, int s>
void f(T (&tab)[s]) { }
```
poprawna specjalizacja ma postać

418. Konkretyzowanie wzorca funkcji odbywa się

419. Dla wzorca klasy
```
template<typename T = int, int ile = 3>
class Kontener { };
```
i definicji
```
Kontener<double,66> k1;
Kontener k2;
Kontener<int> k3;
Kontener<5> k4;
```
poprawne zdefiniowano obiekty

420. Dziedziczenie
```
template <typename T>
class A { public: void fun(T) { }; };
class B : public A<int> { };
```

421. Prawidłowa deklaracja wzorca klasy `A` o parametrach `T` i `U` to

422. Prawidłowa definicja metody `f` wzorca klasy
```
template <typename T> class A {
  int f(double);
};
```
to

423. Prawidłowa definicja wzorca metody `f` klasy
```
class A {
  template <typename T> int f(T param);
};
```
to

424. Prawidłowa definicja wzorca metody `f`
```
template <typename T> class A {
  template <typename U> int f(U param);
};
```
to

425. Deklaracja zaprzyjaźnienia w
```
template <typename T> class A {
  friend
  ostream& operator << <> (ostream&, T);
};
```
oznacza

426. Dana jest definicja wzorca klasy
```
template <typename T> class A {
  typedef int type;
};
```
oraz deklaracja wzorca funkcji
```
template <typename U> int f(A<U>& param);
```
Definicja lokalnej zmiennej typu składowego `type` w definicji funkcji `f` to

427. Która z poniższych definicji zmiennej klasy wzorcowej
```
template <typename T, typename U = int>
class A { };
```
jest błędna?

428. Która z poniższych definicji zmiennej klasy wzorcowej
```
template <typename T = int>
class A { };
```
jest błędna?

429. Definicja pełnej specjalizacji wzorca klasy
```
template <typename T>
class A { };
```
dla typu `char` to

430. Jaką wartość przyjmie zmienna `c`
```
template<int n>
struct A {enum { b = A<n-1>::b*n }; };
template<> struct A<0>{enum { b=1 }; };
// ...
int c = A<3>::b;
```

431. Aby oddzielić kod źródłowy definicji szablonu klasy od definicji jej metod na dwa różne pliki (model separatywny), należy

432. Zasobniki skojarzeniowe (asocjacyjne) to

433. Iterator używany do zapisu elementu do zasobnika, mogący się przesuwać tylko do przodu o jeden element w danej chwili to

434. Iteratory o dostępie swobodnym obsługiwane są przez zasobniki

435. Struktury danych z STL bez obsługi iteratorów to

436. Działania, które można wykonywać na wszystkich rodzajach iteratorów to (gdzie `p` jest iteratorem)

437. Metoda `end()` zasobników sekwencyjnych zwraca

438. Metoda `swap()` jest składową

439. Na zasobniku sekwencyjnym `u` zawierającym elementy typu całkowitego `2, 3, 5, 8, 12, 18, 20` wykonano instrukcję
`u.erase(u.begin()+1);`
w efekcie otrzymano

440. Po wykonaniu instrukcji
`u.insert(u.end()-3, 10);`
na zasobniku sekwencyjnym `u`, zawierającym cztery elementy (typu całkowitego), każdy o wartości `5` otrzymano

441. Przy próbie wstawienia (metodą `insert()`) elementu o wartości `8` do zasobnika klasy `set<int>` zawierającego elementy `12, 3, 8, 5, 7`

442. Dla zasobnika `v` klasy `map<int, double>` zawierającego następujące pary (klucz, wartość) `(5, 100.1), (10,200.2), (20, 400.4), (40, 800.8)` wykonano operację `v[80]=1600.16`, w efekcie

443. Dla którego adaptora (łącznika) zasobnika poniższy szablon funkcji będzie działać poprawnie
```
template<class T>
void foo(T& a)
{
  while (!a.empty()) {
    cout << a.front() << ' ';
    a.pop();
  }
}
```

444. Dla zasobnika sekwencyjnego `v`, zawierającego elementy typu całkowitego `3, 5, 3, 7, 9, 6, 3`, wykonano instrukcję postaci
`replace(v.begin(), v.end()-1, 3, 0);`
w efekcie zawartość zasobnika zmieniła się na

445. Dla zasobnika sekwencyjnego `v`, zawierającego elementy typu całkowitego `3, 5, 3, 7, 9, 6, 3`, wykonano instrukcję
```
swap_ranges(v.begin(),
            v.begin()+3, v.begin()+3);
```
w efekcie zawartość zasobnika zmieniła się na

446. Wspólne metody dla wszystkich zasobników z biblioteki STL to

447. Po wykonaniu instrukcji
```
map<int, string> m;
m.insert(pair<int,string>(10, "kota"));
m.insert(make_pair(10, "ma"));
m.insert(make_pair<int, string>(10, "ma"));
```
Liczba elementów zasobnika `m` wynosi

448. Po wykonaniu instrukcji
```
multimap<int, string> m;
m.insert(pair<int,string>(10, "kota"));
m.insert(make_pair(10, "ma"));
m.insert(make_pair<int, string>(10, "ma"));
```
Liczba elementów zasobnika `m` wynosi

449. Wynik działania
```
vector<int> v(3, 3); v[2] = 2;
ostream_iterator<int> p(cout);
copy(v.begin(),
     remove(v.begin(),v.end(),2), p);
copy(v.begin(), v.end(), p);
```
będzie następujący

450. Wynik działania
```
vector<int> V(4,1); V[1]= 2;
list<int> L;
remove_copy(V.begin(), V.end(),
            front_inserter(L), 2);
copy(V.begin(),V.end(),
     ostream_iterator<int>(cout));
```
będzie następujący

451. Poniższy kod
```
list<int> L(2,0);
list<int>::iterator it =
  find_if(L.begin(), L.end(),
          bind2nd(equal_to<int>(), 0));
```
ma na celu

452. Poniższy kod ma na celu
```
int t[] = {11, 2, 13, 1, -7, 9, -1};
list<int> l(t, t+7);
l.sort(not2(less<int>()));
```

453. Adaptor iteratora zawracany przez `back_inserter()` może być użyty dla

454. Które ze zdań są prawdziwe

   A: iteratorów nie udostępniają zasobniki `stack`, `queue`

   B: `vector` udostępnia iteratory o dostępie swobodnym

   C: `reverse_iterator` daje możliwość zarówno odczytu jak i zapisu.

455. Po wykonaniu kodu
```
int t[] = {1,2,3};
list<int> l(3, 5);
copy(t, t+3, inserter(l, l.begin()));
```
zasobnik `l` będzie zawierał kolejno elementy

456. Po wykonaniu kodu
```
int t[] = {1,2,3};
list<int> l(3, 5);
copy(t, t+3, inserter(l, ++l.begin()));
```
zasobnik `l` będzie zawierał kolejno elementy

457. Po wykonaniu kodu
```
int t[] = {1,2,3};
list<int> l(3, 5);
copy(t, t+3, inserter(l, l.end()));
```
zasobnik `l` będzie zawierał kolejno elementy

458. Jeśli `v` jest zmienną typu `vector<int>`, `s = v.size()` i `c = v.capacity()`, to

459. Które stwierdzenie jest nieprawdziwe dla zasobnika `vector`?

460. Które stwierdzenie jest nieprawdziwe dla zasobnika `list`?

461. Jeśli `z` jest zasobnikiem sekwencyjnym, a `e = z.end()` i `rb = z.rbegin()` , to

462. Dla zmiennych
```
deque<int>::iterator i;
deque<int>::const_iterator ci;
deque<int>::reverse_iterator ri;
deque<int>::const_reverse_iterator rci;
```
błędne jest przypisanie

463. Dla zmiennych
```
list<int>::iterator i;
list<int>::const_iterator ci;
list<int>::reverse_iterator ri;
list<int>::const_reverse_iterator rci;
```
błędne jest przypisanie

464. Dla zmiennych
```
vector<int>::iterator i;
vector<int>::const_iterator ci;
vector<int>::reverse_iterator ri;
vector<int>::const_reverse_iterator rci;
```
błędne jest przypisanie

465. Które stwierdzenie jest nieprawdziwe dla stosu `stack`?

466. Które stwierdzenie jest nieprawdziwe dla kolejki `priority_queue`?

467. Jeśli zmienna `m` jest typu `map<string, int>`, to nie jest prawdą, że instrukcja `m["Ala"] = 5;`

468. Jeśli `s` jest zasobnikiem skojarzeniowym, to aby stwierdzić, że `s` nie zawiera elementu o kluczu `k`, wystarczy sprawdzić, czy prawdziwy jest warunek

469. Algorytmy są powiązane w następujący sposób z zasobnikami, na których operują

470. Algorytmy niezmieniające, np. `for_each`, zawdzięczają swoją nazwę temu, że

471. Algorytm `make_heap`

472. Wartość 2 po wykonaniu kodu
```
vector<int> a(10,0);
vector<int>::iterator i = a.begin();
a.resize(a.capacity()+1); *i = 2;
```
zostanie wpisana

473. Kolejka ze schematem LIFO (`stack`) w STL jest

474. Jakie podstawowe warunki musi spełniać obiekt, aby mógł być przechowywany w kontenerach STL

475. Generator to kategoria obiektów funkcyjnych

476. Predykat dwuargumentowy to kategoria obiektów funkcyjnych

477. Zasobnikiem sekwencyjnym nie jest

478. Zasobnikiem asocjacyjnym nie jest

479. W sytuacji gdy często wstawia się dane do zasobnika pomiędzy inne elementy tegoż zasobnika, należy wybrać ze względów wydajnościowych pojemnik

480. W sytuacji gdy, często wstawia się dane do zasobnika na koniec badź na początek tegoż zasobnika, należy wybrać ze względów wydajnościowych pojemnik

481. Który spośród wymienionych elementów biblioteki standardowej nie udostępnia iteratorów

482. Który spośród wymienionych elementów biblioteki standardowej udostępnia iterator o dostępie swobodnym (bezpośrednim)

483. W wyniku wykonania następującego kodu
```
list<int> lst(2);
vector<int> vec(2, 2);
copy(vec.begin(), vec.end(),
     back_inserter(lst));
```
zasobnik `lst` będzie zawierać – w kolejności – następujące elementy

484. W wyniku wykonania następującego kodu
```
list<int> lst;
vector<int> vec(2, 2);
copy(vec.begin(), vec.end(),
     back_inserter(lst));
```
zasobnik `lst` będzie zawierać – w kolejności – następujące elementy

485. W wyniku wykonania następującego kodu
```
list<int> lst;
vector<int> vec(2, 2);
copy(vec.begin(), vec.end(), lst.begin());
```
zasobnik `lst` będzie zawierać – w kolejności – następujące elementy

486. W wyniku wykonania następującego kodu
```
list<int> lst(2, 2);
vector<int> vec(2, 3);
copy(vec.begin(), vec.end(),
     front_inserter(lst));
```
zasobnik `lst` będzie zawierać – w kolejności – następujące elementy

487. W wyniku wykonania następującego kodu
```
int t[] = {1, 2, 2, 4, 2, 3};
vector<int> vec(t, t+6);
vec.erase(unique(vec.begin(), vec.end()),
          vec.end());
```
zasobnik `vec` będzie zawierał – w kolejności – następujące elementy

488. W wyniku wykonania następującego kodu
```
int t[] = {1, 2, 2, 4, 2, 3};
vector<int> vec(t, t+6);
sort(vec.begin(), vec.end());
vec.erase(unique(vec.begin(), vec.end()),
          vec.end());
```
zasobnik `vec` będzie zawierał – w kolejności – następujące elementy

489. Której z wymienionych właściwości nie udostępnia zasobnik `map`

490. Metody `operator[]` nie udostępnia

491. Jaką wartość będzie miała zmienna `sum` po wykonaniu następującego kodu
```
vector<int> vec(6, 1);
int sum = accumulate(vec.begin(),
                     vec.end(), 4);
```

492. Jaką wartość będzie miała zmienna `sum` po wykonaniu następującego kodu
```
vector<int> vec(6, 1);
int sum = accumulate(vec.begin(), vec.end(),
                     4, multiplies<int>());
```

493. Jaką wartość będzie miała zmienna `sum` po wykonaniu następującego kodu
```
vector<int> vec(6);
int sum = accumulate(vec.begin(),
                     vec.end(), 4);
```

494. W wyniku wykonania następującego kodu
```
int t[] = {1, 2, 2, 4, 2, 3};
vector<int> vec(t, t+6);
replace_if(vec.begin(), vec.end(),
           bind2nd(greater<int>(), 2), 1);
```
zasobnik `vec` będzie zawierał – w kolejności – następujące elementy

495. Jaki rozmiar będzie miał wektor `vec` po wykonaniu następującego kodu
```
int t[] = {1, 2, 2, 4, 2, 3};
vector<int> vec(t, t+6);
remove_if(vec.begin(), vec.end(),
          bind2nd(greater<int>(), 2));
```

496. Jaki rozmiar będzie miał wektor `vec` po wykonaniu następującego kodu
```
int t[] = {1, 2, 2, 4, 2, 3};
vector<int> vec(t, t+6);
greater<int> g;
vec.erase(remove_if(vec.begin(), vec.end(),
                    bind2nd(g, 2)),
          vec.end());
```

497. W wyniku wykonania następującego kodu
```
list<int> lst(2, 3);
int t[] = {1, 2, 4, 2, 3};
copy(t, t+3, front_inserter(lst));
```
zasobnik `lst` będzie zawierać – w kolejności – następujące elementy

498. W wyniku wykonania następującego kodu
```
int t[] = {1, 2, 4, 2, 3};
sort(t+1, t+4, less<int>());
```
tablica `t` będzie zawierać – w kolejności – następujące elementy

499. W wyniku wykonania następującego kodu
```
int t[] = {1, 2, 4, 2, 3};
sort(t+2, t+5, greater<int>());
```
tablica `t` będzie zawierać – w kolejności – następujące elementy

500. Dla następująco zdefiniowanej klasy
```
class A {
public:
  explicit A(const float&);
  A(const A&);
  bool operator< (const A&) const;
protected:
  bool operator> (const A&) const;
};
```
w części operacyjnej programu (funkcja `main()`) wykonano instrukcje
```
set<A> a; a.insert(A(3.14));
```
Instrukcje te są

501. Dla następująco zdefiniowanej klasy
```
class A {
public:
  explicit A(const float&);
  A(const A&);
  bool operator< (const A&) const;
protected:
  bool operator> (const A&) const;
};
```
w części operacyjnej programu (funkcja `main()`) wykonano instrukcje
```
set<A, greater<A> > a; a.insert(A(3.14));
```
Instrukcje te są

502. Dla następująco zdefiniowanej klasy
```
class A {
public:
  explicit A(const float&);
  A(const A&);
  bool operator< (const A&) const;
protected:
  bool operator> (const A&) const;
};
```
w części operacyjnej programu (funkcja `main()`) wykonano instrukcje
```
set<A, greater_equal<A> > a;
a.insert(A(3.14));
```
Instrukcje te są

503. Dla następująco zdefiniowanej klasy
```
class A {
public:
  explicit A(const float&);
  bool operator< (const A&) const;
protected:
  A(const A&);
  bool operator> (const A&) const;
};
```
w części operacyjnej programu (funkcja `main()`) wykonano instrukcje
```
multiset<A> a; a.insert(A(3.14));
```
Instrukcje te są

504. Dla następująco zdefiniowanej klasy
```
class A {
public:
  A();
  explicit A(const float&);
protected:
  A(const A&);
};
```
w części operacyjnej programu (funkcja `main()`) wykonano instrukcje
```
vector<A> a(5);
fill(a.begin(), a.end(), A(2.1));
```
Instrukcje te są

505. Dla następująco zdefiniowanej klasy
```
class A {
public:
  explicit A(const float&);
};
```
w części operacyjnej programu (funkcja `main()`) wykonano instrukcje
```
list<A> a(5);
fill(a.begin(), a.end(), A(2.1));
```
Instrukcje te są

506. Dla następująco zdefiniowanej klasy
```
class A {
public:
  explicit A(const float&);
};
```
w części operacyjnej programu (funkcja `main()`) wykonano instrukcje
```
list<A> a;
fill_n(back_inserter(a), 3, A(2.1));
```
Instrukcje te są

507. Dla następująco zdefiniowanej klasy
```
class A {
public:
  explicit A(const float&);
protected:
  A(const A&);
};
```
w części operacyjnej programu (funkcja `main()`) wykonano instrukcje
```
list<A> a;
fill_n(back_inserter(a), 3, A(2.1));
```
Instrukcje te są