**PODSTAWY C++**

**Dział: Krótko o programowaniu**

**Algorytm** to jednoznacznie określony sposób, w jaki program komputerowy realizuje

jakąś elementarną czynność.

**Język programowania** to forma zapisu instrukcji dla komputera i programów

komputerowych, pośrednia między językiem naturalnym a kodem maszynowym.

**Kompilator** – program zamieniający kod źródłowy, napisany w jednym z języków

programowania, na kod maszynowy w postaci oddzielnych modułów.

**Linker** łączy skompilowane moduły kodu i inne pliki w jeden plik wykonywalny, czyli

program (w przypadku Windows – plik EXE).

**Środowisko programistyczne** (ang. *integrated development environment* – w skrócie

IDE) to pakiet aplikacji ułatwiających tworzenie programów w danym języku

programowania. Umożliwia najczęściej organizowanie plików z kodem w projekty, łatwą

kompilację, czasem też wizualne tworzenie okien dialogowych.

Popularnie, środowisko programistyczne nazywa się po prostu kompilatorem (gdyż jest

jego główną częścią).

**Dział: Z czego składa się program?**

**Projekt** w środowiskach programistycznych to zbiór modułów kodu źródłowego i innych

plików, które po kompilacji i linkowaniu stają się pojedynczym plikiem EXE, czyli

programem.

**Procedura** to wydzielony fragment kodu programu, którego zadaniem jest wykonywanie

jakiejś czynności.

**Funkcja** zawiera kod, którego celem jest obliczenie i zwrócenie jakiejś wartości.

**Stała** to niezmienna wartość, której nadano nazwę celem łatwego jej odróżnienia od

innych, często podobnych wartości, w kodzie programu.

**Operator** to jeden lub kilka znaków (zazwyczaj niebędących literami), które mają

specjalne znaczenie w języku programowania.

**Dział: Działanie programu**

**Parametry funkcji** to dodatkowe dane, przekazywane do funkcji podczas jej wywołania.

**Pętla** to element języka programowania, pozwalający na wielokrotne, kontrolowane

wykonywanie wybranego fragmentu kodu.

**Dział: Operacje na zmiennych**

**Zasięg** (**zakres**, ang. *scope*) zmiennej to część kodu, w ramach której dana zmienna

jest dostępna.

**Zakres lokalny** obejmuje kod od miejsca deklaracji zmiennej aż do końca bloku, wraz z

ewentualnymi blokami zagnieżdżonymi.

**Przeciążanie funkcji** (ang. *function overloading*) to obecność kilku deklaracji funkcji o

tej samej nazwie, lecz posiadających różne listy parametrów i/lub typy zwracanej

wartości.

**Koniunkcja** (&&) jest **prawdziwa** tylko wtedy, kiedy **oba** jej argumenty są **prawdziwe**.

**Alternatywa** (||) jest **fałszywa** jedynie wówczas, gdy **oba** jej argumenty są **fałszywe**.

**Negacja** (!) powoduje **zmianę** prawdy na fałsz lub fałszu na prawdę.

**Dział: Złożone zmienne**

**Tablica** (ang. *array*) to zespół równorzędnych zmiennych, posiadających wspólną nazwę.

Jego poszczególne elementy są rozróżnianie poprzez przypisane im liczby - tak zwane

**indeksy**.

**Typy strukturalne** (zwane też w skrócie strukturami) to zestawy kilku zmiennych,

należących do innych typów, z których każda posiada swoją własną i **unikalną nazwę**.

Owe „podzmienne” nazywamy **polami** struktury.

**Interfejs użytkownika** (ang. *user interface*) to sposób, w jaki aplikacja prowadzi dialog

z obsługującymi ją osobami. Obejmuje to zarówno pobieranie od nich danych

wejściowych, jak i prezentację wyników pracy.

**Prototyp** (**deklaracja**) **funkcji** to wstępne określenie jej nagłówka. Stanowi on

informację dla kompilatora i programisty o sposobie, w jaki funkcja może być wywołana.

**Deklaracja zapowiadająca** (ang. *forward declaration*) to częściowe określenie jakiegoś

programistycznego bytu. Nie definiuje dokładnie wszystkich jego aspektów, ale wystarcza

do skorzystania z niego wewnątrz zakresu umieszczenia deklaracji.

Przykładem może być prototyp funkcji czy użycie słowa extern dla zmiennej.

**Obiekt** może reprezentować **cokolwiek**. Programista wykorzystuje obiekty jako cegiełki,

z których buduje gotowy program.

**Klasa** to złożony typ zmiennych, składający się z **pól**, przechowujących dane, oraz

posiadający **metody**, wykonujące zaprogramowane czynności.

**Prywatne** składowe klasy (wpisane po słowie private: w jej definicji) są dostępne

jedynie **wewnątrz samej klasy**, tj. tylko dla jej własnych metod.

**Publiczne** składowe klasy (wpisane po słowie public: w jej definicji) widoczne są

zawsze i **wszędzie** - nie tylko dla samej klasy (jej metod), ale **na zewnątrz** - np. dla jej

obiektów.

**Konstruktor** to specyficzna funkcja składowa klasy, wywoływana zawsze podczas

tworzenia należącego doń obiektu.

**Destruktor** jest specjalną metodą, przywoływaną podczas niszczenia obiektu

zawierającej ją klasy.

**Zmienne obiektowe** przechowuje obiekty w ten sam sposób, w jaki czynią to zwykłe

zmienne ze swoimi wartościami. Identycznie odbywa się też przypisywanie71 takich

zmiennych - tworzone są wtedy odpowiednie **kopie** obiektów.

**Wskaźnik na obiekt** jest jedynie **odwołaniem** do niego. Wykonanie przypisania do

wskaźnika może więc co najwyżej **skopiować owo odwołanie**, pozostawiając docelowy

obiekt całkowicie **niezmienionym**.

**Operator kropki** (.) pozwala uzyskać dostęp do składników obiektu zawartego w

**zmiennej obiektowej**.

**Operator strzałki** (->) wykonuje analogiczną operację dla **wskaźnika na obiekt**.

**Dział: Programowanie obiektowe**

**Dziedziczenie** (ang. *inheritance*) to tworzenie nowej klasy na podstawie jednej lub kilku

istniejących wcześniej klas bazowych.

**Metoda wirtualna** jest przygotowana na zastąpienie siebie przez nową wersję,

zdefiniowaną w klasie pochodnej.

Metody nazywane **czysto wirtualnymi** (ang. *pure virtual*) nie posiadają **żadnej**

**implementacji** i są przeznaczone **wyłącznie do przedefiniowania** w klasach

pochodnych.

**Klasa abstrakcyjna** zawiera przynajmniej jedną czysto wirtualną metodę i z jej powodu

nie jest przeznaczona do instancjowania (tworzenia z niej obiektów), a **jedynie do**

**wyprowadzania** zeń **klas pochodnych**.

**Polimorfizm** w programowaniu obiektowym oznacza wykorzystanie tego samego kodu

do operowania na obiektach przynależnych różnym klasom, dziedziczącym od siebie.

**Typ polimorficzny** to w C++ klasa zawierająca przynajmniej jedną **metodę wirtualną**.

**Singleton** (‘jedynak’) to klasa, której **jedyna instancja** (obiekt) spełnia kluczową rolę w

całym programie.

**Statyczne składowe** są przypisane do klasy jako całości, a nie do jej poszczególnych

instancji (obiektów).

**Obiekty zasadnicze** to główny budulec programu stworzonego według zasad OOP.

Wchodząc w zależności między sobą oraz przekazując dane, realizują one wszystkie

funkcje aplikacji.

**Obiekty narzędziowe**, zwane też **pomocniczymi** lub **konkretnymi**86, reprezentują

pewien nieskomplikowany typ danych. Zawierają pola służące przechowywaniu jego

danych oraz metody do wykonywania nań prostych operacji.

**Model abstrakcyjny** powinien opisywać założone działanie programu bez precyzowania

szczegółów implementacyjnych.

**Abstrakcja klasy** jest określeniem roli, jaką ta klasa pełni w programie.

W **związku jednokierunkowym** z pierwszego obiektu możemy otrzymać drugi, lecz

odwrotna sytuacja nie jest możliwa.

W **związku dwukierunkowym** oba obiekty mają do siebie wzajemny dostęp.

Nie należy stosować związków dwukierunkowych, jeżeli w konkretnym przypadku

wystarczą relacje jednokierunkowe.

**Dział: Wskaźniki**

**Wskaźnik** (ang. *pointer*) jest adresem pojedynczej komórki pamięci operacyjnej.

**Stos** (ang. *stack*) jest obszarem pamięci, który zostaje automatycznie przydzielony do

wykorzystania dla programu.

**Sterta** (ang. *heap*) to cała pamięć dostępna dla programu i mogąca być mu przydzielona

do wykorzystania.

**Wskaźniki w C++** to zmienne należące do specjalnych **typów wskaźnikowych**.

**Wskaźnik na stałą** (ang. *pointer to constant*) pokazuje na wartość, która może być

poprzez ten wskaźnik **jedynie odczytywana**.

Jeżeli wskaźnik **na stałą** jest dodatkowo wskaźnikiem **na obiekt**, to na jego rzecz

możliwe jest wywołanie jedynie **stałych metod**. Nie modyfikują one bowiem pól obiektu.

**Stały wskaźnik** (ang. *const(ant) pointer*) jest **nieruchomy**, na zawsze przywiązany do

**jednego adresu** pamięci.

Wskaźnik typu void\* może przechowywać **dowolny adres** z pamięci operacyjnej.

**Kolejne elementy** tablicy jednowymiarowej są ułożone **obok siebie**, w **ciągłym**

**obszarze** pamięci.

**Nazwa tablicy** jest także stałym **wskaźnikiem** do jej **zerowego elementu**

(**początku**).

**Łańcuch znaków w stylu C** to jednowymiarowa **tablica** elementów typu char.

**Zawsze** zwalniaj zaalokowaną przez siebie pamięć.

**Referencje** (ang. *references*) to zmienne wskazujące na adresy miejsc w pamięci, ale

pozwalające używać zwyczajnej składni przy odwoływaniu się do tychże miejsc.

W C++ występują **wyłącznie stałe referencje**. Po **koniecznej** inicjalizacji nie mogą już

być zmieniane.

**Wskaźnik do funkcji** (ang. *pointer to function*) to w C++ zmienna, która przechowuje

**adres**, pod jakim istnieje w pamięci operacyjnej dana **funkcja**.

Wyrażenie może być w programowaniu użyte jako instrukcja, natomiast instrukcja nie

może być użyta jako wyrażenie.

**Konwencja wywołania** (ang. *calling convention*) to określony sposób wywoływania

funkcji, precyzujący przede wszystkim kolejność przekazywania jej parametrów.

Parametry funkcji są w C++ przekazywane **przez wartości**.

**Typ wskaźnika na funkcję** określa typ zwracanej wartości, konwencję wywołania oraz

listę parametrów funkcji, na które wskaźnik może pokazywać i które mogą być za jego

pośrednictwem wywoływane.

**Nazwa funkcji** jest także **wskaźnikiem** do niej.

**ZAAWANSOWANE C++**

**Dział: Preprocesor**

**Preprocesor** to specjalny mechanizm języka, który przetwarza **tekst programu** jeszcze

**przed jego kompilacją**.

Dyrektywy preprocesora kończą się zawsze przejściem do następnego wiersza.

Preprocesor **nie dokonuje zastępowania** nazw makr **wewnątrz napisów**.

**Dział: Zaawansowana Obiektowość**

**Przyjaciel** (ang. *friend*) danej klasy ma **dostęp do jej wszystkich składników** - także

tych **chronionych**, a nawet **prywatnych**.

**Deklaracja przyjaźni** może być umieszczona w **każdym miejscu definicji klasy** i

zawsze ma **to samo znaczenie**.

**Jedna zwykła deklaracja przyjaźni** oznacza przyjaźń z **jedną funkcją**.

Funkcje zaprzyjaźnione z klasą **nie są jej składnikami**. Nie posiadają dostępu do

**wskaźnika this tej klasy**, gdyż **nie są jej metodami**.

**Deklaracja przyjaźni z funkcją** może być **jednocześnie deklaracją samej funkcji**.

Wcześniejsza wiedza kompilatora o istnieniu zaprzyjaźnianej funkcji **nie jest niezbędna**,

aby funkcja ta mogła zostać zaprzyjaźniona.

Wszystkie **funkcje zdefiniowane wewnątrz definicji klasy** są **automatycznie *inline***,

jednak tylko te **bez słówka friend są jej metodami**. **Pozostałe są funkcjami**

**globalnymi**, lecz **zaprzyjaźnionymi z klasą**.

Przyjaźń klas w C++ **nie jest automatycznie wzajemna**. Jeżeli klasa A deklaruje

przyjaźń z klasą B, to klasa B nie jest od razu także przyjacielem klasy A. Obiekty klasy B

mają więc dostęp do niepublicznych danych klasy A, lecz nie odwrotnie.

Klasa deklarująca przyjaźń udostępnia przyjacielowi swoje niepubliczne składowe - lecz

nie powoduje to od razu, że klasa zaprzyjaźniona jest tak samo otwarta.

Jeżeli klasa A deklaruje przyjaźń z klasą B, zaś klasa B z klasą C, to **nie znaczy to**, że

klasa C jest od razu przyjacielem klasy A.

Klasa ma **tylko tych przyjaciół**, których **sama sobie zadeklaruje**.

102

**Konstruktor domyślny** (ang. *default constructor*), zwany też domniemanym, jest to

taki konstruktor, który **może być wywołany bez podawania parametrów**.

**Inicjalizacja** (ang. *initialization*) jest to nadanie obiektowi **wartości początkowej** w

chwili jego tworzenia.

**Lista inicjalizacyjna** (ang. *initializers’ list*) ustala sposób inicjalizacji obiektów tworzonej

klasy.

Samowystarczalne obiekty mogą być kopiowane poprzez dosłowne przepisanie wartości

swoich pól.

**Konstruktor kopiujący** (ang. *copy constructor*) służy do **tworzenia nowego obiektu**

danej klasy na **podstawie już istniejącego**, innego obiektu tej klasy.

**Konstruktor kopiujący** jest wywoływany w momencie **inicjalizacji nowotworzonego**

**obiektu przy pomocy innego obiektu** tej samej klasy. Z tego powodu taki konstruktor

jest również zwany **inicjalizatorem kopiującym**.

**Konstruktor konwertujący** może przyjmować dokładnie **jeden parametr**

**określonego typu** i wykonywać jego konwersję na **typ swojej klasy**.

Jeżeli potrzebujesz **konstruktora jednoparametrowego**, który będzie działał

**wyłącznie jako zwykły** (a nie też jako konwertujący), umieść w jego deklaracji słowo

kluczowe explicit.

**Operatorów konwersji** możemy używać, aby zdefiniować niejawną **konwersję typu**

**swojej klasy na inny, dowolny typ**.

**Przeciążanie operatorów** (ang. *operator overloading*), zwane też ich

**przeładowaniem**, polega na nadawaniu operatorom **nowych znaczeń** - tak, aby mogły

być one **wykorzystane w stosunku do obiektów zdefiniowanych klas**.

**Operator** to jeden lub kilka znaków (zazwyczaj niebędących literami), które mają

specjalne znaczenie w języku programowania.

**Prein/dekrementacja** zwraca **wartość już zwiększoną (zmniejszoną) o 1**.

**Postin/dekrementacja** zwraca **oryginalną wartość**.

**L-wartość** (ang. *l-value*) jest wyrażeniem **mogącym wystąpić po lewej stronie**

**operatora przypisania** - stąd ich nazwa.

**R-wartość** (ang. *r-value*), po polsku zwana **p-wartością**, **może wystąpić tylko po**

**prawej stronie operatora przypisania**.

**Każda l-wartość** jest **jednocześnie r-wartością** (p-wartością) - lecz nie odwrotnie!

**Rezultatem przypisania** jest **przypisywana wartość**.

W C++ elementy tablic (oraz łańcuchów znaków) liczymy **od zera**.

const\_cast służy do **usuwania przydomków** const i volatile z opatrzonych nimi

wskaźników do zmiennych.

Funkcja operatorowa przyjmuje tyle argumentów, ile ma przeciążany przy jej pomocy

operator. Do tych argumentów **zalicza się wskaźnik this**, jeżeli funkcja operatorowa

jest metodą klasy.

**Globalna funkcja operatorowa** pozwala kompilatorowi na dokonywanie **niejawnych**

**konwersji** wobec **wszystkich argumentów** operatora.

**Globalna funkcja operatorowa nie musi być zaprzyjaźniona** z klasą, na rzecz której

definiuje znaczenie operatora.

Działanie operatorów wobec typów zdefiniowanych przez programistę powinno w miarę

możliwości pokrywać się z ich funkcjonalnością dla typów wbudowanych.

**Operatory przypisania** (zarówno prosty, jak i te złożone) muszą być zdefiniowane jako

**niestatyczna funkcja składowa** klasy, na której pracują.

**Operator przypisania** powinien zwracać **referencję do \*this**.

**Operator indeksowania** musi być zdefiniowany jako **niestatyczna metoda klasy**.

**Operator indeksowania** powinien **w wyniku zwracać l-wartość**.

**Operator wyłuskania ->** musi być **niestatyczną funkcją składową** klasy.

Funkcja operator->() dokonuje raczej **zamiany obiektu na wskaźnik** niż faktycznego

przedefiniowania znaczenia operatora ->.

**Inteligentny wskaźnik** (ang. *smart pointer*) to klasa będąca opakowaniem dla

normalnych wskaźników i zapewniająca wobec nich dodatkowe, „inteligentne”

zachowanie.

**Operator wywołania funkcji** może być zdefiniowany tylko jako **niestatyczna funkcja**

**składowa** klasy.

Przeciążone operatory new i delete mogą jedynie **zmienić sposób alokowania i**

**zwalniania pamięci**. Nie można ingerować w inicjalizację (wywołanie konstruktorów) i

sprzątanie (przywołanie destruktorów), które temu towarzyszą.

Symbol operatora powinien kojarzyć się z czynnością przez niego wykonywaną.

**Wskaźnik na pole klasy** jest określeniem **miejsca w pamięci**, jakie zajmuje pole

danej klasy, **względem początku obiektu w pamięci**.

**Wskaźniki do zwykłych funkcji** potrafią też pokazywać na **statyczne metody klas**.

**Wskaźnik do metody w klasie** (ang. *pointer-to-member function*) określa **miejsce**

**deklaracji tej metody** w **definicji klasy**.

**Dział: Wyjątki**

**Sytuacja wyjątkowa** (ang. *exceptional state*) ma miejsce wtedy, gdy **warunki**

**zewnętrzne** uniemożliwiają danemu fragmentowi kodu poprawne wykonanie. Ów

fragment **nie jest winny** zaistnienia sytuacji wyjątkowej.

Blok try obejmuje **kod**, w którym **może zajść sytuacja wyjątkowa**.

Instrukcja throw **wewnątrz bloku try** służy do **informowania** o takiej sytuacji przy

pomocy **obiektu wyjątku**.

Blok catch **przechwytuje obiekty** wyrzucone przez throw i **reaguje** na zainstaniałe

sytuacje wyjątkowe.

Typy w blokach catch są sprawdzane wedle ich **kolejności w kodzie**, a wybierana jest

**pierwsza pasująca** możliwość. Przy dopasowywaniu brane są pod uwagę **wszystkie**

**niejawne konwersje**.

Umieszczając kilka bloków catch jeden po drugim, zadbaj o to, aby występowały one w

porządku **rosnącej ogólności**. Niech **najpierw** pojawią się bloki o **najbardziej**

**wyspecjalizowanych typach**, a dopiero potem typy **coraz bardziej ogólne**.

Typ *A* jest **ogólniejszy** od typu *B*, jeżeli istnieje **niejawna konwersja z *B* do *A***,

niepowodująca utraty danych.

Wyjątki powinny być łapane w jak **najbliższym od ich rzucenia** miejscu, w którym

**możliwe jest ich obsłużenie**.

**Specyfikacja** albo **wyszczególnienie wyjątków** (ang. *exceptions’ specification*) mówi

nam, czy dana funkcja wyrzuca z siebie jakieś **nieobsłużone obiekty wyjątków**, a jeśli

tak, to informuje także o ich **typach**.

O **rzuceniu przez funkcję niezadeklarowanego wyjątku** dowiemy się dopiero **w**

**czasie działania programu**

Opuszczanie bloków kodu dokonywane podczas odwijania stosu przebiega tak samo, jak

to się dzieje podczas normalnego wykonywania programu. Obiekty lokalne są więc

**niszczone poprawnie**.

Nie należy rzucać następnego wyjątku w czasie, gdy **kompilator zajmuje się obsługą**

poprzedniego.

**Obsługa wyjątku dokonywana przez kompilator** polega na jego **dostarczeniu go**

**do odpowiedniego bloku catch** przy jednoczesnym **odwinięciu stosu**.

Nie powinno się wykorzystywać wyjątków tam, gdzie z powodzeniem wystarczają inne

techniki sygnalizowania i obsługi błędów.

Kod niższego poziomu powinien być zazwyczaj **niezależny** od kodu wyższego poziomu.

Gdy stosujemy **hierarchię klas wyjątków**, powinniśmy **najpierw** próbować **łapać**

**obiekty klas pochodnych**, a dopiero **potem obiekty klas bazowych**.

**Dział: Szablony**

**Szablon funkcji** reprezentuje zestaw (rodzinę) funkcji, działających dla dowolnej liczby

typów danych.

**Szablon funkcji** może być stosowany dla **tych typów danych**, dla których **poprawne**

**są wszystkie operacje**, dokonywane na **obiektach tychże typów** w **treści szablonu**.

**Parametry szablonu funkcji** są **dedukowane** w oparciu o **parametry jej wywołania**

oraz niejawne konwersje.

Dedukcja parametrów szablonu następuje **od końca** (od prawej strony). Te parametry,

które mogą być wzięte z wywołania funkcji, powinny zatem znajdować się na końcu listy

**Szablon klasy** reprezentuje zestaw (rodzinę) klas, mogących współpracować z różnymi

typami danych.

**Konstruktory** i **destruktory** w **szablonach klas** mają nazwy odpowiadające **nazwom**

**ich macierzystych szablonów i niczemu więcej**, tzn. **nie zawierają parametrów w**

**nawiasach ostrych**.

**Sama nazwa szablonu** może być stosowana **wewnątrz niego** w tych miejscach, gdzie

wymagany jest **typ klasy szablonowej**. Możemy więc posłużyć się nią do skrótowego

deklarowania pól, zmiennych czy parametrów funkcji bez potrzeby pisania nawiasów

ostrych i nazw parametrów szablonu.

Konkretyzacji podlegają **tylko te składowe klasy**, które są faktycznie **używane**.

Kompilator konkretyzuje **wyłącznie te metody klasy szablonowej**, które są

**używane**.

**Parametr szablonu** może mieć **wartość domyślną** tylko wtedy, gdy znajduje się na

**końcu listy** lub gdy wszystkie **parametry za nim też** mają wartość domyślną.

Slowa class i typename w są **synonimami** w deklaracjach **parametrów szablonu**

będących **typami**.

**Parametry pozatypowe szablonów** są traktowane wewnątrz nich jako **stałe**.

**Szablony klas** konkretyzowane **innym zestawem parametrów** są **zupełnie**

**odmiennymi typami**.

**Nazwa zależna** (ang. *dependent name*) to każda **nazwa użyta wewnątrz szablonu**,

**powiązana** w jakiś sposób **z jego parametrami**.

Należy **poprzedzać** słowem typename każdą **nazwę zależną**, która ma być

**interpretowana jako typ danych**.

**Kontenery** albo **pojemniki** (ang. *containers*) to specjalne struktury danych

przeznaczone do zarządzania kolekcjami obiektów tego samego typu w określony sposób.

Wszystkie te definicje zostały spisane na rzecz prowadzenia kursu „Od Zera Do Gier Kodera” i podlegają licencji GNU…

Karol Kuczmarski (Xion)